Perl. Mistrzostwo w programowaniu

Autor: Brian d foy
Tłumaczenie: Grzegorz Werner
ISBN: 978-83-246-1374-8
Tytuł oryginału: Mastering Perl
Format: B5, stron: 304

Profesjonalne programowanie na mistrzowskim poziomie

- Jak wykrywać błędy, których Perl nie raportuje?
- Jak pisać programy jako moduły?
- Jak śledzić działanie programu za pomocą Log4perl?

Perl jest językiem o szerokim zastosowaniu, można go skompilować na prawie wszystkich architekturach i systemach operacyjnych. Wszechstronność Perla pozwala na programowanie w różnych modelach: proceduralnym, funkcyjnym czy obiektowym. Jest doskonałym narzędziem do analizy plików tekstowych oraz tworzenia raportów, aplikacji, modułów i programów. Umożliwia powiązanie systemów i struktur danych, których współpraca nie była przewidywana w momencie projektowania. Twórcy Perla twierdzą, że język ten sprawia, iż rzeczy łatwe pozostają łatwymi, a trudne stają się możliwe do wykonania.

„Perl. Mistrzostwo w programowaniu" to wyjątkowa książka pomagająca w samodzielnej nauce, przeznaczona dla programistów, którzy używali już Perla i znają jego podstawy. Podążając za radami z tego przewodnika, nauczysz się definiować procedury i odwracać zwykły model programowania proceduralnego. Będziesz wiedział, jak zapisywać dane, aby wykorzystać je w innym programie, a także jak poprawiać kod bez modyfikowania pierwotnego kodu źródłowego. Dowiesz się także, jak używać operacji na bitach oraz wektorów bitowych do efektywnego przechowywania danych. Czytając „Perl. Mistrzostwo w programowaniu", zmierzasz prostą drogą do mistrzostwa.

- Tworzenie i zastępowanie nazwanych procedur
- Modyfikowanie i rozszerzanie modułów
- Konfigurowanie programów Perla
- Rejestrowanie błędów i innych informacji
- Utrwalanie danych
- Praca z formatem Pod
- Tworzenie podklas modułu Pod::Simple
- Operatory bitowe
- Przechowywanie łańcuchów bitowych
- Testowanie programu

Dołącz do klasy mistrzów – twórz profesjonalne programy w Perlu!

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

Spis treści

ROZDZIAŁ 2.

Zaawansowane wyrażenia regularne

Wyrażenia regularne stanowią klucz do przetwarzania tekstu w Perlu i z pewnością są jedną z funkcji, dzięki którym Perl zyskał tak dużą popularność. Wszyscy programiści Perla przechodzą fazę, w której próbują pisać wszystkie programy jako wyrażenia regularne, a jeśli to im nie wystarcza — jako *jedno* wyrażenie regularne. Wyrażenia regularne Perla mają więcej możliwości, niż mogę — i chcę — tu zaprezentować, więc omówię te zaawansowane funkcje, które uważam za najbardziej użyteczne i o których każdy programista Perla powinien wiedzieć bez zaglądania do *perlre* (rozdziału dokumentacji poświęconego wyrażeniom regularnym).

Referencje do wyrażeń regularnych

Kiedy piszę jakiś program, nie muszę zawczasu znać wszystkich wzorców. Perl pozwala mi interpolować wyrażenia regularne w zmiennych. Mogę zakodować te wartości „na sztywno", pobrać je z danych dostarczonych przez użytkownika albo uzyskać w dowolny inny sposób. Oto króciutki program Perla, który działa podobnie jak `grep`. Pobiera pierwszy argument wiersza polecenia i używa go jako wyrażenia regularnego w instrukcji `while`. Nie ma w tym nic szczególnego (na razie); pokazaliśmy, jak to robić, w książce *Perl. Wprowadzenie*. Mogę użyć łańcucha w zmiennej `$regex` jako wyrażenia regularnego, a Perl skompiluje je, kiedy zinterpoluje łańcuch w operatorze dopasowania[1]:

```
#!/usr/bin/perl
# perl-grep.pl

my $regex = shift @ARGV;

print "Wyrażenie regularne to [$regex]\n";

while( <> )
        {
        print if m/$regex/;
        }
```

Mogę uruchomić ten program z poziomu wiersza poleceń, aby poszukać wzorca w plikach. W poniższym przykładzie szukam wzorca `new` we wszystkich programach Perla znajdujących się w bieżącym katalogu:

[1] Od wersji 5.6 Perla, jeśli łańcuch się nie zmienia, wyrażenie nie jest kompilowane ponownie. W starszych wersjach trzeba było użyć opcji `/o`, aby uzyskać takie działanie. Można nadal używać opcji `/o`, aby wskazać, że wzorzec nie powinien być rekompilowany nawet wtedy, gdy łańcuch się zmieni.

```
% perl-grep.pl new *.pl
Wyrażenie regularne to [new]
my $regexp = Regexp::English->new
my $graph = GraphViz::Regex->new($regex);
                [ qr/\G(\n)/,                "newline"     ],
                                                { ( $1, "newline char"     ) }
print YAPE::Regex::Explain->new( $ARGV[0] )->explain;
```

Co się stanie, jeśli podam nieprawidłowe wyrażenie regularne? Wypróbuję to na wyrażeniu, które ma nawias otwierający, ale nie ma zamykającego:

```
$ ./perl-grep.pl "(perl" *.pl
Wyrażenie regularne to [(perl]
Unmatched ( in regex; marked by <-- HERE in m/( <-- HERE perl/
        at ./perl-grep.pl line 10, <> line 1.
```

Kiedy interpoluję wyrażenie regularne w operatorze dopasowania, Perl kompiluje wyrażenie i natychmiast zgłasza błąd, przerywając działanie programu. Aby tego uniknąć, muszę skompilować wyrażenie, zanim spróbuję go użyć.

`qr//` to operator przytaczania wyrażeń regularnych, który zapisuje wyrażenie w skalarze (dokumentację tego operatora można znaleźć na stronie *perlop*). Operator `qr//` kompiluje wzorzec, aby był gotowy do użycia, kiedy zinterpoluję `$regex` w operatorze dopasowania. Aby wychwycić błąd, umieszczam `qr//` w bloku operatora `eval`, choć w tym przypadku i tak przerywam działanie programu instrukcją `die`:

```
#!/usr/bin/perl
# perl-grep2.pl

my $pattern = shift @ARGV;

my $regex = eval { qr/$pattern/ };
die "Sprawdź swój wzorzec! $@" if $@;

while( <> )
        {
        print if m/$regex/;
        }
```

Wyrażenie regularne w zmiennej `$regex` ma wszystkie cechy operatora dopasowania, w tym odwołania wsteczne i zmienne pamięciowe. Poniższy wzorzec wyszukuje sekwencję trzech znaków, w której pierwszy i trzeci znak są takie same, a żaden nie jest znakiem odstępu. Dane wejściowe to tekstowa wersja strony dokumentacji *perl*, którą uzyskuję za pomocą polecenia `perldoc -t`:

```
% perldoc -t  perl | perl-grep2.pl "\b(\S)\S\1\b"
    perl583delta        Perl changes in version 5.8.3
    perl582delta        Perl changes in version 5.8.2
    perl581delta        Perl changes in version 5.8.1
    perl58delta         Perl changes in version 5.8.0
    perl573delta        Perl changes in version 5.7.3
    perl572delta        Perl changes in version 5.7.2
    perl571delta        Perl changes in version 5.7.1
    perl570delta        Perl changes in version 5.7.0
    perl561delta        Perl changes in version 5.6.1
http://www.perl.com/       the Perl Home Page
http://www.cpan.org/       the Comprehensive Perl Archive
http://www.perl.org/       Perl Mongers (Perl user groups)
```

Nie jest zbyt łatwo (przynajmniej mnie) stwierdzić na pierwszy rzut oka, co Perl dopasował, więc mogę wprowadzić pewną zmianę w programie `grep`. Zmienna `$&` przechowuje dopasowaną część łańcucha:

```
#!/usr/bin/perl
# perl-grep3.pl

my $pattern = shift @ARGV;

my $regex = eval { qr/$pattern/ };
die "Sprawdź swój wzorzec! $@" if $@;

while( <> )
    {
    print "$_\t\tdopasowano >>>$&<<<\n" if m/$regex/;
    }
```

Teraz widzę, że moje wyrażenie regularne dopasowuje kropkę, znak i kolejną kropkę, jak w `.8.`:

```
% perldoc -t perl | perl-grep3.pl  "\b(\S)\S\1\b"
                perl587delta        Perl changes in version 5.8.7
                                dopasowano >>>.8.<<<
                perl586delta        Perl changes in version 5.8.6
                                dopasowano >>>.8.<<<
                perl585delta        Perl changes in version 5.8.5
                                dopasowano >>>.8.<<<
```

Tak dla zabawy, jak sprawdzić, co zostało dopasowane w każdej grupie pamięciowej, czyli zmienne `$1`, `$2` itd.? Mógłbym spróbować wyświetlić ich zawartość bez względu na to, czy mam związane z nimi grupy przechwytywania, czy nie, ale ile jest takich zmiennych? Perl to „wie", ponieważ śledzi dopasowywanie w specjalnych tablicach `@a-` i `@a+`, które przechowują przesunięcia odpowiednio początku i końca każdego dopasowania. Oznacza to, że dla dopasowanego łańcucha w `$_` liczba grup pamięciowych jest równa ostatniemu indeksowi tablicy `@a-` lub `@a+` (mają one taką samą długość). Pierwszy element w każdej z nich dotyczy dopasowanej części łańcucha (a zatem `$&`), a następny element, o indeksie `1`, dotyczy zmiennej `$1` itd. aż do końca tablicy. Wartość w `$1` jest taka sama jak w poniższym wywołaniu `substr`:

```
my $one = substr(
        $_,             # łańcuch
        $-[1],          # początkowa pozycja $1
        $+[1] - $-[1]   # długość $1 (nie końcowa pozycja!)
        );
```

Aby wyświetlić zmienne pamięciowe, wystarczy przetworzyć w pętli indeksy tablicy `@-`:

```
#!/usr/bin/perl
# perl-grep4.pl

my $pattern = shift @ARGV;

my $regex = eval { qr/$pattern/ };
die "Sprawdź swój wzorzec! $@" if $@;

while( <> )
        {
        if( m/$regex/ )
                {

                print "$_";

                print "\t\t\$&: ",
```

```
                substr( $_, $-[$i], $+[$i] - $-[$i] ),
                "\n";

        foreach my $i ( 1 .. $#- )
                {
                print "\t\t\$$i: ",
                        substr( $_, $-[$i], $+[$i] - $-[$i] ),
                        "\n";
                }
        }
}
```

Teraz mogę zobaczyć dopasowaną część łańcucha, a także dopasowania podrzędne:

```
% perldoc -t perl | perl-grep4.pl  "\b(\S)\S\1\b"
        perl587delta        Perl changes in version 5.8.7
                            $&: .8.
                            $1: .
```

Jeśli dodam do wzorca więcej dopasowań podrzędnych, nie będę musiał niczego zmieniać, aby zobaczyć dodatkowe dopasowania:

```
% perldoc -t perl | perl-grep4.pl  "\b(\S)(\S)\1\b"
        perl587delta        Perl changes in version 5.8.7
                            $&: .8.
                            $1: .
                            $2: 8
```

(?imsx-imsx:WZORZEC)

A gdybym chciał, żeby mój program `grep` robił coś bardziej skomplikowanego, na przykład wyszukiwał łańcuchy bez uwzględniania wielkości liter? Aby użyć mojego programu do wyszukania ciągu „Perl” lub „perl”, mogę użyć kilku sposobów, z których żaden nie wymaga większego wysiłku:

```
% perl-grep.pl "[pP]erl"
% perl-grep.pl "(p|P)erl"
```

Gdybym chciał ignorować wielkość liter w całym wzorcu, miałbym znacznie więcej pracy, a to mi się nie podoba. W przypadku operatora dopasowania wystarczyłoby dodać na końcu opcję `/i`:

```
print if m/$regex/i;
```

Mógłbym to zrobić również w przypadku operatora `qr//`, choć wówczas wszystkie wzorce przestałyby rozróżniać wielkość liter:

```
my $regex = qr/$pattern/i;
```

Aby tego uniknąć, mogę określić opcje dopasowywania wewnątrz wzorca. Specjalna sekwencja `(?imsx)` pozwala włączyć wybrane opcje. Jeśli chcę ignorować wielkość liter, mogę użyć `(?i)` wewnątrz wzorca. Ignorowanie wielkości liter będzie obowiązywać w części wzorca następującej po `(?i)` (albo do końca tekstu w nawiasie):

```
% perl-grep.pl "(?i)perl"
```

Ogólnie rzecz biorąc, mogę włączać opcje dla części wzorca, określając te, które są mi potrzebne, w nawiasie okrągłym, ewentualnie razem z częścią wzorca, której dotyczą, jak pokazano w tabeli 2.1.

Tabela 2.1. Opcje używane w sekwencji (?opcje:WZORZEC)

Wpleciona opcja	Opis
`($i:WZORZEC)`	Ignorowanie wielkości liter
`($m:WZORZEC)`	Wielowierszowy tryb dopasowywania
`($s:WZORZEC)`	Kropka (`.`) dopasowuje znak nowego wiersza
`($x:WZORZEC)`	Tryb wyjaśniania (od ang. *eXplain*)

Opcje można nawet grupować:

```
(?si:WZORZEC)      Kropka dopasowuje nowy wiersz, ignorowanie wielkości liter
```

Jeśli poprzedzę opcję znakiem zapytania, wyłączę ją w danej grupie:

```
(?-s:PATTERN)      Kropka nie dopasowuje nowego wiersza
```

Jest to przydatne szczególnie wtedy, kiedy otrzymuję wzorzec z wiersza polecenia. W rzeczywistości, kiedy używam operatora `qr//` w celu utworzenia wyrażenia regularnego, opcje te są ustawiane automatycznie. Zmienię program tak, aby wyświetlał wyrażenie regularne po utworzeniu go przez operator `qr//`, ale przed użyciem:

```
#!/usr/bin/perl
# perl-grep3.pl

my $pattern = shift @ARGV;

my $regex = eval { qr/$pattern/ };
die "Sprawdź swój wzorzec! $@" if $@;

print "Wyrażenie regularne ---> $regex\n";

while( <> )
        {
        print if m/$regex/;
        }
```

Kiedy wyświetlam wyrażenie regularne, widzę, że początkowo wszystkie opcje są w nim wyłączone. W tekstowej wersji wyrażenia regularnego znajduje się ciąg `(?-OPCJE:WZORZEC)`, który wyłącza wszystkie opcje:

```
% perl-grep3.pl "perl"
Wyrażenie regularne ---> (?-xism:perl)
```

Kiedy włączam ignorowanie wielkości liter, wynikowy łańcuch wygląda nieco dziwnie: wyłącza opcję `i`, aby za chwilę włączyć ją ponownie:

```
% perl-grep3.pl "(?i)perl"
Wyrażenie regularne ---> (?-xism:(?i)perl)
```

Wyrażenie regularne Perla mają wiele podobnych sekwencji w nawiasie okrągłym, a kilka z nich pokażę w dalszej części rozdziału. Każda zaczyna się od nawiasu otwierającego, po którym następują pewne znaki określające żądaną operację. Ich pełną listę można znaleźć na stronie *perlre*.

Referencje jako argumenty

Ponieważ referencje są skalarami, mogę używać skompilowanego wyrażenia regularnego tak jak każdego innego skalara, na przykład zapisać go w tablicy lub asocjacji albo przekazać jako argument procedury. Na przykład moduł `Test::More` ma funkcję `like`, która przyjmuje wyrażenie regularne jako drugi argument. Mogę porównać łańcuch z wyrażeniem regularnym i otrzymać bardziej szczegółowe wyniki, jeśli łańcuch nie zostanie dopasowany:

```
use Test::More 'no_plan';

my $string = "Kolejny programista Perla,";
like( $string, qr/(\S+) haker/, "Co to za haker!" );
```

Ponieważ w łańcuchu `$string` występuje wyraz `programista` zamiast `haker`, test kończy się niepowodzeniem. Wyniki zawierają łańcuch, oczekiwaną wartość oraz wyrażenie regularne, którego spróbowałem użyć:

```
not ok 1 - Co to za haker!
1..1
#   Failed test 'Co to za haker!'
#                   'Kolejny programista Perla,'
#     doesn't match '(?-xism:(\S+) haker)'
# Looks like you failed 1 test of 1.
```

Funkcja `like` nie musi robić niczego specjalnego, aby przyjąć wyrażenie regularne jako argument, choć sprawdza typ referencji[2], zanim zacznie odprawiać swoje czary:

```
if( ref $regex eq 'Regexp' ) { ... }
```

Ponieważ `$regex` jest po prostu referencją (typu `Regexp`), mogę wykonywać na niej różne operacje „referencyjne", na przykład użyć funkcji `isa` do sprawdzenia typu albo pobrać typ za pomocą `ref`:

```
print "Mam wyrażenie regularne!\n" if $regex->isa( 'Regexp' );
print "Typ referencji to ", ref( $regex ), "\n";
```

Grupy nieprzechwytujące, (?:WZORZEC)

Nawiasy okrągłe w wyrażeniach regularnych nie muszą wyzwalać zapisywania dopasowanych części wzorca w pamięci. Można ich użyć wyłącznie do grupowania przez zastosowanie specjalnej sekwencji `(?:WZORZEC)`. Dzięki temu w grupach przechwytujących nie pojawiają się niepożądane dane.

Przypuśćmy, że chcę dopasowywać imiona po obu stronach spójników `i` albo `lub`. W tablicy `@array` mam kilka łańcuchów z takimi parami imion. Spójnik może się zmieniać, więc w wyrażeniu regularnym używam alternacji `i|lub`. Problemem jest pierwszeństwo operatorów. Alternacja ma wyższe pierwszeństwo niż sekwencja, więc aby wzorzec zadziałał, muszę umieścić alternację w nawiasie okrągłym, `(\S+) (i|lub) (\S+)`:

```
#!/usr/bin/perl

my @strings = (
        "Fred i Barney",
        "Jacek lub Agatka",
        "Fred i Ginger",
        );

foreach my $string ( @strings )
        {
        # $string =~ m/(\S+) i|lub (\S+)/; # nie działa
        $string =~ m/(\S+) (i|lub) (\S+)/;

        print "\$1: $1\n\$2: $2\n\$3: $3\n";
        print "-" x 10, "\n";
        }
```

[2] W rzeczywistości dzieje się to w metodzie `maybe_regex` klasy `Test::Builder`.

Wyniki pokazują niepożądaną konsekwencję grupowania alternacji: część łańcucha umieszczona w nawiasie pojawia się wśród zmiennych pamięciowych jako `$2` (tabela 2.2). Jest to skutek uboczny.

Tabela 2.2. Niepożądane przechwytywanie dopasowań

Bez grupowania i|lub	Z grupowaniem i|lub
`$1: Fred`	`$1: Fred`
`$2:`	`$2: i`
`$3:`	`$3: Barney`
`----------`	`----------`
`$1:`	`$1: Jacek`
`$2: Agatka`	`$2: lub`
`$3:`	`$3: Agatka`
`----------`	`----------`
`$1: Fred`	`$1: Fred`
`$2:`	`$2: i`
`$3:`	`$3: Ginger`
`----------`	`----------`

Użycie nawiasów rozwiązało problemy z pierwszeństwem, ale teraz mam dodatkową zmienną pamięciową, która wchodzi mi w drogę, kiedy zmieniam program tak, aby używał dopasowania w kontekście listy. Wszystkie zmienne pamięciowe, łącznie ze spójnikami, pojawiają się w tablicy `@names`:

```
# Dodatkowy element!
my @names = ( $string =~ m/(\S+) (i|lub) (\S+)/ );
```

Chcę po prostu grupować elementy bez ich zapamiętywania. Zamiast zwykłych nawiasów, których używałem do tej pory, dodaję `?:` zaraz za nawiasem otwierającym grupę, przez co otrzymuję nawias nieprzechwytujący. Zamiast `(i|lub)` mam teraz `(?:i|lub)`. Ta postać nie wyzwala zmiennych pamięciowych i nie zmienia ich numerowania. Mogę stosować kwantyfikatory, tak samo jak w zwykłych nawiasach. Teraz w tablicy `@names` nie pojawiają się dodatkowe elementy:

```
# Teraz tylko imiona
my @names = ( $string =~ m/(\S+) (?:i|lub) (\S+)/ );
```

Czytelne wyrażenia regularne, /x i (?#...)

Wyrażenia regularne mają zasłużoną reputację mało czytelnych. Są pisane w zwięzłym języku, który używa bardzo ograniczonej liczby znaków do reprezentowania praktycznie nieskończenie wielu możliwości, i to licząc tylko te elementy, których większość programistów używa na co dzień.

Na szczęście Perl daje mi możliwość znacznego zwiększenia czytelności wyrażeń regularnych. Wystarczy trochę odpowiedniego formatowania, a inni (albo ja sam kilka tygodni później) będą mogli łatwo stwierdzić, co próbuję dopasować. Wspomnieliśmy o tym krótko w książce *Perl. Wprowadzenie*, ale jest to tak dobry pomysł, że zamierzam omówić go dokładniej. Opisuje to również Damian Conway w książce *Perl. Najlepsze rozwiązania* (Helion).

Kiedy dodaję opcję /x do operatora dopasowania albo podstawienia, Perl ignoruje dosłowne odstępy we wzorcu. Oznacza to, że mogę rozdzielić części wyrażenia, aby ułatwić ich identyfikację. Moduł HTTP::Date napisany przez Gislego Aasa dokonuje analizy składniowej daty, wypróbowując kilka różnych wyrażeń regularnych. Oto jedno z jego wyrażeń, zmodyfikowane tak, aby mieściło się w jednym wierszu (tutaj zawinięte w celu dopasowania do strony):

```
/^(\d\d?)(?:\s+|[-\/])(\w+)(?:\s+|[-\/])
➥(\d+)(?:(?:\s+|:)(\d\d?):(\d\d)(?::(\d\d))
➥?)?\s*([-+]?\d{2,4}|(?![APap][Mm]\b)[A-Za-z]+)?\s*(?:\(\w+\))?\s*$/
```

Szybko: czy ktoś potrafi powiedzieć, który z licznych formatów daty dopasowuje to wyrażenie? Ja też nie. Na szczęście Gisle użył opcji /x, aby podzielić wyrażenie na części, i dodał komentarze, które informują, do czego służy każda część. Dzięki opcji /x Perl ignoruje odstępy oraz perlowskie komentarze wewnątrz wyrażenia. Oto rzeczywisty kod Gislego, który jest znacznie bardziej zrozumiały:

```
/^
 (\d\d?)                # dzień
    (?:\s+|[-\/])
 (\w+)                  # miesiąc
    (?:\s+|[-\/])
 (\d+)                  # rok
 (?:
       (?:\s+|:)        # separator przed czasem
    (\d\d?):(\d\d)      # godzina: minuta
    (?::(\d\d))?        # opcjonalne sekundy
 )?                     # opcjonalny zegar
        \s*
 ([-+]?\d{2,4}|(?![APap][Mm]\b)[A-Za-z]+)? # strefa czasowa
        \s*
 (?:\(\w+\))?           # reprezentacja ASCII strefy czasowej w nawiasie
        \s*$
/x
```

Kiedy używam opcji /x w celu dopasowania odstępu, muszę określić go jawnie — albo za pomocą symbolu \s, który pasuje do dowolnego odstępu (\f\r\n\t), albo za pomocą odpowiedniej sekwencji ósemkowej lub szesnastkowej, na przykład \040 lub \x20 dla dosłownej spacji[3]. Podobnie, jeśli potrzebuję dosłownego hasha (#), muszę poprzedzić go ukośnikiem odwrotnym: \#.

Nie muszę używać opcji /x, aby umieszczać komentarze w wyrażeniach regularnych. Mogę to zrobić również za pomocą sekwencji (?#KOMENTARZ), choć prawdopodobnie wyrażenie nie stanie się przez to bardziej czytelne. Mogę objaśnić części łańcucha tuż obok reprezentujących ich części wzorca. Choć jednak można używać sekwencji (?#), nie znaczy to jeszcze, że należy to robić. Myślę, że wzorce są znacznie czytelniejsze, kiedy używa się opcji /x:

```
$isbn = 'O-596-1O2O6-2';

$isbn =~ m/(\d+)(?#kraj)-(\d+)(?#wydawca)-(\d+)(?#pozycja)-([\dX])/i;

print <<"HERE";
Kod kraju:      $1
Kod wydawcy:    $2
Pozycja:        $3
Suma kontrolna: $4
HERE
```

[3] Mogę również poprzedzić dosłowną spację znakiem \, ale ponieważ spacji nie widać, wolę używać czegoś, co jest widoczne, na przykład \x20.

Dopasowywanie globalne

W książce *Perl. Wprowadzenie* wspomnieliśmy o opcji `/g`, której można użyć w celu dokonania wszystkich możliwych podstawień. Jak się okazuje, ma ona również inne zastosowania. Można użyć jej w połączeniu z operatorem dopasowania, a wówczas działa inaczej w kontekście skalarnym i w kontekście listy. Stwierdziliśmy, że operator dopasowania zwraca `true`, jeśli zdoła dopasować wzorzec, a `false` w przeciwnym przypadku. To prawda (przecież byśmy nie kłamali!), ale nie jest to po prostu wartość logiczna. Opcja `/g` jest najbardziej przydatna w kontekście listy. Operator dopasowania zwraca wówczas wszystkie zapamiętane dopasowania:

```
$_ = "Kolejny haker Perla,";
my @words = /(\S+)/g; # "Kolejny" "haker" "Perla,"
```

Choć w wyrażeniu mam tylko jeden nawias pamięciowy, znajduje on tyle dopasowań, ile się da. Po udanym dopasowaniu Perl zaczyna od miejsca, w którym skończył, i próbuje jeszcze raz. Powiem o tym więcej za chwilę. Często trafiam na zbliżony idiom Perla, w którym nie chodzi o rzeczywiste dopasowania, ale o ich liczbę:

```
my $word_count = () = /(\S+)/g;
```

Wykorzystano tu mało znaną, ale ważną zasadę: wynikiem operacji przypisania listy jest liczba elementów listy znajdującej się po prawej stronie. W tym przypadku jest to liczba elementów zwracanych przez operator dopasowania. Działa to tylko w kontekście listy, czyli w przypadku przypisywania jednej listy do drugiej. Dlatego potrzebny jest dodatkowy nawias `()`.

W kontekście skalarnym opcja `/g` wykonuje dodatkową pracę, o której nie wspomniano w poprzednich książkach. Po udanym dopasowaniu Perl zapamiętuje pozycję w łańcuchu, a kiedy znów porównuje ten sam łańcuch z wzorcem, zaczyna od miejsca, w którym skończył poprzednio. Zwraca wynik jednego zastosowania wzorca do łańcucha:

```
$_ = "Kolejny haker Perla,";
my @words = /(\S+)/g; # "Kolejny" "haker" "Perla,"

while( /(\S+)/g ) # kontekst skalarny
        {
        print "Następne słowo to '$1'\n";
        }
```

Kiedy ponownie dopasowuję ten sam łańcuch, Perl zwraca następne dopasowanie:

```
Następne słowo to 'Kolejny'
Następne słowo to 'haker'
Następne słowo to 'Perla,'
```

Mogę nawet sprawdzać pozycję dopasowywania w miarę przetwarzania łańcucha. Wbudowany operator `pos()` zwraca pozycję dopasowywania dla podanego łańcucha (domyślne dla `$_`). Pozycja w każdym łańcuchu jest śledzona oddzielnie. Pierwsza pozycja w łańcuchu to `0`, więc `pos()` zwraca `undef`, jeśli nie znajdzie dopasowania i pozycja zostanie zresetowana. Działa to tylko w przypadku użycia opcji `/g` (inaczej stosowanie operatora `pos()` nie miałoby sensu):

```
$_ = "Kolejny haker Perla,";
my $pos = pos( $_ );       # to samo co pos()
print "Jestem w pozycji [$pos]\n"; # undef

/(Kolejny)/g;
$pos = pos();
print "[$1] kończy się w pozycji $pos\n"; # 7
```

Kiedy dopasowywanie zawodzi, Perl resetuje wartość `pos()` na `undef`. Jeśli będę kontynuował dopasowywanie, zacznę ponownie od początku łańcucha (może to doprowadzić do nieskończonej pętli):

```
my( $third word ) = /(Java)/g;
print "Następna pozycja to " . pos() . "\n";
```

Na marginesie: bardzo nie lubię takich instrukcji `print`, w których operator konkatenacji jest używany w celu dołączenia wyniku wywołania funkcji do danych wyjściowych. Perl nie oferuje specjalnego sposobu interpolowania wywołań funkcji, więc konieczne jest małe oszustwo. Wywołuję funkcję w konstruktorze tablicy anonimowej `[ ... ]` i natychmiast wyłuskuję ją przez umieszczenie w bloku `@{ ... }`[4]:

```
print "Następna pozycja to @{ [ pos( $line ) ] }\n";
```

Operator `pos()` może być również l-wartością, przez co programiści rozumieją, że mogę używać go po lewej stronie przypisania i zmieniać jego wartość. Dzięki temu mogę skłonić operator dopasowania, aby zaczął od wybranego przeze mnie miejsca. Kiedy dopasuję pierwsze słowo w `$line`, pozycja dopasowywania znajduje się gdzieś za początkiem łańcucha. Następnie używam funkcji `index`, aby znaleźć następną literę `h` za bieżącą pozycją dopasowywania. Kiedy znajdę przesunięcie tej litery `h`, przypisuję je do `pos($line)`, aby następne dopasowywanie rozpoczęło się od tej pozycji[5]:

```
my $line = "Oto kolejny haker wyrażeń regularnych,";

$line =~ /(\S+)/g;
print "Pierwsze słowo to $1\n";
print "Następna pozycja to @{ [ pos( $line ) ] }\n";

pos( $line ) = index( $line, 'h', pos( $line) );

$line =~ /(\S+)/g;
print "Następne słowo to $1\n";
print "Następna pozycja to @{ [ pos( $line ) ] }\n";
```

Kotwice dopasowywania globalnego

Dotychczas kolejne dopasowania mogły „pływać”, to znaczy dopasowywanie mogło zaczynać się od dowolnego miejsca za pozycją początkową. Aby zakotwiczyć następne dopasowanie dokładnie tam, gdzie skończyłem poprzednie, używam kotwicy `\G`. Działa ona tak samo jak kotwica początku łańcucha `^`, ale wskazuje bieżącą pozycję dopasowywania. Jeśli dopasowywanie zawiedzie, Perl resetuje `pos()` i znów wracam do początku łańcucha.

W poniższym przykładzie zakotwiczam wzorzec za pomocą `\G`. Następnie używam nawiasów nieprzechwytujących, aby zgrupować opcjonalne odstępy, `\s*`, oraz wzorzec słowa, `\w+`. Używam też opcji `/x`, aby podzielić wyrażenie na części i zwiększyć jego czytelność. Program znajduje tylko cztery pierwsze słowa, ponieważ nie może dopasować przecinka (nie ma go w klasie `\w`) po słowie `regularnych`. Ponieważ następne dopasowanie musi zaczynać się tam, gdzie skoń-

[4] Tego samego triku można użyć do interpolowania wywołań funkcji w łańcuchu: `print "Wynik to: @{ [ funkcja (@argumenty) ] }"`.

[5] *Od tłumacza*: aby ten przykład zadziałał poprawnie (podobnie jak inne, w których występują polskie litery), należy użyć opcji `-CS` (wielkie litery) programu *perl* oraz pragmy `use utf8`; w tekście programu, na przykład:

```
#!/usr/bin/perl -CS
use utf8;
```

czyłem poprzednie, a jedyną rzeczą, jaką da się dopasować, są znaki odstępu albo słów, nie mogę przejść dalej. Następne dopasowanie zawodzi i Perl ustawia pozycję dopasowywania na początek `$line`:

```
my $line = "Kolejny haker wyrażeń regularnych, haker Perla,";

while( $line =~ /  \G (?: \s* (\w+) )  /xg )
        {
        print "Znaleziono słowo '$1'\n";
        print "Bieżąca pozycja to @{ [ pos( $line ) ] }\n";
        }
```

Można jednak zapobiec resetowaniu pozycji dopasowywania przez Perl. W tym celu należy użyć opcji `/c`, która po prostu wskazuje, że pozycja nie powinna być resetowana po nieudanym dopasowaniu. Mogę bezkarnie wypróbować jakiś wzorzec; jeśli to się nie uda, mogę spróbować czegoś innego w tej samej pozycji. Ta funkcja to skaner leksykalny dla ubogich. Oto bardzo uproszczony parser zdań:

```
my $line = "Kolejny haker wyrażeń regularnych, haker Perla; to chyba wszystko!\n";

while( 1 )
    {
    my( $found, $type )= do {
        if( $line =~ /\G(\w+)/igc )
                { ( $1, "słowo"               ) }
        elsif( $line =~ /\G (\n) /xgc               )
                { ( $1, "znak nowego wiersza" ) }
        elsif( $line =~ /\G (\s+) /xgc              )
                { ( $1, "znak odstępu"        ) }
        elsif( $line =~ /\G ( [[:punct:]] ) /xgc    )
                { ( $1, "znak interpunkcyjny" ) }
        else
                { last; ()                    }
        };

    print "Znaleziono $type [$found]\n";
    }
```

Przyjrzyjmy się dokładniej temu przykładowi. A gdybym chciał dodać więcej elementów do dopasowania? Musiałbym dopisać kolejną gałąź struktury decyzyjnej. Nie podoba mi się to. Wielokrotnie powtarzam strukturę kodu, która robi to samo: dopasowuje coś, a następnie zwraca `$1` i opis. Zmienię zatem kod tak, aby usunąć powtarzającą się strukturę. Wyrażenia regularne mogę zapisać w tablicy `@items`. Używam pokazanego wcześniej operatora przytoczenia `qr//` i ustawiam wyrażenia regularne w takiej kolejności, w jakiej mają być wypróbowywane. Pętla `foreach` przetwarza je kolejno, aż znajdzie takie, które pasuje, a wówczas wypisuje komunikat złożony z opisu oraz zawartości zmiennej `$1`. Jeśli zechcę dodać więcej leksemów, po prostu uzupełnię tablicę `@items`:

```
#!/usr/bin/perl
use strict;
use warnings;

my $line = "Kolejny haker wyrażeń regularnych, haker Perla; to chyba wszystko!\n";

my @items = (
        [ qr/\G(\w+)/i,        "słowo"               ],
        [ qr/\G(\n)/,          "znak nowego wiersza" ],
        [ qr/\G(\s+)/,         "znak odstępu"        ],
        [ qr/\G([[:punct:]])/, "znak interpunkcyjny" ],
        );
```

```
LOOP: while( 1 )
        {
        MATCH: foreach my $item ( @items )
                {
                my( $regex, $description ) = @$item;
                my( $type, $found );

                next unless $line =~ /$regex/gc;

                print "Znaleziono $description [$1]\n";
                last LOOP if $1 eq "\n";

                next LOOP;
                }
        }
```

Zobaczmy, co dzieje się w tym przykładowym programie. Wszystkie dopasowania wymagają opcji `/gc`, więc umieszczam te opcje w pętli `foreach`. Wyrażenie dopasowujące słowa wymaga jednak również opcji `/i`. Nie mogę dodać jej do operatora dopasowania, ponieważ w przyszłości mogą pojawić się nowe gałęzie, w których byłaby niepożądana. Dodaję zatem asercję `/i` do odpowiedniego wyrażenia regularnego w tablicy `@items`, włączając ignorowanie wielkości liter tylko w tym wyrażeniu. Gdybym chciał zachować eleganckie formatowanie z wcześniejszych przykładów, mógłbym zastosować sekwencję `(?ix)`. Nawiasem mówiąc, jeśli większość wyrażeń regularnych powinna ignorować wielkość liter, mogę dodać opcję `/i` do operatora dopasowania, a następnie wyłączyć ją za pomocą opcji `(?-i)` w odpowiednich wyrażeniach.

Patrzenie w przód i w tył

Operatory patrzenia (ang. *lookarounds*) to arbitralne kotwice w wyrażeniach regularnych. Kilka kotwic, takich jak `^`, `$` oraz `\b`, omówiliśmy w książce *Perl. Wprowadzenie*, a przed chwilą pokazałem kotwicę `\G`. Za pomocą operatora patrzenia mogę opisać moją własną kotwicę za pomocą wyrażenia regularnego — podobnie jak inne kotwice, nie liczy się ono jako część wzorca ani nie pochłania znaków łańcucha. Określa warunek, który musi być spełniony, ale nie wchodzi w skład części łańcucha dopasowywanej przez ogólny wzorzec.

Operatory patrzenia mają dwa rodzaje: **operatory patrzenia w przód** (ang. *lookaheads*), które sprawdzają pewien warunek tuż za bieżącą pozycją dopasowywania, oraz **operatory patrzenia w tył** (ang. *lookbehinds*), które sprawdzają pewien warunek tuż przed bieżącą pozycją dopasowywania. Wydaje się to proste, ale łatwo o złe zastosowanie tych reguł. Trzeba przypomnieć sobie, że operator zakotwicza się w bieżącej pozycji dopasowywania, a następnie ustalić, której strony dotyczy.

Zarówno operatory patrzenia w przód, jak i patrzenia w tył mają dwie odmiany: **pozytywną** i **negatywną**. Odmiana pozytywna potwierdza, że dany wzorzec pasuje, a odmiana negatywna — że nie pasuje. Bez względu zastosowany operator patrzenia trzeba pamiętać, że dotyczy on bieżącej pozycji dopasowywania, a nie jakiegoś innego miejsca w łańcuchu.

Asercje z patrzeniem w przód, (?=WZORZEC) i (?!WZORZEC)

Asercje z patrzeniem w przód pozwalają mi zerknąć na część łańcucha znajdującą się tuż za bieżącą pozycją dopasowywania. Asercja nie pochłania części łańcucha, a jeśli się powiedzie, dopasowywanie jest kontynuowane tuż za bieżącą pozycją.

Pozytywne asercje z patrzeniem w przód

W książce *Perl. Wprowadzenie* zamieściliśmy ćwiczenie polegające na sprawdzaniu, czy w danym wierszu występuje zarówno słowo „Fred", jak i „Wilma", bez względu na ich kolejność. Chodziło o to, aby pokazać początkującym programistom Perla, że dwa wyrażenia regularne mogą być prostsze od jednego. Jednym z rozwiązań jest powtórzenie łańcuchów `Wilma` i `Fred` w alternacji i wypróbowanie obu kolejności. Drugim — podzielenie ich na dwa wyrażenia regularne:

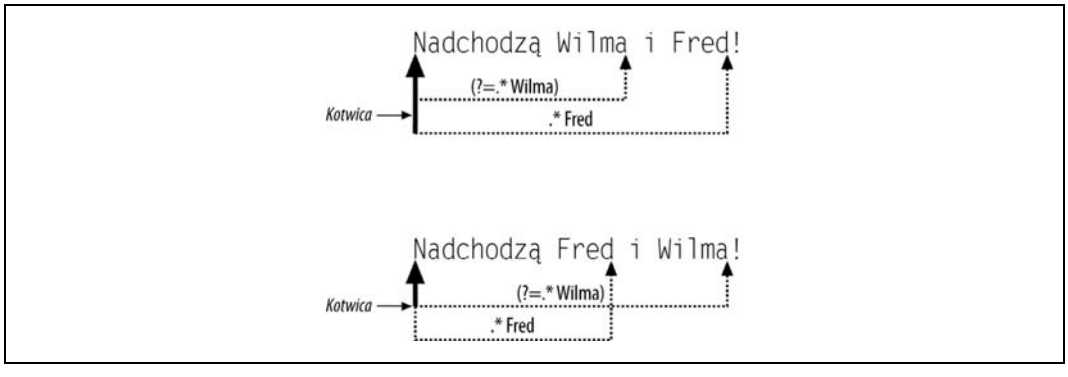
```
#/usr/bin/perl
# fred-and-wilma.pl

$_ = "Nadchodzą Wilma i Fred!";
print "Pasuje: $_" if /Fred.*Wilma|Wilma.*Fred/;
print "Pasuje: $_" if /Fred/ && /Wilma/;
```

Mogę również utworzyć proste, pojedyncze wyrażenie regularne z **pozytywną asercją patrzenia w przód** oznaczoną przez `(?=WZORZEC)`. Ta asercja nie pochłania tekstu w łańcuchu, ale jeśli nie jest spełniona, to całe wyrażenie nie zostanie dopasowane. W tym przykładzie w pozytywnej asercji używam wzorca `.*Wilma`. Wzorzec ten musi zostać znaleziony tuż za bieżącą pozycją dopasowywania:

```
$_ = "Nadchodzą Wilma i Fred!";
print "Pasuje: $_" if /(?=.*Wilma).*Fred/;
```

Umieściłem asercję na początku wzorca, co oznacza, że musi być spełniona na początku łańcucha. Mówiąc ściślej, na początku łańcucha musi zostać dopasowana dowolna liczba znaków (z wyjątkiem znaku nowego wiersza), po którym następuje ciąg `Wilma`. Jeśli to się powiedzie, reszta wzorca zostanie zakotwiczona w pozycji asercji (na początku łańcucha). Na rysunku 2.1 pokazano dwa sposoby działania tego wzorca zależne od kolejności ciągów `Fred` i `Wilma` w badanym łańcuchu. Ciąg `.*Wilma` zakotwicza się tam, gdzie rozpoczęło się jego dopasowywanie. Elastyczny symbol `.*`, który może dopasować dowolną liczbę znaków innych niż znak nowego wiersza, powoduje zatem zakotwiczenie wzorca na początku łańcucha.



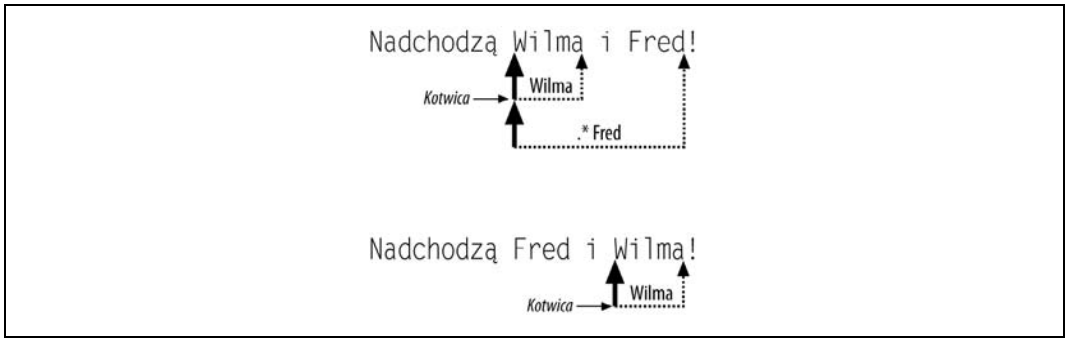
*Rysunek 2.1. Asercja z patrzeniem w przód (?=.*Wilma) zakotwicza wzorzec na początku łańcucha*

Łatwiej jednak zrozumieć operatory patrzenia przez zbadanie przykładów, w których nie działają. Zmienię nieco wzorzec, usuwając `.*` z asercji. Początkowo wydaje się, że będzie działał, ale zawodzi, kiedy zmienię kolejność ciągów `Fred` i `Wilma`:

```
$_ = "Nadchodzą Wilma i Fred!";
print "Pasuje: $_" if /(?=Wilma).*Fred/;

$_ = "Nadchodzą Fred i Wilma!";
print "Pasuje: $_" if /(?=Wilma).*Fred/;
```

Na rysunku 2.2 pokazano, co się dzieje. W pierwszym przypadku operator patrzenia zakotwicza wzorzec na początku ciągu `Wilma`. Perl wypróbował asercję na początku łańcucha, ustalił, że nie jest spełniona, po czym przesunął się o jedną pozycję w przód i spróbował jeszcze raz. Robił to dopóty, dopóki nie dotarł do ciągu `Wilma`. Kiedy pomyślnie dopasował ten ciąg, ustawił kotwicę. Reszta wzorca musi zaczynać się od tej pozycji.



Rysunek 2.2. Asercja z patrzeniem w przód (?=Wilma) zakotwicza wzorzec na ciągu Wilma

W pierwszym przypadku da się dopasować ciąg `.*Fred` od pozycji kotwicy, ponieważ następuje on po ciągu `Wilma`. W drugim przypadku przedstawionym na rysunku 2.2 Perl postępuje podobnie. Wypróbowuje asercję na początku łańcucha, ustala, że nie jest spełniona, po czym przesuwa się o jedną pozycję w przód. Asercję uda się dopasować dopiero po minięciu ciągu `Fred`. Reszta wzorca musi zaczynać się od kotwicy, więc nie da się jej dopasować.

Ponieważ asercje z patrzeniem w przód nie pochłaniają części łańcucha, mogę używać ich we wzorcach `split`, kiedy nie chcę odrzucać dopasowanych części wzorca. W poniższym przykładzie chcę wydzielić słowa z łańcucha zapisanego w notacji „wielbłądziej". Łańcuch trzeba podzielić na części zaczynające się od wielkiej litery. Chcę jednak zachować początkową literę, więc używam asercji z patrzeniem w przód zamiast ciągu pochłaniającego znaki. Różni się to od trybu zachowywania separatorów, ponieważ wzorzec podziału w istocie nie jest separatorem, lecz po prostu kotwicą:

```
my @words = split /(?=[A-Z])/, 'ŁańcuchWNotacjiWielbłądziej';
print join '_', map { lc } @words; # łańcuch_w_notacji_wielbłądziej
```

Negatywne asercje z patrzeniem w przód

Przypuśćmy, że chcę znaleźć wiersze wejściowe, które zawierają ciąg `Perl`, ale tylko pod warunkiem, że nie jest to `Perl6` albo `Perl 6`. Próbuję użyć zanegowanej klasy znakowej, która ma gwarantować, że zaraz za literą `l` w ciągu `Perl` nie pojawi się cyfra `6`. Używam też kotwic granicy słów `\b`, ponieważ nie chcę dopasowywać ciągu `Perl` wewnątrz innych słów, takich jak „BioPerl" lub „PerlPoint":

```
#!/usr/bin/perl
# not-perl6.pl

print "Wypróbowuję zanegowaną klasę znakową:\n";
while( <> )
    {
    print if /\bPerl[^6]\b/;  #
    }
```

Wypróbuję ten wzorzec na przykładowych danych:

```
# Przykładowe dane
Najpierw był Perl 5, a potem Perl6.
W ciągu Perl 6 jest spacja.
Po prostu mówię "Perl".
To jest wiersz języka Perl 5
Perl 5 to bieżąca wersja.
Kolejny haker języka Perl 5,
Na końcu jest Perl
PerlPoint to PowerPoint
BioPerl jest genetyczny
```

Nie działa on prawidłowo dla wszystkich wierszy. Znajduje tylko cztery wiersze, w których występuje ciąg `Perl` bez następującej po nim cyfry `6`, a także wiersz, w którym między `Perl` a `6` jest spacja:

```
Wypróbowuję zanegowaną klasę znakową:
        Najpierw był Perl 5, a potem Perl6.
        W ciągu Perl 6 jest spacja.
        To jest wiersz języka Perl 5
        Perl 5 to bieżąca wersja.
        Kolejny haker języka Perl 5,
```

Wzorzec nie działa, ponieważ po literze `l` w `Perl` musi występować znak, a w dodatku określiłem granicę słowa. Jeśli po literze `l` występuje znak nienależący do klasy znaków słów, taki jak cudzysłów w `Po prostu mówię "Perl"`, granica słowa nie zostanie dopasowana. Jeśli usunę końcowe `\b`, zostanie dopasowany ciąg `PerlPoint`. Nie spróbowałem nawet obsłużyć przypadku, w którym między `Perl` a `6` występuje spacja. Do tego będę potrzebował czegoś znacznie lepszego.

Okazuje się, że mogę to zrobić bardzo łatwo za pomocą negatywnej asercji patrzenia w przód. Nie chcę dopasowywać znaku po `l`, a ponieważ asercja nie dopasowuje znaków, jest właściwym narzędziem do tego celu. Po prostu chcę powiedzieć, że jeśli cokolwiek następuje po ciągu `Perl`, to nie może być to cyfra `6`, nawet jeśli jest przed nią jakiś odstęp. **Negatywna asercja z patrzeniem w przód** ma postać `(?!WZORZEC)`. Aby rozwiązać problem, jako wzorca używam `\s?6`, co oznacza opcjonalny odstęp, po którym następuje `6`:

```
print "Wypróbowuję negatywną asercję z patrzeniem w przód:\n";
while( <> )
        {
        print if /\bPerl(?!\s?6)\b/;  # lub /\bPerl[^6]/
        }
```

Teraz w wynikach pojawiają się wszystkie właściwe wiersze:

```
Wypróbowuję negatywną asercję z patrzeniem w przód:
        Najpierw był Perl 5, a potem Perl 6.
        Po prostu mówię "Perl".
        To jest wiersz języka Perl 5
        Perl 5 to bieżąca wersja.
        Kolejny haker języka Perl 5,
        Na końcu jest Perl
```

Należy pamiętać, że `(?!WZORZEC)` to asercja z patrzeniem **w przód**, więc wyszukuje wzorzec **za** bieżącą pozycją dopasowywania. Właśnie dlatego pokazany niżej wzorzec zostaje dopasowany. Tuż przed literą `b` w `bar` asercja sprawdza, czy dalej nie następuje `foo`. Ponieważ dalej następuje `bar`, a nie `foo`, wzorzec pasuje. Wiele osób błędnie sądzi, że poniższy wzorzec oznacza, że przed ciągiem `foo` nie może występować `bar`, ale oba są dopasowywane od tej samej pozycji, więc oba warunki są spełnione:

```
if( 'foobar' =~ /(?!foo)bar/ )
        {
        print "Pasuje! Nie o to mi chodziło!\n";
        }
else
        {
        print "Nie pasuje! Hura!\n";
        }
```

Asercje z patrzeniem w tył, (?<!WZORZEC) i (?<=WZORZEC)

Zamiast przyglądać się części łańcucha, która dopiero ma nastąpić, mogę spojrzeć w tył i sprawdzić tę część, która została już przetworzona przez mechanizm wyrażeń regularnych. Ze względu na szczegóły implementacyjne Perla asercje z patrzeniem w tył muszą mieć stałą długość, więc nie można stosować w nich kwantyfikatorów o zmiennej długości.

Teraz mogę spróbować dopasować ciąg `bar`, który nie następuje po `foo`. W poprzednim rozdziale nie mogłem użyć negatywnej asercji z patrzeniem w przód, ponieważ sprawdza ona następującą dalej część łańcucha. **Negatywna asercja z patrzeniem w tył**, oznaczana przez `(?<!WZORZEC)`, sprawdza poprzednią część łańcucha. Właśnie tego mi potrzeba. Teraz otrzymuję prawidłową odpowiedź:

```
#!/usr/bin/perl
# correct-foobar.pl

if( 'foobar' =~ /(?<!foo)bar/ )
        {
        print "Pasuje! Nie o to mi chodziło!\n";
        }
else
        {
        print "Nie pasuje! Hura!\n";
        }
```

Ponieważ mechanizm wyrażeń regularnych przetworzył część łańcucha, zanim dotarł do `bar`, moja asercja z patrzeniem w tył nie może być wzorcem o zmiennej długości. Nie mogę używać kwantyfikatorów, ponieważ mechanizm nie cofnie się, aby dopasować asercję. Nie będę mógł więc sprawdzić zmiennej liczby liter `o` w `foooo`:

```
'fooooobar' =~ /(?<!fo+)bar/;
```

Jeśli spróbuję to zrobić, otrzymam komunikat o błędzie. Choć stwierdza on tylko, że funkcja jest niezaimplementowana, nie warto czekać ze wstrzymanym oddechem, aż to się zmieni:

```
Variable length lookbehind not implemented in regex...
```

Pozytywna asercja z patrzeniem w tył również sprawdza poprzednią część łańcucha, ale jej wzorzec **musi** pasować. Asercje te wykorzystuję tylko w podstawieniach w połączeniu z inną asercją. Kiedy używam zarówno asercji z patrzeniem w tył, jak i z patrzeniem w przód, niektóre podstawienia stają się bardziej czytelne.

Na przykład kiedy pisałem tę książkę, używałem różnych odmian słów z łącznikiem, ponieważ nie umiałem zdecydować, która jest właściwa. Czy pisze się `builtin` czy `built-in`? W zależności od nastroju wybierałem jedną albo drugą wersję[6].

[6] Wydawnictwo O'Reilly często ma do czynienia z tym problemem, więc udostępnia listę słów z zalecaną pisownią, ale nie znaczy to jeszcze, że tacy autorzy jak ja ją czytają: *http://www.oreilly.com/oreilly/author/stylesheet.html*.

Musiałem jakoś poradzić sobie z własną niekonsekwencją. Znam część słowa po lewej stronie łącznika oraz część po prawej stronie. Tam, gdzie spotykają się części, powinien być łącznik. Po chwili zastanowienia łatwo dojść do wniosku, że operatory patrzenia będą tu wprost idealne: chcę umieścić coś w określonej pozycji i wiem, co powinno znajdować się naokoło. Oto przykładowy program, który używa pozytywnej asercji z patrzeniem w tył, aby sprawdzić tekst po lewej stronie, i pozytywnej asercji z patrzeniem w przód, aby sprawdzić tekst po prawej. Wyrażenie jest dopasowywane tylko wtedy, gdy te dwie strony się spotykają, co oznacza, że wykryto brakujący łącznik. Kiedy dokonuję podstawienia, umieszczam łącznik w pozycji dopasowania i nie muszę się martwić o konkretny tekst:

```
@hyphenated = qw( built-in );

foreach my $word ( @hyphenated )
        {
        my( $front, $back ) = split /-/, $word;

        $text =~ s/(?<=$front)(?=$back)/-/g;
        }
```

Jeśli ten przykład nie jest wystarczająco skomplikowany, spróbujmy czegoś innego. Użyjmy operatorów patrzenia, aby dodać spacje do liczb. Jeffrey Friedl zamieścił w książce *Wyrażenia regularne* przykładowy wzorzec, który dodaje spacje do liczby obywateli Stanów Zjednoczonych[7]:

```
$pop = 302799312;  # dane z 6 września 2007

# Z książki Jeffreya Friedla
$pop =~ s/(?<=\d)(?=(?:\d\d\d)+$)/ /g;
```

Wzorzec ten działa — do pewnego stopnia. Pozytywne patrzenie w tył `(?<=\d)` próbuje dopasować liczbę, a pozytywne patrzenie w przód `(?=(?:\d\d\d)+$)` znajduje grupy trzech cyfr aż do końca łańcucha. Nie działa to jednak w przypadku liczb zmiennopozycyjnych, na przykład kwot pieniężnych. Mój broker śledzi kursy akcji z dokładnością do czterech miejsc po przecinku. Kiedy próbuję takiego podstawienia, nie otrzymuję spacji po lewej stronie przecinka dziesiętnego, za to pojawia się ona po przecinku. Dzieje się tak ze względu na kotwicę końca łańcucha:

```
$money = '1234,5678';

$money =~ s/(?<=\d)(?=(?:\d\d\d)+$)/ /g;  # 1234,5 678
```

Mogę nieco zmodyfikować ten wzorzec. Zamiast kotwicy końca łańcucha użyję granicy słowa `\b`. Może to wydawać się dziwne, ale pamiętajmy, że cyfry są znakami słów. Dzięki temu otrzymuję spację po lewej stronie przecinka, ale nadal nie pozbyłem się tej po prawej:

```
$money = '1234,5678';

$money =~ s/(?<=\d)(?=(?:\d\d\d)+\b)/ /g;  # 1234.5 678
```

W pierwszej część wyrażenia regularnego tak naprawdę powinienem użyć patrzenia w tył, aby dopasować cyfrę, ale nie wtedy, gdy poprzedza ją przecinek dziesiętny. To jest opis negatywnego patrzenia w tył, `(?<!\,\d)`. Ponieważ wszystkie operatory patrzenia dopasowują wzorce od tej samej pozycji, nie ma znaczenia, że niektóre się nakładają, pod warunkiem że robią to, co chcę:

```
$money = '1234,5678';

$money =~ s/(?<!\,\d)(?<=\d)(?=(?:\d\d\d)+\b)/ /g; # 1234,5678
```

[7] Biuro Spisu Ludności Stanów Zjednoczonych prowadzi „zegar populacji", więc ci, którzy czytają książkę długo po jej wydaniu, mogą znaleźć aktualną liczbę pod adresem *http://www.census.gov/main/www/popclock.html*.

To działa! A szkoda, bo potrzebowałem wymówki, żeby dodać do wzorca negatywne patrzenie w przód. Wyrażenie jest teraz dość skomplikowane, więc dodam opcję `/x`, aby praktykować to, czego nauczam:

```
$money =~ s/
    (?<!\,\d)          # inne znaki niż przecinek-cyfra tuż przed pozycją

    (?<=\d)            # cyfra tuż przed pozycją
                       # <--- BIEŻĄCA POZYCJA DOPASOWYWANIA
    (?=                # ta grupa tuż za pozycją
        (?:\d\d\d)+    # jedna lub więcej grup złożonych z trzech cyfr
        \b             # granica słowa (lewa strona przecinka lub koniec łańcucha)
    )

    / /xg;
```

Odszyfrowywanie wyrażeń regularnych

Kiedy próbuję zrozumieć, o co chodzi w jakimś wyrażeniu regularnym — znalezionym w czyimś kodzie albo napisanym przez mnie (czasem dawno temu) — mogę włączyć tryb debugowania wyrażeń regularnych[8]. Opcja `-D` włącza opcje debugowania interpretera Perla (nie programu, jak w rozdziale 4.). Opcja ta przyjmuje serię liter lub liczb, które określają, co należy włączyć. Opcja `-Dr` włącza debugowanie analizy składniowej oraz wykonywania wyrażeń regularnych.

Do zbadania wyrażenia regularnego wykorzystam krótki program. Pierwszym argumentem będzie łańcuch, a drugim wyrażenie regularne. Zapiszę ten program pod nazwą *explain-regex*:

```
#!/usr/bin/perl

$ARGV[0] =~ /$ARGV[1]/;
```

Kiedy uruchomię ten program z łańcuchem Oto kolejny haker Perla, oraz wzorcem `Oto kolejny haker (\S+),`, zobaczę dwie podstawowe sekcje wyników, które są opisane dokładnie w dokumentacji *perldebguts*. Perl najpierw kompiluje wyrażenie regularne, a wyniki opcji `-Dr` pokazują, w jaki sposób przeprowadzono jego analizę składniową. Widać węzły wyrażenia, takie jak `EXACT` i `NSPACE`, a także optymalizacje w rodzaju `anchored "Oto kolejny haker "`. Następnie Perl próbuje dopasować docelowy łańcuch i pokazuje postępy. To sporo informacji, ale dzięki nim dowiaduję się dokładnie, co robi Perl:

```
$ perl -Dr explain-regex 'Oto kolejny haker Perla,' 'Oto kolejny haker (\S+),'
Omitting $` $& $' support.

EXECUTING...

Compiling REx `Oto kolejny haker (\S+),'
size 15 Got 124 bytes for offset annotations.
first at 1
rarest char , at 0
rarest char j at 8
   1: EXACT <Oto kolejny haker >(7)
   7: OPEN1(9)
   9:   PLUS(11)
  10:     NSPACE(0)
  11: CLOSE1(13)
```

[8] Tryb debugowania wyrażeń regularnych wymaga interpretera Perla skompilowanego z opcją `-DDEBUGGING`. Opcje kompilacyjne interpretera można wyświetlić za pomocą polecenia `perl -V`.

```
  13: EXACT <,>(15)
  15: END(0)
anchored "Oto kolejny haker " at 0 floating "," at 19..2147483647 (checking anchored)
↪minlen 20
Offsets: [15]
        1[18] 0[0] 0[0] 0[0] 0[0] 0[0] 19[1] 0[0] 22[1] 20[2] 23[1] 0[0] 24[1] 0[0] 25[0]
Guessing start of match, REx "Oto kolejny haker (\S+)," against
↪"Oto kolejny haker Perla,"...
Found anchored substr "Oto kolejny haker " at offset 0...
Found floating substr "," at offset 23...
Guessed: match at offset 0
Matching REx "Oto kolejny haker (\S+)," against "Oto kolejny haker Perla,"
  Setting an EVAL scope, savestack=3
   0 <> <Oto kolejny >    |  1:  EXACT <Oto kolejny haker >
  18 <haker > <Perla,>    |  7:  OPEN1
  18 <haker > <Perla,>    |  9:  PLUS
                           NSPACE can match 6 times out of 2147483647...
  Setting an EVAL scope, savestack=3
  23 <haker Perla> <,>    | 11:    CLOSE1
  23 <haker Perla> <,>    | 13:    EXACT <,>
  24 <haker Perla,> <>    | 15:    END
Match successful!
Freeing REx: `"Oto kolejny haker (\\S+),"'
```

Pragma `re` Perla ma tryb debugowania, który nie wymaga interpretera skompilowanego z opcją `-DDEBUGGING`. Instrukcja `use re 'debug'` dotyczy całego programu; nie ma zasięgu leksykalnego, jak większość pragm. Zmodyfikuję poprzedni program tak, aby używał pragmy `re` zamiast opcji wiersza polecenia:

```
#!/usr/bin/perl

use re 'debug';

$ARGV[0] =~ /$ARGV[1]/;
```

Nie muszę modyfikować program w celu użycia pragmy `re`, ponieważ mogę ją włączyć z poziomu wiersza poleceń:

```
$ perl -Mre=debug explain-regex 'Oto kolejny haker Perla,' 'Oto kolejny haker (\S+),'
```

Kiedy uruchomię ten program w pokazany wyżej sposób, otrzymam niemal dokładnie te same wyniki, co w poprzednim przykładzie z opcją `-Dr`.

Moduł `YAPE::Regex::Explain`, choć nieco stary, potrafi objaśnić wyrażenie regularne w zwykłym języku angielskim. Dokonuje analizy składniowej wyrażenia i wyjaśnia, co robi każda jego część. Nie potrafi wyjaśnić semantyki wyrażenia, ale nie można mieć wszystkiego. Za pomocą prostego programu mogę objaśniać wyrażenia podane w wierszu poleceń:

```
#!/usr/bin/perl

use YAPE::Regex::Explain;

print YAPE::Regex::Explain->new( $ARGV[0] )->explain;
```

Kiedy uruchamiam program nawet z krótkim, prostym wyrażeniem, otrzymuję obszerne wyniki:

```
$ perl yape-explain 'Oto kolejny haker (\S+),'
The regular expression:

(?-imsx:Oto kolejny haker (\S+),)

matches as follows:
```

```
NODE                    EXPLANATION
----------------------------------------------------------------------
(?-imsx:                group, but do not capture (case-sensitive)
                        (with ^ and $ matching normally) (with . not
                        matching \n) (matching whitespace and #
                        normally):
----------------------------------------------------------------------
  Oto kolejny haker       'Oto kolejny haker '
----------------------------------------------------------------------
  (                       group and capture to \1:
----------------------------------------------------------------------
    \S+                     non-whitespace (all but \n, \r, \t, \f,
                            and " ") (1 or more times (matching the
                            most amount possible))
----------------------------------------------------------------------
  )                       end of \1
----------------------------------------------------------------------
  ,                       ','
----------------------------------------------------------------------
)                       end of grouping
----------------------------------------------------------------------
```

Końcowe myśli

Dotarłem niemal do końca rozdziału, ale wyrażenia regularne mają znacznie więcej funkcji, które wydają mi się przydatne. Czytelnicy mogą potraktować ten podrozdział jako krótki przewodnik po funkcjach, które mogą przestudiować samodzielnie.

Nie muszę ograniczać się do prostych klas znakowych, takich jak `\w` (znaki słów), `\d` (cyfry) oraz inne sekwencje ukośnikowe. Mogę również używać klas znakowych POSIX. Umieszczam je w nawiasie kwadratowym z dwukropkiem po obu stronach nazwy:

```
print "Znalazłem znak alfabetyczny!\n"  if  $string =~ m/[:alpha:]/;
print "Znalazłem cyfrę szesnastkową!\n" if  $string =~ m/[:xdigit:]/;
```

Aby zanegować te klasy, używam daszka (^) po pierwszym dwukropku:

```
print "Nie znalazłem znaków alfabetycznych!\n" if  $string =~ m/[:^alpha:]/;
print "Nie znalazłem spacji!\n" if  $string =~ m/[:^space:]/;
```

Mogę uzyskać ten sam efekt przez podanie nazwanej właściwości. Sekwencja `\p{`*Nazwa*`}` (mała litera `p`) dołącza znaki odpowiadające nazwanej właściwości, a sekwencja `\P{`*Nazwa*`}` (wielka litera `P`) jest jej dopełnieniem:

```
print "Znalazłem znak ASCII!\n"     if  $string =~ m/\p{IsASCII}/;
print "Znalazłem znak kontrolny!\n" if  $string =~ m/\p{IsCntrl}/;

print "Nie znalazłem znaków interpunkcyjnych!\n" if  $string =~ m/\P{IsPunct}/;
print "Nie znalazłem wielkich liter!\n"          if  $string =~ m/\P{IsUpper}/;
```

Moduł `Regexp::Common` zawiera przetestowane i sprawdzone wyrażenia regularne dla popularnych wzorców, takich jak adresy WWW, liczby, kody pocztowe, a nawet przekleństwa. Oferuje wielopoziomową tablicę asocjacyjną `%RE`, której wartościami są wyrażenia regularne. Jeśli komuś to nie pasuje, może skorzystać z interfejsu funkcyjnego:

```
use Regexp::Common;

print "Znalazłem liczbę rzeczywistą\n" if $string =~ /$RE{num}{real}/;

print "Znalazałem liczbę rzeczywistą\n" if $string =~ RE_num_real;
```

Aby zbudować własny wzorzec, mogę skorzystać z modułu `Regexp::English`, który używa serii połączonych w łańcuch metod do zwrócenia obiektu reprezentującego wyrażenia regularne. Prawdopodobnie nie użylibyśmy go w rzeczywistym programie, ale zapewnia dobrą umysłową rozrywkę:

```
#!/usr/bin/perl

use Regexp::English;

my $regexp = Regexp::English->new
    ->literal( 'Jeszcze' )
        ->whitespace_char
    ->word_chars
        ->whitespace_char
    ->remember( \$type_of_hacker )
    ->word_chars
    ->end
        ->whitespace_char
    ->literal( 'haker' );

$regexp->match( 'Jeszcze jeden Perla haker,' );

print "Typ hakera to [haker $type_of_hacker]\n";
```

Aby dowiedzieć się więcej o wyrażeniach regularnych, warto zajrzeć do książki *Wyrażenia regularne* Jeffreya Friedla. Opisuje ona nie tylko zaawansowane funkcje, ale również działanie wyrażeń regularnych oraz sposoby ich optymalizowania.

Podsumowanie

W tym rozdziale omówiłem kilka przydatnych funkcji mechanizmu wyrażeń regularnych w Perlu. Operator przytaczania `qr()` pozwala skompilować wyrażenie regularne do późniejszego użytku i zwraca je jako referencję. Specjalne sekwencje `(?)` znacznie zwiększają możliwości wyrażeń regularnych, a przy tym zmniejszają ich złożoność. Kotwica `\G` pozwala zakotwiczyć następne dopasowanie w miejscu, w którym skończyło się poprzednie, a opcja `/c` umożliwia wypróbowania kilku możliwości bez resetowania pozycji dopasowywania, jeśli któraś z nich zawiedzie.

Dalsza lektura

perlre to główna dokumentacja poświęcona wyrażeniom regularnym Perla, a strona *perlretut* wyjaśnia, jak się nimi posługiwać. Nie należy jej mylić z *perlreftut*, samouczkiem poświęconym referencjom. Aby jeszcze bardziej skomplikować sprawy, strona *perlreref* to krótka ściąga z wyrażeń regularnych.

Informacje o debugowaniu wyrażeń regularnych znajdują się na stronie *perldebguts*. Wyjaśnia ona wyniki `-Dr` oraz `re 'debug'`.

Książka *Perl. Najlepsze rozwiązania* zawiera rozdział poświęcony wyrażeniom regularnym, a opcja „rozszerzonego formatowania” `/x` została uhonorowana oddzielnym podrozdziałem.

Książka *Wyrażenia regularne* ogólnie omawia wyrażenia regularne i porównuje ich implementacje w różnych językach. Jeffrey Friedl podaje bardzo dobry opis operatorów patrzenia w przód i w tył. Kto chce naprawdę zrozumieć wyrażenie regularne, powinien przeczytać tę książkę.

Simon Cozens wyjaśnia zaawansowane funkcje wyrażeń regularnych w dwóch artykułach opublikowanych przez Perl.com: „Regexp Power" (*http://www.perl.com/pub/a/2003/06/06/regexps.html*) oraz „Power Regexps, Part II" (*http://www.perl.com/pub/a/2003/07/01/regexps.html*).

W witrynie *http://www.regular-expressions.info* można znaleźć dobre omówienie wyrażeń regularnych i ich implementacji w różnych językach.