IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG
TWÓJ KOSZYK

DODAJ DO KOSZYKA
CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK
CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Wyrażenia regularne

Autor: Jeffrey E. F. Friedl
Tłumaczenie: Adam Podstawczyński
ISBN: 83-7197-351-9
Tytuł oryginału: Mastering Regular Expressions
Format: B5, stron: 320

Wyrażenia regularne to niezwykle skuteczny mechanizm przetwarzania tekstów i innych danych. Ci, którzy do tej pory nie zetknęli się z tym pojęciem, odkryją dzięki tej książce nowe, potężne narzędzia, pozwalające w pełni zapanować nad danymi. Prezentowana tu wiedza jest tak szczegółowa i obszerna, że nawet komputerowi weterani znajdą coś nowego dla siebie.

Umiejętne stosowanie wyrażeń regularnych pozwala radykalnie uprościć przetwarzanie wszelkiego rodzaju informacji, poczynając od poczty elektronicznej, poprzez pliki dzienników aż do dokumentów tekstowych. Mechanizm ten odgrywa niezwykle ważną rolę w programowaniu skryptów CGI, często przetwarzających rozmaite dane tekstowe.

Wyrażenia regularne nie funkcjonują samodzielnie. Oprócz doskonale wszystkim znanego programu grep, wchodzą one w skład takich narzędzi programisty, jak:

- translatory języków skryptowych (m.in. Perl, Tcl, awk i Python),
- edytory tekstów (Emacs, vi, Nisus Writer i inne),
- środowiska programowania (m.in. Delphi i Visual C++)
- inne wyspecjalizowane narzędzia (np. lex, Expert czy sed).

Korzystanie z wyrażeń regularnych wymaga nie tylko wiedzy teoretycznej, ale również znajomości pewnych niuansów. Jeffrey Friedl konsekwentnie prowadzi nas przez kolejne etapy tworzenia konstrukcji, które dokładnie zrealizują wszystkie postawione przed nimi zadania.

Wyrażenia regularne nie istnieją oczywiście same dla siebie. Na stronach książki przedstawiono liczne przykłady wykorzystujących je narzędzi, a także wiele praktycznych przykładów. Szczególnie dużo uwagi poświęcono językowi Perl, wyposażonemu w bogaty zestaw funkcji przeznaczonych specjalnie do obsługi wyrażeń regularnych.

Zawarte w tej książce porady pozwolą Czytelnikom uniknąć wszelkich pułapek i skutecznie wykorzystać możliwości wyrażeń regularnych.

„Książka była dla mnie tyleż przyjemna, co pouczająca, nawet w kwestiach związanych z Perlem”
Tom Christiansen, współautor książki Perl. Programowanie

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przedmowa *9*

1. Wprowadzenie do wyrażeń regularnych *17*

Rozwiązywanie prawdziwych problemów 18
Wyrażenia regularne jako język 19
    Analogia do nazw plików 19
    Analogia do języka 20
Myślenie wyrażeniami regularnymi 21
    Przeszukiwanie plików tekstowych — egrep 22
Metaznaki programu egrep 23
    Początek i koniec wiersza 23
    Klasy znaków 23
    Kropka, czyli dowolny znak 26
    Alternacja 27
    Granice słów 28
    W skrócie 29
    Elementy opcjonalne 30
    Inne kwantyfikatory — powtarzanie 31
    Ignorowanie wielkości znaków 33
    Nawiasy i odwołania wsteczne 33
    Szybki unik 35
Ponad podstawy 35
    Różnorodność językowa 35
    Cel tworzenia wyrażenia regularnego 35
    Jeszcze kilka przykładów 36
    Terminologia wyrażeń regularnych 38
    Podwyższanie kwalifikacji 40
    Podsumowanie 42
Uwagi osobiste 43

2. Przykłady wyrażeń regularnych 45

O przykładach 45
Perl — krótkie wprowadzenie 46
Wyszukiwanie tekstu za pomocą wyrażeń regularnych 48
Przykład bliższy rzeczywistości 49
Skutki uboczne udanego dopasowania 50
Przeplatanie wyrażeń regularnych 53
Chwila odpoczynku 56
Modyfikowanie znalezionego tekstu 57
Edycja zautomatyzowana 60
Proste przetwarzanie wiadomości e-mailowych 61
I znów powtarzające się słowa 66

3. Przegląd funkcji i odmian wyrażeń regularnych 71

Spacerkiem po krainie wyrażeń regularnych 72
Świat według grepa 72
Czas wszystko zmienia 73
Najkrótszy przegląd 74
POSIX 76
„Otoczka" wyrażeń regularnych 77
Identyfikacja wyrażenia regularnego 78
Operacje na dopasowanym tekście 78
Inne przykłady 79
„Otoczka" wyrażeń regularnych — podsumowanie 81
Silniki a błyszczący lakier 82
Lakier 82
Silnik i kierowca 82
Typowe metaznaki 82
Skróty znakowe 83
Łańcuchy jako wyrażenia regularne 86
Skrótowy zapis klas, kropka i klasy znaków 88
Zakotwiczanie 92
Grupowanie i wydobywanie informacji 94
Kwantyfikatory 94
Alternacja 95
Przewodnik po dalszych rozdziałach 95
Informacje specyficzne dla konkretnego narzędzia 96

4. Mechanika przetwarzania wyrażeń 97

Przekręcamy klucz w stacyjce 97
Dwa typy silników 97
Nowe standardy 98
Typy mechanizmów wyrażeń regularnych 98
Kilka dodatkowych pytań 100
Podstawy dopasowywania 100
O przykładach 100
Zasada 1. Najwcześniejsze dopasowanie wygrywa 101

Skrzynia biegów ... 102
Części silnika ... 102
Zasada 2. Niektóre metaznaki są „zachłanne” ... 103
Sterowanie wyrażeniem a sterowanie tekstem ... 108
Mechanizm NFA — sterowanie wyrażeniem ... 108
Mechanizm DFA — sterowanie tekstem ... 109
Wyjaśnienie „zagadki istnienia” ... 110
Wycofywanie ... 111
„Krucha” analogia ... 111
Dwie istotne sprawy dotyczące wycofywania ... 112
Zachowane stany ... 113
Wycofywanie a zachłanność ... 114
Więcej o zachłanności ... 117
Problemy z zachłannością ... 117
„Cudzysłowy” wieloznakowe ... 118
Lenistwo? ... 118
Zachłanność zawsze sprzyja dopasowaniu ... 119
Czy alternacja jest zachłanna? ... 120
Sposoby wykorzystania alternacji niezachłannej ... 121
Alternatywa zachłanna z perspektywy ... 123
Klasy znaków a alternacja ... 123
NFA, DFA i POSIX ... 123
„Najdłuższe najbardziej z lewej” ... 123
POSIX a zasada „najdłuższe najbardziej z lewej” ... 125
Szybkość i wydajność ... 126
Porównanie mechanizmów DFA i NFA ... 126
Techniki tworzenia wyrażeń regularnych ... 128
O czym należy pamiętać ... 129
Warto być konkretnym ... 130
Trudności i niemożliwości ... 133
Uwaga na niepożądane dopasowania ... 134
Dopasowanie ograniczonego tekstu ... 136
Wiedza o przetwarzanych danych ... 139
Dalsze przykłady zachłanności ... 139
Podsumowanie ... 142
Podsumowanie mechaniki dopasowania ... 142
Praktyczne efekty działania mechanizmów dopasowania wzorca ... 143

5. Dopracowywanie wyrażeń regularnych ... 145

Przykład otrzeźwiający ... 146
Prosta zmiana — najlepszą chorągiew posyłamy przodem ... 146
Krok dalej: znalezienie źródła zachłanności ... 147
Twarda rzeczywistość ... 149
O wycofywaniu ogólnie ... 151
POSIX NFA — więcej pracy ... 152
Gdy nie ma dopasowania ... 152
Dążenie do precyzji ... 153
Alternacja może kosztować ... 154

Silne prowadzenie ... 155
Wpływ nawiasów okrągłych ... 155
Optymalizacja wewnętrzna ... 159
Rozpoznawanie pierwszego znaku ... 159
Sprawdzanie obecności stałego fragmentu tekstu ... 160
Proste powtarzanie ... 160
Niepotrzebne małe kwantyfikatory ... 162
Rozpoznanie długości ... 162
Rozpoznanie dopasowania ... 162
Rozpoznanie potrzeb ... 162
Zakotwiczenia łańcucha lub wiersza ... 162
Buforowanie postaci skompilowanej ... 163
Identyfikacja mechanizmu ... 165
NFA czy DFA? ... 165
NFA tradycyjny czy posiksowy? ... 166
Rozwijanie pętli ... 166
Metoda pierwsza: tworzenie wyrażenia na podstawie doświadczenia ... 167
Faktyczny wzór na „rozwinięcie pętli” ... 168
Metoda druga: pogląd z góry ... 171
Metoda trzecia: nazwa hosta internetowego w cudzysłowach ... 171
Obserwacje ... 172
Rozwijanie komentarzy w C ... 173
Wyrażeniowy zawrót głowy ... 173
Sposób naiwny ... 173
Rozwijanie pętli w języku C ... 176
Swobodne wyrażenie ... 177
Dopasowanie wspomagane ... 177
Czy można jeszcze szybciej? ... 178
Opakowanie ... 180
Nic nie zastąpi myślenia ... 181
Różne oblicza optymalizacji ... 181

6. O konkretnych narzędziach ... 185

Pytania, które powinny się pojawić ... 185
Pozornie prosty grep… ... 185
W tym rozdziale ... 187
Język awk ... 187
Różnice pomiędzy odmianami wyrażeń regularnych w awku ... 188
Funkcje i operatory wyrażeń regularnych w awku ... 190
Tcl ... 192
Argumenty wyrażeń regularnych ... 192
Korzystanie z wyrażeń regularnych Tcl-a ... 193
Optymalizacja wyrażeń w języku Tcl ... 195
GNU Emacs ... 195
Łańcuchy Emacsa jako wyrażenia regularne ... 196
Emacsowa odmiana wyrażeń regularnych ... 197
Wyniki dopasowania w Emacsie ... 199

Pomiar wydajności w Emacsie 200
Optymalizacja wyrażeń regularnych w Emacsie 201

7. Wyrażenia regularne Perla 203

Metoda Perla 204
Wyrażenia regularne jako składnik języka 205
Największa siła Perla 206
Największa słabość Perla 207
Perl a problem jajka i kury 207
Przykład wprowadzający: analiza tekstu w formacie CSV 208
Wyrażenia regularne a metoda Perla 210
Perl bez tajemnic 211
„Perlizmy" związane z wyrażeniami regularnymi 212
Kontekst wyrażenia regularnego 213
Zasięg dynamiczny a wynik dopasowania 214
Zmienne specjalne po dopasowaniu 219
„Przetwarzanie cudzysłowowe" i interpolacja zmiennych 221
Perlowa odmiana wyrażeń regularnych 226
Kwantyfikatory zachłanne i leniwe 226
Grupowanie 228
Punkty zakotwiczenia łańcucha 233
Zakotwiczenie poprzedniego dopasowania 237
Punkty zakotwiczenia słów 241
Wygodne skróty i inne sposoby notacji 242
Klasy znaków 244
Prawdziwe kłamstwa, czyli modyfikacja z użyciem symbolu \Q i znaków pokrewnych 246
Operator dopasowania 247
Ograniczniki argumentu dopasowania 247
Modyfikatory dopasowania 249
Określanie argumentu docelowego 250
Inne efekty uboczne operatora dopasowania 251
Wartość zwracana przez operator dopasowania 252
Czynniki zewnętrzne wpływające na operator dopasowania 254
Operator podstawiania 255
Argument podstawienia 255
Modyfikator /e 256
Kontekst i wartość zwracana 258
Użycie modyfikatora /g w wyrażeniu, które może dopasować „nic" 258
Operator split 259
Podstawowe działanie operatora split 259
Zaawansowane działanie operatora split 260
Zaawansowany argument dopasowania w split 261
Operator split w kontekście skalarnym 263
Argument dopasowania operatora split a nawiasy przechwytujące 263
Aspekty wydajności 264
„Jest na to wiele sposobów" 265
Kompilacja wyrażenia, modyfikator /o i wydajność 266

Nietowarzyska zmienna $& i spółka ... 271
Spadek wydajności wywołany modyfikatorem /i ... 277
Aspekty wydajności związane z podstawianiem ... 279
Testowanie ... 281
Usuwanie błędów wyrażenia regularnego ... 282
Funkcja study ... 284
Składamy wszystkie klocki ... 286
Usuwanie końcowych i początkowych białych znaków ... 287
Dodawanie przecinków do liczb ... 288
Usuwanie komentarzy z kodu w języku C ... 288
Dopasowanie adresu poczty elektronicznej ... 289
Kilka słów na koniec ... 299
Uwagi na temat Perla4 ... 300

A Informacje dostępne w sieci ... 303

Informacje ogólne ... 303
Rzemiosło wyrażeń regularnych ... 303
O'Reilly & Associates ... 303
Wirtualna biblioteka oprogramowania OAK ... 304
Archiwum GNU ... 304
Yahoo! ... 304
Inne adresy ... 304
Awk ... 304
Pakiety biblioteczne dla języka C ... 304
Klasa wyrażeń regularnych Javy ... 304
Egrep ... 305
Emacs ... 305
Perl ... 305
Python ... 305
Tcl ... 305

B Program do obsługi adresów e-mail ... 307

Spis tabel ... 311

O Autorze ... 313

Skorowidz ... 315

2

Przykłady wyrażeń regularnych

Przypomnijmy sobie problem powtórzonych słów, przedstawiony w poprzednim rozdziale. Wspomniano tam, że pełne rozwiązanie można byłoby napisać w zaledwie kilku linijkach Perla. Przykład znajduje się poniżej:

```
$/ = ".\n";
while (<>) {
    next if !s/\b([a-z]+)((\s|<[^>]+>)+)(\1\b)/\e[7m$1\e[m$2\e[7m$4\e[m/ig;
    s/^([^\e]*\n)+//mg;     # usuwamy nieoznaczone wiersze
    s/^/$ARGV: /mg;         # rozpoczynamy wiersze od nazwy pliku
    print;
}
```

Tak, to już *cały* program.

Zapewne nie jest on jeszcze całkowicie zrozumiały, chodzi tu tylko o ukazanie możliwości, których nie posiada *egrep* i zwiększenie apetytu Czytelnika na prawdziwą siłę wyrażeń regularnych — niemal całe działanie powyższego programu opiera się na trzech takich wyrażeniach:

```
\b([a-z]+)((\s|<[^>]+>)+)(\1\b)
^([^\e]*\n)+
^
```

Na pewno zrozumiałe jest ostatnie z nich, ⌈^⌋. W pozostałych jednak występują elementy nie omawiane w poprzednim rozdziale (choć o symbolu ⌈\b⌋ napisaliśmy krótko na stronie 39, wspominając, że czasem reprezentuje on *granicę słowa* — tę samą rolę pełni on także tutaj). Wynika to z faktu, że odmiana wyrażeń regularnych zastosowana w Perlu różni się od tej z *egrepa*. Różne są niektóre sposoby zapisu, a poza tym Perl udostępnia o wiele bogatszy zestaw metaznaków. Przekonamy się o tym na przykładach zamieszczonych w tym rozdziale.

O przykładach

Perl umożliwia o wiele bardziej zaawansowane użycie wyrażeń regularnych niż *egrep*. Dzięki przykładom w Perlu poznamy dalsze sposoby wykorzystania wyrażeń regularnych i — co ważniejsze — zobaczymy, jak zachowują się w innym kontekście niż w przypadku programu *egrep*.

Przedstawiana tu odmiana wyrażeń regularnych jest podobna do opisanej w poprzednim rozdziale, choć trochę się od niej różni.

Omawiane w tym rozdziale przykładowe problemy, takie jak weryfikacja danych wprowadzanych przez użytkownika czy operacje na nagłówkach poczty elektronicznej, staną się okazją do dalszego zagłębiania się w krainę wyrażeń regularnych. Przyjrzymy się pokrótce Perlowi i przeanalizujemy niektóre procesy myślowe związane z budowaniem wyrażeń regularnych. Z tak wyznaczonej ścieżki będziemy jednak co pewien czas zbaczać i omawiać inne ważne pojęcia. Jako język programowania, Perl nie jest jakimś wyjątkowym zjawiskiem. Równie dobrze można by było wykorzystać dowolny inny zaawansowany język (np. Tcl, Python, czy nawet *elisp* programu GNU Emacs) jednak Perl jest tu najodpowiedniejszy dlatego, że spośród wymienionych języków w nim właśnie wyrażenia regularne są najgłębiej zakorzenione; jest też chyba najszerzej dostępny. Perl posiada również wiele użytecznych, zwięzłych konstrukcji do operowania na danych, dzięki którym sam wykonuje znaczną część „ciężkiej roboty" i pozwala skoncentrować się na właściwych wyrażeniach regularnych. Aby powyższe słowa brzmiały bardziej wiarygodnie, przypomnimy przykład z analizowaniem plików ze strony 18. Wykorzystano tu właśnie Perla, a całe polecenie brzmiało następująco:

```
% perl -0ne 'print "$ARGV\n" if s/ResetSize//ig != s/SetSize//ig' *
```

Być może nie jest ono jeszcze dla Czytelnika zrozumiałe, ale samą zwartością rozwiązania na pewno robi wrażenie.

Trzeba tu jednak pamiętać o uniknięciu pułapek języka, miejmy bowiem na uwadze fakt, że rozdział ten koncentruje się na *wyrażeniach regularnych*. Przypomina to nieco słowa, które pewien profesor informatyki skierował do studentów pierwszego roku: „Teraz będziemy poznawać zagadnienia informatyczne, a do ich zademonstrowania posłużymy się Pascalem" (Pascal to tradycyjny język programowania, pierwotnie przeznaczony do nauczania).[1]

Ponieważ nie wymaga się tu od Czytelnika znajomości Perla, przed omówieniem przykładów zamieszczono wprowadzenie (pewnej podstawowej wiedzy o Perlu wymaga za to rozdział 7, przedstawiający istotne szczegóły tego języka). Nawet osobom, które mają doświadczenie z różnymi językami programowania, Perl może na pierwszy rzut oka wydać się odmienny — ma bardzo uproszczoną i czasem dziwną składnię. Prezentowane przykłady nie są może „złe" pod względem stylu programowania w tym języku, ale nie są też „doskonałe". Dążąc do zrozumiałości przykładów może nie wykorzystaliśmy wszystkiego, co Perl może zaoferować — programy są tu przedstawione raczej w sposób ogólny, niemal jako „pseudokod". Można natomiast tu znaleźć naprawdę ciekawe zastosowanie wyrażeń regularnych.

Perl — krótkie wprowadzenie

Perl to język programowania o ogromnych możliwościach. Został stworzony przez Larry'ego Walla w późnych latach osiemdziesiątych, a przy jego budowie czerpano pomysły z innych języków. Wiele sposobów przetwarzania tekstu i wyrażeń regularnych pochodzi z *awka* i *seda*, te natomiast znacznie się różnią od „tradycyjnych" języków, takich jak C czy Pascal. Perl jest dostępny dla wielu systemów, w tym dla DOS-u, Windows, MacOS, OS/2, VMS i Uniksa. Jego możliwości objawiają się szczególnie przy przetwarzaniu tekstu; jest też bardzo często wykorzystywany do tworzenia skryptów CGI na potrzeby serwisów WWW (programy CGI służą do tworzenia i wyświetlania dynamicznych stron WWW). Informacje o tym, jak zdobyć kopię Perla dla swojego

[1] Podziękowania za przykład należą się Williamowi F. Matonowi oraz jego profesorowi.

systemu zamieszczono w dodatku A. Omówienie w tej książce dotyczyć będzie Perla w wersji 5.003, ale prezentowane tu przykłady zostały napisane tak, że będą działały w wersji 4.036 lub późniejszych.[2]

Spójrzmy na prosty przykład:

```
$celsjusz = 30;
$fahrenheit = ($celsjusz * 9 / 5) + 32;   # obliczamy stopnie Fahrenheita
print "$celsjusz C to $fahrenheit F.\n";  # drukujemy obie temperatury
```

Po wykonaniu programu otrzymamy taki wynik:

```
30 C to 86 F.
```

Proste zmienne, takie jak `$fahrenheit` czy `$celsjusz`, zawsze rozpoczynają się znakiem dolara i mogą zawierać liczbę lub dowolną ilość tekstu (w tym przykładzie przypisano im tylko liczby). Komentarze rozpoczynają się znakiem # i kończą wraz z końcem wiersza. Dla osób przyzwyczajonych do tradycyjnych języków programowania, takich jak C czy Pascal, prawdopodobnie najbardziej zaskakujące jest, iż zmienne mogą znajdować się wewnątrz łańcuchów objętych cudzysłowami. W łańcuchu `"$celsjusz C to $fahrenheit F.\n"` pod obie zmienne podstawiane są ich wartości. Tak uzyskany wiersz jest następnie wyświetlany (`\n` reprezentuje znak nowego wiersza).

W Perlu można też używać instrukcji sterujących, podobnie jak w innych popularnych językach:

```
$celsjusz = 20;
while ($celsjusz <= 45)
{
  $fahrenheit = ($celsjusz * 9 / 5) + 32; # obliczamy Fahrenheita
  print "$celsjusz C to $fahrenheit F.\n";
  $celsjusz = $celsjusz + 5;
}
```

Treść zawarta wewnątrz pętli `while` jest wykonywana dopóty, dopóki warunek (w tym przypadku brzmiący: **`$celsjusz <= 45`**) jest prawdą. Jeśli powyższy tekst umieścimy w pliku, np. o nazwie *temperatury*, będziemy mogli uruchomić taki program wprost z wiersza poleceń:

```
% perl -w temperatury
20 C to 68 F.
25 C to 77 F.
30 C to 86 F.
35 C to 95 F.
40 C to 104 F.
45 C to 113 F.
```

Opcja `-w` nie jest konieczna, nie ma też bezpośrednio nic wspólnego z wyrażeniami regularnymi. Informuje po prostu Perla, aby dokładniej sprawdzał wykonywany program i ostrzegał użytkownika za każdym razem, gdy znajdzie jakąś nieprawidłowość (np. niezainicjalizowane zmienne — zmienne nie muszą być w Perlu wcześniej deklarowane). Opcję tę zastosowano tu tylko po to, aby wpoić Czytelnikom nawyk jej używania.

---

[2] Choć wszystkie przykłady w tym rozdziale dają się uruchomić we wcześniejszych wersjach Perla, autor *bardzo* namawia do korzystania z wersji 5.002 lub późniejszej, bardzo natomiast odradza korzystanie z archaicznej wersji 4.036, chyba że naprawdę nie ma innego wyjścia.

Wyszukiwanie tekstu za pomocą wyrażeń regularnych

Wyrażenia regularne są w Perlu wykorzystywane na rozmaite sposoby. Najprostszy z nich polega na sprawdzeniu, czy wyrażenie regularne może zostać dopasowane do tekstu przechowywanego w zmiennej. Poniższy fragment programu sprawdza, czy w zmiennej `$odpowiedz` znajdują się wyłącznie cyfry:

```
if ($odpowiedz =~ m/^[0-9]+$/) {
    print "tylko cyfry\n";
} else {
    print "nie tylko cyfry\n";
}
```

Składnia pierwszego wiersza może wydawać się nieco osobliwa. Wyrażenie regularne to ⌈`^[0-9]+$`⌋, natomiast otaczająca je konstrukcja `m/.../` wskazuje Perlowi, co trzeba z tym wyrażeniem zrobić. Litera `m` oznacza, że Perl ma podjąć próbę *dopasowania wyrażenia regularnego* do zmiennej, natomiast ukośniki wyznaczają granice samego wyrażenia. Symbol `=~` łączy konstrukcję `m/.../` z dopasowywanym łańcuchem znaków — w tym przypadku łańcuch ten jest zawarty w zmiennej `$odpowiedz`.

Nie należy mylić sekwencji `=~` z `=` lub `==`, ponieważ jest to coś zupełnie innego. Operator `==` testuje, czy dwie liczby są takie same (do testowania identyczności *łańcuchów* służy, jak się wkrótce przekonamy, operator `eq`). Operator `=` wykorzystywany jest do przypisywania wartości zmiennej, np. **`$celsjusz = 20`**. Wreszcie zapis `=~` służy do łączenia konstrukcji wyszukującej z przeszukiwanym łańcuchem znaków (w przykładzie konstrukcją wyszukującą jest `m/^[0-9]+$/`, a przeszukiwanym łańcuchem — `$odpowiedz`). Być może wygodniej byłoby czytać zapis `=~` jako „pasuje do” — wtedy zapis:

```
if ($odpowiedz =~ m/^[0-9]+$/) {
```

można by przeczytać jako:

„jeśli tekst w zmiennej `$odpowiedz` pasuje do wyrażenia regularnego ⌈`^[0-9]+$`⌋, wtedy...”

Wynik całego wyrażenia **`$odpowiedz =~ m/^[0-9]+$/`** ma wartość *prawda*, jeśli wyrażenie ⌈`^[0-9]+$`⌋ pasuje do łańcucha `$odpowiedz`; w przeciwnym razie ma wartość *fałsz*. Instrukcja `if` korzysta potem z tej wartości, decydując na jej podstawie, który komunikat wydrukować.

Zauważmy, że test o postaci **`$odpowiedz =~ m/[0-9]+/`** (taki sam jak poprzednio, ale bez daszka na początku i dolara na końcu) zwróciłby wartość *prawda*, jeśli zmienna `$odpowiedz` zawierałaby przynajmniej jedną cyfrę w dowolnym miejscu. Symbole ⌈`^...$`⌋ zapewniają, że zmienna `$odpowiedz` musi zawierać *wyłącznie* cyfry.

Połączmy dwa ostatnie przykłady. Poprosimy użytkownika o wpisanie jakiejś wartości, przypiszemy tę wartość zmiennej, a potem za pomocą wyrażenia regularnego sprawdzimy, czy jest liczbą. Jeśli tak — program wyświetli odpowiednik w stopniach Fahrenheita. Jeśli nie, pojawi się ostrzeżenie.

```
print "Wpisz temperaturę w stopniach Celsjusza:\n";
$celsjusz = <STDIN>;  # wczytujemy jeden wiersz podany przez użytkownika
chop($celsjusz);      # usuwamy znak końca wiersza ze zmiennej $celsjusz
if ($celsjusz =~ m/^[0-9]+$/) {
     $fahrenheit = ($celsjusz * 9 / 5) + 32;  # obliczamy stopnie Fahrenheita
```

```
        print "$celsjusz C = $fahrenheit F\n";
    } else {
        print "Oczekiwano wprowadzenia liczby, zapis \"$celsjusz\" jest
        ↳niezrozumiały.\n";
    }
```

Można zauważyć, że sposób „anulowania" znaków cudzysłowu w ostatniej instrukcji `print` jest podobny do sposobu, w jaki eliminuje się specjalne znaczenie metaznaków w wyrażeniu regularnym. Więcej szczegółów na ten temat zamieszczono kilka stron dalej (☞54), w *części Mała dygresja — Firmament metaznaków*.

Spróbujemy zapisać nasz program w pliku *c2f* i uruchomić go:

```
% perl -w c2f
Wpisz temperaturę w stopniach Celsjusza:
22
22 C = 71.599999999999994316 F
```

Niestety, wygląda na to, że zwykła funkcja `print` nie radzi sobie najlepiej z liczbami zmiennoprzecinkowymi. Aby nie zagłębiać się tutaj we wszystkie detale Perla, zasugerujemy użycie w tym miejscu funkcji `printf` (ang. *print formatted*) — wydruk będzie wyglądać wtedy lepiej (`printf` przypomina funkcję `printf` z języka C oraz funkcje formatujące tekst z języków Pascal, Tcl, *elisp* i Python):

```
printf "%.2f C = %.2f F\n", $celsjusz, $fahrenheit;
```

Nie spowoduje to zmiany wartości zmiennych, a jedynie sposobu ich wyświetlania. Działanie programu powinno teraz przebiegać następująco:

```
% perl -w c2f
Wpisz temperaturê w stopniach Celsjusza:
22
22.00 C = 71.60 F
```

co oczywiście wygląda o wiele lepiej.

Przykład bliższy rzeczywistości

Nasz przykład dobrze byłoby rozszerzyć tak, by można było podawać wartości ujemne i ułamkowe. Część matematyczna programu nie wymaga zmian — liczby całkowite i zmiennoprzecinkowe Perl traktuje tak samo. Trzeba jednak zmienić wyrażenie regularne tak, by nie powodowało wyświetlenia komunikatu o błędzie w przypadku podania liczby ułamkowej lub ujemnej. W tym celu należy na początku dopisać wyrażenie ⌈`-?`⌋, co pozwoli rozpocząć treść zmiennej znakiem minusa. Właściwie można nawet dodać ciąg ⌈`[-+]?`⌋, tak by na początku dopuszczalny był również plus.

Aby dopuścić możliwość wpisania części dziesiętnej, dopiszemy wyrażenie ⌈`(\.[0-9]*)?`⌋. Sekwencja unikowa z kropką pasuje do dosłownego znaku kropki (w krajach anglojęzycznych, a także w Perlu, część ułamkową oddziela się od całkowitej kropką, a nie, jak w Polsce, przecinkiem), a więc wyrażenie ⌈`\.[0-9]*`⌋ odpowiada kropce, po której znajduje się dowolna liczba dowolnych cyfr. Ponieważ wyrażenie ⌈`\.[0-9]*`⌋ jest ujęte w znaki ⌈`(...)?`⌋, będzie ono opcjonalne jako całość (jest to logicznie różne od ⌈`\.?[0-9]*`⌋ — tutaj cyfry zostałyby dopasowane nawet wtedy, gdyby nie dopasowano kropki).

Cały wiersz kontrolujący poprawność wpisu wygląda więc następująco:

```
if ($celsjusz =~ m/^[-+]?[0-9]+(\.[0-9]*)?$/) {
```

Teraz już można wpisywać takie liczby jak 32, -3.723 czy +98.6. Wyrażenie to nie jest jednak idealne: nie pozwala na wpisywanie liczb rozpoczynających się od kropki (np. .357). Oczywiście użytkownik może zawsze podać na początku zero (czyli 0.357), nie jest to więc jakaś ogromna wada. Problem ułamków ma jeszcze inne ciekawe aspekty, o których szczegółowo powiemy w rozdziale 4. (☞135).

Skutki uboczne udanego dopasowania

Rozbudujmy nasz przykład jeszcze bardziej i pozwólmy na wpisywanie wartości albo w stopniach Celsjusza, albo Fahrenheita. Użytkownik będzie do wpisywanej liczby dodawał odpowiednio C lub F. Aby nasze wyrażenie regularne na to pozwoliło, wystarczy dodać — po części dopasowującej liczbę — zapis ⌈[CF]⌋. Ale przecież musimy także zmienić resztę programu tak, by rozpoznawał, w jakiej skali wpisano temperaturę i na jaką należy ją przeliczyć.

Perl, podobnie jak inne języki wykorzystujące wyrażenia regularne, posiada zestaw użytecznych zmiennych pozwalających na odwoływanie się do tekstu dopasowanego wcześniej do wyrażeń regularnych ujętych w nawiasy. W pierwszym rozdziale powiedzieliśmy, że niektóre wersje programu *egrep* rozpoznają metasekwencje ⌈\1⌋, ⌈\2⌋, ⌈\3⌋ itd. umieszczone wewnątrz samego wyrażenia regularnego. Perl także obsługuje te metasekwencje, a oprócz tego pozwala odwoływać się do fragmentów wyrażenia regularnego na zewnątrz niego, już po zakończeniu dopasowywania. Takie odwołania realizowane są za pomocą zmiennych $1, $2, $3 itd. Wygląda to może trochę dziwnie, ale to *są* zmienne, tyle że ich nazwy są liczbami. Perl nadaje im wartości za każdym razem, gdy dopasowanie wyrażenia regularnego się powiedzie. Metaznaku ⌈\1⌋ można użyć wewnątrz wyrażenia regularnego w celu odwołania do tekstu dopasowanego wcześniej podczas tej samej próby dopasowywania. Natomiast zmiennej $1 należy użyć, by odwołać się do dopasowanego tekstu w programie już po zakończeniu udanego dopasowania.

Aby zachować przejrzystość przykładu i ułatwić przyswojenie nowych wiadomości, usuniemy na chwilę z naszego wyrażenia część związaną z ułamkami dziesiętnymi (powrócimy do nich wkrótce). Żeby poznać działanie zmiennej $1, porównajmy poniższe dwa wyrażenia:

```
$celsjusz =~ m/^[-+]?[0-9]+[CF]$/
$celsjusz =~ m/^([-+]?[0-9]+)([CF])$/
```

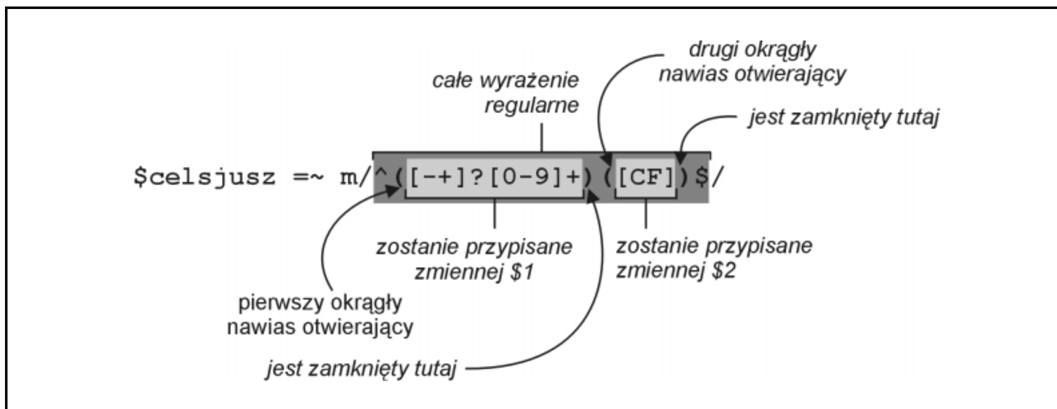
Czy dodanie nawiasów okrągłych zmienia znaczenie wyrażenia? Aby uzyskać odpowiedź na to pytanie, musimy sprawdzić, czy nawiasy te:

- grupują elementy na potrzeby gwiazdki lub innego kwantyfikatora;
- ograniczają wyrażenia rozdzielone znakiem ⌈|⌋.

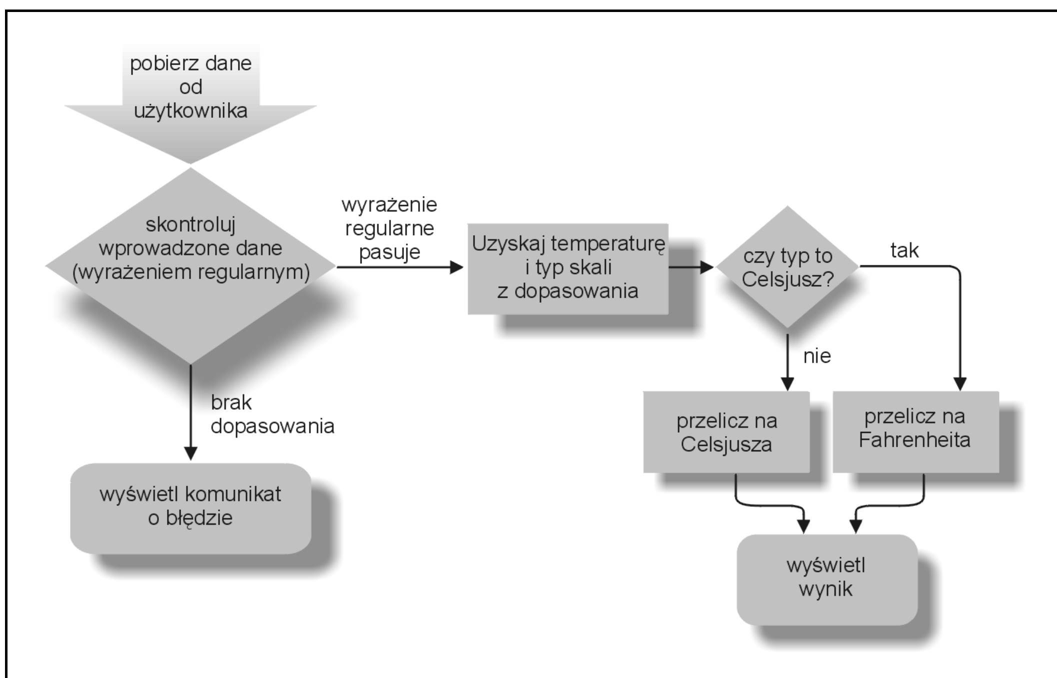
Odpowiedź na powyższe pytania brzmi „nie", a więc obydwa wyrażenia pasują do tego samego ciągu. Jednak nawiasy okrągłe obejmują dwa podwyrażenia — te „interesujące" z naszego punktu widzenia. Jak pokazano na rysunku 2.1, zmienna $1 będzie zawierała wpisaną liczbę, zaś zmienna $2 — albo literę C, albo F. Jeśli teraz spojrzymy na sieć działań przedstawioną na rysunku 2.2, zobaczymy, że znając zawartość tych zmiennych możemy w prosty sposób pokierować działaniem programu po dopasowaniu.

Zakładając, że pokazany dalej program nazwiemy *konwersja*, jego użycie wyglądałoby następująco:

```
% perl -w konwersja
Wpisz temperaturę (np. 32F, 100C):
39F
```



Rysunek 2.1. Nawiasy „przechwytujące”



Rysunek 2.2. Algorytm działania programu konwertującego temperatury

```
3.89 C = 39.00 F
% perl -w konwersja
Wpisz temperaturę (np. 32F, 100C):
39C
39.00 C = 102.20 F
% perl -w konwersja
Wpisz temperaturę (np. 32F, 100C):
ojejku
Oczekiwano wprowadzenia temperatury, zapis "ojejku" jest niezrozumiały.
print "Wpisz temperaturę (np. 32F, 100C):\n";
$lancuch = <STDIN>;  # wczytujemy jeden wiersz podany przez użytkownika
```

```
chop($lancuch);    # usuwamy znak końca wiersza ze zmiennej $celsjusz
if ($lancuch =~ m/^([-+]?[0-9]+)([CF])$/)
{
                   # jeżeli znajdziemy się w tym miejscu programu,
                   # to znaczy, że dopasowanie się powiodło
                   # $1 zawiera liczbę, a $2 literę "C" lub "F"
    $liczba = $1;  # zachowujemy wartości w nazwanych zmiennych,
    $typ    = $2;  # co uprości czytanie pozostałej części programu

if ($typ eq "C") { # "eq" sprawdza, czy 2 łańcuchy są identyczne
                   # podano temperaturę w Celsjuszach, przeliczamy na F.
        $celsjusz = $liczba;
        $fahrenheit = ($celsjusz * 9 / 5) + 32;
    } else {
                   # a więc jednak podano Fahrenheita -- przeliczamy na C.
        $fahrenheit = $liczba;
        $celsjusz = ($fahrenheit - 32) * 5 / 9;
    }
                   # w tej chwili znamy już obie temperatury, więc
                   # wyświetlamy wynik
    printf "%.2f C = %.2f F\n", $celsjusz, $fahrenheit;
} else {
     print "Oczekiwano wprowadzenia temperatury, zapis \"$lancuch\" jest
     ↳niezrozumiały.\n";
}
```

Dopasowania zanegowane

Nasz program ma następującą strukturę logiczną:

```
if ( test logiczny ) {
  … DŁUGIE PRZETWARZANIE jeżli zwrócono wynik prawda …
} else {
  … tylko trochę przetwarzania, jeżli zwrócono wynik falsz …
}
```

Każdy student programowania strukturalnego wie (lub powinien wiedzieć), że kiedy jedna gałąź konstrukcji `if` jest krótka, a druga długa, to — o ile to tylko jest praktycznie wykonalne — lepiej umieścić tę krótką na początku. Dzięki temu `else` jest bliżej `if`, co ułatwia przetwarzanie kodu.

Aby zrobić tak z naszym programem, musimy odwrócić sens testu. Krótsza część to ta, która mówi „jeśli nie pasuje", a więc test powinien zwracać wartość *prawda* wtedy, gdy dopasowanie się nie powiedzie. Można to, zrobić zamieniając zapis =~ na !~, tak jak to pokazano poniżej:

```
$lancuch !~ m/^([-+]?[0-9]+([CF])$/
```

Wyrażenie regularne i badany łańcuch pozostają bez zmian. Jedyna różnica polega na tym, że wynik całej kombinacji przyjmuje teraz wartość *fałsz*, jeśli wyrażenie regularne *pasuje* do łańcucha oraz wartość *prawda* w przeciwnym przypadku. Jeśli nastąpi dopasowanie, zmiennym `$1`, `$2` itd. zostaną tak samo nadane wartości. Dlatego ta część naszego programu wyglądałaby teraz następująco:

```
if ($lancuch !~ m/^([-+]?[0-9]+)([CF])$/) {
   print "Oczekiwano wprowadzenia temperatury, zapis \"$lancuch\" jest
   ↳niezrozumiały.\n";
} else {
    # jeżeli znajdziemy się w tym miejscu programu,
    # to znaczy, że dopasowanie się powiodło
    # $1 zawiera liczbę, a $2 literę m "C" lub "F"
    …
}
```

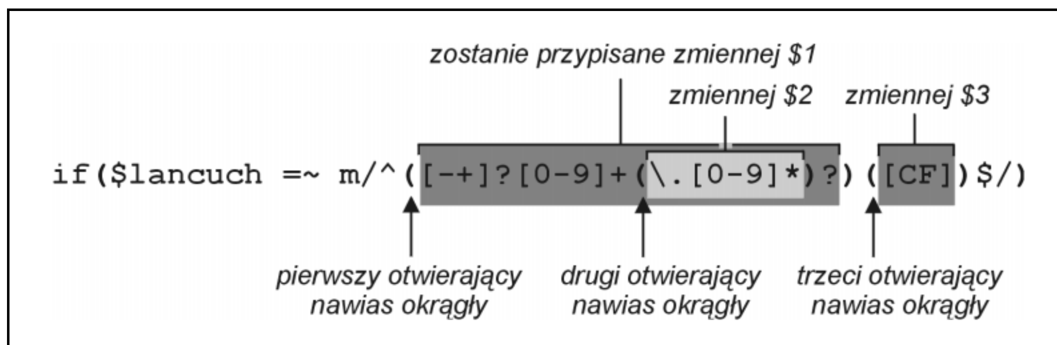
Przeplatanie wyrażeń regularnych

W zaawansowanych językach programowania, takich jak Perl, wyrażenia regularne mogą w dużym stopniu przeplatać się z resztą programu. Aby przedstawić to na przykładzie, wprowadzimy teraz do naszego programu trzy pożyteczne zmiany: umożliwimy wpisywanie liczb ułamkowych, tak jak to robiliśmy wcześniej, zezwolimy na wpisywanie małych liter `f` i `c` oraz zezwolimy na występowanie spacji pomiędzy liczbą a literą. Po takich zmianach użytkownik będzie mógł wpisać np. `98.6•f`.

Jak wspomniano wcześniej, obsługę ułamków realizuje się poprzez dodanie wyrażenia ⌈`(\.[0-9]*)?`⌋:

```
if ($lancuch =~ m/^([-+]?[0-9]+(\.[0-9]*)?)([CF])$/)
```

Zauważmy, że nowy fragment dodano *wewnątrz* pierwszej pary nawiasów okrągłych. Ponieważ za pomocą tej pary nawiasów identyfikujemy liczbę do przeliczenia, musimy przechwycić ją wraz z ułamkami. Chociaż dodatkowa para nawiasów okrągłych służy wyłącznie do grupowania elementów na potrzeby znaku zapytania, powoduje ona efekt uboczny polegający na przypisaniu do zmiennej znajdującej się wewnątrz nich wartości. Ponieważ nawias otwierający jest w tym przypadku drugim nawiasem otwierającym od lewej strony, wartość przypisana zostaje do zmiennej `$2`. Dobrze obrazuje to rysunek 2.3.



Rysunek 2.3. Nawiasy zagnieżdżone

Widzimy, w jaki sposób nawiasy otwierające i zamykające są zagnieżdżone jedne w drugich. Dodanie pary nawiasów przed wyrażeniem ⌈`[CF]`⌋ nie powoduje bezpośrednio zmiany jego znaczenia. Jest jednak wpływ pośredni: teraz nawiasy otaczające to wyrażenie stanowią już trzecią parę nawiasów, a to znaczy, że zmiennej `$typ` musimy przypisać wartość zmiennej `$3`, a nie `$2`.

Umożliwienie wpisywania spacji pomiędzy liczbą a literą nie jest aż takie skomplikowane. Wiemy, że „czysta" spacja w wyrażeniu regularnym odpowiada dokładnie jednej spacji w dopasowywanym tekście, a więc ciąg ⌈`•*`⌋ pasuje do dowolnej liczby spacji (może też nie być ich wcale):

```
if ($lancuch =~ m/^([-+]?[0-9]+(\.[0-9]*)?) *([CF])$/)
```

Już w ten sposób zapewnia się pewną elastyczność dopasowania. Ponieważ jednak chcemy stworzyć coś, co ma być rzeczywiście użyteczne w praktyce, spróbujmy stworzyć wyrażenie regularne pozwalające na wprowadzanie także innych białych znaków, na przykład często spotykanych zna-

ków tabulacji. Oczywiście wyrażenie ⌈⇒*⌋ uniemożliwi użycie zwykłych spacji, musimy więc stworzyć klasę znaków zawierającą i znaki tabulacji, i spacje: ⌈[•⇒]*⌋. A teraz krótka kartkówka: czym różni się to wyrażenie od ⌈(•*|⇒*)⌋? ❖ Odpowiedź znajduje się na stronie 56.

W tej książce spacje i znaki tabulacji łatwo odróżnić, zostały bowiem zaznaczone specjalnymi symbolami • oraz ⇒. Niestety, na ekranie nie jest już tak łatwo. Jeśli widzimy zapis w rodzaju `[  ]*`, możemy zgadywać, że pewnie jest to spacja i znak tabulacji, ale nie upewnimy się, dopóki tego nie sprawdzimy. Perl ułatwia nam nieco życie, udostępniając metaznak ⌈\t⌋. Pasuje on po prostu do znaku tabulacji, a jego jedyną zaletą w stosunku do „prawdziwego" znaku jest to, że lepiej go widać (i dlatego będziemy z niego korzystali w wyrażeniach). Zatem zamieniamy zapis ⌈[•⇒]*⌋ na ⌈[•\t]*⌋

A oto kilka innych wygodnych metaznaków: ⌈\n⌋ (*newline* — znak nowego wiersza), ⌈\f⌋ (*form feed* — znak wysuwu strony) oraz ⌈\b⌋ (*backspace* — znak cofania). Ale chwileczkę — wcześniej przypisaliśmy ⌈\b⌋ do granicy słowa. Cóż więc oznacza ten znak? Otóż oznacza on i to, i to!

Mała dygresja — firmament metaznaków

We wcześniejszych przykładach występowała sekwencja `\n`, jednak nie znajdowała się ona w wyrażeniu regularnym, tylko w łańcuchu znaków. *Łańcuchy* Perla wykorzystują własne metaznaki, w ogóle niezwiązane z metaznakami *wyrażeń regularnych*. Nowi programiści często mylą te dwa obszary.

Jak się jednak okazuje, niektóre metaznaki łańcuchów mają — dla ułatwienia — podobne znaczenia, jak te same metaznaki wyrażeń regularnych. Na przykład metaznak `\t` służy do wstawiania znaku tabulacji do łańcucha, a podobny metaznak ⌈\t⌋ — odpowiada znakowi tabulacji wewnątrz wyrażenia regularnego.

Takie podobieństwa są wygodne, trzeba jednak pamiętać, by ich nie mylić. Może wydaje się to mało ważne dla prostego przykładu znaku `\t`, ale, jak zobaczymy przy okazji omawiania innych języków i narzędzi, wiedza o tym, które metaznaki są wykorzystywane w jakiej sytuacji, jest niezwykle istotna.

Konflikty pomiędzy metaznakami nie powinny być już niczym nowym. W rozdziale 1., przy okazji omawiania programu *egrep*, wyrażenie regularne ujmowaliśmy w apostrofy. Cały wiersz poleceń wpisywany jest po znaku zachęty, a interpreter rozpoznaje własne metaznaki. W pojęciu interpretera metaznakiem jest np. spacja — oddziela ona polecenie od argumentów oraz poszczególne argumenty między sobą. W wielu interpreterach apostrofy informują, że wewnątrz nich inne metaznaki mają pozostać niezinterpretowane (w przypadku DOS-u takie znaczenie mają cudzysłowy).

Użycie apostrofów w przypadku interpretera pozwala na zastosowanie w wyrażeniu regularnym znaków spacji. Bez apostrofów znaki te zostałyby przechwycone przez interpreter poleceń, a nie przekazane do programu *egrep*. Wiele interpreterów rozpoznaje także inne metaznaki — `$`, `*`, `?` itd. — a przecież te z dużym prawdopodobieństwem mogą pojawić się w wyrażeniu regularnym.

Cała ta opowieść o metaznakach interpretera i metaznakach w łańcuchach Perla nie ma nic wspólnego z samymi wyrażeniami regularnymi, jest natomiast silnie związana ze sposobem *korzystania* z wyrażeń regularnych w praktyce. W wielu miejscach książki natkniemy się na złożone nieraz problemy, w których będzie trzeba wykorzystać możliwość „nakładania się" różnych poziomów interpretacji metaznaków.

Wróćmy jednak do podwójnego znaczenia metaznaku ⌈\b⌋. Tym razem sprawa jak najbardziej dotyczy wyrażeń regularnych. W Perlu sekwencja taka oznacza normalnie granicę słowa (ang. *boundary*), ale wewnątrz klasy znaków służy do dopasowania znaku cofania (*backspace*). Wstawianie granicy słowa wewnątrz klasy znaków nie miałoby sensu, a więc w takich miejscach Perl może go interpretować inaczej. Zamieszczone w pierwszym rozdziale ostrzeżenie, że „podjęzyk” klasy znaków jest inny od właściwego języka wyrażeń regularnych, jest na pewno aktualne w przypadku Perla (i wszystkich innych odmian wyrażeń regularnych).

Białe znaki hurtowo: \s

Przy omawianiu białych znaków poprzestaliśmy na wyrażeniu ⌈[•\t]*⌋. Takie rozwiązanie działa, ale mechanizm wyrażeń regularnych Perla pozwala zapisać je znacznie prościej. Podobnie jak metaznak ⌈\t⌋ reprezentuje znak tabulacji, ⌈\s⌋ to skrótowy zapis oznaczający całą klasę znaków zawierającą dowolny „biały znak”, czyli między innymi spację, znak tabulacji, nowego wiersza i powrotu karetki (ang. *carriage return*). W naszym przykładzie znaki nowego wiersza i powrotu karetki nie są i tak do niczego potrzebne, ale zapis ⌈\s*⌋ jest prostszy niż ⌈[•\t]*⌋. Wkrótce przyzwyczaimy się do widoku tego symbolu; jego znaczenie łatwo można zrozumieć nawet w złożonych wyrażeniach regularnych.

Nasz test przybierze teraz postać:

```
$lancuch =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/
```

Ostatnim udogodnieniem miało być umożliwienie wprowadzania zarówno wielkich, jak i małych liter. Oczywiście, najprostszym sposobem byłoby dodanie małych liter do klasy znaków: ⌈[CFcf]⌋, jednak przedstawimy tu jeszcze inne rozwiązanie:

```
$lancuch =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i
```

Znak `i` jest tutaj *modyfikatorem*, a umieszczenie go po konstrukcji `m/…/` jest dla Perla wskazówką, że przy dopasowywaniu nie należy brać pod uwagę wielkości liter. Litera `i` nie jest częścią wyrażenia regularnego, lecz konstrukcji składniowej `m/…/`, która mówi Perlowi, co należy zrobić z podanym wyrażeniem regularnym. Ponieważ nieco kłopotliwe byłoby ciągłe nazywanie wspomnianego znaku „modyfikatorem `i`”, zazwyczaj wykorzystuje się zapis `/i` (choć w praktyce dodatkowy ukośnik przed literą jest zwykle pomijany). W tym rozdziale poznamy jeszcze modyfikator `/g`; na inne przyjdzie czas w dalszych częściach książki.

Wypróbujmy teraz nasz program:

```
% perl -w konwersja
Wpisz temperaturę (np. 32F, 100C):
32 f
10.0 C = 32.00 F
% perl -w konwersja
Wpisz temperaturę (np. 32F, 100C):
50 c
10.00 C = 50.00 F
```

Coś się nie zgadza. Kiedy za drugim razem wpisano 50° Celsjusza, zostały one zinterpretowane jako 50° Fahrenheita. Dlaczego? Spójrzmy jeszcze raz na odpowiednią część programu:

```
if ($lancuch =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i)
{
        …
        $typ    = $3;    # zachowujemy wartości w nazwanych zmiennych,
```

Jaka jest różnica pomiędzy wyrażeniem ⌈[•⇒]*⌋ a ⌈(•*|⇒*)⌋?

❖ Odpowiedź na pytanie ze strony 54

Wyrażenie ⌈(•*|⇒*)⌋ odpowiada albo ⌈•*⌋ albo ⌈⇒*⌋, czyli pasuje do kilku spacji (lub żadnej), albo kilku znaków tabulacji (lub żadnego). Nie umożliwia ono jednak dopasowania *mieszaniny* spacji i znaków tabulacji.

⌈[•⇒]*⌋ pasuje do wyrażenia ⌈[•⇒]⌋ powtórzonego dowolną liczbę razy. W łańcuchu „⇒••" takie wyrażenie dopasowywane jest trzy razy, najpierw do znaku tabulacji, a potem dwóch spacji.

⌈[•⇒]*⌋ to logicznie to samo co ⌈(•|⇒)*⌋, jednak użycie klasy znaków jest często o wiele wydajniejsze (z powodów przedstawionych w rozdziale 4).

```
                     # co uprości czytanie pozostałej części programu
if ($typ eq "C") {   # "eq" sprawdza, czy 2 łańcuchy są identyczne
    …
} else {
    …
```

Co prawda zmodyfikowane wyrażenie regularne zezwala na wpisywanie małego `f`, ale nie wzięliśmy tego pod uwagę w pozostałej części programu. Jeśli zmienna `$typ` nie równa się dokładnie „`C`", program zakłada, że użytkownik wpisał temperaturę w stopniach Fahrenheita. Ponieważ temperaturę w stopniach Celsjusza możemy teraz oznaczać także małą literą `c`, musimy dokonać odpowiedniej zmiany[3]:

```
if ($typ eq "C" or $typ eq "c") {
```

Teraz wszystko działa już tak, jak należy. Powyższe przykłady pokazują, w jaki sposób wyrażenia regularne przeplatają się z resztą programu.

Warto zauważyć, że sprawdzanie obecności litery `c` można też wykonać za pomocą wyrażenia regularnego. Jak? ❖ Odpowiedź zamieszczono na stronie 58.

Chwila odpoczynku

Choć większość tego rozdziału poświęcono poznawaniu Perla, wspomnieliśmy też o kilku sprawach związanych z wyrażeniami regularnymi. Przypomnijmy:

- Wyrażenia regularne Perla różnią się od tych wbudowanych w program *egrep*; większość narzędzi posiada własne odmiany wyrażeń regularnych. Perl kwalifikuje się do tej samej ogólnej grupy co *egrep*, ale posiada bogatszy zestaw metaznaków.
- Perl potrafi dopasować wyrażenie regularne do łańcucha przechowywanego w zmiennej. Służy do tego konstrukcja **`$zmienna =~ m/…/`**. Litera `m` wskazuje, że oczekujemy *dopasowania* (ang. *match*), zaś ukośniki ograniczają (ale nie są częścią) wyrażenia. Test jako całość może przyjmować wartość *prawda* lub *fałsz*.

[3] W starszych wersjach Perla zamiast **`or`** wykorzystywano symbol `||`.

- Pojęcie metaznaków (znaków interpretowanych w sposób specjalny) nie jest domeną samych wyrażeń regularnych. Jak wspomnieliśmy przy omawianiu cudzysłowów w interpreterze poleceń, różne konteksty „walczą" o interpretację swoich metaznaków. Znajomość kontekstów (interpretera, wyrażeń regularnych, łańcucha itd.), ich metaznaków oraz sposobów wzajemnego oddziaływania staje się szczególnie ważna w przypadku obcowania z Perlem, Tcl-em, GNU Emacsem, *awkiem*, Pythonem lub innymi zaawansowanymi językami skryptowymi.
- Oto bardziej przydatne „skróty" dopuszczalne wewnątrz wyrażeń regularnych Perla (o niektórych jeszcze nie mówiliśmy):

  `\t` znak tabulacji

  `\n` znak nowego wiersza

  `\r` znak powrotu karetki

  `\s` klasa dopasowująca dowolny „biały znak" (spację, znak tabulacji, nowego wiersza, wysuwu strony itd.)

  `\S` wszystko inne niż ⌈`\s`⌋

  `\w` ⌈`[a-zA-Z0-9_]`⌋ (przydaje się w wyrażeniu ⌈`\w+`⌋, co dość ogólnikowo pasuje do „słowa")

  `\W` wszystko inne niż ⌈`\w`⌋, a więc ⌈`[^a-zA-Z0-9_]`⌋

  `\d` ⌈`[0-9]`⌋, czyli cyfra

  `\D` wszystko inne niż ⌈`\d`⌋, czyli ⌈`[^0-9]`⌋

- Modyfikator `/i` powoduje, że w teście ignorowana jest wielkość liter. Choć w opisach używa się sekwencji „`/i`", właściwie za zamykającym ogranicznikiem wyrażenia umieszcza się tylko literę `(i)`.
- Po udanym dopasowaniu udostępnione zostają zmienne `$1`, `$2`, `$3` itd., zawierające tekst pasujący do odpowiednich podwyrażeń ujętych w nawiasy okrągłe. Podwyrażenia są numerowane począwszy od pierwszego otwierającego nawiasu okrągłego od lewej i mogą być zagnieżdżane, np. ⌈**(**`(Rzeczpospolita•)?Polska`**)**⌋.

  Nawiasy okrągłe mogą służyć tylko do grupowania, ale (jako działanie uboczne) ich zawartość przypisywana jest jednej ze wspomnianych wyżej zmiennych. Z drugiej strony, często wstawia się je tylko po to, aby przypisać dopasowany przez podwyrażenie tekst do zmiennej. Pozwala to na pobieranie informacji z dopasowanego łańcucha do programu.

Modyfikowanie znalezionego tekstu

Jak do tej pory pisaliśmy jedynie o wyszukiwaniu informacji z łańcuchów znaków. Teraz spróbujemy dokonać *podstawiania*, czy też operacji „znajdź i zamień". Taką możliwość oferuje zarówno Perl, jak i wiele innych narzędzi.

Jak przedstawiono wcześnie, konstrukcja **`$zmienna =~ m/wyrazenie/`** służy do dopasowania wyrażenia regularnego do podanej zmiennej, a w zależności od tego, czy dopasowanie się powiodło, zwraca wartości *prawda* lub *fałsz*.

Sposób sprawdzenia obecności jednej litery w zmiennej

❖ Odpowiedź na pytanie ze strony 56.

Sprawdzenie obecności litery c w zmiennej może wyglądać następująco: **$typ =~ m/^[cC]$/**. Ponieważ wiemy na pewno, że zmienna $typ będzie zawierała ni mniej ni więcej, tylko jedną literę, tutaj akurat możemy pominąć konstukcję ⌈^...$⌋. Co więcej, można nawet zastosować prostszy zapis: **$typ =~ m/c/i** — literka i wykonuje za nas część pracy.

Ponieważ łatwo jest sprawdzić bezpośrednio, czy zmienna zawiera c lub C, stosowanie wyrażenia regularnego do tego celu jest być może przesadą. Gdybyśmy jednak mieli zezwolić na dowolną wielkość liter w całym słowie „Celsjusza", użycie wyrażenia regularnego zamiast sprawdzania bezpośrednio słów „celsjusza", „Celsjusza", „CelSJuSZa" itd. (512 kombinacji!) z pewnością miałoby sens.

W miarę poznawania Perla można znaleźć jeszcze lepsze sposoby rozwiązania tego problemu, np. **lc($typ) eq "celsjusza"**.

Podobna konstrukcja, **$zmienna =~ s/wyrazenie/** *podstawienie***/**, robi więcej: jeśli dopasowanie wyrażenia do łańcucha znaków w zmiennej powiedzie się, dopasowany fragment zostanie zastąpiony podstawieniem. Wyrażenie regularne działa dokładnie tak, jak w konstrukcji m/.../, ale podstawienie (znajdujące się pomiędzy środkowym a końcowym ukośnikiem) traktowane jest jak łańcuch umieszczony w cudzysłowach. Oznacza to, że można w nim odwołać się do zmiennych (w tym również — i jest to bardzo przydatne — do $1, $2 itd., czyli zmiennych reprezentujących dopasowane fragmenty).

Tak więc w konstrukcji **$zmienna =~s/.../.../** wartość zmiennej ulega zmianie (chyba że dopasowanie się nie powiodło — wtedy nie jest dokonywane jakiekolwiek podstawianie i zmienna pozostanie bez zmian). Na przykład, jeśli $zmienna zawiera łańcuch Jeff•Friedl, wykonanie instrukcji:

```
$zmienna =~ s/Jeff/Jeffrey/;
```

spowoduje umieszczenie w zmiennej $zmienna łańcucha Jeffrey•Friedl. Gdyby jednak $zmienna zawierała już wcześniej łańcuch Jeffrey•Friedl, po wykonaniu tego fragmentu programu otrzymalibyśmy Jeffreyrey•Friedl. Może trzeba więc wykorzystać metaznak reprezentujący granicę słowa. Jak wspomniano w rozdziale 1. niektóre wersje *egrepa* obsługują znaki ⌈\<⌋ oraz ⌈\>⌋, oznaczające odpowiednio *początek słowa* i *koniec słowa*. W Perlu natomiast na oba przypadki mamy jeden znak ⌈\b⌋:

```
$zmienna =~ s/\bJeff\b/Jeffrey/;
```

Pamiętając wskazówkę, że w konstrukcji s/.../.../ (podobnie jak w m/.../) możemy używać modyfikatorów takich jak /i, odpowiedzmy na trochę podchwytliwe pytanie: Jak w praktyce zachowa się poniższa instrukcja?

```
$zmienna =~ s/\bJeff\b/Jeffrey/i;
```

Rozwiązanie znajduje się na stronie 60.

Spójrzmy teraz na dość humorystyczny przykład ukazujący sposób wykorzystania zmiennej w tekście podstawianym. Możemy sobie wyobrazić system rozsyłający listy, który jako podstawę wykorzystuje następujący tekst:

```
Szanowny Panie <NAZWISKO>!
Z przyjemnością informujemy, że został Pan wybrany do wzięcia udziału w
konkursie o <BUBEL>! Całkowicie za darmo!
Proszę sobie wyobrazić słowa zazdrości: "Tak, to ten <FRAJER>, który wygrał
<BUBEL>!". Na co więc czekać? Wystarczy zadzwoić pod numer...
```

Przetworzenie takiego tekstu pod kątem konkretnego odbiorcy będzie wymagało najpierw przypisania wartości odpowiednim zmiennym:

```
$imie = 'Jaś';
$nazwisko = 'Kowalski';
$wygrana = '100-procentowo oryginalny sygnet ze sztucznym diamentem';
```

Po ustawieniu zmiennych można już „wypełnić formularz":

```
$list =~ s/<NAZWISKO>/$nazwisko/g;
$list =~ s/<FRAJER>/$imie $nazwisko/g;
$list =~ s/<BUBEL>/wspaniały, $wygrana/g;
```

Poszczególne wyrażenia regularne wyszukują proste znaczniki i po ich odnalezieniu zamieniają je na pożądany tekst. W pierwszym przypadku tekst pobierany jest po prostu ze zmiennej (tak, jak gdyby w łańcuchu znaków występowała tylko sama zmienna, np. `"$nazwisko"`). W drugim wierszu znaleziony tekst zamieniany jest na odpowiednik `"$imie $nazwisko"`, a w trzecim na `"wspaniały, $wygrana"`. Gdybyśmy mieli tylko ten jeden list, moglibyśmy pominąć zmienne i bezpośrednio wpisać pożądany tekst. Ale powyższy sposób pozwala na wprowadzenie automatyzacji — np. pobieranie nazwisk z listy.

Nie wspominaliśmy jeszcze o modyfikatorze `/g`, czyli o „dopasowaniu globalnym". Oznacza on, że konstrukcja `s/…/…/` ma po wykonaniu pierwszego podstawienia kontynuować wyszukiwanie (i dokonywać kolejnych podstawień). Jest to konieczne, jeśli chcemy, aby jedna instrukcja wykonała wszystkie możliwe podstawienia, a nie tylko jedno.

Łatwo przewidzieć wynik podstawienia — nieraz znajdujemy takie zabawne liściki w naszych skrzynkach…

> Szanowny Panie Kowalski!
> Z przyjemnością informujemy, że został Pan wybrany do wzięcia udziału w konkursie o wspaniały,
> 100-procentowo oryginalny sygnet ze sztucznym diamentem! Całkowicie za darmo!
> Proszę sobie wyobrazić słowa zazdrości: „Tak, to ten Jaś Kowalski, który wygrał wspanialy, 100-procentowo
> oryginalny sygnet ze sztucznym diamentem!" Na co więc czekać? Wystarczy zadzwonić pod numer…

Spróbujmy teraz rozważyć inny problem, z którym autor zetknął się podczas pisania w Perlu oprogramowania do obsługi cennika i magazynu. Otóż często zdarzało się, że ceny wyprowadzane były w postaci „9.0500000037272". Oczywiście, cena wynosiła 9.05, ale Perl czasem drukuje liczby w taki sposób, w jaki przetwarza je wewnętrznie komputer, chyba że użyje się odpowiedniego formatowania. W przypadku zwykłych cen można po prostu użyć funkcji `printf` i za jej pomocą wyświetlić dokładnie dwie cyfry po przecinku — tak jak w przykładzie z konwersją skali temperatur. Jednak w tym konkretnym przypadku nie można było tej funkcji zastosować, gdyż cena magazynowa, która kończy się np. ułamkiem 1/8, to w zapisie dziesiętnym „.125" — a w takich przypadkach wymagane byłyby trzy cyfry po przecinku, a nie tylko dwie.

Jak właściwie działa instrukcja $zmienna =~ s/\bJeff\b/Jeffrey/i; ?

❖ Odpowiedź na pytanie ze strony 58.

Pytanie mogło być podchwytliwe ze względu na sposób, w jaki zostało sformułowane. Gdyby użyto wyrażenia regularnego ⌈\bJEFF\b⌋ lub ⌈\bjeff\b⌋ czy może ⌈\bjEfF\b⌋, być może łatwiej byłoby się domyślić, o co chodzi. Dzięki modyfikatorowi `/i` słowo „Jeff" zostanie odnalezione bez względu na to, jakiej wielkości liter użyto. Następnie znaleziony tekst zostanie zamieniony na „`Jeff`", napisane dokładnie tak, jak tu widzimy (`/i` nie ma wpływu na tekst podstawiany; w rozdziale 7. powiemy o modyfikatorach, które mają taki wpływ).

Problem sprowadzał się do realizacji następującego polecenia: „Zawsze bierz pierwsze dwie cyfry, a trzecią tylko wtedy, gdy jest różna od zera. Następnie usuń pozostałe cyfry". Tak więc zapis `12.3750000000392` albo już poprawny `12.375` zwracany byłby jako „12.375", natomiast `37.500` byłby skracany do „37.50" — i o to właśnie chodziło.

Jak to zrealizować? Zmienna `$cena` zawiera badany łańcuch, a więc odpowiednia instrukcja miałaby postać: `$cena =~ s/(\.\d\d[1-9]?)\d*/$1/`. Następujące po kropce (w wersji polskiej należałoby użyć zapisu `,` — *przyp. tłum.*) symbole ⌈\d\d⌋ reprezentują dwie cyfry. Następny fragment, ⌈[1-9]?⌋, to dodatkowa cyfra niezerowa, o ile taka tam się znajduje. Wszystko dopasowane do tego miejsca należy *zachować*, więc obejmujemy odpowiednią część nawiasami okrągłymi i tym samym „przechwytujemy" ją do zmiennej `$1`. Następnie zmienną tę można wykorzystać w podstawianym łańcuchu. Gdyby nawiasy okrągłe obejmowały całe wyrażenie regularne, znaleziony tekst zostałby wymieniony na identyczny — byłaby to niezbyt użyteczna operacja. Ale tutaj dopasowujemy także elementy poza nawiasami. Nie mają one odpowiednika w łańcuchu podstawianym, więc w efekcie zostaną po prostu usunięte. W tym przypadku dotyczy to opcjonalnych dodatkowych cyfr — ⌈\d*⌋ — na końcu wyrażenia regularnego.

Zapamiętajmy ten przykład, ponieważ wrócimy do niego w rozdziale 4. kiedy przyjrzymy się temu, co tak naprawdę dzieje się „za kulisami" procesu dopasowywania. Eksperymenty z tym przykładem bardzo nam się wtedy przydadzą.

Edycja zautomatyzowana

Jeszcze jeden przykład zaczerpnięty z życia autora: podczas pracy na komputerze zdalnym, znajdującym się po drugiej stronie Pacyfiku trzeba było dokonać naprawdę niewielkich zmian — zamienić w pliku każde wystąpienie `sysread` na `read`. Niestety połączenie było tak wolne, że odpowiedź na wciśnięcie klawisza Enter trwała około minuty. W takiej sytuacji szaleństwem byłoby uruchamianie pełnoekranowego edytora.

Oto, co wystarczyło zrobić:

```
% perl -p -i -e 's/sysread/read/g' plik
```

Takie polecenie wykonuje program Perla **`s/sysread/read/g`** (to już cały program — opcja `-e` oznacza, że cała treść programu zostanie podana w wierszu poleceń). Mamy tutaj także opcje

`-p` oraz `-i`, a program operuje na podanym jako argument pliku. Krótko mówiąc, połączenie powyższych opcji powoduje, że podstawianie będzie wykonywane dla każdego wiersza, zaś po zakończeniu tej operacji zmiany zostaną zapisane do pliku.

Zauważmy, że nie podano żadnego łańcucha, dla którego ma być wykonywane podstawianie (nie mamy konstrukcji **`$zmienna =~`** ...) — użyte opcje powodują, że przetwarzane będą po kolei wszystkie wiersze. Ponieważ w programie występuje modyfikator `/g`, na pewno zostaną podstawione wszystkie wystąpienia wzorca w dowolnym wierszu.

Jako argument podano tylko jeden plik, ale nic nie stoi na przeszkodzie, by to samo zrobić dla wielu plików — Perl dokona wtedy podstawienia w każdym wierszu każdego pliku. W ten sposób możemy zbiorczo modyfikować całe grupy plików — wszystko jednym poleceniem.

Proste przetwarzanie wiadomości e-mailowych

Spróbujmy stworzyć inne przykładowe narzędzie. Załóżmy, że w pliku znajduje się wiadomość przesłana pocztą elektroniczną, na którą należy odpowiedzieć. Pierwotna wiadomość ma zostać zacytowana, tak by odpowiedź była lepiej widoczna. Usuniemy także niepożądane wiersze z nagłówka wiadomości i przygotujemy nagłówek do odpowiedzi.

Załóżmy, że nasza wiadomość wygląda następująco:

```
From wodnik Mon Jul 03 20:29:40 2000
Received: from wodnik@localhost by szuwarki.pl (6.2.12) id VAA26883
Received: from szuwarki.pl by gateway.net.net (8.6.5/2) id CV3342BK
Received: from gateway.net.net Mon Jul 03 20:31:33 2000
To: adam@english.w3.pl (Adam Podstawczynski)
From: wodnik@szuwarki.pl (Wodnik Szuwarek)
Date:   Mon, 3 Jul 2000 22:20:16 +0200
Message-ID: <001c01bfe52c$199e7500$7d3f19d5@szuwarki.pl>
Subject: odpowiedź na pytanie
MIME-Version: 1.0
Content-Type: text/plain;
    charset="iso-8859-2"
Content-Transfer-Encoding: 8bit
X-Mailer: Glonojad [version 2.4 PL23]
Witam serdecznie.
Bardzo dziękuję za nadesłanie propozycji współpracy.
Niestety, ostatnio niemal w ogóle nie ruszam się z wody,
więc trudno będzie zrealizować to zamierzenie na suchym lądzie.
Niemniej jednak wspólnie z Syrenką zastanowimy się jeszcze nad
całą sprawą.
Pozdrowienia od Kurki Wodnej.
    Wodniś
```

W nagłówku znajdują się pola interesujące — data, temat itp. — ale także te, które nie są nam potrzebne i które można usunąć. Zakładając, że nasz skrypt będzie nazywał się *odpowiedz*, zaś pierwotna wiadomość znajduje się w pliku `wodnik.wej`, szablon odpowiedzi należałoby sporządzić poleceniem:

```
% perl -w odpowiedz wodnik.wej > wodnik.wyj
```

(pamiętajmy, że opcja `-w` powoduje wysyłanie przez Perla dodatkowych ostrzeżeń; ☞ 47).

Chcemy, żeby plik wynikowy `wodnik.wyj` zawierał mniej więcej taki tekst:

```
To: wodnik@szuwarki.pl (Wodnik Szuwarek)
From: Adam Podstawczynski <adam@english.w3.pl>
Subject: Re: odpowiedź na pytanie
```

```
Dnia   Mon, 3 Jul 2000 22:20:16 +0200 Wodnik Szuwarek napisał(a) co następuje:
|> Witam serdecznie.
|> Bardzo dziękuję za nadesłanie propozycji współpracy.
|> Niestety, ostatnio niemal w ogóle nie ruszam się z wody,
|> więc trudno będzie zrealizować to zamierzenie na suchym lądzie.
|> Niemniej jednak wspólnie z Syrenką zastanowimy się jeszcze nad
|> całą sprawą.
|> Pozdrowienia od Kurki Wodnej.
|>
|>     Wodniś
```

Przeanalizujmy nasze zadanie. Żeby wydrukować nowy nagłówek, trzeba znać adres przeznaczenia (w tym wypadku `wodnik@szuwarki.pl`), prawdziwe nazwisko odbiorcy (`Wodnik Szuwarek`), własny adres i nazwisko oraz temat. Oprócz tego, żeby wstawić wiersz wprowadzający cytowaną treść, należy znać datę.

Całe zadanie można rozbić na trzy fazy:

- pobranie informacji z nagłówka wiadomości;
- wydrukowanie nagłówka wiadomości;
- wydrukowanie wiadomości z wierszami poprzedzonymi znakami „|>•".

Trochę się zapędziliśmy. Nie ma co myśleć o przetwarzaniu danych, zanim nie dowiemy się, jak te dane wczytać do programu. Na szczęście w Perlu wczytywanie danych to pestka: służy do tego „magiczny" operator `"<>"`. Ta zabawnie wyglądająca konstrukcja, przypisana do zmiennej `$zmienna`, zwraca kolejne wiersze podane na wejściu. Dane wejściowe pochodzą z plików wymienionych po nazwie skryptu w wierszu poleceń (w naszym przypadku z pliku `wodnik .wej`).

Dwuznakowego operatora <> nie należy mylić z udostępnianym przez interpreter operatorem przeadresowania („`> plik`") ani ze znakami większości i mniejszości, będącymi również operatorami Perla. Znak <> to po prostu trochę nietypowy sposób Perla na wyrażenie czegoś w rodzaju funkcji `getline()`.

Po wczytaniu wszystkich danych wejściowych, operator <> zwraca wartość niezdefiniowaną (która jest interpretowana jako boolowski *fałsz*), tak więc całe przetworzenie pliku można zawrzeć w następującej konstrukcji:

```
while ($wiersz = <>) {
    …tutaj przetwarzamy $wiersz…
}
```

Podobnie będzie wyglądało przetwarzanie tekstu naszej wiadomości, ale z powodu natury zadania musimy potraktować nagłówek specjalnie. Nagłówek to wszystko to, co znajduje się przed pierwszym pustym wierszem; dalej jest już treść wiadomości. Żeby odczytać tylko nagłówek, można wykorzystać następującą konstrukcję:

```
# przetwarzanie nagłówka
while ($wiersz = <>) {
    if ($wiersz =~m/^\s*$/) {
    last;  # kończymy przetwarzanie w tej pętli while, idziemy dalej
    }
   …tutaj przetwarzanie jednego wiersza…
}
…przetwarzanie pozostałej części wiadomości…
…
```

Pustego wiersza kończącego nagłówek poszukujemy wyrażeniem ⌈`^\s*$`⌋. Sprawdza ono, czy przetwarzany łańcuch ma początek (wszystkie mają), po którym może wystąpić dowolna liczba białych znaków (ale nie muszą żadne), a po nich koniec łańcucha.[4] Dlaczego nie można użyć prostrzej konstrukcji **`$wiersz eq ""`** — dowiemy się później. Słowo kluczowe `last` powoduje przerwanie zawierającej je pętli `while` i zatrzymanie przetwarzania nagłówka.

Wewnątrz pętli, po sprawdzeniu obecności pustego wiersza, możemy robić z wierszami nagłówka praktycznie wszystko. Trzeba będzie wydobyć z nich informacje, takie jak temat wiadomości i datę.

Aby wydobyć temat, można wykorzystać popularne rozwiązanie, które nieraz przyda nam się w przyszłości :

```
if ($wiersz =~ m/^Subject: (.*)/) {
    $temat = $1;
}
```

W taki sposób staramy się wyszukać łańcuch rozpoczynający się od `Subject:•`. Kiedy już taką część uda się dopasować, tekst następujący dalej, czyli ⌈`.*`⌋, będzie pasował do reszty wiersza. Ponieważ ta część wyrażenia znajduje się w nawiasach, do treści tematu można się później odwołać za pomocą zmiennej `$1`. W tym przypadku przepisujemy po prostu jej wartość do zmiennej `$temat`. Oczywiście, jeśli wyrażenie nie zostało dopasowane do łańcucha (a tak będzie dla większości wierszy), instrukcja `if` zwróci wartość *fałsz*, a zmiennej `$temat` nie zostanie dla danego wiersza przypisana żadna wartość.

W podobny sposób znajdziemy pola `Date` i `Reply-To`:

```
if ($wiersz =~ m/^Date: (.*)/) {
    $data = $1;
}
if ($wiersz =~ m/^Reply-To: (.*)/) {
    $adres_zwrotny = $1;
}
```

Wiersz `From:` wymaga nieco większego nakładu pracy. Po pierwsze, chcemy uzyskać wiersz rozpoczynający się od `From:`, a nie ten rozpoczynający się od `From•`. Poszukiwany wiersz to:

```
From: wodnik@szuwarki.pl (Wodnik Szuwarek)
```

Zawiera on adres oraz (w nawiasach) pełną nazwę nadawcy, którą chcemy zeń uzyskać.

Dopasowanie adresu można zrealizować za pomocą wyrażenia ⌈`^From:•(\S+)`⌋. Symbol ⌈`\S`⌋ pasuje do wszystkiego, co *nie* jest białym znakiem (☞ 57), a więc ⌈`\S+`⌋ pasuje aż do pierwszego białego znaku (czy też do końca dopasowywanego tekstu) — w tym przypadku do adresu nadawcy. Po dopasowaniu wyszukamy tekst znajdujący się wewnątrz nawiasów. Oczywiście trzeba dopasować także same nawiasy, co wykonamy za pomocą sekwencji ⌈`\(...\)`⌋ (znosi to specjalne znaczenie nawiasów okrągłych). Wewnątrz nawiasów ma zostać dopasowane cokolwiek (oprócz

[4] Użyto tu słowa „łańcuch" zamiast „wiersz", ponieważ — choć nie jest to ściśle związane akurat z tym przykładem — Perl potrafi dopasowywać wyrażenie regularne do fragmentu tekstu składającego się z wielu wierszy, a wtedy punkty zakotwiczenia (znaki daszka i dolara) dotyczą początku i końca tego fragmentu jako całości (omówienie ogólne ☞ 92; szczegółowe omówienie odnośnie Perla: ☞ 232). Tak czy inaczej, rozróżnienie to nie ma tu znaczenia, ponieważ — co wynika z natury naszego algorytmu — *wiemy*, że `$wiersz` będzie zawierał nie więcej niż jeden wiersz logiczny.

Ostrożnie z wyrażeniem ⌈.⌋*

Wyrażenie ⌈.*⌋ jest często wykorzystywane do dopasowywania „grupy dowolnych znaków” — kropka to dowolny znak (lub, w niektórych narzędziach, dowolny oprócz znaku nowego wiersza i znaku pustego NULL — dotyczy to zazwyczaj także Perla), a gwiazdka oznacza dowolną liczbę powtórzeń (także zero powtórzeń) ostatniego znaku. Taki zapis jest więc bardzo przydatny.

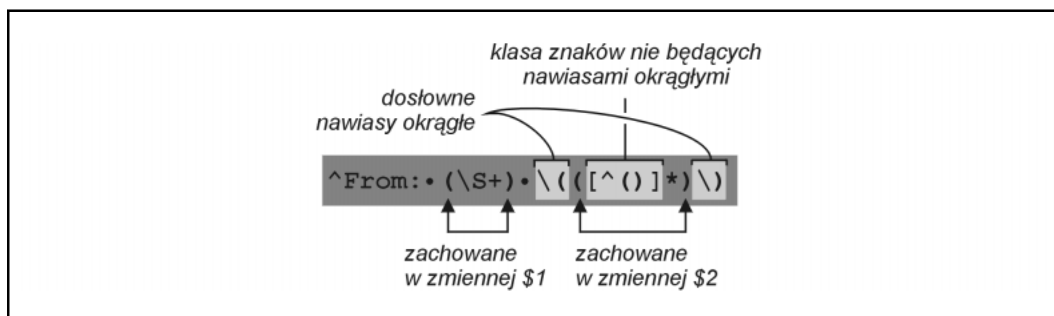
Trzeba jednak uważać na pułapki, które ujawniają się przy oddziaływaniu takiego fragmentu na resztę dużego wyrażenia i które mogą zaskoczyć użytkownika nie znającego właściwego mechanizmu działania wyrażeń regularnych. Szczegółowo omówimy ten temat w rozdziale 4.

dalszych nawiasów!), czyli ⌈[^()]*⌋ (pamiętajmy, że metaznaki klasy znaków różnią się od „normalnych” metaznaków wyrażeń regularnych; wewnątrz klasy znaków nawiasy okrągłe nie mają specjalnego znaczenia i nie trzeba ich ujmować w sekwencje unikowe).

W rezultacie powstało więc następujące wyrażenie:

⌈^From:•(\S+)•\(([^()]*)\)⌋

Wszystkie te nawiasy mogą na początku wprowadzić pewne zamieszanie, przyjrzyjmy się więc rysunkowi 2.4, ukazującemu nasze wyrażenie w bardziej przejrzysty sposób.



Rysunek 2.4. Nawiasy zagnieżdżone

Po dopasowaniu wyrażenia regularnego pokazanego na rysunku 2.4, nazwa nadawcy znajduje się w zmiennej `$2`, a w zmiennej `$1` mamy jego adres zwrotny:

```
if ($wiersz =~ m/^From: (\S+) \(([^()]*)\)/) {
    $adres_zwrotny = $1;
    $nazwa_nadawcy = $2;
}
```

Ponieważ nie wszystkie wiadomości zawierają nagłówek `Reply-To`, zmiennej `$1` użyjemy jako zapasowego adresu zwrotnego. Jeśli w nagłówku znajdzie się wiersz `Reply-To`, zmiennej `$adres_zwrotny` przypiszemy nową wartość. Cała pętla prezentuje się więc następująco:

```
while ($wiersz = <>) {
    if ($wiersz =~ m/^\s*$/ ) { # jeżli wiersz jest pusty
       last;                    # natychmiast kończymy pętlę while
```

```
    }
        if ($wiersz =~ m/^Subject: (.*)/) {
            $temat = $1;
        }
        if ($wiersz =~ m/^Date: (.*)/) {
            $data = $1;
        }
        if ($wiersz =~ m/^Reply-To: (\S+)/) {
            $adres_zwrotny = $1;
        }
        if ($wiersz =~ m/^From: (\S+) \(([^()]*)\)/) {
            $adres_zwrotny = $1;
            $nazwa = $2;
        }
}
```

Kolejne wiersze nagłówka porównywane są z wszystkimi wyrażeniami regularnymi, a jeśli dopasowanie się powiedzie, ustawiana jest określona zmienna. Większość wierszy nagłówka nie będzie pasowała do żadnego wyrażenia; wiersze te zostaną po prostu zignorowane.

Po zakończeniu pętli `while` możemy już wydrukować nasz nagłówek:

```
print "To: $adres_zwrotny ($nazwa)\n";
print "From: Adam Podstawczynski <adam\@english.w3.pl>\n";
print "Subject: Re: $temat\n";
print "\n" ; # wiersz pusty, oddzielający nagłówek od treści
```

Zauważmy, że w temacie wiadomości dodano skrót `Re:` oznaczający, że mamy do czynienia z odpowiedzią. Jeszcze przed wydrukowaniem treści wiadomości możemy dodać:

```
print "Dnia $data $nazwa napisał(a) co następuje:\n";
```

Teraz pozostaje już tylko wydrukowanie pozostałych wierszy wejściowych (treści wiadomości), poprzedzonych znakami |>•:

```
while ($wiersz = <>) {
    print "|> $wiersz";
}
```

Nie trzeba przy tym dopisywać znaków nowego wiersza, ponieważ znajdują się one już w danych wejściowych.

Zamiast instrukcji **`print "|> $wiersz"`** można użyć wyrażenia regularnego:

```
$wiersz =~s/^/|> /;
print $wiersz;
```

W tym wyrażeniu poszukujemy miejsca ⌈^⌋ i (oczywiście) znajdujemy je na samym początku. Nie dopasowujemy zatem żadnego znaku, a pod „znalezione nic" podstawiamy ciąg |>•, co w efekcie wstawia łańcuch |>• na samym początku wiersza. Jest to podręcznikowy przykład użycia wyrażenia regularnego do czegoś, co można zrobić o wiele prościej. Inny przykład poznamy na końcu tego rozdziału (wskazówka: pojawił się on już na początku, ale Czytelnik mógł tego nie zauważyć).

Prawdziwe problemy, prawdziwe rozwiązania

Trudno wskazać praktyczny przykład bez pokazywania wad, jakie mogą się pojawić w praktyce. Po pierwsze, jak już wcześniej wspomniano, celem tych przykładów jest pokazanie działania wyrażeń regularnych — Perl jest tu tylko narzędziem. Opisywany tutaj przykładowy kod nie jest najwydajniejszy ani „najpiękniejszy", ale dobrze pokazuje praktyczne działanie wyrażeń regularnych.

„Prawdziwy" program Perla wykonujący powyżej opisane zadanie mieściłby się pewnie w jakichś dwóch linijkach.[5]

Prawdziwe problemy „z życia wzięte" są jednak o wiele bardziej złożone niż powyższe proste zadanie. Wiersz `From:` może mieć najrozmaitsze formaty, a nasz program obsługuje tylko jeden z nich. Jeśli wzorzec nie zostanie dopasowany dokładnie, zmienna `$nazwa` nigdy nie uzyska wartości i w razie próby użycia zostanie przedstawiona jako niezdefiniowana (czyli jej wartość będzie właściwie „brakiem wartości"). Można by rozbudować wyrażenie regularne tak, by obsługiwało różne formaty adresów i nazw, ale to trochę za dużo jak na ten rozdział (rozwiązania należy szukać pod koniec rozdziału 7.). Póki co, po sprawdzeniu nagłówków (ale przed wydrukowaniem szablonu odpowiedzi) możemy wstawić taki kod:[6]

```
if (    not defined($adres_zwrotny)
     or not defined($nazwa)
     or not defined($temat)
     or not defined($data)  )
{
    die "nie znaleziono wymaganych informacji!";
}
```

Funkcja `defined` Perla informuje, czy zmienna ma wartość, zaś funkcja `die` wyświetla komunikat o błędzie i kończy program.

Inny problem polega na tym, że nasz program zakłada, iż wiersz `From:` pojawi się przed ewentualnym `Reply-To:`. Jeśli będzie odwrotnie, zmienna `$adres_zwrotny` pobrana z wiersza `Reply-To` zostanie zastąpiona wartością wziętą z wiersza `From:`.

„Prawdziwie" prawdziwe problemy

Wiadomości poczty elektronicznej tworzone są przy użyciu różnych programów, z których każdy stosuje własny standard — dlatego ich obsłużenie stanowi często spore wyzwanie. Jak przekonał się autor, próbując napisać program w Pascalu, może to być *niezwykle* trudne bez użycia wyrażeń regularnych. Tak trudne, że prostsze okazało się napisanie pakietu wyrażeń regularnych „w stylu Perla" dla języka Pascal niż pisanie wszystkiego w czystym Pascalu! Potęgę i elastyczność wyrażeń regularnych zwykle przyjmuje się jako coś zwyczajnego, dopóki nie wkroczy się na teren, gdzie w ogóle ich nie ma.

I znów powtarzające się słowa

Problem powtarzających się słów, nakreślony w rozdziale 1., zaostrzył apetyt Czytelnika na poznawanie wyrażeń regularnych. Na początku tego rozdziału mieliśmy okazję obejrzeć garść enigmatycznych symboli i przyjąć na wiarę, że jest to rozwiązanie. Znając już trochę Perla, można zrozumieć przynajmniej ogólną budowę programu, czyli np. konstrukcje <>, `s/…/…/` czy `print`. Wciąż jednak jest to przykład dla nie lada głowy! Jeśli cała nasza przygoda z Perlem sprowadza się do przeczytania tego rozdziału (a cała przygoda z wyrażeniami regularnymi — do przeczytania dotychczasowej części książki), ten przykład może okazać się za trudny.

---

[5] No, to może trochę przesada. Ale jak widać na przykładzie pierwszego programu zamieszczonego w tym rozdziale, Perl pozwala na upakowanie dużej ilości przetwarzania w niewielkiej objętości kodu. Częściowo wynika to z jego dużych możliwości w zakresie wyrażeń regularnych.

[6] W tym fragmencie kodu wykorzystujemy konstrukcje nieobecne w Perlu sprzed wersji 5 (dzięki temu przykład jest czytelniejszy). Użytkownicy starszych wersji muszą zamienić `not` na `!`, a `or` na `||`.

Samo wyrażenie regularne nie jest tak bardzo skomplikowane, jak mogło wydawać się na początku. Zanim ponownie zerkniemy na program, dobrze byłoby spojrzeć na zadania, jakie postawiliśmy przed nim w rozdziale 1., oraz na przykładowy wynik działania:

```
% perl -w FindDbl 01.txt
01.txt: powtórzone słowa (np. "jest jest") — takie błędy często występują w
01.txt: * Znajdzie powtórzone słowa mimo różnic w wielkości liter, np.
↪"Taki taki…" oraz
01.txt: czcionkę słowa, stosuje się zapis: jest <B>bardzo</B>
01.txt: bardzo istotne…".
01.txt: rozdziału. Gdybyśmy wiedzieli, jakie jakie słowo w tekście jest
↪zdublowane (np.
01.txt: musi oznaczać to błąd (jak choćby w wyrażeniu "ho ho..."), ale
```

Teraz przyjrzyjmy się samemu programowi. Tym razem wykorzystamy bardzo praktyczną funkcję nowszych odmian Perla, która pozwala na używanie w wyrażeniach regularnych komentarzy i dowolnej liczby odstępów. Poza tym składniowym wybrykiem, pokazana poniżej wersja programu niczym nie różni się od zamieszczonej na początku rozdziału. Wykorzystano w niej wiele rzeczy do tej pory nie omawianych. Pokrótce wyjaśnimy teraz budowę programu, ale po szczegóły trzeba będzie sięgnąć do podręcznika Perla (albo, jeśli chodzi o wyrażenia regularne, rozdziału 7.). Występujące w poniższym opisie pojęcie „magia" oznacza „funkcję Perla, z którą Czytelnik może jeszcze nie być zaznajomiony".

❶ Ponieważ problem powtarzających się słów wykracza poza ramy jednego wiersza, nie możemy użyć normalnego przetwarzania wiersz po wierszu, tak jak było to w przykładzie z pocztą elektroniczną. Ustawienie specjalnej zmiennej `$/` (tak, to również zmienna) w taki sposób, jak to widać w programie, powoduje, że następujący po niej symbol <> działa w „magicznym" trybie, operując nie na pojedynczych wierszach, ale na czymś przypominającym mniej więcej akapity. Zwrócona wartość będzie jednym łańcuchem, ale zawierającym wiele fragmentów, które normalnie należy uznać za logiczne wiersze.

❷ Zwróćmy uwagę, że wartość pobrana z operatora <> nie została do niczego przypisana. Kiedy symbol <> umieszczony jest w pętli warunkowej `while`, podobnie jak tutaj, łańcuch pobrany z <> jest w „magiczny" sposób przypisywany do pewnej domyślnej zmiennej.[7] Ta sama zmienna przechowuje domyślny łańcuch, na którym operuje konstrukcja `s/…/…/` i który drukuje funkcja `print`. Korzystanie z takich zmiennych domyślnych powoduje, że program jest mniej zagmatwany, ale także trudniejszy w zrozumieniu dla początkującego. Jeśli więc używanie jawnych argumentów jest wygodniejsze, można dalej tak robić.

❸ Litera `s` w tym wierszu pochodzi z konstrukcji podstawiania `s/…/…/`, która tak naprawdę jest o wiele elastyczniejsza, niż przedstawiono do tej pory. Bardzo wygodne jest to, że nie trzeba wcale wykorzystywać ukośników do ograniczania wyrażenia regularnego i podstawienia — można tu użyć innych symboli, np. `s{wyrażenie}"podstawienie"`. Teraz spójrzmy aż na punkt ❹. Wykorzystano tam modyfikator `/x`, który pozwala na korzystanie z komentarzy i dowolnej liczby białych znaków w wyrażeniu (ale nie w łańcuchu podstawianym). Te dwadzieścia chyba linijek wyrażenia regularnego to głównie komentarze — „prawdziwe" wyrażenie regularne jest co do bajta identyczne z pokazanym na początku rozdziału.

Wyrażenie **`next unless`** przed poleceniem podstawiania powoduje, że Perl przerywa przetwarzanie bieżącego łańcucha (by przejść do następnego), jeżeli podstawianie nie

[7] Ta zmienna to `$_`(tak, to również zmienna). Pełni ona rolę domyślnego argumentu wielu funkcji i operatorów.

Problem powtórzonych słów rozwiązany za pomocą nowszej odmiany Perla

```
$/ = ".\n"; ❶ # specjalny tryb operowania na "blokach tekstu"
              # bloki kończą się kropką i znakiem nowego wiersza
while (<>)  ❷
{
    next unless s   ❸
  { # (tu zaczyna się wyrażenie regularne)
        ### chcemy dopasować jedno słowo
        \b         # początek słowa
        ( [a-z]+ ) # przechwytujemy słowo do $1 (i \1)
        ### teraz chcemy znaleźć dowolną liczbę spacji i/lub <ZNACZNIKÓW>
        (          # to co pomiędzy, zachowujemy do $2
          (        # ten nawias ($3) tylko do grupowania alternacji
             \s       # białe znaki (w tym nowy wiersz, o który właśnie
                        ↳chodzi)
             |        # lub
              <[^>]+> # element typu <ZNACZNIK>
          )+       # przynajmniej jedno z powyższych, może więcej
        )
        ### teraz dopasowujemy jeszcze raz to samo słowo
         (\1\b)    # \b zapewnia, że jest to samodzielne słowo; zapisano
         ↳w $4
     # (tu wyrażenie regularne się kończy)
  }
  # powyżej wyrażenie, poniżej podstawienie z modyfikatorami /i, /g i /x
  "e[7m$1\e[m$2\e[7m$4\e[m"igx;   ❹
    s/^([^\e]*\n)+//mg;     ❺ # usuwamy nieoznaczone wiersze
    s/^/$ARGV: /mg;         ❻ # rozpoczynamy wiersze od nazwy pliku
    print;
}
```

spowodowało żadnej zmiany. Nie ma po co przetwarzać dalej łańcucha, w którym nie znaleziono powtórzonych słów.

❹ Podstawianie to zaledwie `"$1$2$4"`, przeplatane jedynie sekwencjami sterującymi ANSI powodującymi wyróżnianie zdublowanych słów (ale nie tego, co pomiędzy nimi). Sekwencja `\e[7m` rozpoczyna wyróżnienie, a `\e[m` je kończy (`\e` to skrót Perla umieszczany w wyrażeniach i łańcuchach i oznaczający początek sekwencji sterujących ANSI).

Jeśli spojrzeć na rozkład nawiasów okrągłych w wyrażeniu regularnym, można zauważyć, że `"$1$2$4"` odpowiada dokładnie temu, co zostało wcześniej dopasowane. A więc poza dodaniem wyróżnienia, całe polecenie podstawiania praktycznie nic nie robi (i w dodatku działa bardzo powoli).

Wiemy, że symbole `$1` i `$4` reprezentują dopasowania tego samego słowa (na tym polega cały program), a więc być może wystarczyłoby wykorzystać w podstawieniu tylko jedną zmienną. Skoro jednak słowa te mogą się różnić wielkością liter, musimy użyć jawnie obu zmiennych.

❺ Kiedy w czasie podstawiania zostaną już wyróżnione wszystkie powtórzone słowa w łańcuchu (także te znajdujące się w różnych wierszach), przychodzi kolej na usunięcie tych wierszy logicznych, które nie zawierają sekwencji sterujących (i pozostawienie tylko tych, które nas

interesują).[8] Znak `/m` wykorzystany w tym i następnym podstawieniu w niemal magiczny sposób traktuje łańcuch docelowy (czyli wspomnianą wcześniej zmienną domyślną `$_`) jako zbiór wierszy logicznych. Zmienia to znaczenie znaku daszka z „początku *łańcucha*" na „początek *wiersza logicznego*"; dzięki temu znaczek ten może zostać dopasowany gdzieś w środku łańcucha, o ile tylko to „gdzieś" jest początkiem wiersza logicznego. Wyrażenie regularne ⌈`^([^\e]*\n)+`⌋ odnajduje sekwencje znaków innych niż sterujące (unikowe), zakończone znakiem nowego wiersza. Podstawienie z wykorzystaniem tego wyrażenia powoduje usunięcie takich sekwencji, w wy wyniku czego pozostają tylko te wiersze logiczne, które zawierają znaki sterujące — czyli tylko te, które zawierają powtórzone słowa.

❻ Zmienna `$ARGV` to „magiczny" sposób na pobranie pliku wejściowego. W połączeniu z metaznakami `/m` i `/g` takie podstawianie powoduje dołączenie nazwy pliku wejściowego na początku wszystkich wierszy logicznych pozostałych w łańcuchu.

Na koniec funkcja `print` drukuje to, co pozostało z łańcucha, wraz ze znakami sterującymi i całą resztą. Pętla `while` powtarza całe przetwarzanie dla wszystkich łańcuchów (fragmentów tekstu stanowiących akapity) odczytanych z wejścia programu.

Perl wcale nie jest wyjątkowy

Jak podkreślaliśmy na początku rozdziału, Perl jest tu wykorzystywany jako narzędzie do zademonstrowania pewnej koncepcji. To bardzo użyteczne narzędzie, ale trzeba podkreślić, że nasz problem można rozwiązać równie prosto w dowolnym języku. W rozdziale 3. zobaczymy podobne rozwiązanie zrealizowane w programie GNU Emacs, natomiast poniższy wydruk pozwala na bezpośrednie porównanie rozwiązania w Pythonie. Nawet ci, którzy widzą Pythona pierwszy raz w życiu, powinni zauważyć różnice w sposobie traktowania wyrażeń regularnych.

Najwięcej różnic widać w sposobie rozmieszczenia poszczególnych fragmentów programu. Perl stworzony został do przetwarzania tekstu i w „magiczny" sposób robi rozmaite rzeczy za nas.

Python jest o wiele bardziej „ogólny" — konstrukcja wszystkich interfejsów jest bardziej spójna. Ale oznacza to, że wiele prozaicznych czynności, takich jak otwieranie plików, trzeba zaprogramować samemu.[9]

Odmiana wyrażeń regularnych zastosowana w Pythonie różni się od odmiany dostępnej w Perlu i *egrepie* w jednym aspekcie: wymaga stosowania niewiarygodnej liczby lewych ukośników. Na przykład, ⌈`(`⌋ nie jest metaznakiem — grupowanie i przechwytywanie zapewnia ciąg ⌈`\(…\)`⌋. Oprócz tego `\e` nie jest skrótowym zapisem znaku unikowego, a więc trzeba wypisywać zwykły kod ASCII, tj. `\033`. Poza jednym detalem dotyczącym znaku ⌈`^`⌋ (o którym powiemy za chwilę) różnice te są tylko powierzchowne; wyrażenia regularne w Pythonie są funkcjonalnie identyczne z tymi, które widzieliśmy w przykładzie w Perlu.[10]

---

[8] Zakładamy tutaj, że w danych wejściowych nie ma znaków unikowych ASCII. Gdyby takowe istniały, program wyprowadzałby nieodpowiednie wiersze.

[9] Właśnie to wielbiciele Perla kochają w Perlu i nienawidzą w Pythonie, a wielbiciele Pythona z kolei kochają w Pythonie i nienawidzą w Perlu.

[10] Trzeba przyznać, że w oczy *może* się rzucać jeszcze jedna różnica. Znak ⌈`\s`⌋ Perla został zamieniony na ⌈`\n\r\t\f\v•`⌋. Wersja perlowa opiera definicję „białego znaku" na bibliotece C, w oparciu o którą Perl został skompilowany. W wyrażeniu pythonowym trzeba jawnie określić, co ma być uznane za „biały znak".

Problem powtarzających się słów rozwiązany w Pythonie

```
import sys; import regex; import regsub
### Przygotowujemy trzy wymagane wyrażenia regularne
wyr1 = regex.compile(
        '\\b\([a-z]+\)\(\([\n\r\t\f\v ]\|<[^>]+>\)+\)\(\\1\\b\)',
        regex.casefold)
wyr2 = regex.compile('^\([^\033]*\n\)+')
wyr3 = regex.compile('^\(.\)')
for filename in sys.argv[1:]:                 # dla każdego pliku...
      try:
          file = open(filename)               # próbuj otworzyć plik
      except IOError, info:
          print '%s: %s' % (filename, info[1])  # informuj, jeżeli nie
                                                ↳ się uda
          continue                            # i rozpocznij pętlę od
                                                ↳nowa
       dane = file.read() # cały plik wczytaj do 'dane', wykonaj wyrażenia
                          # regularne, wydrukuj
       dane = regsub.gsub(wyr1, '\033[7m\\1\033[m\\2\033[7m\\4\033[m',
       dane)
       dane = regsub.gsub(wyr2, '', dane)
       dane = regsub.gsub(reg3, filename + ': \\1', dane)
       print dane,
```

Ciekawe jest też to, że w łańcuchu podstawianym dla metody `gsub` (ang. *global substitution*, działa analogicznie do perlowego `s/…/…/`) wykorzystywany jest ten sam symbol `\\1`, co w samym wyrażeniu regularnym. Dostęp do tej samej informacji poza wyrażeniem i podstawieniem wykonywany jest w Pythonie za pomocą zapisu `regex.group(1)`. Pamiętamy, że w Perlu metaznak ⌈`\1`⌋ wykorzystywany jest wewnątrz wyrażenia regularnego, a zmienna `$1` wszędzie poza nim. Podobne koncepcje, lecz różne rozwiązania.

Ważną i wcale nie powierzchowną różnicą związaną z wyrażeniami regularnymi jest to, że pythonowy zapis ⌈`^`⌋ „postrzega" ostatni znak nowego wiersza jako początek pustego wiersza. Objawia się to obecnością dodatkowego wiersza `filename:`, jeśli w trzecim wyrażeniu użyjemy tego samego znaku ⌈`^`⌋, co w Perlu. Aby zniwelować tę różnicę, zapisaliśmy trzecie wyrażenie tak, aby wyszukiwało jeszcze coś po znaku daszka i wymieniało to „coś" na samo siebie. W efekcie po zakończeniu ostatniego wiersza logicznego wyrażenie nie zostanie dopasowane.