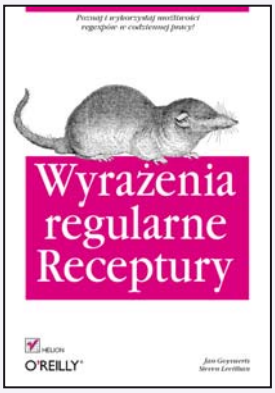
Wyrażenia regularne. Receptury



Autorzy: Jan Goyvaerts, Steven Levithan
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 978-83-246-2510-9
Tytuł oryginału: Regular Expressions Cookbook
Format: 168×237, stron: 520

Poznaj i wykorzystaj możliwości regexpów w codziennej pracy!

- Jak wyrażenia regularne mogą przyśpieszyć Twoją pracę?
- Jak sprawdzić poprawność danych?
- Jak wykorzystać wyrażenia regularne w pracy z plikami XML?

Wyrażenie regularne (ang. regexp) to inaczej wzorzec, który określa zbiór dopasowanych łańcuchów znaków. Brzmi to prosto. Jednak przy pierwszym spotkaniu z wyrażeniami wcale tak nie jest. Zbiór znaków i symboli składający się na wyrażenie regularne w niczym nie przypomina rzeczy, którą chciałbyś się zająć. Wyrażenia regularne zawsze kojarzą się początkującemu użytkownikowi co najmniej z wiedzą tajemną, a często wręcz z magią. Warto im się jednak przyjrzeć, poznać je i polubić, a następnie wykorzystać możliwości, jakie w nich drzemią.

Jedno jest pewne – te możliwości są spore. Autorzy błyskawicznie zaprzyjaźnią Cię z wyrażeniami regularnymi – książka należy bowiem do znanej serii Receptury, cechującej się tym, że proces nauki jest oparty na analizie rozwiązań prawdziwych problemów. Na samym początku zdobędziesz elementarną wiedzę dotyczącą różnych typów dopasowania oraz dowiesz się, jak unikać najczęstszych problemów.
Na kolejnych stronach nauczysz się stosować wyrażenia regularne w różnych językach programowania oraz wykorzystywać je do kontroli poprawności danych i formatowania ciągów znaków. Ponadto dowiesz się, jak operować na słowach, wierszach, znakach specjalnych oraz liczbach. Osobny rozdział został poświęcony operacjom na adresach URL oraz ścieżkach dostępu. Dzięki tej książce szybko zgłębisz tajniki wyrażeń regularnych. Kolejny krok to wykorzystanie tej wiedzy w codziennej pracy!

- Dopasowanie stałego tekstu
- Dopasowanie znaków niedrukowanych
- Dopasowania na początku i końcu wiersza
- Wyrażenia regularne dla całych wyrazów
- Wykorzystanie alternatywnych wyrażeń
- Grupowanie dopasowań
- Eliminowanie nawrotów
- Sposoby komentowania wyrażeń
- Wyrażenia regularne w językach programowania
- Weryfikacja i formatowanie danych z wykorzystaniem wyrażeń regularnych
- Dopasowanie kompletnego wiersza
- Praca z liczbami
- Operacje na adresach URL, ścieżkach i adresach internetowych
- Wykorzystanie wyrażeń regularnych w pracy z plikami XML

Sprawdź, jak wyrażenia regularne mogą przyśpieszyć Twoją pracę!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 3.

Programowanie z wykorzystaniem wyrażeń regularnych

Języki programowania i odmiany wyrażeń regularnych

W tym rozdziale wyjaśnimy, jak implementować wyrażenia regularne w wybranym przez Ciebie języku programowania. W recepturach składających się na ten rozdział zakładamy, że dysponujesz już prawidłowymi wyrażeniami regularnymi (w ich konstruowaniu powinny Ci pomóc poprzednie rozdziały). Koncentrujemy się więc tylko na zadaniu umieszczania wyrażeń regularnych w kodzie źródłowym i wykorzystywaniu ich do właściwego działania.

W tym rozdziale robimy, co w naszej mocy, aby możliwie precyzyjnie wyjaśnić, jak i dlaczego poszczególne fragmenty kodu działają w ten czy inny sposób. Właśnie z uwagi na wysoki poziom szczegółowości czytanie tego rozdziału od początku do końca może być dość nużące. Jeśli czytasz tę książkę po raz pierwszy, zachęcamy tylko do przejrzenia tego rozdziału, aby dysponować ogólną wiedzą o tym, co jest możliwe, a co jest konieczne. W przyszłości, kiedy będziesz implementował wyrażenia regularne proponowane w kolejnych rozdziałach, będziesz mógł wrócić do tego materiału, aby dokładnie dowiedzieć się, jak integrować te wyrażenia z wybranym językiem programowania.

W rozdziałach 4. – 8. będziemy wykorzystywali wyrażenia regularne do rozwiązywania rzeczywistych problemów programistycznych. W tych pięciu rozdziałach będziemy koncentrowali się na samych wyrażeniach regularnych, a wiele receptur w ogóle nie będzie zawierało kodu źródłowego. Aby wyrażenia prezentowane w tych rozdziałach mogły być stosowane w praktyce, należy je przenieść do fragmentów kodu źródłowego z niniejszego rozdziału.

Ponieważ w pozostałych rozdziałach koncentrujemy się na wyrażeniach regularnych, prezentujemy rozwiązania dla konkretnych odmian wyrażeń regularnych zamiast dla poszczególnych języków programowania. Odmiany wyrażeń regularnych nie są związane relacją jeden do jednego z odpowiednimi językami programowania. Języki skryptowe zwykle oferują własne, wbudowane odmiany wyrażeń regularnych, a pozostałe języki programowania najczęściej korzystają z odpowiednich bibliotek. Niektóre z tych bibliotek są dostępne w wersjach dla wielu języków programowania, a część języków oferuje swoim programistom więcej niż jedną bibliotekę.

W punkcie „Różne odmiany wyrażeń regularnych" w rozdziale 1. opisano wszystkie odmiany wyrażeń regularnych prezentowanych w tej książce. W punkcie „Zastępowanie tekstu w różnych odmianach" także w rozdziale 1. wymieniono odmiany zastępowania tekstu stosowane podczas operacji przeszukiwania i zastępowania danych z wykorzystaniem wyrażeń regularnych. Wszystkie języki programowania omawiane w tym rozdziale korzystają z jednej z tych odmian.

Języki programowania omawiane w tym rozdziale

W tym rozdziale omówimy siedem języków programowania. Każda receptura zawiera odrębne rozwiązania dla wszystkich ośmiu języków programowania, a w wielu recepturach sporządzono nawet osobne analizy rozwiązań pod kątem poszczególnych języków. Jeśli jakaś technika ma zastosowanie w więcej niż jednym języku, wspominamy o niej w analizie dla każdego z tych języków. Zdecydowaliśmy się na takie rozwiązanie, abyś mógł bezpiecznie pomijać języki programowania, którymi nie jesteś zainteresowany.

C#
: Język programowania C# korzysta z frameworku Microsoft .NET. Klasy przestrzeni nazw `System.Text.RegularExpressions` stosują więc odmianę wyrażeń regularnych i zastępowania tekstu, które w tej książce nazywamy odmianami platformy .NET. W tej książce omówimy język C# w wersjach od 1.0 do 3.5 (stosowane odpowiednio w środowiskach Visual Studio od wersji 2002 do wersji 2008).

VB.NET
: W tej książce będziemy używali terminów VB.NET i Visual Basic.NET w kontekście języka programowania Visual Basic 2002 i nowszych, aby uniknąć mylenia tych wersji z językiem Visual Basic 6 i starszymi. Współczesne wersje Visual Basica korzystają z frameworku Microsoft .NET. Wspomniana już przestrzeń nazw `System.Text.RegularExpressions` implementuje odmianę wyrażeń regularnych i zastępowania tekstu, które w tej książce nazywamy odmianami platformy .NET. W tej książce ograniczymy się do prezentacji języka Visual Basic w wersjach 2002 – 2008.

Java
: Java 4 jest pierwszym wydaniem oferującym wbudowaną obsługę wyrażeń regularnych w formie pakietu `java.util.regex`. Właśnie pakiet `java.util.regex` implementuje odmianę wyrażeń regularnych i zastępowanego tekstu, które w tej książce nazywamy odmianą Javy. W tej książce omawiamy Javę 4, 5 i 6.

JavaScript
: Tę odmianę wyrażeń regularnych stosuje się w języku programowania powszechnie znanym jako JavaScript. Wspomniany język jest implementowany przez wszystkie współczesne przeglądarki internetowe: Internet Explorer (przynajmniej w wersji 5.5), Firefox, Opera, Safari oraz Chrome. Także wiele innych aplikacji wykorzystuje JavaScript w roli języka skryptowego.

  Precyzyjnie mówiąc, w tej książce będziemy używali terminu **JavaScript** w kontekście języka programowania zdefiniowanego w trzeciej wersji standardu ECMA-262. Wspomniany standard definiuje język programowania ECMAScript znany lepiej dzięki implementacjom nazwanym JavaScript i JScript, oferowanym w rozmaitych przeglądarkach internetowych.

Standard ECMA-262v3 definiuje też stosowane w JavaScripcie odmiany wyrażeń regularnych i zastępowanego tekstu. W tej książce będziemy określali te odmiany mianem odmian JavaScriptu.

PHP

PHP oferuje trzy zbiory funkcji operujących na wyrażeniach regularnych. Ponieważ sami jesteśmy zwolennikami korzystania z rodziny funkcji `preg`, w tej książce będziemy koncentrowali się właśnie na nich (dostępnych począwszy od wydania PHP 4.2.0). W tej książce omówimy język PHP 4 i 5. Funkcje z rodziny `preg` są w istocie opakowaniami funkcji biblioteki PCRE. Odmianę wyrażeń regularnych implementowaną przez tę bibliotekę będziemy nazywali odmianą PCRE. Ponieważ jednak biblioteka PCRE nie oferuje funkcji przeszukiwania i zastępowania, twórcy języka PHP opracowali własną składnię zastępowanego tekstu na potrzeby funkcji `preg_replace`. Samą odmianę zastępowanego tekstu nazywamy w tej książce odmianą PHP.

Funkcje z rodziny `mb_ereg` wchodzą w skład zbioru tzw. funkcji wielobajtowych języka PHP, które zaprojektowano z myślą o językach tradycyjnie kodowanych za pomocą wielobajtowych zbiorów znaków, na przykład o językach japońskim i chińskim. W PHP 5 funkcje `mb_ereg` korzystają z biblioteki wyrażeń regularnych Oniguruma, którą początkowo tworzono dla języka programowania Ruby. Odmianę wyrażeń regularnych zaimplementowaną w bibliotece Oniguruma będziemy nazywali odmianą języka Ruby 1.9. Stosowanie funkcji z rodziny `mb_ereg` zaleca się tylko tym programistom, którzy muszą operować na wielobajtowych stronach kodowych i którzy opanowali już techniki korzystania z funkcji `mb_`.

Grupa funkcji `ereg` to najstarszy zbiór funkcji PHP stworzonych z myślą o przetwarzaniu wyrażeń regularnych. Funkcje z tego zbioru oficjalnie uznano za przestarzałe i niezalecane wraz z wydaniem PHP 5.3.0. Funkcje `ereg` nie korzystają z żadnych bibliotek zewnętrznych i implementują odmianę POSIX ERE. Wspomniana odmiana oferuje jednak dość ograniczony zakres funkcji i jako taka nie jest omawiana w tej książce. Funkcje odmiany POSIX ERE stanowią podzbiór funkcji oferowanych przez odmiany języka Ruby 1.9 i biblioteki PCRE. Każde wyrażenie regularne obsługiwane przez funkcje `ereg` jest obsługiwane także przez funkcje z rodziny `mb_ereg` lub `preg`. Funkcje `preg` wymagają jednak stosowania separatorów Perla (patrz receptura 3.1).

Perl

Wbudowana obsługa wyrażeń regularnych Perla to jeden z głównych powodów obserwowanej obecnie popularności tych wyrażeń. Odmiany wyrażeń regularnych i zastępowanego tekstu wykorzystywane przez operatory `m//` i `s///` języka Perl nazywamy w tej książce odmianami Perla. Skoncentrujemy się na wersjach 5.6, 5.8 i 5.10.

Python

W języku Python obsługę wyrażeń regularnych zaimplementowano w module `re`. W tej książce odmiany wyrażeń regularnych i zastępowanego tekstu nazywamy odmianami Pythona. W książce omawiamy język Python w wersjach 2.4 i 2.5.

Ruby

Język Ruby oferuje wbudowaną obsługę wyrażeń regularnych. W tej książce omówimy wersje 1.8 i 1.9 tego języka. Wymienione wersje języka Ruby domyślnie stosują różne moduły wyrażeń regularnych. Język Ruby 1.9 korzysta z modułu Oniguruma, który oferuje nieporównanie więcej funkcji niż klasyczny silnik stosowany w domyślnej kompilacji języka 1.8. Szczegółowych informacji na ten temat należy szukać w punkcie „Odmiany wyrażeń regularnych prezentowane w tej książce" w rozdziale 1.

W tym rozdziale nie będziemy poświęcać zbyt wiele uwagi różnicom dzielącym moduły wyrażeń regularnych wersji 1.8 i 1.9. Wyrażenia prezentowane w tym rozdziale będą na tyle proste, że nie będą potrzebne nowe funkcje zaimplementowane w języku Ruby 1.9. Ponieważ mechanizmy odpowiedzialne za obsługę wyrażeń regularnych są włączane do samego języka Ruby na etapie kompilacji, kod wykorzystywany do implementowania wyrażeń regularnych jest taki sam niezależnie od wybranego modułu (klasycznego lub biblioteki Oniguruma). Oznacza to, że istnieje możliwość ponownej kompilacji języka Ruby 1.8, aby korzystał z biblioteki Oniguruma (jeśli na przykład potrzebujemy rozszerzonych funkcji tej biblioteki).

Inne języki programowania

Języki programowania wymienione na poniższej liście nie będą omawiane w tej książce, mimo że korzystają z prezentowanych przez nas odmian wyrażeń regularnych. Jeśli pracujesz w którymś z tych języków, możesz pominąć ten rozdział i jednocześnie z powodzeniem korzystać z materiału zawartego w pozostałych rozdziałach.

ActionScript
: ActionScript jest implementacją standardu ECMA-262 opracowaną przez firmę Adobe. W wersji 3.0 język ActionScript zawiera pełną obsługę wyrażeń regularnych zdefiniowanych w standardzie ECMA-262v3. W tej książce będziemy nazywali tę odmianę odmianą JavaScriptu. Język ActionScript jest bardzo podobny do języka JavaScript, zatem przeniesienie fragmentów kodu JavaScriptu do języka ActionScript nie powinno Ci sprawić najmniejszego problemu.

C
: Programiści języka C mają do dyspozycji wiele różnych bibliotek wyrażeń regularnych. Biblioteka PCRE typu open source jest bodaj najlepszym rozwiązaniem tego typu spośród wszystkich odmian omówionych w tej książce. Kompletny kod źródłowy tej biblioteki (w języku C) można pobrać z witryny internetowej *http://www.pcre.org*. Kod napisano w taki sposób, aby umożliwić jego kompilację z wykorzystaniem rozmaitych kompilatorów dla wielu różnych platform.

C++
: Także programiści języka C++ mają do wyboru wiele różnych bibliotek wyrażeń regularnych. Biblioteka PCRE typu open source jest bodaj najlepszym rozwiązaniem tego typu spośród wszystkich odmian omówionych w tej książce. Istnieje możliwość korzystania albo bezpośrednio z interfejsu API języka C, albo z opakowań w formie klas języka C++ dostępnych wraz z samą biblioteką PCRE (patrz witryna internetowa *http://www.pcre.org*).

  W systemie Windows można dodatkowo zaimportować obiekt COM nazwany VBScript 5.5 RegExp (patrz materiał poświęcony językowi Visual Basic 6). Takie rozwiązanie jest korzystne, jeśli chcemy zachować spójność wewnętrznych mechanizmów zaimplementowanych w C++ i elementów interfejsu zaimplementowanych w JavaScripcie.

Delphi dla platformy Win32
: W czasie, kiedy pisano tę książkę, wersja języka Delphi dla platformy Win32 nie oferowała żadnych wbudowanych mechanizmów obsługi wyrażeń regularnych. Istnieje jednak wiele komponentów VCL implementujących obsługę wyrażeń regularnych. Sami polecamy wybór komponentu stworzonego na bazie biblioteki PCRE. Delphi oferuje możliwość

dołączania do budowanych aplikacji plików wynikowych języka C — większość opakowań biblioteki PCRE w formie komponentów VCL ma postać właśnie takich plików wynikowych. Takie rozwiązanie umożliwia umieszczanie aplikacji w pojedynczych plikach *.exe*.

Komponent nazwany TPerlRegEx (mojego autorstwa) można pobrać ze strony internetowej *http://www.regexp.info/delphi.html*. TPerlRegEx ma postać komponentu VCL instalowanego automatycznie w palecie komponentów, zatem jego przeciąganie na formularz nie stanowi żadnego problemu. Innym popularnym opakowaniem biblioteki PCRE dla Delphi jest klasa `TJclRegEx` wchodząca w skład biblioteki `JCL` (dostępnej pod adresem *http://www.delphi-jedi.org*). Ponieważ jednak `TJclRegEx` jest klasą potomną klasy `TObject`, nie jest możliwe jej przenoszenie na formularz.

Obie biblioteki mają charakter oprogramowania open source i są oferowane na zasadach licencji Mozilla Public License.

Delphi Prism

W Delphi Prism można wykorzystać mechanizm obsługi wyrażeń regularnych zaimplementowany w ramach frameworku .NET. Wystarczy do klauzuli `uses` dodać przestrzeń nazw `System.Text.RegularExpressions`, aby dana jednostka języka Delphi Prism mogła korzystać ze wspomnianej implementacji wyrażeń regularnych.

Po wykonaniu tego kroku można z powodzeniem stosować te same techniki, które w tym rozdziale proponujemy dla języków C# i VB.NET.

Groovy

Podobnie jak w Javie, w języku Groovy do obsługi wyrażeń regularnych można wykorzystać pakiet `java.util.regex`. W praktyce wszystkie prezentowane w tym rozdziale rozwiązania dla Javy powinny działać prawidłowo także w języku Groovy. Składnia wyrażeń regularnych tego języka różni się tylko dodatkowymi skrótami notacji. Stałe wyrażenie regularne otoczone prawymi ukośnikami jest traktowane jako obiekt klasy `java.lang.String`, a operator `=~` tworzy obiekt klasy `java.util.regex.Matcher`. Możemy swobodnie mieszać składnię języka Groovy ze standardową składnią Javy, ponieważ w obu przypadkach korzystamy z tych samych klas i obiektów.

PowerShell

PowerShell jest językiem skryptowym firmy Microsoft zaprojektowanym na bazie frameworku .NET. Wbudowane operatory `-match` i `-replace` tego języka korzystają z odmian wyrażeń regularnych i zastępowanego tekstu platformy .NET, czyli z odmian prezentowanych w tej książce.

R

W projekcie R zaimplementowano obsługę wyrażeń regularnych za pośrednictwem funkcji `grep`, `sub` i `regexpr` pakietu `base`. Wszystkie te funkcje otrzymują na wejściu argument oznaczony etykietą `perl`, który — w razie pominięcia — ma przypisywaną wartość `FALSE`. Jeśli za pośrednictwem tego argumentu przekażemy wartość `TRUE`, wymusimy użycie opisanej w tej książce odmiany wyrażeń regularnych biblioteki PCRE. Wyrażenia regularne tworzone z myślą o bibliotece PCRE 7 mogą być z powodzeniem stosowane w języku R, począwszy od wersji 2.5.0. W starszych wersjach tego języka należy stosować wyrażenia regularne, które w tej książce opisujemy jako tworzone z myślą o bibliotece PCRE 4 lub nowszych. Obsługiwane w języku R odmiany „podstawowa" i „rozszerzona", które są starsze i mocno ograniczone, nie będą omawiane w tej książce.

REALbasic

Język REALbasic oferuje wbudowaną klasę `RegEx`. Wspomniana klasa wewnętrznie wykorzystuje bibliotekę PCRE w wersji przystosowanej do pracy z formatem UTF-8. Oznacza to, że istnieje możliwość korzystania z biblioteki PCRE w wersji z obsługą standardu Unicode, jednak konwersja znaków spoza zbioru ASCII na znaki UTF-8 (przed przekazaniem do klasy `RegEx`) wymaga użycia klasy `TextConverter` języka REALbasic.

Wszystkie prezentowane w tej książce wyrażenia regularne dla biblioteki PCRE 6 można z powodzeniem stosować także w języku REALbasic. Warto jednak pamiętać, że w tym języku opcje ignorowania wielkości liter i dopasowywania znaków podziału wiersza do karety i dolara (tzw. tryb wielowierszowy) są domyślnie włączone. Oznacza to, że jeśli chcesz używać w języku REALbasic wyrażeń regularnych, które nie wymagają włączenia tych trybów dopasowywania, powinieneś je wprost wyłączyć.

Scala

Język Scala oferuje wbudowaną obsługę wyrażeń regularnych w formie pakietu `scala.↪util.matching`. Pakiet ten zaprojektowano na podstawie modułu wyrażeń regularnych stosowanego w Javie (czyli pakietu `java.util.regex`). Odmiany wyrażeń regularnych i zastępowanego tekstu obowiązujące w językach Java i Scala nazywamy w tej książce po prostu odmianami Javy.

Visual Basic 6

Visual Basic 6 był ostatnią wersją tego języka, która nie wymagała frameworku .NET. Oznacza to, że programiści korzystający z tej wersji nie dysponują doskonałymi mechanizmami obsługi wyrażeń regularnych tego frameworku. Przykładów kodu języka VB.NET prezentowanych w tym rozdziale nie można więc przenosić do języka VB 6.

Z drugiej strony Visual Basic 6 znacznie ułatwia korzystanie z funkcji implementowanych przez biblioteki ActiveX i COM. Jednym z takich rozwiązań jest biblioteka skryptowa VBScript firmy Microsoft. Począwszy od wersji 5.5, w bibliotece VBScript implementowano uproszczoną obsługę wyrażeń regularnych. Wspomniana biblioteka skryptowa implementuje tę samą odmianę wyrażeń regularnych, która jest stosowana w JavaScripcie (zgodną ze standardem ECMA-262v3). Biblioteka VBScript jest częścią przeglądarki Internet Explorer 5.5 i nowszych, zatem jest dostępna na wszystkich komputerach z systemem operacyjnym Windows XP lub Windows Vista (oraz starszymi systemami operacyjnymi, jeśli tylko ich użytkownicy zaktualizowali przeglądarkę do wersji 5.5 lub nowszej). Oznacza to, że biblioteka VBScript jest dostępna na praktycznie wszystkich komputerach z systemem Windows wykorzystywanych do łączenia się z internetem.

Aby użyć tej biblioteki w aplikacji tworzonej w Visual Basicu, z menu *Project* zintegrowanego środowiska programowania (IDE) należy wybrać opcję *References*. Na wyświetlonej liście powinieneś odnaleźć pozycję *Microsoft VBScript Regular Expressions 5.5* (dostępną bezpośrednio pod pozycją *Microsoft VBScript Regular Expressions 1.0*). Upewnij się, że na liście jest zaznaczona wersja 5.5, nie wersja 1.0. Wersja 1.0 ma na celu wyłącznie zapewnienie zgodności wstecz, a jej możliwości są dalekie od satysfakcjonujących.

Po dodaniu tej referencji uzyskujesz dostęp do wykazu klas i składowych klas wchodzących w skład wybranej biblioteki. Warto teraz wybrać z menu *View* opcję *Object Browser*. Z listy rozwijanej w lewym górnym rogu okna *Object Browser* wybierz z bibliotekę *VBScript_RegExp_55*.

3.1. Stałe wyrażenia regularne w kodzie źródłowym

Problem

Otrzymałeś wyrażenie regularne `<[$"'\n\d/\\]>` jako rozwiązanie pewnego problemu. Wyrażenie to składa się z pojedynczej klasy znaków pasującej do znaku dolara, cudzysłowu, apostrofu, znaku nowego wiersza, dowolnej cyfry (0 – 9) oraz prawego i lewego ukośnika. Twoim zadaniem jest trwałe zapisanie tego wyrażenia regularnego w kodzie źródłowym (w formie stałej łańcuchowej lub operatora wyrażenia regularnego).

Rozwiązanie

C#

W formie zwykłego łańcucha:

```
"[$\"'\n\\d/\\\\]"
```

W formie łańcucha dosłownego:

```
@"[$""'\n\d/\\]"
```

VB.NET

```
"[$""'\n\d/\\]"
```

Java

```
"[$\"'\n\\d/\\\\]"
```

JavaScript

```
/[$"'\n\d\/\\]/
```

PHP

```
'%[$"\'\n\d/\\\\]%'
```

Perl

Operator dopasowywania wzorców:

```
/[\$"'\n\d\/\\]/
m![\$"'\n\d/\\]!
```

Operator podstawiania:

```
s![\$"'\n\d/\\]!!
```

Python

Standardowy (surowy) łańcuch otoczony potrójnymi cudzysłowami:

```
r"""[$"'\n\d/\\]"""
```

Zwykły łańcuch:

```
"[$\"'\n\\d/\\\\]"
```

Ruby

Stałe wyrażenie regularne otoczone prawymi ukośnikami:

```
/[$"'\n\d\/\\]/
```

Stałe wyrażenie regularne otoczone wybranymi znakami interpunkcyjnymi:

```
%r![$"'\n\d/\\]!
```

Analiza

Kiedy w tej książce proponujemy Ci samo wyrażenie regularne (czyli wyrażenie niebędące częścią większego fragmentu kodu źródłowego), zawsze formatujemy je w standardowy sposób. Ta receptura jest jedynym wyjątkiem od tej reguły. Jeśli korzystasz z testera wyrażeń regularnych, jak RegexBuddy czy RegexPal, powinieneś wpisywać swoje wyrażenia właśnie w ten sposób. Jeśli Twoja aplikacja operuje na wyrażeniach regularnych wpisywanych przez użytkownika, także użytkownik powinien wpisywać swoje wyrażenia w ten sposób.

Jeśli jednak chcesz zapisywać stałe wyrażenia regularne w swoim kodzie źródłowym, musisz się liczyć z dodatkowymi zadaniami. Bezmyślne, nieostrożne kopiowanie i wklejanie wyrażeń regularnych z testera do kodu źródłowego (i w przeciwnym kierunku) często prowadziłoby do błędów, a Ciebie zmuszałoby do gruntownych analiz obserwowanych zjawisk. Musiałbyś poświęcić sporo czasu na odkrywanie, dlaczego to samo wyrażenie regularne działa w testerze, ale nie działa w kodzie źródłowym, lub nie działa w testerze, mimo że zostało skopiowane z prawidłowego kodu źródłowego. Wszystkie języki programowania omawiane w tej książce wymagają otaczania stałych wyrażeń regularnych określonymi separatorami — część języków korzysta ze składni łańcuchów, inne wprowadzają specjalną składnię stałych wyrażeń regularnych. Jeśli Twoje wyrażenie regularne zawiera separatory danego języka programowania lub inne znaki, które mają w tym języku jakieś specjalne znaczenie, musisz zastosować sekwencje ucieczki.

Najczęściej stosowanym symbolem ucieczki jest lewy ukośnik (`\`). Właśnie dlatego większość rozwiązań zaproponowanych dla tego problemu zawiera dużo więcej lewych ukośników niż cztery ukośniki z oryginalnego wyrażenia regularnego (w punkcie „Problem”).

C#

W języku C# wyrażenia regularne można przekazywać na wejściu konstruktora `Regex()` i rozmaitych funkcji składowych klasy `Regex`. Parametry reprezentujące wyrażenia regularne zawsze są deklarowane jako łańcuchy.

C# obsługuje dwa rodzaje stałych łańcuchowych. Najbardziej popularnym rodzajem takich stałych są łańcuchy otoczone cudzysłowami, czyli konstrukcje doskonale znane z takich języków programowania, jak C++ czy Java. W ramach łańcuchów otoczonych cudzysłowami inne cudzysłowy i lewe ukośniki muszą być poprzedzane lewymi ukośnikami. W łańcuchach można też stosować sekwencje ucieczki ze znakami niedrukowanymi, na przykład `<\n>`. Jeśli włączono tryb swobodnego stosowania znaków białych (patrz receptura 2.18) za pośrednictwem

`RegexOptions.IgnorePatternWhitespace`, konstrukcje `"\n"` i `"\\n"` są traktowane w odmienny sposób (patrz receptura 3.4). O ile konstrukcja `"\n"` jest traktowana jako stała łańcuchowa z podziałem wiersza, która nie pasuje do znaków białych, o tyle `"\\n"` jest łańcuchem z tokenem wyrażenia regularnego `<\n>`, który pasuje do nowego wiersza.

Tzw. łańcuchy dosłowne (ang. *verbatim strings*) rozpoczynają się od znaku `@` i cudzysłowu, a kończą się samym cudzysłowem. Umieszczenie cudzysłowu w łańcuchu dosłownym wymaga użycia dwóch następujących po sobie cudzysłowów. W ramach tego rodzaju łańcuchów nie trzeba jednak stosować sekwencji ucieczki dla lewych ukośników, co znacznie poprawia czytelność wyrażeń regularnych. Konstrukcja `@"\n"` zawsze reprezentuje token wyrażenia regularnego `<\n>`, który pasuje do znaku nowego wiersza (także w trybie swobodnego stosowania znaków białych). Łańcuchy dosłowne co prawda nie obsługują tokenu `<\n>` na poziomie samych łańcuchów, ale mogą obejmować wiele wierszy. Konstrukcje łańcuchów dosłownych wprost idealnie nadają się więc do zapisywania wyrażeń regularnych.

Wybór jest dość prosty — najlepszym sposobem zapisywania wyrażeń regularnych w kodzie źródłowym języka C# jest stosowanie łańcuchów dosłownych.

VB.NET

W języku VB.NET istnieje możliwość przekazywania stałych wyrażeń na wejściu konstruktora `Regex()` oraz rozmaitych funkcji składowych klasy `Regex`. Parametr reprezentujący wyrażenie regularne zawsze jest deklarowany jako łańcuch.

W Visual Basicu stosuje się łańcuchy otoczone cudzysłowami. Cudzysłowy w ramach tych łańcuchów należy zapisywać podwójnie. Żadne inne znaki nie wymagają stosowania sekwencji ucieczki.

Java

W Javie stałe wyrażenia regularne można przekazywać na wejściu fabryki (wytwórni) klas `Pattern.compile()` oraz rozmaitych funkcji klasy `String`. Parametry reprezentujące wyrażenia regularne zawsze deklaruje się jako łańcuchy.

W Javie łańcuchy otacza się cudzysłowami. Ewentualne cudzysłowy i lewe ukośniki w ramach tych łańcuchów należy poprzedzać symbolem ucieczki, czyli lewym ukośnikiem. W łańcuchach można też umieszczać znaki niedrukowane (na przykład `<\n>`) oraz sekwencje ucieczki standardu Unicode (na przykład `<\uFFFF>`).

Jeśli włączono tryb swobodnego stosowania znaków białych (patrz receptura 2.18) za pośrednictwem `Pattern.COMMENTS`, konstrukcje `"\n"` i `"\\n"` są traktowane w odmienny sposób (patrz receptura 3.4). O ile konstrukcja `"\n"` jest interpretowana jako stała łańcuchowa z podziałem wiersza, która nie pasuje do znaków białych, o tyle `"\\n"` jest łańcuchem z tokenem wyrażenia regularnego `<\n>`, który pasuje do nowego wiersza.

JavaScript

W JavaScripcie najlepszym sposobem tworzenia wyrażeń regularnych jest korzystanie ze składni zaprojektowanej specjalnie z myślą o deklarowaniu stałych wyrażeń regularnych. Wystarczy umieścić wyrażenie regularne pomiędzy dwoma prawymi ukośnikami. Jeśli samo wyrażenie zawiera prawe ukośniki, należy każdy z nich poprzedzić lewym ukośnikiem.

Mimo że istnieje możliwość tworzenia obiektów klasy `RegExp` na podstawie łańcuchów, stosowanie notacji łańcuchowej dla stałych wyrażeń regularnych definiowanych w kodzie źródłowym nie miałoby większego sensu, ponieważ wymagałoby stosowania sekwencji ucieczki dla cudzysłowów i lewych ukośników (co zwykle prowadzi do powstania prawdziwego gąszczu lewych ukośników).

PHP

Stałe wyrażenia regularne na potrzeby funkcji `preg` języka PHP są przykładem dość nietypowego rozwiązania. Inaczej niż Java czy Perl, PHP nie definiuje rdzennego typu wyrażeń regularnych. Podobnie jak łańcuchy, wyrażenia regularne zawsze muszą być otoczone apostrofami. Dotyczy to także funkcji ze zbiorów `ereg` i `mb_ereg`. Okazuje się jednak, że w swoich dążeniach do powielenia rozwiązań znanych z Perla twórcy funkcji-opakowań biblioteki PCRE dla języka PHP wprowadzili pewne dodatkowe wymaganie.

Wyrażenie regularne umieszczone w łańcuchu musi być dodatkowo otoczone separatorami stosowanymi dla stałych wyrażeń regularnych Perla. Oznacza to, że wyrażenie regularne, które w Perlu miałoby postać `/wyrażenie/`, w języku PHP (stosowane na wejściu funkcji `preg`) musiałoby mieć postać `'/wyrażenie/'`. Podobnie jak w Perlu, istnieje możliwość wykorzystywania w roli separatorów par dowolnych znaków interpunkcyjnych. Jeśli jednak separator danego wyrażenia regularnego występuje w ramach tego wyrażenia, każde takie wystąpienie należy poprzedzić lewym ukośnikiem. Można uniknąć tej konieczności, stosując w roli separatora znak, który nie występuje w samym wyrażeniu regularnym. Na potrzeby tej receptury użyto znak procenta, ponieważ — w przeciwieństwie do prawego ukośnika — nie występuje w wyrażeniu regularnym. Gdyby nasze wyrażenie nie zawierało prawego ukośnika, powinniśmy otoczyć je właśnie tym znakiem, ponieważ to on jest najczęściej stosowanym separatorem w Perlu oraz wymaganym separatorem w językach JavaScript i Ruby.

PHP obsługuje zarówno łańcuchy otoczone apostrofami, jak i łańcuchy otoczone cudzysłowami. Każdy apostrof, cudzysłów i lewy ukośnik występujący wewnątrz wyrażenia regularnego wymaga zastosowania sekwencji ucieczki (poprzedzenia lewym ukośnikiem). W łańcuchach otoczonych cudzysłowami sekwencję ucieczki należy dodatkowo stosować dla znaku dolara. Jeśli nie planujesz włączania zmiennych do swoich wyrażeń regularnych, powinieneś konsekwentnie zapisywać je w formie łańcuchów otoczonych apostrofami.

Perl

W Perlu stałe wyrażenia regularne wykorzystuje się łącznie z operatorem dopasowywania wzorców oraz operatorem podstawiania. Operator dopasowywania wzorców składa się z dwóch prawych ukośników oraz znajdującego się pomiędzy nimi wyrażenia regularnego. Prawe ukośniki w ramach tego wyrażenia wymagają zastosowania sekwencji ucieczki poprzez poprzedzenie każdego z nich lewym ukośnikiem. Żaden inny znak nie wymaga stosowania podobnej sekwencji (może z wyjątkiem znaków `$` i `@`, o czym napisano na końcu tego podpunktu).

Alternatywna notacja operatora dopasowywania wzorców polega na umieszczaniu wyrażenia regularnego pomiędzy parą znaków interpunkcyjnych poprzedzoną literą `m`. Jeśli w roli separatora używasz dowolnego rodzaju otwierających lub zamykających znaków interpunkcyjnych (nawiasów okrągłych, kwadratowych lub klamrowych), za wyrażeniem regularnym należy umieścić prawy odpowiednik znaku otwierającego, na przykład `m{regex}`. W przypadku pozostałych znaków interpunkcyjnych wystarczy dwukrotnie użyć tego samego sym-

bolu. W rozwiązaniu dla tej receptury wykorzystano dwa wykrzykniki. W ten sposób uniknęliśmy konieczności stosowania sekwencji ucieczki dla prawego ukośnika użytego w ramach wyrażenia regularnego. Jeśli zastosowano inne separatory otwierające i zamykające, symbol ucieczki (lewy ukośnik) jest niezbędny tylko w przypadku separatora zamykającego (jeśli ten separator występuje w wyrażeniu regularnym).

Operator podstawiania pod wieloma względami przypomina operator dopasowywania wzorców. Operator podstawiania rozpoczyna się od litery `s` (zamiast `m`), po której następuje tekst docelowy operacji wyszukiwania i zastępowania. Jeśli w roli separatorów korzystasz z nawiasów kwadratowych lub podobnych znaków interpunkcyjnych, będziesz potrzebował dwóch par: `s[`*`wyrażenie`*`][`*`docelowy`*`]`. Wszystkich pozostałych znaków interpunkcyjnych należy użyć trzykrotnie: `s/`*`wyrażenie`*`/`*`docelowy`*`/`.

Perl traktuje operatory dopasowywania wzorców i podstawiania tak jak łańcuchy otoczone cudzysłowami. Jeśli więc skonstruujemy wyrażenie `m/Mam na imię $name/` i jeśli `$name` reprezentuje `"Jan"`, otrzymamy wyrażenie regularne `<Mam•na•imię•Jan>`. Także `$"` jest w języku Perl traktowane jak zmienna, stąd konieczność poprzedzenia znaku dolara (dopasowywanego dosłownie) w klasie znaków symbolem ucieczki.

Sekwencji ucieczki nigdy nie należy stosować dla znaku dolara, który ma pełnić funkcję kotwicy (patrz receptura 2.5). Znak dolara poprzedzony symbolem ucieczki zawsze jest dopasowywany dosłownie. Perl dysponuje mechanizmami niezbędnymi do prawidłowego rozróżniania znaków dolara występujących w roli kotwic oraz znaków dolara reprezentujących zmienne (w pierwszym przypadku znaki dolara mogą występować tylko na końcu grupy lub całego wyrażenia regularnego bądź przed znakiem nowego wiersza). Oznacza to, że jeśli chcemy sprawdzić, czy *`wyrażenie`* w ramach konstrukcji `<m/^`*`wyrażenie`*`$/>` pasuje do całego przetwarzanego tekstu, nie powinniśmy stosować sekwencji ucieczki dla znaku dolara.

Znak `@` stosowany w wyrażeniach regularnych nie ma co prawda żadnego specjalnego znaczenia, jednak jest wykorzystywany podczas przetwarzania zmiennych. Oznacza to, że każde jego wystąpienie w stałym wyrażeniu regularnym Perla należy poprzedzić symbolem ucieczki (podobnie jak w przypadku łańcuchów otoczonych cudzysłowami).

Python

Funkcje modułu `re` języka Python otrzymują na wejściu wyrażenia regularne w formie łańcuchów. Oznacza to, że dla wyrażeń regularnych definiowanych na potrzeby tych funkcji mają zastosowanie rozmaite sposoby definiowania łańcuchów Pythona. W zależności od znaków występujących w Twoim wyrażeniu regularnym wybór właściwego sposobu definiowania łańcuchów może znacznie ograniczać zakres znaków wymagających stosowania sekwencji ucieczki (poprzedzania lewymi ukośnikami).

Ogólnie najlepszym rozwiązaniem jest stosowanie standardowych (surowych) łańcuchów. Standardowe łańcuchy Pythona nie wymagają stosowania sekwencji ucieczki dla żadnych znaków. Oznacza to, że jeśli zdecydujesz się na tę formę definiowania wyrażeń regularnych, nie będziesz musiał podwajać lewych ukośników. Konstrukcja `r"\d+"` jest bardziej czytelna od konstrukcji `"\\d+"`, szczególnie jeśli całe wyrażenie regularne jest znacznie dłuższe.

Jedyny przypadek, w którym standardowe łańcuchy nie są najlepszym rozwiązaniem, ma miejsce wtedy, gdy wyrażenie regularne zawiera zarówno apostrofy, jak i cudzysłowy. Standardowy łańcuch nie może wówczas być otoczony apostrofami ani cudzysłowami, ponieważ nie ma

możliwości zastosowania sekwencji ucieczki dla tych znaków wewnątrz wyrażenia regularnego. W takim przypadku należy otoczyć standardowy łańcuch trzema cudzysłowami — jak w rozwiązaniu tej receptury dla języka Python (dla porównania pokazano też zwykły łańcuch).

Gdybyśmy chcieli korzystać w naszych wyrażeniach regularnych Pythona z możliwości, jakie daje nam standard Unicode (patrz receptura 2.7), powinniśmy zastosować łańcuchy tego standardu. Standardowy łańcuch można przekształcić w łańcuch Unicode, poprzedzając go przedrostkiem `u`.

Standardowe łańcuchy nie obsługują znaków niedrukowanych poprzedzanych znakami ucieczki, na przykład konstrukcji `\n`. Standardowe łańcuchy interpretują tego rodzaju sekwencje dosłownie. Okazuje się jednak, że wspomniana cecha standardowych łańcuchów nie stanowi problemu w przypadku modułu `re`, który obsługuje tego rodzaju sekwencje w ramach składni wyrażeń regularnych oraz składni docelowego tekstu operacji przeszukiwania i zastępowania. Oznacza to, że konstrukcja `\n` będzie interpretowana jako znak nowego wiersza w wyrażeniu regularnym lub tekście docelowym, nawet jeśli zdefiniujemy go w formie standardowego łańcucha.

Jeśli włączono tryb swobodnego stosowania znaków białych (patrz receptura 2.18) za pośrednictwem `re.VERBOSE`, łańcuch `"\n"` jest traktowany inaczej niż łańcuch `"\\n"` i surowy łańcuch `r"\n"` (patrz receptura 3.4). O ile konstrukcja `"\n"` jest interpretowana jako stała łańcuchowa z podziałem wiersza, która nie pasuje do znaków białych, o tyle konstrukcje `"\\n"` i `r"\n"` to łańcuchy z tokenem wyrażenia regularnego `<\n>`, który pasuje do nowego wiersza.

W trybie swobodnego stosowania znaków białych najlepszym rozwiązaniem jest definiowanie standardowych łańcuchów otoczonych trzema cudzysłowami (na przykład `r"""\n"""`), ponieważ łańcuchy w tej formie mogą się składać z wielu wierszy. Ponieważ konstrukcja `<\n>` nie jest interpretowana na poziomie łańcucha, może być interpretowana na poziomie wyrażenia regularnego, gdzie reprezentuje token pasujący do podziału wiersza.

Ruby

W Ruby najlepszym sposobem tworzenia wyrażeń regularnych jest korzystanie ze składni stworzonej specjalnie z myślą o stałych wyrażeniach tego typu. Wystarczy umieścić wyrażenie regularne pomiędzy dwoma prawymi ukośnikami. Jeśli samo wyrażenie zawiera jakiś prawy ukośnik, należy ten znak poprzedzić lewym ukośnikiem.

Jeśli nie chcesz stosować sekwencji ucieczki dla prawych ukośników, możesz poprzedzić swoje wyrażenie regularne przedrostkiem `%r`, po czym użyć w roli separatora dowolnego wybranego przez siebie znaku interpunkcyjnego.

Mimo że istnieje możliwość tworzenia obiektów klasy `Regexp` na podstawie łańcuchów, stosowanie notacji łańcuchowej dla stałych wyrażeń regularnych definiowanych w kodzie źródłowym nie miałoby większego sensu, ponieważ wymagałoby stosowania sekwencji ucieczki dla cudzysłowów i lewych ukośników (co zwykle prowadzi do powstania prawdziwego gąszczu lewych ukośników).

Pod tym względem język Ruby bardzo przypomina język JavaScript, z tą różnicą, że w języku Ruby odpowiednia klasa nosi nazwę `Regexp`, a w JavaScripcie nazwano ją `RegExp`.

Patrz także

W recepturze 2.3 wyjaśniono sposób działania klas znaków. Opisano też, dlaczego w wyrażeniu regularnym należy używać podwójnych lewych ukośników dla każdego lewego ukośnika wchodzącego w skład klasy znaków.

W recepturze 3.4 wyjaśnimy, jak ustawiać opcje wyrażeń regularnych, co w niektórych językach programowania jest możliwe w ramach stałych wyrażeń regularnych.

3.2. Importowanie biblioteki wyrażeń regularnych

Problem

Aby korzystać z wyrażeń regularnych w tworzonych aplikacjach, należy najpierw zaimportować do kodu źródłowego bibliotekę lub przestrzeń nazw wyrażeń regularnych.

W pozostałych fragmentach kodu źródłowego w tej książce (począwszy od następnej receptury) będziemy zakładali, że w razie konieczności zaimportowałeś już niezbędne biblioteki lub przestrzenie nazw.

Rozwiązanie

C#

```
using System.Text.RegularExpressions;
```

VB.NET

```
Imports System.Text.RegularExpressions
```

Java

```
import java.util.regex.*;
```

Python

```
import re
```

Analiza

Niektóre języki programowania oferują wbudowaną obsługę wyrażeń regularnych. Korzystanie z wyrażeń regularnych w tych językach nie wymaga żadnych dodatkowych kroków. Pozostałe języki programowania udostępniają obsługę wyrażeń regularnych za pośrednictwem bibliotek, które należy zaimportować przy użyciu odpowiednich wyrażeń w kodzie źródłowym. Co więcej, niektóre języki w ogóle nie oferują obsługi wyrażeń regularnych — w ich przypadku konieczne jest samodzielne skompilowanie i łączenie modułu implementującego obsługę wyrażeń regularnych.

C#

Jeśli umieścisz przytoczone wyrażenie `using` na początku swojego pliku źródłowego języka C#, będziesz mógł bezpośrednio korzystać z mechanizmów obsługi wyrażeń regularnych (bez konieczności każdorazowego kwalifikowania stosowanych wywołań). Będziesz mógł na przykład użyć wywołania `Regex()` zamiast wywołania `System.Text.RegularExpressions.Regex()`.

VB.NET

Jeśli umieścisz przytoczone wyrażenie `Imports` na początku swojego pliku źródłowego języka VB.NET, będziesz mógł bezpośrednio korzystać z mechanizmów obsługi wyrażeń regularnych (bez konieczności każdorazowego kwalifikowania stosowanych wywołań). Będziesz mógł na przykład użyć wywołania `Regex()` zamiast wywołania `System.Text.RegularExpressions.Regex()`.

Java

Korzystanie z wbudowanej biblioteki wyrażeń regularnych Javy wymaga uprzedniego zaimportowania pakietu `java.util.regex` do budowanej aplikacji.

JavaScript

W języku JavaScript mechanizmy obsługi wyrażeń regularnych są wbudowane i zawsze dostępne.

PHP

Funkcje z rodziny `preg` są wbudowane i zawsze dostępne w języku PHP, począwszy od wersji 4.2.0.

Perl

Mechanizmy obsługujące wyrażenia regularne są wbudowanymi i zawsze dostępnymi elementami języka Perl.

Python

Warunkiem korzystania z funkcji obsługujących wyrażenia regularne w języku Python jest uprzednie zaimportowanie modułu `re` do tworzonego skryptu.

Ruby

Mechanizmy obsługujące wyrażenia regularne są wbudowanymi i zawsze dostępnymi elementami języka Ruby.

3.3. Tworzenie obiektów wyrażeń regularnych

Problem

Chcesz skonkretyzować obiekt wyrażenia regularnego lub tak skompilować swoje wyrażenie, aby umożliwić efektywne używanie tego wyrażenia w całej swojej aplikacji.

Rozwiązanie

C#

Jeśli wiesz, że Twoje wyrażenie regularne jest prawidłowe:

```
Regex regexObj = new Regex("wzorzec wyrażenia regularnego");
```

Jeśli wyrażenie regularne zostało wpisane przez użytkownika końcowego (gdzie `UserInput` jest zmienną łańcuchową):

```
try {
    Regex regexObj = new Regex(UserInput);
} catch (ArgumentException ex) {
    // Błąd składniowy we wpisanym wyrażeniu regularnym.
}
```

VB.NET

Jeśli wiesz, że Twoje wyrażenie regularne jest prawidłowe:

```
Dim RegexObj As New Regex("wzorzec wyrażenia regularnego")
```

Jeśli wyrażenie regularne zostało wpisane przez użytkownika końcowego (gdzie `UserInput` jest zmienną łańcuchową):

```
Try
    Dim RegexObj As New Regex(UserInput)
Catch ex As ArgumentException
    'Błąd składniowy we wpisanym wyrażeniu regularnym.
End Try
```

Java

Jeśli wiesz, że Twoje wyrażenie regularne jest prawidłowe:

```
Pattern regex = Pattern.compile("wzorzec wyrażenia regularnego");
```

Jeśli wyrażenie regularne zostało wpisane przez użytkownika końcowego (gdzie `userInput` jest zmienną łańcuchową):

```
try {
    Pattern regex = Pattern.compile(userInput);
} catch (PatternSyntaxException ex) {
    // Błąd składniowy we wpisanym wyrażeniu regularnym.
}
```

Aby dopasować to wyrażenie regularne dla łańcucha, należy utworzyć obiekt klasy `Matcher`:

```
Matcher regexMatcher = regex.matcher(subjectString);
```

Dopasowanie tego wyrażenia regularnego do innego łańcucha wymaga albo utworzenia nowego obiektu klasy `Matcher` (jak w powyższym wyrażeniu), albo ponownego użycia obiektu już istniejącego:

```
regexMatcher.reset(anotherSubjectString);
```

JavaScript

Stałe wyrażenie regularne w Twoim kodzie może mieć następującą postać:

```
var myregexp = /wzorzec wyrażenia regularnego/;
```

Wyrażenie regularne wpisane przez użytkownika ma postać łańcucha reprezentowanego przez zmienną `userinput`:

```
var myregexp = new RegExp(userinput);
```

Perl

```
$myregex = qr/wzorzec wyrażenia regularnego/
```

W tym przypadku wyrażenie regularne wpisane przez użytkownika jest reprezentowane przez zmienną `$userinput`:

```
$myregex = qr/$userinput/
```

Python

```
reobj = re.compile("wzorzec wyrażenia regularnego")
```

Wyrażenie regularne wpisane przez użytkownika ma postać łańcucha reprezentowanego przez zmienną `userinput`:

```
reobj = re.compile(userinput)
```

Ruby

Stałe wyrażenie regularne w Twoim kodzie może mieć następującą postać:

```
myregexp = /wzorzec wyrażenia regularnego/;
```

Wyrażenie regularne wpisane przez użytkownika ma postać łańcucha reprezentowanego przez zmienną `userinput`:

```
myregexp = Regexp.new(userinput);
```

Analiza

Zanim moduł wyrażeń regularnych może dopasować jakieś wyrażenie do łańcucha, należy to wyrażenie skompilować. Kompilacja wyrażenia regularnego ma miejsce dopiero w czasie działania naszej aplikacji. Konstruktor wyrażenia regularnego lub odpowiednia funkcja kompilatora poddaje łańcuch zawierający nasze wyrażenie analizie składniowej i konwertuje go na strukturę drzewa lub maszynę stanów. Funkcja odpowiedzialna za właściwe dopasowywanie wzorców przeszukuje to drzewo lub maszynę stanów w trakcie przetwarzania tego łańcucha. Języki programowania, które obsługują stałe wyrażenia regularne, kompilują te wyrażenia w momencie osiągnięcia operatora wyrażenia regularnego.

.NET

W językach C# i VB.NET klasa `System.Text.RegularExpressions.Regex` frameworku .NET reprezentuje jedno skompilowane wyrażenie regularne. Najprostsza wersja konstruktora tej klasy otrzymuje na wejściu tylko jeden parametr — łańcuch zawierający nasze wyrażenie regularne.

W razie występowania jakiegoś błędu składniowego w przekazanym wyrażeniu regularnym konstruktor `Regex()` generuje wyjątek `ArgumentException`. Komunikat dołączony do tego wyjątku precyzyjnie określa rodzaj napotkanego błędu. Jeśli wyrażenie regularne zostało wpisane przez użytkownika naszej aplikacji, niezwykle ważne jest przechwycenie ewentualnego wyjątku. W razie jego wystąpienia należy wyświetlić stosowny komunikat i poprosić użytkownika o poprawienie wpisanego wyrażenia. Jeśli wyrażenie regularne trwale zakodowano w formie stałej łańcuchowej, możemy zrezygnować z przechwytywania wyjątku (warto jednak użyć narzędzia badającego pokrycie kodu, aby upewnić się, że odpowiedni wiersz nie powoduje wyjątków). Trudno sobie wyobrazić, by wskutek zmian stanu lub trybu to samo stałe wyrażenie regularne w jednej sytuacji było kompilowane prawidłowo, a w innej odrzucane przez kompilator. Warto przy tym pamiętać, że w razie błędu składniowego w stałym wyrażeniu regularnym odpowiedni wyjątek będzie generowany dopiero w czasie wykonywania aplikacji (nie na etapie jej kompilacji).

Obiekt klasy `Regex` należy skonstruować w sytuacji, gdy dane wyrażenie regularne ma być wykorzystywane w pętli lub wielokrotnie w różnych częściach kodu aplikacji. Konstruowanie obiektu wyrażenia regularnego nie wiąże się z żadnymi dodatkowymi kosztami. Okazuje się bowiem, że także statyczne składowe klasy `Regex`, które otrzymują wyrażenia regularne za pośrednictwem parametrów łańcuchowych, wewnętrznie konstruują obiekty tej klasy (na własne potrzeby). Oznacza to, że równie dobrze można to zrobić samodzielnie w kodzie źródłowym i zyskać możliwość swobodnego dysponowania odwołaniem do tego obiektu.

Jeśli planujemy użyć danego wyrażenia regularnego zaledwie raz lub kilka razy, możemy użyć statycznych składowych klasy `Regex` i — tym samym — oszczędzić sobie konieczności wpisywania dodatkowego wiersza kodu. Statyczne składowe tej klasy co prawda nie zwracają wewnętrznie konstruowanego obiektu wyrażenia regularnego, ale przechowują w wewnętrznej pamięci podręcznej piętnaście ostatnio użytych wyrażeń regularnych. Rozmiar tej pamięci można zmienić za pośrednictwem właściwości `Regex.CacheSize`. Przeszukiwanie wewnętrznej pamięci podręcznej klasy `Regex` polega na odnajdywaniu łańcucha z odpowiednim wyrażeniem regularnym. Nie należy jednak przeciążać tej pamięci — jeśli często odwołujesz się do wielu różnych obiektów wyrażeń regularnych, stwórz własną pamięć podręczną, którą będziesz mógł przeszukiwać nieporównanie szybciej niż w modelu odnajdywania łańcuchów.

Java

W Javie klasa `Pattern` reprezentuje pojedyncze, skompilowane wyrażenie regularne. Obiekty tej klasy można tworzyć za pośrednictwem fabryki (wytwórni) klas w formie metody `Pattern.compile()`, która otrzymuje na wejściu tylko jeden parametr — nasze wyrażenie regularne.

W razie występowania błędu składniowego w przekazanym wyrażeniu regularnym fabryka `Pattern.compile()` generuje wyjątek `PatternSyntaxException`. Komunikat dołączony do tego wyjątku precyzyjnie określa rodzaj napotkanego błędu. Jeśli wyrażenie regularne zostało wpisane przez użytkownika naszej aplikacji, niezwykle ważne jest przechwycenie ewentualnego wyjątku. W razie jego wystąpienia należy wyświetlić stosowny komunikat i poprosić użytkownika o poprawienie wpisanego wyrażenia. Jeśli wyrażenie regularne trwale zakodowano

w formie stałej łańcuchowej, możemy zrezygnować z przechwytywania wyjątku (warto jednak użyć narzędzia badającego pokrycie kodu, aby upewnić się, że odpowiedni wiersz nie powoduje wyjątków). Trudno sobie wyobrazić, by wskutek zmian stanu lub trybu to samo stałe wyrażenie regularne w jednej sytuacji było kompilowane prawidłowo, a w innej odrzucane przez kompilator. Warto przy tym pamiętać, że w razie błędu składniowego w stałym wyrażeniu regularnym odpowiedni wyjątek będzie generowany dopiero w czasie wykonywania aplikacji (nie na etapie jej kompilacji).

Jeśli nie planujesz użyć swojego wyrażenia regularnego zaledwie raz, powinieneś skonstruować obiekt klasy `Pattern`, zamiast korzystać ze statycznych składowych klasy `String`. Skonstruowanie tego obiektu wymaga co prawda kilku dodatkowych wierszy kodu, jednak kod w tej formie będzie wykonywany szybciej. Nie dość, że wywołania statyczne każdorazowo kompilują Twoje wyrażenie regularne, to jeszcze Java oferuje wywołania statyczne dla zaledwie kilku najprostszych zadań związanych z przetwarzaniem wyrażeń regularnych.

Obiekt klasy `Pattern` ogranicza się do przechowywania skompilowanego wyrażenia regularnego — nie wykonuje właściwych zadań związanych z dopasowywaniem tego wyrażenia. Za dopasowywanie wyrażeń regularnych odpowiada klasa `Matcher`. Utworzenie obiektu tej klasy wymaga wywołania metody `matcher()` dla skompilowanego wyrażenia regularnego. Za pośrednictwem jedynego argumentu metody `matcher()` należy przekazać łańcuch, do którego ma być dopasowane dane wyrażenie.

Metodę `matcher()` można wywołać dowolną liczbę razy dla tego samego wyrażenia regularnego i wielu łańcuchów do przetworzenia. Co więcej, istnieje możliwość jednoczesnego korzystania z wielu metod dopasowujących to samo wyrażenie regularne, ale pod warunkiem realizacji wszystkich tych zadań w ramach pojedynczego wątku. Klasy `Pattern` i `Matcher` nie gwarantują bezpieczeństwa przetwarzania wielowątkowego. Jeśli więc chcemy korzystać z tego samego wyrażenia w wielu wątkach, powinniśmy w każdym z tych wątków użyć osobnego wywołania metody `Pattern.compile()`.

Kiedy już zakończymy stosowanie naszego wyrażenia regularnego dla jednego łańcucha i postanowimy zastosować to samo wyrażenie dla innego łańcucha, będziemy mogli ponownie użyć istniejącego obiektu klasy `Matcher`, wywołując metodę składową `reset()`. Za pośrednictwem jedynego argumentu tej metody należy przekazać kolejny łańcuch do przetworzenia. Takie rozwiązanie jest bardziej efektywne niż każdorazowe tworzenie nowego obiektu klasy `Matcher`. Metoda `reset()` zwraca bowiem ten sam obiekt klasy `Matcher`, dla którego została wywołana. Oznacza to, że można bez trudu przywrócić pierwotny stan i ponownie użyć obiektu dopasowującego w jednym wierszu kodu, na przykład stosując konstrukcję `regexMatcher.reset ↪(nextString).find()`.

JavaScript

Notacja dla stałych wyrażeń regularnych, którą pokazano w recepturze 3.2, tworzy nowy obiekt wyrażenia regularnego. Jedynym warunkiem ponownego użycia tego samego obiektu jest przypisanie go jakiejś zmiennej.

Jeśli dysponujemy wyrażeniem regularnym reprezentowanym przez zmienną łańcuchową (na przykład wyrażeniem wpisanym przez użytkownika aplikacji), możemy użyć konstruktora `RegExp()` do jego skompilowania. Warto pamiętać, że wyrażenie regularne przechowywane

w formie łańcucha nie jest otoczone prawymi ukośnikami. Prawe ukośniki są częścią notacji JavaScriptu stosowanej dla stałych obiektów klasy `RegExp` i jako takie nie wchodzą w skład samego wyrażenia regularnego.

Ponieważ przypisywanie stałych wyrażeń regularnych zmiennym jest dziecinnie proste, większość prezentowanych w tym rozdziale rozwiązań dla JavaScriptu nie będzie zawierała tego wiersza kodu — będziemy raczej bezpośrednio korzystali ze stałego wyrażenia regularnego. Jeśli w swoim kodzie korzystasz z tego samego wyrażenia regularnego więcej niż raz, powinieneś przypisać to wyrażenie jakiejś zmiennej i w kolejnych odwołaniach używać właśnie tej zmiennej (zamiast każdorazowo kopiować i wklejać to samo stałe wyrażenie regularne). Takie rozwiązanie nie tylko poprawia wydajność tworzonych aplikacji, ale też zwiększa czytelność kodu.

PHP

Język PHP nie oferuje mechanizmu przechowywania skompilowanych wyrażeń regularnych w zmiennych. Jeśli chcesz wykonać jakąś operację na wyrażeniu regularnym, musisz to wyrażenie przekazać w formie łańcucha na wejściu odpowiedniej funkcji `preg`.

Funkcje z rodziny `preg` przechowują w wewnętrznej pamięci podręcznej maksymalnie 4096 skompilowanych wyrażeń regularnych. Chociaż przeszukiwanie pamięci podręcznej z wykorzystaniem skrótów nie jest tak efektywne jak odwołania do konkretnych zmiennych, koszty w wymiarze wydajności są nieporównanie mniejsze niż w przypadku każdorazowego kompilowania tego samego wyrażenia regularnego. W momencie zapełnienia pamięci podręcznej automatycznie jest z niej usuwane wyrażenie regularne skompilowane najwcześniej.

Perl

Do skompilowania wyrażenia regularnego i przypisania go zmiennej możemy użyć operatora `qr` (od ang. *quote regex*). Składnia tego operatora jest identyczna jak w przypadku operatora dopasowywania (patrz receptura 3.1), z tą różnicą, że zamiast litery `m` należy użyć liter `qr`.

Ogólnie Perl dość efektywnie radzi sobie z problemem wielokrotnego wykorzystywania skompilowanych wcześniej wyrażeń regularnych. Właśnie dlatego w przykładach kodu prezentowanych w tym rozdziale nie korzystamy z konstrukcji `qr//` (do pokazania zastosowań tej konstrukcji ograniczymy się w recepturze 3.5).

Operator `qr//` jest szczególnie przydatny podczas interpretowania zmiennych użytych w wyrażeniu regularnym lub w sytuacji, gdy całe wyrażenie regularne jest reprezentowane przez łańcuch (na przykład po wpisaniu przez użytkownika). Konstrukcja `qr/$łańcuchWyrażenia/` daje nam kontrolę nad czasem ponownej kompilacji danego wyrażenia regularnego z uwzględnieniem nowej zawartości łańcucha `$łańcuchWyrażenia`. Gdybyśmy użyli operatora `m/$łańcuch` ↪`Wyrażenia/`, nasze wyrażenie byłoby każdorazowo kompilowane, a w przypadku użycia operatora `m/$łańcuchWyrażenia/o` wyrażenie regularne nigdy nie byłoby ponownie kompilowane. Znaczenie opcji `/o` wyjaśnimy w recepturze 3.4.

Python

Funkcja `compile()` modułu `re` języka Python otrzymuje na wejściu łańcuch z naszym wyrażeniem regularnym i zwraca obiekt reprezentujący skompilowane wyrażenie regularne.

Funkcję `compile()` powinieneś wywołać wprost, jeśli planujesz wielokrotne użycie tego samego wyrażenia regularnego. Wszystkie inne funkcje modułu `re` rozpoczynają działanie właśnie od wywołania funkcji `compile()` — dopiero potem wywołują właściwe funkcje operujące na obiekcie skompilowanego wyrażenia regularnego.

Funkcja `compile()` utrzymuje odwołania do ostatnich stu skompilowanych przez siebie wyrażeń regularnych. Takie rozwiązanie skraca czas ponownej kompilacji tych wyrażeń regularnych do czasu potrzebnego do przeszukania słownika. Zawartość tej wewnętrznej pamięci podręcznej jest w całości usuwana w momencie osiągnięcia limitu stu skompilowanych wyrażeń.

Jeśli wydajność tworzonych rozwiązań nie jest najważniejsza, stosunkowo wysoka efektywność opisanego mechanizmu pamięci podręcznej powinna nam umożliwić bezpośrednie wywoływanie funkcji modułu `re`. Jeśli jednak zależy nam na najwyższej wydajności, warto rozważyć wywołanie funkcji `compile()`.

Ruby

Notacja stałych wyrażeń regularnych pokazana w recepturze 3.2 automatycznie tworzy nowy obiekt wyrażenia regularnego. Aby wielokrotnie użyć tego samego obiektu, wystarczy przypisać go jakiejś zmiennej.

Jeśli dysponujesz wyrażeniem regularnym przechowywanym w zmiennej łańcuchowej (na przykład po wpisaniu tego wyrażenia przez użytkownika aplikacji), możesz skompilować to wyrażenie za pomocą fabryki `Regexp.new()` (lub jej synonimu `Regexp.compile()`). Warto przy tym pamiętać, że wyrażenie regularne w ramach łańcucha nie jest otoczone prawymi ukośnikami. Prawe ukośniki są częścią notacji języka Ruby dla stałych obiektów klasy `Regexp`, a nie notacji samych wyrażeń regularnych.

Ponieważ przypisywanie stałych wyrażeń regularnych zmiennym jest dziecinnie proste, większość prezentowanych w tym rozdziale rozwiązań dla języka Ruby nie będzie zawierała tego wiersza kodu — będziemy bezpośrednio używali stałych wyrażeń regularnych. Jeśli w swoim kodzie korzystasz z tego samego wyrażenia regularnego więcej niż raz, powinieneś przypisać to wyrażenie jakiejś zmiennej i w kolejnych odwołaniach używać właśnie tej zmiennej (zamiast każdorazowo kopiować i wklejać to samo stałe wyrażenie regularne). Takie rozwiązanie nie tylko poprawia wydajność tworzonych aplikacji, ale też zwiększa czytelność kodu.

Kompilowanie wyrażeń regularnych do wspólnego języka pośredniego (CIL)

C#

```
Regex regexObj = new Regex("wzorzec wyrażenia regularnego", RegexOptions.Compiled);
```

VB.NET

```
Dim RegexObj As New Regex("wzorzec wyrażenia regularnego", RegexOptions.Compiled)
```

Analiza

Podczas konstruowania obiektu klasy `Regex` we frameworku .NET bez dodatkowych opcji wskazane wyrażenie regularne jest kompilowane w sposób opisany w punkcie „Analiza" na początku tej receptury. Jeśli za pośrednictwem drugiego parametru konstruktora `Regex()` przekażemy opcję `RegexOptions.Compiled`, działanie tej klasy będzie nieco inne — wyrażenie regularne zostanie skompilowane do tzw. wspólnego języka pośredniego (ang. *Common Intermediate Language — CIL*). CIL to niskopoziomowy język programowania bliższy asemblerowi niż takim językom, jak C# czy Visual Basic. Kod wspólnego języka pośredniego jest generowany przez wszystkie kompilatory frameworku .NET. Podczas pierwszego uruchamiania aplikacji framework .NET kompiluje kod języka CIL do postaci kodu maszynowego właściwego danemu komputerowi.

Zaletą kompilowania wyrażeń regularnych z opcją `RegexOptions.Compiled` jest blisko dziesięciokrotnie szybsze działanie niż w przypadku wyrażeń regularnych kompilowanych bez tej opcji.

Wadą tego rozwiązania jest czas trwania samej kompilacji — o dwa rzędy wielkości dłuższy od zwykłej analizy składniowej łańcucha z wyrażeniem regularnym (do postaci odpowiedniej struktury drzewa). Kod języka CIL staje się trwałym składnikiem naszej aplikacji do czasu zakończenia działania i jako taki nie podlega procedurom odzyskiwania pamięci.

Opcji `RegexOptions.Compiled` powinieneś używać tylko wtedy, gdy Twoje wyrażenie regularne jest albo na tyle złożone, albo musi przetwarzać na tyle dużo tekstu, że operacje z jego wykorzystaniem zajmują zauważalnie dużo czasu. Z drugiej strony nie ma sensu tracić czasu na wielokrotnie dłuższą kompilację, jeśli Twoje wyrażenia regularne są dopasowywane do przetwarzanego tekstu w ułamku sekundy.

Patrz także

Receptury 3.1, 3.2 i 3.4.

3.4. Ustawianie opcji wyrażeń regularnych

Problem

Chcesz skompilować wyrażenie regularne ze wszystkimi dostępnymi trybami dopasowywania — swobodnego stosowania znaków białych, ignorowania wielkości liter, dopasowywania znaków podziału wiersza do kropek oraz dopasowywania znaków podziału wiersza do karet i znaków dolara.

Rozwiązanie

C#

```
Regex regexObj = new Regex("wzorzec wyrażenia regularnego",
    RegexOptions.IgnorePatternWhitespace | RegexOptions.IgnoreCase |
    RegexOptions.Singleline | RegexOptions.Multiline);
```

VB.NET

```
Dim RegexObj As New Regex("wzorzec wyrażenia regularnego",
    RegexOptions.IgnorePatternWhitespace Or RegexOptions.IgnoreCase Or
    RegexOptions.Singleline Or RegexOptions.Multiline)
```

Java

```
Pattern regex = Pattern.compile("wzorzec wyrażenia regularnego",
    Pattern.COMMENTS | Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE |
    Pattern.DOTALL | Pattern.MULTILINE);
```

JavaScript

Stałe wyrażenie regularne w Twoim kodzie źródłowym:

```
var myregexp = /wzorzec wyrażenia regularnego/im;
```

Wyrażenie regularne wpisane przez użytkownika i reprezentowane w formie łańcucha:

```
var myregexp = new RegExp(userinput, "im");
```

PHP

```
regexstring = '/wzorzec wyrażenia regularnego/simx';
```

Perl

```
m/regex pattern/simx;
```

Python

```
reobj = re.compile("wzorzec wyrażenia regularnego",
    re.VERBOSE | re.IGNORECASE |
    re.DOTALL | re.MULTILINE)
```

Ruby

Stałe wyrażenie regularne w Twoim kodzie źródłowym:

```
myregexp = /wzorzec wyrażenia regularnego/mix;
```

Wyrażenie regularne wpisane przez użytkownika i reprezentowane w formie łańcucha:

```
myregexp = Regexp.new(userinput,
    Regexp::EXTENDED or Regexp::IGNORECASE or
    Regexp::MULTILINE);
```

Analiza

Wiele wyrażeń regularnych prezentowanych w tej książce (ale też znaczna część wyrażeń proponowanych w innych publikacjach) jest zapisywanych z myślą o dopasowywaniu w określonych trybach. Istnieją cztery podstawowe tryby obsługiwane przez niemal wszystkie współczesne odmiany wyrażeń regularnych. Okazuje się jednak, że twórcy niektórych odmian przyjęli niespójne i mylące nazewnictwo opcji implementujących te tryby. Wykorzystanie niewłaściwego trybu zwykle uniemożliwia prawidłowe dopasowanie wyrażenia regularnego.

Wszystkie rozwiązania zaproponowane na początku tej receptury wykorzystują flagi lub opcje udostępniane przez języki programowania lub klasy wyrażeń regularnych i umożliwiające ustawianie właściwych trybów. Alternatywnym sposobem ustawiania trybów jest korzystanie z tzw. modyfikatorów trybów w ramach samych wyrażeń regularnych. Modyfikatory trybów zawsze mają wyższy priorytet niż opcje i (lub) flagi ustawione poza wyrażeniem regularnym.

.NET

Na wejściu konstruktora `Regex()` można przekazać opcjonalny drugi parametr reprezentujący opcje dopasowywania danego wyrażenia regularnego. Dostępne opcje zdefiniowano w ramach typu wyliczeniowego `RegexOptions`.

Swobodne stosowanie znaków białych: `RegexOptions.IgnorePatternWhitespace`
Ignorowanie wielkości liter: `RegexOptions.IgnoreCase`
Dopasowywanie znaków podziału wiersza do kropek: `RegexOptions.Singleline`
Dopasowywanie znaków podziału wiersza do karet i znaków dolara: `RegexOptions.Multiline`

Java

Na wejściu fabryki klas `Pattern.compile()` można przekazać opcjonalny drugi parametr reprezentujący opcje dopasowywania danego wyrażenia regularnego. Klasa `Pattern` definiuje wiele stałych ustawiających rozmaite opcje. Istnieje możliwość ustawienia wielu opcji jednocześnie — wystarczy połączyć te opcje bitowym operatorem alternatywy niewykluczającej (`|`).

Swobodne stosowanie znaków białych: `Pattern.COMMENTS`
Ignorowanie wielkości liter: `Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE`
Dopasowywanie znaków podziału wiersza do kropek: `Pattern.DOTALL`
Dopasowywanie znaków podziału wiersza do karet i znaków dolara: `Pattern.MULTILINE`

W rzeczywistości istnieją dwie opcje dla trybu ignorowania wielkości liter; włączenie pełnego trybu ignorowania wielkości liter wymaga ustawienia obu tych opcji. Gdybyśmy użyli samej opcji `Pattern.CASE_INSENSITIVE`, tylko angielskie litery od *A* do *Z* byłyby dopasowywane bez względu na wielkość. Po ustawieniu obu opcji wszystkie znaki ze wszystkich alfabetów będą dopasowywane w trybie ignorowania wielkości liter. Jedynym powodem, dla którego można by rozważyć rezygnację z opcji `Pattern.UNICODE_CASE`, jest wydajność (oczywiście jeśli mamy pewność, że przetwarzany tekst nie będzie zawierał znaków spoza zbioru ASCII). W ramach wyrażeń regularnych można te opcje zastąpić modyfikatorem trybu `<(?i)>` wymuszającym ignorowanie wielkości liter ze zbioru ASCII oraz modyfikatorem trybu `<(?iu)>` wymuszającym ignorowanie wielkości liter ze wszystkich alfabetów.

JavaScript

W JavaScripcie można ustawiać opcje wyrażeń regularnych, dopisując jedną lub wiele jednoliterowych flag do stałej typu `RegExp` (bezpośrednio za prawym ukośnikiem kończącym dane wyrażenie regularne). Kiedy mówimy o tych flagach, zwykle zapisujemy je w formie `/i` bądź `/m`, mimo że sama flaga składa się z zaledwie jednej litery (prawy ukośnik nie wchodzi w skład flagi). Stosowanie flag dla wyrażeń regularnych nie wymaga dopisywania żadnych prawych ukośników.

Jeśli kompilujesz łańcuch do postaci wyrażenia regularnego za pomocą konstruktora `RegExp()`, możesz przekazać flagi dodatkowych opcji za pomocą drugiego, opcjonalnego parametru tego konstruktora. Drugi parametr powinien mieć postać łańcucha złożonego z liter reprezentujących ustawiane opcje (nie należy umieszczać w tym łańcuchu żadnych ukośników).

Swobodne stosowanie znaków białych: Tryb nieobsługiwany w JavaScripcie
Ignorowanie wielkości liter: `/i`
Dopasowywanie znaków podziału wiersza do kropek: Tryb nieobsługiwany w JavaScripcie
Dopasowywanie znaków podziału wiersza do karet i znaków dolara: `/m`

PHP

Jak już wspomnieliśmy w recepturze 3.1, funkcje `preg` języka PHP wymagają otaczania stałych wyrażeń regularnych dwoma znakami interpunkcyjnymi (zwykle prawymi ukośnikami) i powinny być formatowane tak jak stałe łańcuchowe. Opcje wyrażenia regularnego można określić, dopisując do odpowiedniego łańcucha jeden lub wiele jednoliterowych modyfikatorów. Oznacza to, że modyfikatory trybów należy umieścić za separatorem kończącym wyrażenie regularne, ale w ramach łańcucha (przed zamykającym apostrofem lub cudzysłowem). Kiedy mówimy o tych modyfikatorach, zwykle zapisujemy je na przykład jako `/x`, mimo że flaga składa się z samej litery, a separatorem oddzielającym wyrażenie regularne od modyfikatorów nie musi być prawy ukośnik.

Swobodne stosowanie znaków białych: `/x`
Ignorowanie wielkości liter: `/i`
Dopasowywanie znaków podziału wiersza do kropek: `/s`
Dopasowywanie znaków podziału wiersza do karet i znaków dolara: `/m`

Perl

W Perlu opcje przetwarzania wyrażeń regularnych można określać, dopisując jeden lub wiele jednoliterowych modyfikatorów trybów do operatora dopasowywania wzorców lub podstawiania. Kiedy mówimy o tych modyfikatorach, zwykle zapisujemy je na przykład jako `/x`, mimo że flaga składa się z samej litery, a separatorem oddzielającym wyrażenie regularne od modyfikatorów nie musi być prawy ukośnik.

Swobodne stosowanie znaków białych: `/x`
Ignorowanie wielkości liter: `/i`
Dopasowywanie znaków podziału wiersza do kropek: `/s`
Dopasowywanie znaków podziału wiersza do karet i znaków dolara: `/m`

Python

Na wejściu funkcji `compile()`, której działanie wyjaśniono w poprzedniej recepturze, można przekazać opcjonalny, drugi parametr reprezentujący opcje przetwarzania danego wyrażenia regularnego. Można skonstruować ten parametr, korzystając z operatora `|`, który umożliwia łączenie różnych stałych zdefiniowanych w module `re`. Także wiele innych funkcji modułu `re`, które otrzymują na wejściu stałe wyrażenia regularne, dodatkowo akceptuje opcje przetwarzania tych wyrażeń przekazywane za pośrednictwem ostatniego, opcjonalnego parametru.

Stałe reprezentujące opcje wyrażeń regularnych występują w parach. Każda opcja jest reprezentowana zarówno przez stałą z pełną nazwą, jak i przez jednoliterowy skrót. Znaczenie obu form jest identyczne. Jedyną różnicą jest większa czytelność pełnych nazw (szczególnie z perspektywy programistów, którzy nie zdążyli w dostatecznym stopniu opanować jednoliterowych skrótów reprezentujących opcje przetwarzania wyrażeń regularnych). Podstawowe jednoliterowe opcje opisane w tym punkcie są takie same jak w Perlu.

Swobodne stosowanie znaków białych: `re.VERBOSE` lub `re.X`
Ignorowanie wielkości liter: `re.IGNORECASE` lub `re.I`
Dopasowywanie znaków podziału wiersza do kropek: `re.DOTALL` lub `re.S`
Dopasowywanie znaków podziału wiersza do karet i znaków dolara: `re.MULTILINE` lub `re.M`

Ruby

W języku Ruby opcje przetwarzania wyrażeń regularnych można określać, dopisując jedną lub wiele jednoliterowych flag do stałej typu `Regexp`, za prawym ukośnikiem kończącym właściwe wyrażenie regularne. Kiedy mówimy o tych flagach w tej książce, zwykle zapisujemy je na przykład jako `/i` lub `/m`, mimo że sama flaga składa się tylko z litery. Dla flag określających tryb przetwarzania wyrażeń regularnych nie są wymagane żadne dodatkowe prawe ukośniki.

Na wejściu fabryki `Regexp.new()` kompilującej łańcuch do postaci wyrażenia regularnego możemy przekazać opcjonalny, drugi parametr reprezentujący flagi opcji. Drugi parametr może mieć albo wartość `nil` (wówczas wyłącza wszystkie opcje), albo zawierać kombinację stałych składowych klasy `Regexp` połączonych za pomocą operatora `or`.

Swobodne stosowanie znaków białych: `/r` lub `Regexp::EXTENDED`
Ignorowanie wielkości liter: `/i` lub `Regexp::IGNORECASE`
Dopasowywanie znaków podziału wiersza do kropek: `/m` lub `Regexp::MULTILINE`. W języku Ruby użyto dla tego trybu litery `m` (od ang. *multiline*), natomiast wszystkie pozostałe odmiany wyrażeń regularnych stosują listerę `s` (od ang. *singleline*).
Dopasowywanie znaków podziału wiersza do karet i znaków dolara: W języku Ruby znaki podziału wiersza domyślnie są dopasowywane do znaków karety (`^`) i dolara (`$`). Co więcej, nie można tej opcji wyłączyć. Dopasowywanie wyrażenia do początku lub końca przetwarzanego tekstu wymaga odpowiednio stosowania konstrukcji `<\A>` i `<\Z>`.

Dodatkowe opcje właściwe poszczególnym językom programowania

.NET

Opcja `RegexOptions.ExplicitCapture` powoduje, że żadna grupa (z wyjątkiem grup nazwanych) nie ma charakteru grupy przechwytującej. Użycie tej opcji sprawia, że konstrukcja `<(`*grupa*`)>` ma takie samo znaczenie jak konstrukcja `<(?:`*grupa*`)>`. Jeśli zawsze nazywasz swoje grupy przechwytujące, powinieneś włączyć tę opcję, aby podnieść efektywność przetwarzania swoich wyrażeń regularnych bez konieczności stosowania składni `<(?:`*grupa*`)>`. Zamiast korzystać z opcji `RegexOptions.ExplicitCapture`, można włączyć ten sposób interpretowania grup, umieszczając na początku wyrażenia regularnego konstrukcję `<(?n)>`. Szczegółowe omówienie techniki grupowania można znaleźć w recepturze 2.9; w recepturze 2.11 wyjaśniliśmy działanie grup nazwanych.

Jeśli korzystasz z tego samego wyrażenia regularnego w swoim kodzie frameworku .NET oraz w kodzie JavaScriptu i jeśli chcesz mieć pewność, że w obu językach Twoje wyrażenie będzie interpretowane w ten sam sposób, użyj opcji `RegexOptions.ECMAScript`. Takie rozwiązanie jest szczególnie korzystne w sytuacji, gdy pracujemy nad aplikacją internetową, której część kliencka jest implementowana w JavaScripcie, a część działająca po stronie serwera jest

implementowana w technologii ASP.NET. Najważniejszym skutkiem opisanej opcji jest ograniczenie zakresu znaków pasujących do tokenów `\w` i `\d` do znaków ASCII (a więc do zbioru pasującego do tych tokenów w JavaScripcie).

Java

Opcja `Pattern.CANON_EQ` jest unikatowym rozwiązaniem dostępnym tylko w Javie i reprezentuje tryb tzw. kanonicznej równoważności (ang. *canonical equivalence*). Jak wyjaśniono w podpunkcie „Grafem standardu Unicode" w rozdziale 2., standard Unicode oferuje różne sposoby reprezentowania znaków diakrytycznych. Po włączeniu tej opcji Twoje wyrażenie regularne będzie pasowało do znaków nawet wtedy, gdy zakodujesz je inaczej, niż zakodowano przetwarzany łańcuch. Na przykład wyrażenie `<\u00E0>` zostanie dopasowane zarówno do znaku `"\u00E0"`, jak i do sekwencji `"\u0061\u0300"`, ponieważ obie formy są kanonicznie równoważne. Obie formy reprezentują znak wyświetlany na ekranie jako *à*, zatem z perspektywy użytkownika są nie do odróżnienia. Gdybyśmy nie włączyli trybu kanonicznej równoważności, wyrażenie `<\u00E0>` nie zostałoby dopasowane do łańcucha `"\u0061\u0300"` (tak też działają wszystkie pozostałe odmiany wyrażeń regularnych prezentowane w tej książce).

I wreszcie opcja `Pattern.UNIX_LINES` wymusza na Javie interpretowanie jako znaku podziału wiersza wyłącznie konstrukcji `<\n>` (dopasowywanej do kropki, karety i dolara). Domyślnie za znak podziału wiersza uważa się wszystkie znaki podziału standardu Unicode.

JavaScript

Jeśli chcesz wielokrotnie stosować to samo wyrażenie regularne dla tego samego łańcucha (na przykład po to, by iteracyjnie przeszukać wszystkie dopasowania lub odnaleźć i zastąpić wszystkie pasujące podłańcuchy zamiast pierwszego), powinieneś użyć flagi `/g` (tzw. trybu globalnego).

PHP

Opcja `/u` wymusza na bibliotece PCRE interpretowanie zarówno samego wyrażenia regularnego, jak i przetwarzanego łańcucha jako łańcuchów w formacie UTF-8. Wspomniany modyfikator dodatkowo umożliwia stosowanie takich tokenów standardu Unicode, jak `<\p{FFFF}>` czy `<\p{L}>`. Znaczenie tych tokenów wyjaśniono w recepturze 2.7. Bez tego modyfikatora biblioteka PCRE traktuje każdy bajt jako odrębny znak, a tokeny wyrażeń regularnych standardu Unicode powodują błędy.

Modyfikator `/U` odwraca działanie zachłannych i leniwych kwantyfikatorów definiowanych z wykorzystaniem znaku zapytania. W normalnych okolicznościach `<.*>` jest zachłannym, a `<.*?>` jest leniwym kwantyfikatorem. Opcja `/U` powoduje, że to `<.*>` jest leniwym kwantyfikatorem, a `<.*?>` jest zachłannym kwantyfikatorem. Stanowczo odradzamy stosowanie tej flagi, ponieważ opisane działanie może być mylące dla programistów czytających Twój kod, którzy z natury rzeczy mogą nie dostrzec tego modyfikatora (występującego tylko w języku PHP). Jeśli będziesz miał okazję czytać kod innego programisty, w żadnym razie nie powinieneś mylić modyfikatora `/U` z modyfikatorem `/u` (wielkość liter w modyfikatorach trybów wyrażeń regularnych ma znaczenie).

Perl

Jeśli chcesz wielokrotnie zastosować to samo wyrażenie regularne dla tego samego łańcucha (na przykład po to, by iteracyjnie przeszukać wszystkie dopasowania lub odnaleźć i zastąpić wszystkie pasujące podłańcuchy zamiast pierwszego), powinieneś użyć flagi `/g` (tzw. trybu globalnego).

Jeśli w swoim wyrażeniu regularnym interpretujesz jakąś zmienną (na przykład w wyrażeniu `m/Mam na imię $name/`), Perl będzie ponownie kompilował to wyrażenie przed każdym użyciem, ponieważ zawartość zmiennej `$name` może być zmieniana. Możesz temu zapobiec, stosując modyfikator trybu `/o`. Wyrażenie `m/Mam na imię $name/o` zostanie skompilowane w momencie, w którym Perl po raz pierwszy będzie musiał go użyć (we wszystkich kolejnych przypadkach będzie ponownie wykorzystywał już skompilowane wyrażenie). Jeśli więc zawartość zmiennej `$name` ulegnie zmianie, skompilowane wcześniej wyrażenie regularne z opcją `/o` nie będzie uwzględniało tej zmiany. Techniki sterowania procesem kompilacji wyrażeń regularnych omówiono w recepturze 3.3.

Python

Python oferuje dwie dodatkowe opcje zmieniające znaczenie granic wyrazów (patrz receptura 2.6) oraz skróconych form klas znaków `<\w>`, `<\d>` i `<\s>` (a także ich zanegowanych odpowiedników — patrz receptura 2.3). Wymienione tokeny domyślnie operują wyłącznie na literach, cyfrach i znakach białych ze zbioru ASCII.

Opcja `re.LOCALE` (lub `re.L`) uzależnia interpretację tych tokenów od bieżących ustawień regionalnych. Właśnie ustawienia regionalne decydują o tym, który znak jest traktowany jako litera, który jako cyfra, a który jako znak biały. Z tej opcji należy korzystać zawsze wtedy, gdy przetwarzany łańcuch nie jest łańcuchem standardu Unicode i zawiera na przykład litery ze znakami diakrytycznymi (które chcemy właściwie interpretować).

Opcja `re.UNICODE` (lub `re.U`) powoduje, że w procesie dopasowywania wymienionych tokenów uwzględnia się specyfikę standardu Unicode. Wszystkie znaki zdefiniowane w tym standardzie jako litery, cyfry i znaki białe są odpowiednio interpretowane przez te tokeny. Z tej opcji należy korzystać zawsze wtedy, gdy łańcuch, dla którego stosujemy dane wyrażenie regularne, reprezentuje tekst w formacie Unicode.

Ruby

Na wejściu fabryki `Regexp.new()` można przekazać opcjonalny, trzeci parametr określający schemat kodowania, który ma być obsługiwany przez tworzone wyrażenie regularne. Jeśli nie określimy żadnego schematu kodowania, zostanie użyty ten sam schemat, w którym zapisano dany plik z kodem źródłowym. W większości przypadków takie rozwiązanie jest naturalne i w pełni prawidłowe.

Aby bezpośrednio wskazać konkretny schemat kodowania za pomocą tego parametru, należy przekazać pojedynczą literę. Wielkość użytej litery nie ma znaczenia. Obsługiwane wartości wymieniono i krótko opisano poniżej:

`n`

: Litera `n` oznacza brak kodowania (od ang. *none*). Każdy bajt przetwarzanego łańcucha będzie więc traktowany jako pojedynczy znak. Należy stosować tę opcję dla tekstu w formacie ASCII.

`e`
: Włącza kodowanie EUC dla języków dalekowschodnich.

`s`
: Włącza kodowanie Shift-JIS dla języka japońskiego.

`u`
: Włącza kodowanie UTF-8, w którym każdy znak jest reprezentowany przez cztery bajty i który obsługuje wszystkie języki standardu Unicode (w tym wszystkie, nawet dość rzadko spotykane żywe języki).

Dla stałego wyrażenia regularnego można określić schemat kodowania za pomocą modyfikatora trybu `/n`, `/e`, `/s` lub `/u`. Dla pojedynczego wyrażenia regularnego można użyć tylko jednego z wymienionych modyfikatorów. Można jednak łączyć te modyfikatory z dowolnym lub wszystkimi modyfikatorami ze zbioru `/x`, `/i` i `/m`.

Modyfikatora `/s` stosowanego w języku Ruby nie należy mylić z odpowiednim modyfikatorem obowiązującym w odmianach języków Perl i Java oraz frameworku .NET. W języku Ruby modyfikator `/s` wymusza stosowanie schematu kodowania Shift-JIS. W Perlu i większości innych odmian wyrażeń regularnych wspomniany modyfikator włącza tryb dopasowywania znaków podziału wiersza do kropki. W języku Ruby można ten tryb włączyć za pomocą modyfikatora `/m`.

Patrz także

Skutki stosowania poszczególnych trybów dopasowywania wyrażeń regularnych szczegółowo omówiono w rozdziale 2. Można tam znaleźć także wyjaśnienie technik stosowania modyfikatorów trybów w samych wyrażeniach regularnych.

Swobodne stosowanie znaków białych: Receptura 2.18
Ignorowanie wielkości liter: Podpunkt „Dopasowywanie bez względu na wielkość liter" w recepturze 2.1
Dopasowywanie znaków podziału wiersza do kropek: Receptura 2.4
Dopasowywanie znaków podziału wiersza do karet i znaków dolara: Receptura 2.5

W recepturach 3.1 i 3.3 wyjaśniono, jak korzystać ze stałych wyrażeń regularnych w kodzie źródłowym poszczególnych języków programowania i jak tworzyć obiekty wyrażeń regularnych. Opcje przetwarzania wyrażenia regularnego można ustawić na etapie konstruowania odpowiedniego obiektu.

3.5. Sprawdzanie możliwości odnalezienia dopasowania w przetwarzanym łańcuchu

Problem

Chcemy sprawdzić, czy istnieje możliwość znalezienia dopasowania określonego wyrażenia regularnego w określonym łańcuchu. W zupełności wystarczy dopasowanie częściowe. Na przykład wyrażenie regularne `<wzorzec•wyrażenia•regularnego>` częściowo pasuje do tekstu *Ten*

wzorzec wyrażenia regularnego można dopasować. Nie interesują nas żadne szczegóły tego dopasowania — chcemy tylko wiedzieć, czy nasze wyrażenie pasuje do danego łańcucha.

Rozwiązanie

C#

Do jednorazowego przeprowadzenia tego prostego testu można użyć następującego wywołania statycznego:

```
bool foundMatch = Regex.IsMatch(subjectString, "wzorzec wyrażenia regularnego");
```

Jeśli wyrażenie regularne zostało wpisane przez użytkownika końcowego aplikacji, należy użyć tego wywołania statycznego z pełną obsługą wyjątków:

```
bool foundMatch = false;
try {
    foundMatch = Regex.IsMatch(subjectString, UserInput);
} catch (ArgumentNullException ex) {
    // Wyrażenie regularne ani łańcuch do przetworzenia nie mogą mieć wartości null.
} catch (ArgumentException ex) {
    // W przekazanym wyrażeniu regularnym wystąpił błąd składniowy.
}
```

Aby wielokrotnie używać tego samego wyrażenia regularnego, należy skonstruować obiekt klasy `Regex`:

```
Regex regexObj = new Regex("wzorzec wyrażenia regularnego");
bool foundMatch = regexObj.IsMatch(subjectString);
```

Jeśli wyrażenie regularne zostało wpisane przez użytkownika końcowego aplikacji, także obiekt klasy `Regex` powinniśmy stosować z pełną obsługą wyjątków:

```
bool foundMatch = false;
try {
    Regex regexObj = new Regex(UserInput);
    try {
        foundMatch = regexObj.IsMatch(subjectString);
    } catch (ArgumentNullException ex) {
        // Wyrażenie regularne ani łańcuch do przetworzenia nie mogą mieć wartości null.
    }
} catch (ArgumentException ex) {
    // W przekazanym wyrażeniu regularnym wystąpił błąd składniowy.
}
```

VB.NET

Do jednorazowego przeprowadzenia tego prostego testu można użyć następującego wywołania statycznego:

```
Dim FoundMatch = Regex.IsMatch(SubjectString, "wzorzec wyrażenia regularnego")
```

Jeśli wyrażenie regularne zostało wpisane przez użytkownika końcowego aplikacji, należy użyć tego wywołania statycznego z pełną obsługą wyjątków:

```
Dim FoundMatch As Boolean
Try
    FoundMatch = Regex.IsMatch(SubjectString, UserInput)
Catch ex As ArgumentNullException
```

```
        'Wyrażenie regularne ani łańcuch do przetworzenia nie mogą mieć wartości Nothing.
    Catch ex As ArgumentException
        'W przekazanym wyrażeniu regularnym wystąpił błąd składniowy.
    End Try
```

Aby wielokrotnie używać tego samego wyrażenia regularnego, należy skonstruować obiekt klasy `Regex`:

```
Dim RegexObj As New Regex("wzorzec wyrażenia regularnego")
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

W tym przypadku zmienna `SubjectString` powinna być jedynym parametrem przekazanym na wejściu metody `IsMatch()`, a metodę `IsMatch()` należy wywołać dla obiektu `RegexObj` klasy `Regex`, a nie dla samej klasy `Regex`:

```
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Jeśli wyrażenie regularne zostało wpisane przez użytkownika końcowego aplikacji, także obiekt klasy `Regex` powinniśmy stosować z pełną obsługą wyjątków:

```
Dim FoundMatch As Boolean
Try
    Dim RegexObj As New Regex(UserInput)
    Try
        FoundMatch = Regex.IsMatch(SubjectString)
    Catch ex As ArgumentNullException
        'Wyrażenie regularne ani łańcuch do przetworzenia nie mogą mieć wartości Nothing.
    End Try
Catch ex As ArgumentException
    'W przekazanym wyrażeniu regularnym wystąpił błąd składniowy.
End Try
```

Java

Jedynym sposobem sprawdzenia, czy częściowe dopasowanie jest możliwe, jest skonstruowanie obiektu klasy `Matcher`:

```
Pattern regex = Pattern.compile("wzorzec wyrażenia regularnego");
Matcher regexMatcher = regex.matcher(subjectString);
boolean foundMatch = regexMatcher.find();
```

Jeśli wyrażenie regularne zostało wpisane przez użytkownika końcowego aplikacji, należy zastosować mechanizm obsługi wyjątków:

```
boolean foundMatch = false;
try {
    Pattern regex = Pattern.compile(UserInput);
    Matcher regexMatcher = regex.matcher(subjectString);
    foundMatch = regexMatcher.find();
} catch (PatternSyntaxException ex) {
    // W przekazanym wyrażeniu regularnym wystąpił błąd składniowy.
}
```

JavaScript

```
if (/wzorzec wyrażenia regularnego/.test(subject)) {
    // Udane dopasowanie.
} else {
    // Próba dopasowania zakończona niepowodzeniem.
}
```

PHP

```
if (preg_match('/wzorzec wyrażenia regularnego/', $subject)) {
    # Udane dopasowanie.
} else {
    # Próba dopasowania zakończona niepowodzeniem.
}
```

Perl

Jeśli przetwarzany łańcuch jest przechowywany w specjalnej zmiennej `$_`, można użyć następującej konstrukcji:

```
if (m/wzorzec wyrażenia regularnego/) {
    # Udane dopasowanie.
} else {
    # Próba dopasowania zakończona niepowodzeniem.
}
```

Jeśli przetwarzany łańcuch jest przechowywany w zmiennej `$subject`, można użyć konstrukcji:

```
if ($subject =~ m/wzorzec wyrażenia regularnego/) {
    # Udane dopasowanie.
} else {
    # Próba dopasowania zakończona niepowodzeniem.
}
```

Można też użyć skompilowanego wcześniej wyrażenia regularnego:

```
$regex = qr/wzorzec wyrażenia regularnego/;
if ($subject =~ $regex) {
    # Udane dopasowanie.
} else {
    # Próba dopasowania zakończona niepowodzeniem.
}
```

Python

Do jednorazowego przeprowadzenia tego prostego testu można użyć funkcji globalnej:

```
if re.search("wzorzec wyrażenia regularnego", subject):
    # Udane dopasowanie.
else:
    # Próba dopasowania zakończona niepowodzeniem.
```

Aby wielokrotnie użyć tego samego wyrażenia regularnego, należy posłużyć się skompilowanym obiektem:

```
reobj = re.compile("wzorzec wyrażenia regularnego")
if reobj.search(subject):
    # Udane dopasowanie.
else:
    # Próba dopasowania zakończona niepowodzeniem.
```

Ruby

```
if subject =~ /wzorzec wyrażenia regularnego/
    # Udane dopasowanie.
else
    # Próba dopasowania zakończona niepowodzeniem.
end
```

Poniższy kod działa dokładnie tak samo:

```
if /wzorzec wyrażenia regularnego/ =~ subject
    # Udane dopasowanie.
else
    # Próba dopasowania zakończona niepowodzeniem.
end
```

Analiza

Najprostszym zadaniem związanym z przetwarzaniem wyrażenia regularnego jest sprawdzenie, czy dane wyrażenie pasuje do jakiegoś łańcucha. W większości języków programowania częściowe dopasowanie wystarczy, by odpowiednia funkcja zwróciła pozytywny wynik. Funkcja dopasowująca analizuje cały przetwarzany łańcuch pod kątem możliwości dopasowania danego wyrażenia regularnego do części tego łańcucha. Funkcja dopasowująca zwraca pozytywny wynik zaraz po znalezieniu tego dopasowania. Ewentualna wartość negatywna zwracana jest dopiero w momencie osiągnięcia końca przetwarzanego łańcucha bez znalezienia dopasowania.

Przykładowe fragmenty kodu zaproponowane w tej recepturze są szczególnie przydatne podczas sprawdzania, czy przetwarzany łańcuch zawiera określone dane. Gdybyśmy chcieli sprawdzić, czy dany łańcuch w całości pasuje do określonego wzorca (na przykład celem weryfikacji danych wejściowych), powinniśmy użyć rozwiązania z następnej receptury.

C# i VB.NET

Klasa `Regex` udostępnia cztery przeciążone wersje metody `IsMatch()`, z których dwie mają postać składowych statycznych. Oznacza to, że metodę `IsMatch()` można wywoływać z różnymi parametrami. Pierwszym parametrem zawsze jest łańcuch do przetworzenia, czyli łańcuch, w którym próbujemy znaleźć dopasowanie do danego wyrażenia regularnego. Za pośrednictwem tego parametru nie można przekazać wartości `null` (w przeciwnym razie metoda `IsMatch()` wygeneruje wyjątek `ArgumentNullException`).

Możliwość dopasowania naszego wyrażenia możemy sprawdzić za pomocą zaledwie jednego wiersza — wystarczy wywołać metodę `Regex.IsMatch()` bez konieczności konstruowania obiektu klasy `Regex`. Za pośrednictwem drugiego parametru tej metody należy przekazać wyrażenie regularne, a za pośrednictwem opcjonalnego, trzeciego parametru można przekazać ewentualne opcje przetwarzania tego wyrażenia. Jeśli nasze wyrażenie zawiera jakiś błąd składniowy, metoda `IsMatch()` wygeneruje wyjątek `ArgumentException`. Jeśli przekazane wyrażenie jest prawidłowe, w razie znalezienia częściowego dopasowania zostanie zwrócona wartość `true`; w razie braku takiego dopasowania zostanie zwrócona wartość `false`.

Gdybyśmy chcieli użyć tego samego wyrażenia regularnego dla wielu łańcuchów, moglibyśmy podnieść efektywność tego kodu, konstruując najpierw obiekt klasy `Regex` i wywołując metodę `IsMatch()` dla tego obiektu. W takim przypadku pierwszym i jedynym wymaganym parametrem metody `IsMatch()` byłby łańcuch do przetworzenia. Można by też użyć drugiego, opcjonalnego parametru, który wskazywałby indeks znaku, od którego metoda `IsMatch()` miałaby rozpocząć dopasowywanie danego wyrażenia regularnego. W praktyce przekazana wartość reprezentuje liczbę początkowych znaków przetwarzanego łańcucha, które mają być ignorowane przez dane wyrażenie regularne. Taka możliwość jest cenna w sytuacji, gdy przetworzyliśmy już jakąś część łańcucha (do pewnego punktu) i planujemy poddać dalszej analizie

pozostałe znaki. Jeśli zdecydujemy się użyć tego parametru, powinniśmy przekazać wartość większą lub równą zero oraz mniejszą lub równą długości przetwarzanego łańcucha (w przeciwnym razie metoda `IsMatch()` wygeneruje wyjątek `ArgumentOutOfRangeException`).

Statyczne, przeciążone wersje tej metody nie obsługują parametru wskazującego początek podłańcucha dopasowywanego do wyrażenia regularnego. Nie istnieje też wersja metody `IsMatch()`, która umożliwiałaby przerywanie dopasowywania przed osiągnięciem końca łańcucha. Można ten cel osiągnąć, wywołując metodę `Regex.Match("subject", start, stop)` i sprawdzając właściwość `Success` zwróconego obiektu klasy `Match`. Więcej informacji na ten temat znajdziesz w recepturze 3.8.

Java

Aby sprawdzić, czy dane wyrażenie regularne pasuje do całości lub części jakiegoś łańcucha, należy skonstruować obiekt klasy `Matcher` (patrz receptura 3.3). Powinniśmy następnie wywołać metodę `find()` dla nowo utworzonego lub właśnie wyzerowanego obiektu dopasowującego.

Podczas realizacji tego zadania nie należy korzystać z metod `String.matches()`, `Pattern.matches()` ani `Matcher.matches()`. Wszystkie te metody dopasowują wyrażenie regularne do całego łańcucha.

JavaScript

Aby sprawdzić, czy dane wyrażenie regularne można dopasować do fragmentu jakiegoś łańcucha, należy wywołać metodę `test()` dla obiektu tego wyrażenia. Za pośrednictwem jedynego parametru tej metody powinniśmy przekazać łańcuch do przetworzenia.

Metoda `regexp.test()` zwraca wartość `true`, jeśli dane wyrażenie regularne pasuje do części lub całości przetwarzanego łańcucha; w przeciwnym razie metoda `regexp.test()` zwraca wartość `false`.

PHP

Funkcję `preg_match()` można z powodzeniem wykorzystywać do wielu różnych celów. Najprostszym sposobem jej wywołania jest przekazanie tylko dwóch wymaganych parametrów — łańcucha z wyrażeniem regularnym oraz łańcucha z tekstem do przetworzenia (dopasowania do danego wyrażenia). W razie odnalezienia dopasowania funkcja `preg_match()` zwraca wartość `1`; w przeciwnym razie funkcja `regexp.test()` zwraca wartość `0`.

W dalszej części tego rozdziału wyjaśnimy znaczenie opcjonalnych parametrów funkcji `preg_match()`.

Perl

W Perlu konstrukcja `m//` pełni funkcję operatora wyrażeń regularnych (nie — jak można by przypuszczać — kontenera wyrażeń regularnych). Jeśli użyjemy samego operatora `m//`, w roli źródła tekstu do przetworzenia zostanie wykorzystana zmienna `$_`.

Gdybyśmy chcieli użyć operatora dopasowania dla zawartości innej zmiennej, powinniśmy użyć operatora wiązania `=~`, aby skojarzyć operator wyrażenia regularnego z odpowiednią zmienną. Zastosowanie operatora wiążącego wyrażenie regularne z łańcuchem powoduje

natychmiastowe przetworzenie tego wyrażenia. Operator dopasowywania wzorców zwraca wartość `true`, jeśli dane wyrażenie pasuje do części lub całości danego łańcucha; w przeciwnym razie (w przypadku braku dopasowania) operator zwraca wartość `false`.

Gdybyśmy chcieli sprawdzić, czy dane wyrażenie regularne nie pasuje do łańcucha, powinniśmy użyć operatora `!~`, czyli zanegowanej wersji operatora `=~`.

Python

Funkcja `search()` modułu `re` przeszukuje wskazany łańcuch pod kątem możliwości dopasowania danego wyrażenia regularnego do jego części. Za pośrednictwem pierwszego parametru tej funkcji należy przekazać wyrażenie regularne; za pośrednictwem drugiego parametru powinniśmy przekazać łańcuch do przetworzenia. Opcjonalny, trzeci parametr służy do przekazywania ewentualnych opcji wyrażenia regularnego.

Funkcja `re.search()` wywołuje funkcję `re.compile()`, po czym wywołuje metodę `search()` już dla obiektu reprezentującego skompilowane wyrażenie regularne. Sama metoda `search()` otrzymuje na wejściu tylko jeden parametr — łańcuch do przetworzenia.

Jeśli dopasowanie zostanie znalezione, metoda `search()` zwróci obiekt klasy `MatchObject`. W przeciwnym razie metoda `search()` zwróci `None`. Jeśli analizujemy zwróconą wartość w wyrażeniu warunkowym `if`, obiekt klasy `MatchObject` jest traktowany jako `True`, natomiast `None` jest traktowane jako `False`. Sposoby korzystania z informacji reprezentowanych przez obiekt `MatchObject` omówimy w dalszej części tego rozdziału.

Nie należy mylić funkcji `search()` i `match()`. Funkcji `match()` nie można użyć do odnajdywania dopasowania w środku przetwarzanego łańcucha. Funkcję `match()` wykorzystamy w następnej recepturze.

Ruby

W języku Ruby `=~` pełni funkcję operatora dopasowywania wzorców. Aby znaleźć pierwsze dopasowanie wyrażenia regularnego do przetwarzanego tekstu, należy umieścić ten operator pomiędzy interesującym nas wyrażeniem a odpowiednim łańcuchem. Operator `=~` zwraca liczbę całkowitą reprezentującą początkową pozycję dopasowania znalezionego w danym łańcuchu. Jeśli nie uda się znaleźć dopasowania, operator `=~` zwróci wartość `nil`.

Operator `=~` zaimplementowano zarówno w klasie `Regexp`, jak i w klasie `String`. W języku Ruby 1.8 nie ma znaczenia, którą klasę umieścimy na lewo, a którą na prawo od tego operatora. W języku Ruby 1.9 kolejność operandów ma istotny wpływ na sposób przetwarzania nazwanych grup przechwytujących (wyjaśnimy ten mechanizm w recepturze 3.9).

We wszystkich pozostałych fragmentach kodu języka Ruby prezentowanych w tej książce będziemy umieszczali łańcuch z przetwarzanym tekstem na lewo, a wyrażenie regularne na prawo od operatora `=~`. W ten sposób zachowamy spójność z Perlem, z którego zaczerpnięto koncepcję operatora `=~`, i jednocześnie unikniemy niespodzianek związanych z obsługą nazwanych grup przechwytujących w języku Ruby 1.9, które często prowadzą do poważnych utrudnień.

Patrz także

Receptury 3.6 i 3.7.

3.6. Sprawdzanie, czy dane wyrażenie regularne pasuje do całego przetwarzanego łańcucha

Problem

Chcemy sprawdzić, czy dany łańcuch w całości pasuje do pewnego wyrażenia regularnego. Oznacza to, że chcemy sprawdzić, czy wyrażenie regularne reprezentujące pewien wzorzec pasuje do danego łańcucha od jego początku do końca. Gdybyśmy na przykład dysponowali wyrażeniem `<wzorzec•wyrażenia•regularnego>`, nasze rozwiązanie powinno go dopasować do tekstu *wzorzec wyrażenia regularnego*, ale nie do tekstu *Ten wzorzec wyrażenia regularnego można dopasować*.

Rozwiązanie

C#

Do jednorazowego przeprowadzenia tego prostego testu można użyć następującego wywołania statycznego:

```
bool foundMatch = Regex.IsMatch(subjectString, @"\Awzorzec wyrażenia regularnego\Z");
```

Aby wielokrotnie używać tego samego wyrażenia regularnego, należy skonstruować obiekt klasy `Regex`:

```
Regex regexObj = new Regex(@"\Awzorzec wyrażenia regularnego\Z");
bool foundMatch = regexObj.IsMatch(subjectString);
```

VB.NET

Do jednorazowego przeprowadzenia tego prostego testu można użyć następującego wywołania statycznego:

```
Dim FoundMatch = Regex.IsMatch(SubjectString, "\Awzorzec wyrażenia regularnego\Z")
```

Aby wielokrotnie używać tego samego wyrażenia regularnego, należy skonstruować obiekt klasy `Regex`:

```
Dim RegexObj As New Regex("\Awzorzec wyrażenia regularnego\Z")
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Za pośrednictwem jedynego parametru metody `IsMatch()` należy przekazać `SubjectString`, a samo wywołanie należy wykonać dla obiektu `RegexObj` klasy `Regex`, nie dla samej klasy `Regex`:

```
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Java

Gdybyśmy chcieli sprawdzić tylko jeden łańcuch, moglibyśmy użyć następującego wywołania statycznego:

```
boolean foundMatch = subjectString.matches("wzorzec wyrażenia regularnego");
```

Gdybyśmy chcieli zastosować to samo wyrażenie regularne dla wielu łańcuchów, powinniśmy skompilować to wyrażenie i utworzyć obiekt dopasowujący (klasy `Matcher`):

```
Pattern regex = Pattern.compile("wzorzec wyrażenia regularnego");
Matcher regexMatcher = regex.matcher(subjectString);
boolean foundMatch = regexMatcher.matches(subjectString);
```

JavaScript

```
if (/^wzorzec wyrażenia regularnego$/.test(subject)) {
    # Udane dopasowanie.
} else {
    # Próba dopasowania zakończona niepowodzeniem.
}
```

PHP

```
if (preg_match('/\Awzorzec wyrażenia regularnego\Z/', $subject)) {
# Successful match
} else {
# Match attempt failed
}
```

Perl

```
if ($subject =~ m/\Awzorzec wyrażenia regularnego\Z/) {
    # Udane dopasowanie.
} else {
    # Próba dopasowania zakończona niepowodzeniem.
}
```

Python

Do przeprowadzenia jednorazowego testu można użyć funkcji globalnej:

```
if re.match(r"wzorzec wyrażenia regularnego\Z", subject):
    # Udane dopasowanie.
else:
    # Próba dopasowania zakończona niepowodzeniem.
```

Aby wielokrotnie użyć tego samego wyrażenia regularnego, należy wykorzystać skompilowany obiekt:

```
reobj = re.compile(r"wzorzec wyrażenia regularnego\Z")
if reobj.match(subject):
    # Udane dopasowanie.
else:
    # Próba dopasowania zakończona niepowodzeniem.
```

Ruby

```
if subject =~ /\Awzorzec wyrażenia regularnego\Z/
    # Udane dopasowanie.
```

```
else
    # Próba dopasowania zakończona niepowodzeniem.
end
```

Analiza

W normalnych okolicznościach udane dopasowanie oznacza dla programisty tylko tyle, że wskazany wzorzec występuje **gdzieś** w przetwarzanym tekście. W wielu sytuacjach chcemy mieć dodatkowo pewność, że nasz wzorzec pasuje do **całego** tekstu, tj. że przetwarzany tekst nie zawiera żadnych innych, niepasujących fragmentów. Bodaj najczęstszym zastosowaniem operacji kompletnych dopasowań jest weryfikacja poprawności danych wejściowych. Jeśli na przykład użytkownik wpisuje numer telefonu lub adres IP z dodatkowymi, nieprawidłowymi znakami, próba zapisania tych danych powinna zostać odrzucona.

Rozwiązanie polegające na użyciu kotwic `<$>` i `<\Z>` można by z powodzeniem zastosować podczas przetwarzania kolejnych wierszy pliku (patrz receptura 3.21), a mechanizm wykorzystywany do uzyskiwania wierszy pomija znaki podziału z końca tych wierszy. Jak wspomniano w recepturze 2.5, wymienione kotwice są dopasowywane także do tekstu sprzed ostatniego podziału wiersza, zatem umożliwiają ignorowanie tego znaku podziału.

W kolejnych podpunktach szczegółowo wyjaśnimy rozwiązania dla poszczególnych języków programowania.

C# i VB.NET

Klasa `Regex` frameworku .NET nie udostępnia funkcji sprawdzającej, czy dane wyrażenie regularne pasuje do całego przetwarzanego łańcucha. Właściwym rozwiązaniem jest więc umieszczenie kotwicy początku łańcucha (`<\A>`) na początku wyrażenia regularnego oraz kotwicy końca łańcucha (`<\Z>`) na końcu wyrażenia regularnego. W ten sposób wymuszamy albo dopasowanie wyrażenia regularnego do całego przetwarzanego łańcucha, albo brak dopasowania. Jeśli nasze wyrażenie zawiera podwyrażenia alternatywne, na przykład `<jeden|dwa|trzy>`, koniecznie powinniśmy pogrupować te podwyrażenia i otoczyć kotwicami całą grupę: `<\A(?:jeden|dwa|trzy)\Z>`.

Po wprowadzeniu odpowiednich poprawek do wyrażenia regularnego możemy użyć tej samej metody `IsMatch()`, którą posługiwaliśmy się w poprzedniej recepturze.

Java

Programiści Javy mają do dyspozycji trzy metody nazwane `matches()`. Wszystkie te metody sprawdzają, czy dane wyrażenie regularne można w całości dopasować do pewnego łańcucha. Metody `matches()` umożliwiają błyskawiczną weryfikację danych wejściowych (bez konieczności umieszczania wyrażenia regularnego pomiędzy kotwicami początku i końca łańcucha).

Klasa `String` definiuje metodę `matches()`, która otrzymuje za pośrednictwem jedynego parametru wyrażenie regularne. W zależności od tego, czy dopasowanie tego wyrażenia do całego łańcucha jest możliwe, czy nie, metoda `matches()` zwraca odpowiednio wartość `true` lub `false`. Klasa `Pattern` definiuje statyczną metodę `matches()`, która otrzymuje na wejściu dwa łańcuchy — pierwszy reprezentuje wyrażenie regularne, drugi zawiera tekst do przetworzenia.

W praktyce na wejściu metody `Pattern.matches()` (w roli drugiego parametru) można przekazać dowolny obiekt klasy `CharSequence`. Możliwość przekazywania obiektów tego typu to jedyny powód, dla którego warto korzystać z metody `Pattern.matches()` (zamiast metody `String.matches()`).

Zarówno metoda `String.matches()`, jak i metoda `Pattern.matches()` każdorazowo kompiluje otrzymane wyrażenie regularne, wywołując metodę `Pattern.compile("regex").matcher ↪(subjectString).matches()`. W tej sytuacji powinniśmy korzystać z tych metod tylko wtedy, gdy dane wyrażenie regularne ma być użyte tylko raz (na przykład na potrzeby weryfikacji zawartości pojedynczego pola formularza wejściowego) lub gdy efektywność naszego kodu jest nieistotna. Wymienione metody nie zapewniają możliwości definiowania opcji dopasowywania poza samymi wyrażeniami regularnymi. W razie występowania błędu składniowego w przekazanym wyrażeniu regularnym opisane metody generują wyjątek `PatternSyntax ↪Exception`.

Gdybyśmy chcieli efektywnie użyć tego samego wyrażenia regularnego do sprawdzenia wielu łańcuchów, powinniśmy skompilować to wyrażenie, po czym skonstruować i wielokrotnie wykorzystać obiekt klasy `Matcher` (patrz receptury 3.3). Możemy następnie wywoływać metodę `matches()` dla tego obiektu. Metoda `matches()` nie otrzymuje na wejściu żadnych parametrów, ponieważ łańcuch do przetworzenia jest przekazywany już na etapie tworzenia lub zerowania obiektu dopasowującego.

JavaScript

JavaScript nie oferuje funkcji umożliwiającej sprawdzanie, czy wyrażenie regularne pasuje do całego przetwarzanego łańcucha. Właściwym rozwiązaniem jest więc umieszczenie kotwicy początku łańcucha (`<^>`) na początku wyrażenia regularnego oraz kotwicy końca łańcucha (`<$>`) na końcu wyrażenia regularnego. Powinniśmy się przy tym upewnić, że dla naszego wyrażenia nie ustawiliśmy flagi `/m`, ponieważ brak tej flagi jest warunkiem dopasowywania symboli karety i dolara wyłącznie do początku i końca przetwarzanego łańcucha. Flaga `/m` powoduje, że oba symbole są dopasowywane także do znaków podziału wiersza w środku łańcucha.

Po dodaniu kotwic do wyrażenia regularnego można użyć metody `regexp.test()` zgodnie z procedurą opisaną w poprzedniej recepturze.

PHP

PHP nie oferuje funkcji umożliwiającej sprawdzanie, czy wyrażenie regularne pasuje do całego przetwarzanego łańcucha. Właściwym rozwiązaniem jest więc umieszczenie kotwicy początku łańcucha (`<\A>`) na początku wyrażenia regularnego oraz kotwicy końca łańcucha (`<\Z>`) na końcu wyrażenia regularnego. W ten sposób wymuszamy albo dopasowanie wyrażenia regularnego do całego przetwarzanego łańcucha, albo brak dopasowania. Jeśli nasze wyrażenie zawiera podwyrażenia alternatywne, na przykład `<jeden|dwa|trzy>`, koniecznie powinniśmy pogrupować te podwyrażenia i otoczyć kotwicami całą grupę: `<\A(?:jeden|dwa|trzy)\Z>`.

Po wprowadzeniu odpowiednich poprawek do wyrażenia regularnego możemy użyć tej samej metody `preg_match()`, którą posługiwaliśmy się w poprzedniej recepturze.

Perl

Perl udostępnia tylko jeden operator dopasowywania wzorców, który zadowala się częściowymi dopasowaniami. Jeśli więc chcemy sprawdzić, czy nasze wyrażenie regularne pasuje do całego przetwarzanego łańcucha, powinniśmy umieścić kotwicę początku łańcucha (`<\A>`) na początku wyrażenia regularnego oraz kotwicę końca łańcucha (`<\Z>`) na końcu wyrażenia regularnego. W ten sposób wymuszamy albo dopasowanie wyrażenia regularnego do całego przetwarzanego łańcucha, albo brak dopasowania. Jeśli nasze wyrażenie zawiera podwyrażenia alternatywne, na przykład `<jeden|dwa|trzy>`, koniecznie powinniśmy pogrupować te podwyrażenia i otoczyć kotwicami całą grupę: `<\A(?:jeden|dwa|trzy)\Z>`.

Po wprowadzeniu niezbędnych zmian w naszym wyrażeniu regularnym możemy posłużyć się rozwiązaniem opisanym w poprzedniej recepturze.

Python

Funkcja `match()` pod wieloma względami przypomina opisaną w poprzedniej recepturze funkcję `search()`. Najważniejsza różnica dzieląca obie funkcje polega na tym, że funkcja `match()` dopasowuje dane wyrażenie regularne tylko do początku przetwarzanego łańcucha. Jeśli to wyrażenie nie pasuje do początku łańcucha, funkcja `match()` zwraca wartość `None`. Nieco inaczej działa funkcja `search()`, która próbuje dopasować wyrażenie regularne do fragmentu łańcucha na dowolnej pozycji i albo odnajduje dopasowanie, albo osiąga koniec tego łańcucha.

Funkcja `match()` nie wymaga, by wyrażenie regularne pasowało do całego łańcucha — dopasowanie częściowe jest akceptowane, pod warunkiem że rozpoczyna się na początku przetwarzanego łańcucha. Jeśli więc chcemy sprawdzić, czy nasze wyrażenie regularne można dopasować do całego łańcucha, powinniśmy dopisać do tego wyrażenia kotwicę końca łańcucha (`<\Z>`).

Ruby

Klasa `Regexp` języka Ruby nie udostępnia funkcji sprawdzającej, czy dane wyrażenie regularne pasuje do całego przetwarzanego łańcucha. Właściwym rozwiązaniem jest więc umieszczenie kotwicy początku łańcucha (`<\A>`) na początku wyrażenia regularnego oraz kotwicy końca łańcucha (`<\Z>`) na końcu wyrażenia regularnego. W ten sposób wymuszamy albo dopasowanie wyrażenia regularnego do całego przetwarzanego łańcucha, albo brak dopasowania. Jeśli nasze wyrażenie zawiera podwyrażenia alternatywne, na przykład `<jeden|dwa|trzy>`, koniecznie powinniśmy pogrupować te podwyrażenia i otoczyć kotwicami całą grupę: `<\A(?:jeden|dwa|trzy)\Z>`.

Po wprowadzeniu odpowiednich poprawek do wyrażenia regularnego możemy użyć tego samego operatora `=~`, którym posługiwaliśmy się w poprzedniej recepturze.

Patrz także

W recepturze 2.5 szczegółowo wyjaśniono działanie kotwic.

W recepturach 2.8 i 2.9 wyjaśniono zagadnienia związane z wyrażeniami alternatywnymi i grupowaniem. Jeśli Twoje wyrażenie obejmuje wyrażenia alternatywne, które nie wchodzą w skład żadnych grup, przed dodaniem kotwic musisz je pogrupować. Jeśli Twoje wyrażenie nie zawiera podwyrażeń alternatywnych lub jeśli wszystkie alternatywy wchodzą w skład grup, żadne dodatkowe grupowanie nie jest potrzebne do prawidłowego działania kotwic.