Systemy operacyjne

Laboratorium 9

Perl – wyrażenia regularne

Jarosław Rudy
Politechnika Wrocławska

28 lutego 2017

Temat obejmuje wykorzystanie wyrażeń regularnych w `perl`u. Wyrażenia same w sobie są w zasadzie identyczne do tych wykorzystanych w przypadku `awk`a. W przypadku `perl`a nie powinno być jednak problemów z niektórymi rodzajami regexów. Ponadto, regexy w `perl`u oferują więcej możliwości. W związku z tym poniższa instrukcja obejmuje głównie sposoby stosowania regexów w `perl`u.

1 Rekordy, pola i separatory

W instrukcji do laboratorium 7 omówiono separatory wyjściowe `$\` oraz `$,`, które są tekstem. Separatorem wejściowym jest `$/`, który TAKŻE jest tekstem, a nie regexem. Jest to istotna różnica w stosunku do `awk`a. Jak wspomniano na poprzednim laboratorium, w `perl` rekord wczytywany jest wraz z jego separatorem. Aby "wyciąć" separator z końca rekordu należy użyć funkcji `chomp` – próbuje ona usunąć z końca podanego argumentu to co aktualnie zapisane jest w `$/`.

Perl domyślnie nie dzieli rekordu na pola, można jednak w tym przypadku posłużyć się funkcją `split`, która podzieli tekst zgodnie z podanym regexem. Funkcja `split` zwraca LISTĘ "słów" po podziale (kontekst listy/tablicy) lub LICZBĘ słów (kontekst skalarny).

2 Match

Podstawowe użycie regexów w `perl`u dotyczy operatorów dopasowania (match, m), podstawiania (substitute, s) oraz transliteracji (transliteration, tr, y). Bazowa składnia operatora match jest następująca:

```
$tekst =~ m/regex/opcje
```

Powyższy kod spróbuje znaleźć tekst dany `regex`em wewnątrz zmiennej `$tekst`. Opcje są zawsze pojedynczymi literami (ale może być więcej niż jedna opcja naraz). Przykładowo, opcja `'i'` sprawia, że dopasowanie nie rozróżnia małych i wielkich liter. Ważną opcją jest `'g'` (global), która sprawia, że `match` spróbuje dopasować się wielokrotnie wewnątrz tego samego tekstu. Oprócz zwykłego `=~` dostępna jest także wersja `!~`, która odwraca wynik `matcha`:

```
$tekst !~ m/regex/opcje
```

Przejdziemy teraz do nieco bardziej skomplikowanego aspektu `matcha`, czyli określania jego wyniku. Jeśli chcemy zapisać wynik `matcha` do zmiennej, to oczywiście używamy operatora przypisania, ale potrzebna jest nowa zmienna (np. `$wynik`):

```
$wynik = $tekst =~ m/regex/opcje
```

Podobieństwo operatorów `=` oraz `=~` czyni ten zapis nieco dziwnym. Dodajmy do tego fakt, że dosyć łatwo w perlu wykorzystać zawartość zmiennej jako regex:

```
$regex = "jakis_regex";
$wynik = $tekst =~ m/$regex/
```

Widzimy więc tutaj TRZY zmienne. Nie należy ich ze sobą mylić! Zanim przejdziemy dalej, zauważmy, że możemy użyć `//` zamiast `m//`:

```
$tekst =~ /regex/opcje
```

Jeśli używamy zmiennej jako regexa i nie stosujemy opcji, to możliwe jest nawet prostsze:

```
$tekst =~ $regex
```

Przejdźmy teraz do określenia jaki wynik operator `match` w zasadzie zwraca. Jeśli dopasowanie się NIE udało, to otrzymamy PUSTY TEKST (kontekst skalarny) lub PUSTĄ LISTĘ (kontekst listy). Jeśli dopasowanie się UDAŁO, to w kontekście skalarnym zawsze otrzymamy liczbę `1`. Z kolei wynik UDANEGO `matcha` w kontekście listowym jest zależne od tego czy w naszym regexie zdefiniowaliśmy grupy (podwyrażenia) z użyciem nawiasów okrągłych oraz od tego, czy użyliśmy opcji `'g'`. Możliwe są 4 przypadki[1]:

- Grupy: nie, global: nie. Wtedy `match` zwraca listę zawierającą jeden element, którym jest liczba `1`.
- Grupy: tak, global: nie. Jeśli zdefiniowano K grup, to wtedy `match` zwraca listę (tablicę) K elementów, zaś i-ty element zawiera to co dopasowało się do i-tej grupy. Zauważmy, że rozmiar tej listy jest zawsze $\geqslant 1$.
- Grupy: nie, global: tak. Jeśli udało się dopasować regex N razy, to wtedy `match` zwraca listę N elementów, zaś i-ty element zawiera i-te dopasowanie. Ponownie rozmiar listy jest zawsze $\geqslant 1$.

[1] Przykład można znaleźć na stronie.

- Grupy: tak, global: tak. Połączenie dwóch powyższych trybów. Jeśli zdefiniowaliśmy K grup, a dopasowanie nastąpiło N razy, to otrzymamy listę o liczbie elementów równej $N \cdot K$. Pierwsze K elementów listy zawiera to co dopasowało się do grup w pierwszym dopasowaniu, następne K elementów zawiera grupy z drugiego dopasowania itp.

Przy okazji – jeśli definiujemy regex w zmiennej, to należy zwracać uwagę na odpowiednią liczbę odwrotnych ukośników, gdyż, podobnie jak dla `awk`a, występują tu dwa poziomy interpretacji (poziom łańcucha tekstowego i poziom regexa).

Należy jeszcze zauważyć, że po wykonaniu `match`a zmienne `$1`, `$2` itp. zostaną ustawione na to co dopasowało się do danej grupy (nawiasu). Należy zauważyć, że jeśli użyliśmy opcji `'g'` i, przykładowo, wystąpiły 3 dopasowania, to zmienne typu `$1` będą dotyczyły tylko ostatniego "zestawu" (tzn. ostatnie K elementów w $N \cdot K$-elementowej liście).

Ostatnia uwaga – skrócona wersja `match` (bez `'m'`) używa tylko ukośników, ale dla wersji z `'m'` możliwe są inne "ograniczniki". Np. zamiast:

```
$tekst =~ m/regex/opcje
```

możemy użyć:

```
$tekst =~ m|regex|opcje
```

Jest to szczególnie przydatne, jeśli "lewy" ukośnik często występuje w naszym regexie. Większość znaków "specjalnych" działa w ten sposób. Znaki w stylu nawiasów OTWIERAJĄCYCH muszą być zamykane przez swoje ZAMYKAJĄCE odpowiedniki:

```
$tekst =~ m(regex)opcje
```

a nie:

```
$tekst =~ m(regex(opcje
```

Nawiasy zamykające działają "normalnie":

```
$tekst =~ m)regex)opcje
```

a nie:

```
$tekst =~ m)regex(opcje
```

3 Podstawianie

Operator podstawiania ma następującą składnię:

```
$tekst =~ s/regex/zast/opcje
```

Jest więc podobnie do `match`a, ale występuje `'s'` zamiast `'m'`, nie ma wersji skróconej (bez `'s'`), ale dalej możemy stosować inne ograniczniki:

```
$tekst =~ s|regex|zast|opcje
```

Widzimy też, że podstawianie ma dodatkowy człon `zast`, który określa jaki tekst należy wstawić w zmiennej `$tekst` w miejsce, w którym nastąpiło dopasowanie do `regex`a. Ten dodatkowy człon sprawia, że ograniczniki z nawiasami otwartymi przybierają formę:

```
$tekst =~ s(regex)(zast)opcje
```

Zauważmy też, że `zast` jest od razu traktowany jako tekst (chyba, że np. zaczyna się od dolara, co oznacza wstawienie wartości ze zmiennej), więc nie należy wstawiać naokoło niego cudzysłowów.

Uwagi dotyczące tego w czym podstawiamy (`$tekst`) oraz tego w czym zapisujemy wynik (`$wynik`) są podobne jak dla `match`a. Normalnie podstawiamy tylko pierwsze dopasowanie, chyba, że użyto opcji `'g'` – wtedy wykona się tyle podstawień, ile wykryto dopasowań. Normalnie podstawianie jest "niszczące" tzn. podstawienie zmienia tekst zawarty w `$tekst`. W takim trybie operator zwraca liczbę dokonanych podstawień. Jeśli jednak podanie zostanie opcja `'r'`, to oryginalna zmienna `$tekst` pozostaje nienaruszona, a operator zwraca tekst po podstawieniu. Bez opcji `'r'` operator zachowuje się więc jak `awk`owy `(g)sub`, zaś z tą opcją zachowuje się jak `awk`owy `gensub`.

Zmienne `$1`, `$2` itp. zawierają tekst, który dopasował się do poszczególnych grup. Zmienne te są ustawione po wykononaniu dopasowania, ale PRZED samym podstawieniem, można więc je bezpiecznie użyć w członie `zast`. Tak więc poniższe wywołanie:

```
$tekst =~ s/([0-9]+)/$1.00/g
```

zastąpi w `$tekst` wszystkie liczby całkowite z postaci `X` na `X.00`. Zamiast zmiennych `$N` możemy użyć formy `\N`, ale wtedy jesteśmy ograniczeni do $N \leqslant 9$.

4 Transliteracja

Transliteracja działa podobnie do komendy `tr`. Bazowa składnia jest następująca:

```
$tekst =~ tr/pierwszy/drugi/opcje
```

lub

```
$tekst =~ y/pierwszy/drugi/opcje
```

Operator zamienia znaki ze zbioru `pierwszy` na odpowiednie znaki ze zbioru `drugi`. Aby zamienić w danym tekście cyfry (nie liczby!) na znaki gwiazdki wystarczy użyć:

```
$tekst =~ tr/0123456789/**********/
```

lub

```
$tekst =~ y/0123456789/**********/
```

Jeśli `drugi` zbiór ma za mało znaków, to ostatni znak tego zbioru będzie dotyczył pozostałych znaków ze zbioru `pierwszy`. Oznacza to, że powyższy przykład można równie dobrze napisać jako:

```
$tekst =~ tr/0123456789/*/opcje
```

Podobnie jak w regexach możemy definiować "zasięg" zbioru, co jeszcze bardziej upraszcza ten sam przykład:

```
$tekst =~ tr/0-9/*/opcje
```

Opcja `'r'` działa podobnie jak dla podstawiania – jeśli jej nie podamy, to transliteracja modyfikuje dany tekst i zwraca liczbę zmienionych (lub usuniętych, patrz dalej) znaków. Z tą opcją transliteracja zwraca tekst po zmianie, a oryginał nie jest zmieniany. Opcja `'c'` sprawia, że zbiór `pierwszy` zawiera wszystkie znaki, których w nim NIE podaliśmy, tak więc wywołanie:

```
$tekst =~ tr/0-9/*/c
```

zamieni w gwiazdki wszystko poza cyframi. Opcja `'d'`, sprawia, że znaki ze zbioru `pierwszy`, dla których nie ma dopasowania zostaną usunięte. Tak więc:

```
$tekst =~ tr/0-9/*/d
```

zamieni wszystkie zera w gwiazdki oraz usunie pozostałe cyfry.