Wyrażenia regularne w elektronicznym przetwarzaniu tekstów

Grzegorz Wójcik
Instytut Łączności,
ul. Szachowa 1, 04-894 Warszawa
G.Wojcik@itl.waw.pl

Streszczenie

W artykule opisano podstawy wyrażeń regularnych oraz możliwości ich zastosowania do przetwarzania tekstów. Przedstawione przykłady pokazują jak można ich używać w edytorach *vi* i *sed* oraz przy programowaniu w językach AWK i Perl.

Wprowadzenie

Często przy przygotowywaniu publikacji stykamy się z sytuacją, że niektóre partie materiału dostępne są w postaci elektronicznej. Trzeba jedynie tak dopasować ich strukturę, aby stały się elementem tworzonego przez nas dokumentu. Ponieważ danych w postaci elektronicznej przybywa: są to przede wszystkim bazy danych, ale także wszelkiego rodzaju spisy, listy, czy wreszcie dokumenty tworzone przy użyciu innych niż TEX narzędzi, warto przypomnieć sobie niektóre bardzo już stare narzędzia, które wciąż znakomicie ułatwiają przetwarzanie dużej ilości tekstu.

Narzędzia te wywodzą się z początków wykorzystywanych dziś systemów operacyjnych. Ich protoplastami były edytory liniowe – czyli takie edytory, które pozwalały na raz wprowadzać lub poprawiać tylko jedną linię tekstu. Przykładem tego typu programów jest występujący wciąż w wielu systemach uniksowych (jako ostatnia deska ratunku dla administratora) edytor *ed*. To w tych edytorach pojawiły się po raz pierwszy wyrażenia regularne, które stanowią obecnie podstawę wszystkich uniksowych (i od nich się wywodzących, a działających pod kontrolą innych systemów operacyjnych) programów i języków programowania do przetwarzania tekstów.

Najprostsze operacje na tekście

Najprostsze operacje na tekście, przy użyciu wyrażeń regularnych, można wykonywać korzystając z edytorów takich jak *vi* (ew. jego unowocześnionej i rozbudowanej wersji jaką jest *vim*) lub *sed*. *Vi* jest edytorem ekranowym, a więc każda zmiana jaką wykonamy w tekście jest natychmiast widoczna na ekranie. Natomiast *sed* to tzw. edytor strumieniowy, tj. taki, który odczytuje tekst ze swojego standardowego wejścia, przetwarza go zgodnie z podanymi przez nas regułami, a następnie przetworzony tekst wyrzuca na standardowe wyjście. Oba te edytory bazują na mechanizmach zaczerpniętych z edytora liniowego *ex*[1]. Dlatego w obu komendy związane z wyrażeniami regularnymi są identyczne. Różnią się one tylko tym, że jeśli nie zdefiniujemy zakresu działania komendy, to w przypadku edytora *vi* działa ona na bieżącej linii, natomiast w przypadku *sed*-a na całym pliku.

Jedną z najczęściej wykonywanych operacji jest zastąpienie jednego tekstu innym. Wyobraźmy sobie, że chcemy zastąpić słowo `Ala` słowem `Ola`. W najprostszym przypadku wyglądałoby to tak:

```
s/Ala/Ola/
```

Komenda `s` (od *substitute*) zamienia w tekście wzorzec podany jako pierwszy parametr (tekst pomiędzy pierwszymi ukośnikami), na napis podany jako drugi parametr. Jeśli nie podamy żadnych opcji, to zamianie podlegać będzie tylko pierwsze wystąpienie wzorca w linii (tylko pierwsze słowo `Ala` zostanie zmienione na `Ola`). Żeby zmiana dotyczyła wszystkich wystąpień wzorca w linii, należy posłużyć się opcją `g`:

```
s/Ala/Ola/g
```

Opcje, z których możemy skorzystać to[2]:

Opcja	*Znaczenie*
`g`	zamiana wszystkich wystąpień wzorca w linii
`c`	(w przypadku *vi*) zapytanie o zgodę przed każdą zamianą
`i`	przy dopasowywaniu tekstu do wzorca nie będą rozróżniane wielkie i małe litery
`p`	(w przypadku edytora *sed*) drukowanie wyniku działania komendy `s`

[1] Aby przejść do trybu komend liniowych w edytorze *vi* należy nacisnąć dwukropek(`:`).

[2] Podane opcje zależą od implementacji edytora. W niektórych wersjach programów nie występują wszystkie podane opcje.

Oczywiście zamiana jednego tekstu na drugi nie jest żadną rewelacją – praktycznie każdy edytor tekstu to potrafi. Jednak w tych dwóch edytorach we wzorcu mogą występować wyrażenia, które pozwalają na określenie bardziej ogólnych zasad zamiany. Są to właśnie wyrażenia regularne:

Wyrażenie	*Znaczenie*
`.`	dowolny znak
`*`	zero lub więcej wystąpień wyrażenia
`^`	początek linii
`$`	koniec linii
`\c`	znak specjalny
`\(...\)`	grupowanie (podwyrażenia)
`[abc]`	klasa znaków (litera `a` lub `b` lub `c`)
`[a-z]`	zakres znaków (jedna z liter od `a` do `z`)
`[^...]`	negacja zakresu lub klasy znaków (wszystkie znaki, które nie zostały wymienione w nawiasach kwadratowych)

Warto zapamiętać, że jeśli we wzorcu stosujemy znak `*`, to do wzorca dopasuje się zawsze najdłuższy ciąg znaków spełniających dane wyrażenie, np. wykonanie komendy:

```
s/a*/d/
```

na tekście:

```
aaaaa bbbb ccc
```

da w wyniku:

```
d bbbb ccc
```

Jednocześnie trzeba też zauważyć, że wyrażenia regularne są dopasowywane tak wcześnie jak to tylko jest możliwe. Jeśli więc na wyniku działania poprzedniej komendy, wykonamy polecenie:

```
s/b*/e/
```

to zauważymy (być może z pewnym zaskoczeniem), że otrzymaliśmy następujący wynik:

```
ed bbbb ccc
```

Wyrażenie regularne zostało dopasowane jeszcze zanim rozpoczął się tekst linii – sami przecież napisaliśmy „zero lub więcej” liter b – dopasowało się „zero”.

Również w tekście zastępującym wzorzec (drugim parametrze komendy `s`) mogą występować specjalne wyrażenia:

Wyrażenie	*Znaczenie*
`&`	wszystko co zostało dopasowane do wzorca
`\1`	to co pasowało do pierwszego podwyrażenia
`\2`	to co pasowało do drugiego podwyrażenia itd.
`\L`	rozpocznij zmianę na małe litery
`\l`	zmień następny znak na małą literę
`\U`	rozpocznij zmianę na duże litery
`\u`	zmień następny znak na dużą literę
`\E`	zakończ zmianę pocztu liter

Spróbujmy prześledzić działanie wyrażeń regularnych na kilku prostych przykładach:

1. Zamiana myślnika na początku linii na komendę `\item`:

   ```
   s/^- /\\item /
   ```

   Podwójny ukośnik jest konieczny, aby następujący po nim znak nie został potraktowany jak znak specjalny.

2. To samo dla wyliczeń liczbowych (zamieniamy liczbę, po której następuje kropka na `\item` pod warunkiem, że liczba rozpoczyna linię):

   ```
   s/^[0-9][0-9]*\./\\item/
   ```

   Pierwszy zakres `[0-9]` jest po to, aby na początku linii rzeczywiście była liczba. Gdybyśmy od razu napisali `[0-9]*`, to takie wyrażenie oznaczałoby „zero lub więcej cyfr”. W innych programach, wykorzystujących bardziej rozbudowane wyrażenia regularne, moglibyśmy napisać: `[0-9]+`; w tym przypadku w `+` oznaczałby „jedna lub więcej cyfr”.

3. Przestawienie pierwszego słowa w linii na jej koniec:

   ```
   s/\([^ ]*\) \(.*\)/\2 \1/
   ```

   Zapis `[^ ]` oznacza dowolny znak, który nie jest spacją. Następująca po nim gwiazdka oznacza, że tych znaków może być dowolnie dużo. Zapis `.*` oznacza ciąg dowolnych znaków. Tak więc pierwsze wyrażenie to najdłuższy ciąg liter nie zawierający spacji, drugie to cała reszta linii.

4. Zamiana napisu zaczynającego się i kończącego znakiem podkreślenia na LaTeX-owe polecenie `{\em ...}`:

   ```
   s/_\([^_]*\)_/{\\em \1}/g
   ```

Zarówno w *vi* jak i w edytorze *sed* wyrażenia regularne mogą być również stosowane do adresowania poszczególnych linii tekstu. Np. polecenie:

```
/begin/,/end/s/^- /\\item /
```

wstawia polecenia `\item` zamiast rozpoczynających linię myślników tylko w liniach zawartych pomiędzy `begin` i `end`. W edytorze *vi*, gdy chcemy objąć zasięgiem działania komendy `s` cały plik, powinniśmy poprzedzić ją znakiem procentu (`%`) np:

```
%s/\([^ ]*\) \(.*\)/\2 \U\1\E/
```

Inną komendą obu edytorów, która również wykorzystuje wyrażenia regularne jest komenda `g`. Służy ona do wykonania innej komendy trybu liniowego na wszystkich liniach zawierających podany wzorzec np:

```
g/==/d
```

kasuje wszystkie linie zawierające `==`. Istnieje również możliwość użycia negacji warunku – służy do tego wykrzyknik (`!`):

```
g!/==/s/^\([a-zA-Z]*\)/[\1]/
```

Programowanie z wyrażeniami regularnymi

Istnieje co najmniej kilka języków programowania pozwalających na stosowanie wyrażeń regularnych. Najbardziej znane z nich to AWK i Perl. Popularność zdobywa również Python. Dostępne są także biblioteki, pozwalające na korzystanie z wyrażeń regularnych w klasycznych językach programowania – takich chociażby jak C/C++.

Chciałbym parę słów poświęcić tym językom programowania, które przede wszystkim były tworzone z myślą o przetwarzaniu tekstów. Wyrażenia regularne są podstawą ich działania. Do ich używania w programowaniu trzeba się przyzwyczaić. Jest to szczególnie istotne dla tych, którzy dużo programują w klasycznych językach programowa zapomnieć o tradycyjnych konstrukcjach języków wysokiego poziomu i typowych strukturach danych, a zacząć szukać ukrytej w danych ich wewnętrznej struktury i próbować przenieść jej zapis na język wyrażeń regularnych. Jest to bardzo opłacalne. Zazwyczaj dobrze napisany program w Perlu, czy AWK, służący do przetwarzania tekstów, jest od trzech do dziesięciu razy krótszy niż analogiczny program napisany w języku C. Przy tym dobre zastosowanie wzorców zazwyczaj bardzo poprawia czytelność programu.

W przypadku tych języków programowania mamy do czynienia z jeszcze bardziej rozbudowanym repertuarem wyrażeń regularnych, niż ma to miejsce w (bardzo prostych przecież) edytorach. Pojęciem, które tam się pojawia jest *kwantyfikator* – czyli określenie: ile razy ma się powtórzyć ostatnie wyrażenie. Do znanych już nam kwantyfikatorów należy `*`. Zdefiniowane zostały:

Kwantyfikator	*Znaczenie*
`*`	0 lub więcej razy
`+`	1 lub więcej razy
`?`	0 lub 1 raz
`{`*n*`}`	dokładnie *n* razy
`{`*n*,*m*`}`	co najmniej *n* razy, nie mniej niż *m*
`{`*n*,`}`	co najmniej *n* razy

Ostatnie trzy kwantyfikatory nie występują w standardowym AWK (o ile w mnogości implementacji tego języka można mówić o standardzie).

Dodatkowo, dla uproszczenia zapisu, do grupowania służą zwykłe nawiasy okrągłe zamiast nawiasów poprzedzonych odwrotnymi ukośnikami („w-tył-ciach"). Mamy też możliwość podania kilku alternatyw wyrażenia – do ich rozdzielenia służy pionowa kreska:

```
/^([0-9]+\.?[0-9]*|\.[0-9]+)$/
```

Język AWK jest chyba najmocniej związany z wyrażeniami regularnymi. Każdy skrypt[3] składa się z ciągu par (oddzielonych od siebie średnikami bądź znakami nowej linii):

<*wyrażenie*> { <*akcja*> }

Jeśli *wyrażenie* jest prawdziwe, wykonywana jest *akcja* i interpreter przechodzi do kolejnej pary. W przeciwnym przypadku od razu brana jest kolejna para.

Analiza prowadzona jest dla każdej linii osobno. W terminologii AWK należałoby powiedzieć, że dla każdego *rekordu* osobno. Standardowo rekord to linia tekstu. Możemy to jednak zmienić wpisując inny niż znak nowej linii (`\n`) separator rekordów (zmienna `RS`). Poszczególne rekordy dzielą się z kolei na *pola*. Domyślnie separatorem poszczególnych pól jest spacja lub znak tabulacji. Możemy to jednak zmienić ustawiając odpowiednio wartość zmiennej `FS`. Za ustawienia separatorów dla rekordów wyjściowych odpowiadają odpowiednio zmienne: `ORS` i `OFS`.

Wyrażenia w AWK mogą być dowolnymi wyrażeniami logicznymi, wyrażeniami regularnymi dopasowywanymi do całego rekordu, bądź jednego z jego pól, wyrażeniami adresowymi (określającymi: od której linii do której ma być wykonana jakaś akcja). Istnieją również dwa specjalne wyrażenia: `BEGIN` i `END`. Akcja odpowiadająca `BEGIN` wykonywana jest na samym początku działania programu – jeszcze przed odczytaniem pierwszego rekordu. Zazwyczaj służy ona do przypisania pewnych wartości początkowych zmiennym używanym w programie (w tym zmiennym wbudowanym). Analogicznie – akcja odpowiadająca `END` jest wykonywana jako ostatnia operacja przetwarzania pliku.

[3] Tak zwykle nazywa się krótkie programy wykonywane przez interpretery.

Akcja jest to zbiór instrukcji oddzielonych od siebie średnikami. Składnia instrukcji w AWK jest wzorowana na języku C. Występują tam typowe instrukcje warunkowe i pętle, te same operatory i funkcje wbudowane co w C. Można definiować funkcje, możliwa jest rekurencja. Ciekawą strukturą danych jest tablica asocjacyjna – czyli tablica, której indeksami mogą być dowolne wartości – niekoniecznie liczby. Dokładny opis języka można znaleźć w [2], bardzo polecam [3] – można tam także znaleźć kilka bardzo interesujących skryptów.

Spróbujmy na kilku prostych przykładach pokazać użycie wyrażeń regularnych w programach w AWK:

1. Skrypt, którego zadaniem jest poinformowanie użytkownika, czy ma pocztę w skrzynce pocztowej, czy jest coś nowego lub nieczytanego:

```
/^From /           { from++ }
/^Status: .*O/     { status++ }
/^Status: (OR|RO)/ { read++ }
END {
    if ( from )
      if ( from > status )
         print "You have new mail.";
      else if ( from > read )
         print "You have unread mail.";
      else
         print "You have mail.";
    }
```

2. Lista studentów mających konta na komputerze uniksowym. Informacja o kontach przechowywana jest w pliku `/etc/passwd`. Poszczególne linie w tym pliku wyglądają następująco:

```
gw:x:238:50:Grzegorz Wójcik:/home/gw:/bin/sh
jasio:x:150:10:Jan Nowak:/home/jasio:/bin/sh
```

Grupa studentów ma numer 10. Skrypt drukuje kolejny numer użytkownika, jego nazwę oraz imię i nazwisko:

```
BEGIN       { FS = ":" }
$4 ~ /^10$/ {
  printf "%4d. %-10s %s\n", ++i, $1, $5
  }
```

Zmienne `$1`, `$2`, … oznaczają kolejne pola rekordu.

3. Tabelka LaTeX-owa z listą ocen studentów. Dane do tabelki pochodzą z arkusza kalkulacyjnego. Separatorem pól jest znak tabulacji:

```
Lp.     Nazwisko i imię         Ocena
1.      Adam Abacki             5
2.      Bronisław Babacki       4
3.      Czesław Cacko           3
```

Jak widać nagłówek tabeli to linia zaczynająca się od litery, wszystkie pozostałe linie, rozpoczynające się liczbą, tworzyć będą zawartość tabeli. Skrypt mógłby wyglądać w ten sposób:

```
BEGIN       { FS = "\t"; ORS = "";
   print "\\begin{longtable}{|r|l|c|}\n";
   print "\\hline\n"; }

/^[A-Za-z]/ {
   for ( i = 1; i <= NF; i++ )
     {
     if ( i > 1 ) print " & ";
     print "\\multicolumn{1}{|c|}{\\em"
     print $i, "}";
     }
   print "\\\\\n\\hline\\hline\n";
   print "\\endhead\\hline\\endfoot\n"; }

/^ *[0-9]/  {
   for ( i = 1; i <= NF; i++ )
     {
     if ( i > 1 ) print " & ";
     print $i;
     }
   print "\\\\\n"; }

END         { print "\\end{longtable}\n" }
```

Język Perl jest znacznie młodszy od przedstawionych wcześniej programów. Stanowi on przedziwną mieszaninę komend i funkcji bibliotecznych języka C, niektórych poleceń AWK, rozkazów powłoki uniksowej i … pomysłów, które trudno znaleźć gdzie indziej. Wszystko to razem daje programiście ogromne możliwości. Przede wszystkim jest to język przeznaczony do obróbki danych tekstowych i tu możliwość zastosowania wyrażeń regularnych odgrywa ogromną rolę. Równie ważne wydają mi się nowoczesne struktury danych – tablice, które zostały zaimplementowane w taki sposób, że mogą również służyć jako listy i stosy. W Perlu występują także tablice asocjacyjne, które właściwie użyte stanowią potężne narzędzie. Perl w przetwarzaniu tekstów bardzo dobrze został omówiony w [1]. Jeśli ktoś szukałby dogłębnych informacji na temat tego języka, to polecam [5][4] oraz [4]. Obie te książki opisują już piątą wersję języka – wzbogaconą m.in. o pewne możliwości programowania obiektowego i złożone struktury danych.

Wyrażenia regularne są w Perlu chyba jeszcze szerzej wykorzystywane niż w AWK (chociaż w AWK całą strukturę programu podporządkowano wyrażeniom). W Perlu występuje komenda `s///`

[4] Nie polecam natomiast polskiego tłumaczenia tej książki, ponieważ zawiera bardzo liczne błędy, uniemożliwiające w praktyce zrozumienie działania niektórych konstrukcji języka.

o składni takiej jak w edytorach *sed* i *vi*. Wyrażenia regularne mogą być także częścią instrukcji warunkowej np. polecenie wydrukowania wszystkich linii zawierających słowo `Ala` wyglądałoby w ten sposób:

```
print if /Ala/;
```

Wrażenia regularne są też bardzo często wykonywane na ciągach danych np. przy użyciu polecenia `grep` możemy dokonać operacji na każdym elemencie tablicy:

```
@wew = grep { s/^$pref// } @telefony;
```

(Ta instrukcja wybiera z tablicy `@telefony`, tylko telefony wewnętrzne, czyli takie, których numer zaczyna się od prefiksu zapisanego w zmiennej `$pref` i zapamiętuje je w tablicy `@wew` po uprzednim wykasowaniu prefiksu[5]).

Bardzo często używaną instrukcją jest `split`, która dzieli tekst na kawałki (tworzy tablicę lub jak kto woli listę danych). Separatorem jest podane przez nas wyrażenie regularne (w tym przypadku przecinek, po który może wystąpić dowolnie dużo spacji):

```
@telefony = split /, */, $tel;
```

Spróbujmy prześledzić działanie jeszcze dwóch przykładów w Perlu, wykorzystujących wyrażenia regularne:

1. Skrypt służący do znalezienia w skrzynce poczty elektronicznej przesyłek wysłanych przez użytkownika, którego adres podamy jako pierwszy argument wywołania (`$ARGV[0]`):

```
while ( <STDIN> )
 {
 $tak = undef $tekst if /^From /;
 $tekst .= $_  if /^From / .. /^From: /;
 print if $tak;
 if ( /^From: / )
    {
    $tak = /$ARGV[0]/;
    print $tekst if $tak;
    }
 }
```

   Przesyłka poczty elektronicznej zaczyna się zawsze linią `From` (bez dwukropka). Dobry adres nadawcy powinien znajdować się w linii `From:` (już z dwukropkiem) – linia ta jest gdzieś wewnątrz nagłówka przesyłki. Jedyną trudnością jest więc zapamiętanie treści listu, od linii `From` do linii `From:` – kiedy to będziemy mogli podjąć decyzję, czy można ją wydrukować, czy nie. Pierwsza instrukcja warunkowa zeruje na początku każdej przesyłki zmienne wykorzystywane w programie. Druga zapamiętuje tekst przesyłki pomiędzy dwoma liniami `From`. Trzecia drukuje zawartość wybranej przesyłki. Czwarta określa, czy to jest ta przesyłka, o którą prosiliśmy i jeśli tak – drukuje zapamiętany poprzednio fragment nagłówka. Jeśli nie podamy na czym ma działać wyrażenie regularne, to domyślnie działa ono na właśnie wczytanej linii przechowywanej w zmiennej `$_`.

2. Znaczniki dla edytora *vi*. *Vi* został wyposażony w pewne ułatwienia dla programisty m.in. umożliwia szybkie poruszanie się pomiędzy definicjami poszczególnych funkcji i procedur. Trzeba tylko przygotować odpowiedni plik ze znacznikami zazwyczaj nazywający się `tags`. Dla języka C taki plik jest tworzony przez program `ctags`. Spróbujmy stworzyć skrypt, który będzie go tworzył dla skryptów w Perlu. Budowa tego pliku jest następująca:

```
wartosc kp.pl   /^sub wartosc()$/
kwota   kp.pl   /^sub kwota()$/
pesel   kp.pl   /^sub pesel()$/
```

   W pierwszej kolumnie jest nazwa procedury, w drugiej nazwa pliku, w trzeciej wyrażenie regularne pozwalające na odszukanie danej procedury w pliku. Nasz skrypt mógłby wyglądać następująco:

```
while ( $file = shift @ARGV )
 {
 open FILE, $file;
 while ( <FILE> )
    {
    next if /^#/;  chomp;
    print "$1\t$file\t/^$_\$/\n"
        if /^\s*sub\s*([^(]*)\(\)/;
    }
 close FILE;
 }
```

   `shift` jest funkcją pobierającą pierwszy element z listy (w naszym przypadku jest to lista argumentów wywołania). Dla każdego pliku źródłowego wyszukujemy te linie, w których występuje słowo kluczowe `sub` i nazwa procedury i drukujemy odpowiednią linię do pliku `tags`.

Podsumowanie

Wyrażenia regularne są na pewno znakomitym narzędziem przetwarzania dużych ilości informacji tekstowych. Szczególnie pożyteczne mogą się one okazać w połączeniu z językami programowania, takimi jak AWK, Perl, czy zdobywający popularność

[5] Jak widać język Perl jest bardzo zwięzły w zapisie – ile się trzeba napisać, żeby wytłumaczyć co robi jedna krótka linia Perla.

Python. Zwłaszcza, gdy języki te oferują moduły umożliwiające łatwe stworzenie interfejsów do baz danych, aplikacji sieciowych i systemów informacyjnych. Oczywiście programowanie z użyciem wyrażeń regularnych wymaga zmiany przyzwyczajeń i nieco innego spojrzenia na dane wejściowe. Konieczne jest odkrycie ich (często nieudokumentowanej) struktury. Wydaje mi się, że wysiłek związany z poznaniem jednego z języków programowania do przetwarzania tekstów i nauczeniem się bardzo prostych reguł działania wyrażeń regularnych opłaci się po stokroć – odciążeni zostaniemy od niewdzięcznej ręcznej pracy, którą równie dobrze może wykonać komputer.

Bibliografia

[1] Adam Dawidziuk, Piotr Bolek. Perl w przetwarzaniu tekstów. *Biuletyn GUST*, 7:26–33, 1996.

[2] Dale Dougherty. *sed & awk*. O'Reilly & Associates, Inc., 1st edition, 1992.

[3] Bogusław Lichoński, Tomasz Przechlewski. AWK – opis języka z przykładami. *Biuletyn GUST*, 7:10–25, 1996.

[4] Sriram Srinivasan. *Advanced Perl Programming*. O'Reilly & Associates, Inc., 1st edition, 1997.

[5] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 2nd edition, 1996.

◇ Grzegorz Wójcik
`G.Wojcik@itl.waw.pl`