Notatki 3

Wyrażenia regularne, sed, awk

Wyrażenie regularne (*ang. regular expression*, regex) – reguła opisująca ciąg znaków. Pozwala ona na podanie zwartego przepisu, który określi interesującą nas klasę łańcuchów symboli. Jest to niezwykle przydatne przy przeszukiwaniu i modyfikacji dużej ilości danych. Niestety, występuje kilka konwencji takich reguł: podstawowe (BRE), rozszerzeone (ERE), kompatybilne z językiem Perl (PCRE). Tak naprawdę różnią się one między sobą jedynie pewnymi niuansami – przede wszystkim tym, które znaki uznawane są domyślnie za specjalne, a które nie, co wiąże się z koniecznością używania lewego ukośnika `\` w odpowiednich miejscach. My będziemy używać BRE, gdyż jest to domyślny typ wyrażeń w wielu programach, także w edytorze `vim`. Jeżeli chcemy użyć innych wariantów, część programów udostępnia odpowiednie opcje (`-E` w `grep`-ie i `sed`-dzie).

Słowniczek:

- `^` – początek linii
- `$` – koniec linii
- `.` – dowolny znak
- `*` – dowolna (w tym zerowa!) liczba wystąpień znaku poprzedzającego
- `\+` – dowolna (ale niezerowa!) liczba wystąpień znaku poprzedzającego
- `\?` – jedno lub brak wystąpień znaku poprzedzającego
- `\{n\}` – `n` wystąpień znaku poprzedzającego
- `\{n,m\}` – pomiędzy `n` a `m` wystąpień znaku poprzedzającego
- `[ ]` – zbiór znaków
- `[^ ]` – odwrócenie zbioru (dowolny znak nie należący do podanego zbioru)
- `[:alpha:]`,`[:alnum:]`,`[:digit:]`... – klasy znaków[1]
- `\( \)` – grupa dopasowania
- `\&` – całe dopasowane wyrażenie
- `\|` – alternatywa

Jeśli przy powyższych symbolach posiadających `\` pominiemy go, symbol straci specjalne znaczenie (np. `\+` stanie się zwykłym plusem). Analogicznie dodanie ukośnika przy tych powyższych symbolach, które go nie posiadają, również odbierze im specjalny charakter (np. `\.` to zwykła kropka). W konwencji ERE jest dokładnie na odwrót (czyli np. `\.` to dowolny znak, a `.` to zwykła kropka itp.). Nie należy się specjalnie martwić tym zamieszaniem notacyjnym – w razie konieczności nietrudno się przestawić (lub wymusić na programie użycie innej konwencji, albo nawet przekonwertować wyrażenie... z wykorzystaniem wyrażeń regularnych).

Przejdźmy do praktyki. Przygotujmy plik o nazwie `tekst.txt` i wypełnijmy go jakąś treścią, np.:

```
raz
dwa
trzy
cztery
aa
aaa
atlas
astronomia
apokalipsa
apteka
apteka
burza
aastronom1a
a5tr0n0m1a
a5tr0nom1a
astr0nom1a
astr0nomia
ąśżźćłó
czyjśadres@fuw.edu.pl
jakiśadres1@gmail.com
23:54:12
45:67:23
www.fuw.edu.pl
```

Wywołanie polecenia

[1] Więcej można poczytać tutaj: `https://www.gnu.org/software/sed/manual/html_node/Character-Classes-and-Bracket-Expressions.html#Character-Classes-and-Bracket-Expressions`

```
$ grep '^a.*' tekst.txt
```

spowoduje wypisanie z pliku `tekst.txt` wszystkich linii, które zaczynają się od litery `a` i mają za nią (lub nie) dowolną liczbę dowolnych znaków. UWAGA! Jeżeli w `grep`-ie używamy wyrażeń regularnych, warto brać je w cudzysłów, gdyż powłoka (`bash`) może próbować traktować znaki specjalne po swojemu. Np. znak `^` służy w Bashu do zmodyfikowania i wywołania poprzednio użytej komendy. Należy też pamiętać, że w przypadku użycia podwójnego cudzysłowu (`"`) Bash nadal może interpretować znak dolara na swoją modłę (do oznaczania zmiennych, trybu arytmetycznego i wyniku zagnieżdżonych poleceń) – co może być przez nas pożądane bądź nie. Np.

```
$ grep "$((1+1))" tekst.txt
```

wyszuka w pliku linie zawierające liczbę dwa. Nie jest to aż tak częsty problem, gdyż `$` i tak zwykle jeśli występuje, to jako ostatni znak w wyrażeniu regularnym. Są jednak możliwe przypadki, gdy nie tak jest. Dlatego też dalej będę na wszelki wypadek – i dla wyrabiania dobrych nawyków – używał pojedynczego cudzysłowu (`'`). Wracając do przykładu – gdy zmodyfikujemy go do

```
$ grep '^a.*a$' tekst.txt
```

wyszukane zostaną te linie, które składają się z litery `a`, dowolnej (w tym zerowej) liczby dowolnych znaków oraz kolejnej litery `a` (i tu kończy się linia). Jeśli chcemy, by był co najmniej jeden znak, wystarczy drobna modyfikacja:

```
$ grep '^a.\+a$' tekst.txt
```

Jeżeli ma być to *co najwyżej* jeden znak, również jest na to sposób:

```
$ grep '^a.\?a$' tekst.txt
```

Możemy doprecyzować liczbę znaków, np.

```
$ grep '^a........a$' tekst.txt
```

ograniczy wyszukania do sytuacji, gdzie pomiędzy literami `a` jest ich dokładnie osiem. Nie jest to zbyt wygodny zapis, toteż można go skrócić:

```
$ grep '^a.\{8\}a$' tekst.txt
```

Notacja ta jest dość elastyczna i pozwala też na operowanie zakresami – wystarczy podać dwie liczby:

```
$ grep '^a.\{0,1\}a$' tekst.txt
```

co odpowiada poprzednio zaprezentowanemu symbolowi `\?` . Pominięcie którejś z cyfr jest interpretowane jako pozostawienie dowolności – np. `\{1,\}` jest tożsame użyciu symbolu `\+` .

Możemy uszczegółowić rodzaj znaków – np. jeżeli chcemy, by były to jedynie litery:

```
$ grep '^a[a-z]\{8\}a$' tekst.txt
```

Rozpoznawane są także bardziej „opisowe” nazwy zakresów – przykładowo poprzednie wyrażenie można zapisać też tak:

```
$ grep '^a[[:alpha:]]\{8\}a$' tekst.txt
```

Oczywiście możemy żądać bardziej wyrafinowanych dopasowań. Wybierzmy te linie, w których występuje od jednej do trzech cyfr. Zrobimy to za pomocą:

```
$ grep '^[^0-9]*\([0-9][^0-9]*\)\{1,3\}$' tekst.txt
```

W wyrażeniu regularnym powyżej szukamy najpierw (po znaku początku linii `^` oczywiście) dowolnej liczby znaków nie będących cyframi (`[^0-9]*`), po czym od jednego do trzech powtórzeń (`\(  \)\{1,3\}`) grupy składającej się z jednej cyfry oraz dowolnej liczby znaków nie będących cyframi (`[0-9][^0-9]*`). Warto tu podkreślić, że znak `^` umieszczony w kontekście listy możliwych do dopasowania znaków (a więc w nawiasach kwadratowych) nie oznacza początku linii, ale odwrócenie dopasowania.

Operowanie zakresami pozwala na precyzyjny wybór interesującego nas ciągu znaków. Np. aby wyszukać godziny (zapisane w formacie 24-godzinnym), możemy użyć:

```
$ grep '[0-2][0-9]:[0-5][0-9]:[0-5][0-9]' tekst.txt
```

Pominie to inne liczby zapisane w podobnym formacie, ale nie mieszczące się w poprawnych zakresach godzin, minut i sekund.

Oczywiście nie musimy dopasowywać tylko zestawów znaków dających się zapisać jako przedziały. Możemy np. chcieć znaleźć linie zawierające cyfrę `1`, literę `i` bądź kropkę. Uzyskamy to przez:

```
$ grep '[1i.]' tekst.txt
```

UWAGA: znaki w nawiasach kwadratowych traktowane są dosłownie – tzn. znaków specjalnych nie należy poprzedzać ukośnikiem \. Podobny efekt da polecenie

```
$ grep '\(1\|i\|\.\)' tekst.txt
```

Zawiera ono symbol alternatywy („lub") \|. W przypadku wymieniania pojedynczych znaków, nawiasy kwadratowe są wygodniejsze (i prawdopodobnie nieco szybsze ze względu na mechanizm działania). Symbol \| pozwala jednak na alternatywę pomiędzy całymi zestawami znaków – przykładowo:

```
$ grep '\(fuw\|gmail\)' tekst.txt
```

znajdzie linie zawierające ciąg `fuw` lub `gmail`.

Warto tu przypomnieć, że program `grep` domyślnie wypisuje całe linie zawierające szukane przez nas wyrażenie. Np. poniższe polecenie znajdzie symbol `@`, po którym następuje dowolnie wiele znaków alfanumerycznych i kropek.

```
$ grep '@[[:alnum:].]*' tekst.txt
```

Jeśli dodamy opcję `-o`, program ograniczy się do wypisania jedynie samego dopasowanego wyrażenia.

```
$ grep -o '@[[:alnum:].]*' tekst.txt
```

Program sed – *ang.* **s***tream* **ed***itor*, program (jak sama nazwa wskazuje) służący do edycji strumieni. Jak dotąd w pracy z wyrażeniami regularnymi ograniczaliśmy się do prostego wyszukiwania informacji. Jest to bardzo pożyteczne ich zastosowanie, ale przecież można znacznie więcej. Na początek zmodyfikujmy nasz przykład z adresem email tak, by – wykorzystując `sed`-a – zastępował wszystkie adresy email w pliku przez słowo „email" (np. w celu ich ocenzurowania przed publikacją pliku w sieci):

```
$ sed 's/[[:alnum:]]\+@[[:alnum:].]*/email/' tekst.txt
```

Jak widać składnia jest dość prosta: `sed 's/znajdź/zamień/' plik`. Możemy też wykorzystywać znalezione wyrażenia gdy komponujemy zastępujący je tekst. W tym celu wykorzystujemy cicho przemyconą już wyżej funkcjonalność wyrażeń regularnych – grupy.

```
$ sed 's/\([[:alnum:]]\+\)@\([[:alnum:].]*\)/domena: \2\nlogin: \1/' tekst.txt
```

Powyższe polecenie wyróżnia w wyrażeniu regularnym dwie grupy (objęte nawiasami okrągłymi) – pierwsza zawiera wszystkie znaki aż do „małpy", następna – wszystkie po niej. Odwołujemy się potem do tychże grup przez ich numery (nadawane w kolejności w jakiej są znajdowane). Symbol \n nakazuje złamać linię – tak by domena i login były w oddzielnych wierszach.

Jeśli interesują nas same emaile, możemy albo posłużyć się strumieniami powłoki i najpierw użyć `grep`-a do wyłuskania wierszy, a potem `sed`-a do ich zamiany:

```
$ grep '[[:alnum:]]\+@[[:alnum:].]*' tekst.txt | sed 's/\([[:alnum:]]\+\)@\([[:alnum:].]*\)/domena: \2\nlogin: \1/'
```

Może się wydać nieco nieeleganckie i niewygodne powtarzanie dwukrotnie tego samego wyrażenia (i męczenie się potem gdy chcemy je zmienić). Możemy to poprawić zapisując wyrażenie w zmiennej[2].

```
$ reg='\([[:alnum:]]\+\)@\([[:alnum:].]*\)'
$ grep "$reg" tekst.txt | sed "s/$reg/domena: \2\nlogin: \1/"
```

Warto tu zwrócić uwagę, że jest to właśnie ten przypadek gdy chcemy podwójnych cudzysłowów – aby odwołać się do zapisanej zmiennej powłoki Bash. Tak naprawdę jednak można to zrobić jeszcze prościej – przy użyciu samego `sed`-a:

```
$ sed -n 's/\([[:alnum:]]\+\)@\([[:alnum:].]*\)/domena: \2\nlogin: \1/p' tekst.txt
```

Dodaliśmy przełącznik `-n`, który wyłącza domyślne zachowanie `sed`-a polegające na przedrukowaniu wszystkich danych wejściowych, a także dodaliśmy na końcu jawne żądanie wydrukowania zmienionych linii (`p`), bez którego opcja `-n` wygasiłaby całe wyjście programu.

Posługując się wyrażeniami regularnymi należy pamiętać, że są one dość „zachłanne" (*ang. greedy*), tzn. dopasowują najwięcej znaków, ile tylko są w stanie. Przykładowo zastąpmy w `www.fuw.edu.pl` ciąg składający się z początku linii oraz dowolnej liczby dowolnych znaków, po których następuje kropka:

```
$ echo 'www.fuw.edu.pl' | sed 's/^.*\./chomp\!/'
```

[2] Jeśli chcemy jeszcze bardziej ułatwić sobie życie, możemy zapisać obie te linie w jednej, rozdzielając je średnikiem

Dostaniemy `chomp!pl` – czyli wyrażenie zatrzymało się po ostatniej możliwej kropce. Niektóre programy mają opcje umożliwiające zmianę trybu zachłannego na „leniwy" (*ang. lazy*), ale możemy to osiągnąć także po prostu staranniej dobierając wyrażenie:

```
$ echo 'www.fuw.edu.pl' | sed 's/^[^.]*\./chomp\!/'
```

Różnica polega na zażądaniu zamiast dowolnej liczby *dowolnych* znaków – dowolnej liczby znaków *nie będących kropką*. Wówczas dopasowanie zatrzymuje się po pierwszej napotkanej kropce. Jeśli chcemy zatrzymać się po kropce drugiej, możemy to doprecyzować jako:

```
$ echo 'www.fuw.edu.pl' | sed 's/^\([^.]*\.\)\{2\}/chomp\!/'
```

Jeśli chcemy wykorzystać cały dopasowany ciąg, nie musimy go obejmować grupą – możemy użyć symbolu `&`.

```
$ sed 's/.\{8\}/&\n/g' tekst.txt
```

Powyższy przykład spowoduje złamanie wszystkich wierszy po przekroczeniu długości ośmiu znaków (może być to czasami przydatne podczas programowania – kompilatory dopuszczają niekiedy jakąś maksymalną długość linii kodu). Modyfikator `g` powoduje modyfikację (w tym wypadku wstawienie znaku łamania wiersza po dopasowanym ciągu) dla *każdego* wystąpienia wyrażenia (w tym wypadku dowolnych ośmiu znaków), nie tylko dla pierwszego w każdej linii.

Z przydatnych funkcji warto wymienić także zamianę małych liter na wielkie:

```
$ sed 's/\(.*\)/\U&/' tekst.txt
```

Analogicznie oczywiście można użyć `\L` dla operacji odwrotnej.

Jeśli chcemy wstawić linię przed każdym trafieniem, możemy użyć:

```
$ sed 's/[[:alnum:]]\+@[[:alnum:].]*/&\n\^to był email/' tekst.txt
```

Analogiczne wstawienie po każdym trafieniu:

```
$ sed 's/[[:alnum:]]\+@[[:alnum:].]*/To będzie email: \n&/' tekst.txt
```

Program `sed` ma więcej możliwości niż jedynie zamianę. Jeśli np. chcemy skasować wszystkie pasujące linie (w tym wypadku zawierające email), możemy użyć polecenia `d`:

```
$ sed '/[[:alnum:]]\+@[[:alnum:].]*/d' tekst.txt
```

Aby usunąć wszystkie *nie pasujące* linie, musimy dołożyć wykrzyknik:

```
$ sed '/[[:alnum:]]\+@[[:alnum:].]*/!d' tekst.txt
```

To be continued...