# Systemy operacyjne

Laboratorium 6

## Awk – wyrażenia regularne

Jarosław Rudy
Politechnika Wrocławska

28 lutego 2017

Temat obejmuje tworzenie wyrażeń regularnych, zwłaszcza w połączeniu z komendą `awk`. Strony takie jak `http://regex101.com` mogą zostać wykorzystane do projektowania i testowania wyrażeń regularnych. Jednakże mogą występować pewne różnice pomiędzy wyrażeniami regularnymi `awk`a a tymi stosowanymi poza nim, zwłaszcza w zależności od opcji podanych przy wywołaniu `awk`.

## 1 Wstęp

*Wyrażenia regularne* (*regex*) to pożyteczne narzędzie używane do wyszukiwania określonych wzorców w łańcuchach tekstowych. Regexy mogą być użyte np. do późniejszej zamiany tekstu na inny tekst. Składnia regexów jest podobna w większości narzędzi ich używających (`awk`, `grep`, `perl`).

Zanim przejdziemy do użycia regexów w `awk`u omówimy najpierw składnię regexów jako takich. Większość znaków używana jest wprost (dosłownie). Przykładowo, by "dopasować" znak `a`, używamy po prostu tego znaku:

```
a
```

Żaden inny tekst niż pojedyncza litera `a` nie dopasuje się do tego regexa (nawet `A`).

Podstawą techniką (operatorem) dla regexów jest połączenie (konkatenacja, koniunkcja) regexów – do dwóch regexów napisanych bezpośrednio po sobie dopasuje się tekst, który jest połączeniem (sklejeniem) tekstu, który dopasuje się do pierwszego regexu i tekstu, który dopasuje się do drugiego regexu. Przykładowo, do regexu:

```
abc
```

dopasuje się tylko tekst, w którym bezpośrednio po `a` występuje `b`, a bezpośrednio dalej występuje `c`.

Skoro mamy możliwość uzyskania koniunkcji (`AND`), to należy spodziewać się także możliwości użycia alternatywy (`OR`) regexów. Służy do tego operator `|`:

```
a|b
```

Do tego regexu dopasuje się *pojedynczy* znak, którym jest albo `a`, albo `b`. Pojawia się tutaj pierwszy problem. Znak `|` ma specjalne znaczenie, więc jak można sprawić, by regex "poszukiwał" tego znaku dosłownie? Należy poprzedzić go odwrotnym ukośnikiem:

```
\|
```

Będziemy używać tej techniki dosyć często, by dosłownie dopasować znaki specjalne. W szczególności kropkę, gwiazdkę i sam odwrotny ukośnik:

```
\. \* \\
```

Analogiczna sytuacja zachodzi, gdy chcemy dopasować tabulator lub koniec wiersza:

```
\t \n
```

Zwykłe ukośniki (`/`) nie są specjalne, ale składnia wykorzystania regexów w `awku` może również powodować konieczność poprzedzenia ich backslashem (o tym później).

Drugim problemem z wykorzystaniem alternatywy jest priorytet tego operatora. Przykładowo:

```
ab|cde
```

Koniunkcja (`AND`) wiąże silniej niż alternatywa (`OR`), więc domyślne nawiasy zostaną ustawione jako:

```
(ab)|(cde)
```

co oznacza, że dopasuje się do tego albo 2-znakowy ciąg `ab`, albo 3-znakowy ciąg `cde`. Jest to zupełnie różne od wersji:

```
a(b|c)de
```

która dopasuje się do 4-znakowego ciągu: pojedynczego `a`, po którym jest pojedynczy znak `b` *lub* `c`, po którym nastąpi ciąg `de`. Przy okazji widzimy, że okrągłe nawiasy również są znakami specjalnymi, więc muszą być poprzedzone backslashem, gdy chcemy je dopasować dosłownie. Nawiasy pozwalają więc kontrolować kolejność "działań". Ponadto nawias definiuje tzw. *grup* (możemu ją też nazwać "podregexem"). Pozwala to wygodnie określić "zasięg" pewnych podwyrażeń. Co więcej, po dopasowaniu można odnieść się do fragmentów tekstu, które dopasowały się do poszczególnych grup (o tym później).

Do tej pory poszukiwaliśmy konkretnych znaków (np. litera `a`), jednak moc regexów polega na możliwości określenia ogólnych reguł (wzorców) poszukiwania. Przykładowo, aby dopasować *dowolny*[1] *pojedynczy* znak należy użyć kropki:

[1] Wyjątkiem może być znak nowej linii, zależnie od programu i ustawień.

.

Kolejną, często niezrozumiałą, techniką są nawiasy kwadratowe (`[` oraz `]`). Do takiej konstrukcji dopasuje się `jeden` znak, którym może być dowolny ze znaków w nawiasach. Przykładowo do:

```
[abc]
```

dopasuje się znak `a` lub `b` lub `c`. Jest to nieco podobne do alternatywy:

```
a|b|c
```

Kolejność podawania znaków w nawiasach kwadratowych nie ma znaczenia (przy alternatywie również). Zauważmy też, że wielokrotne podanie tego samego znaku jest nadmiarowe:

```
[aaaaabbb]
```

jest równoważne:

```
[ab]
```

W nawiasach kwadratowych większość znaków traci swoje specjalne znaczenie (ale część *zyskuje* specjalne znaczenie!), więc nawiasy kwadratowe mogą zostać użyte do dosłownego dopasowania:

```
[.*()]
```

W tym przypadku odwrotne ukośniki nie są potrzebne, bo wewnątrz nawiasów powyższe znaki nie są specjalne. Od razu zauważamy też, że same znaki `[` i `]` są standardowo specjalne. Wewnątrz konstrukcji `[]` znak `[` nie jest już specjalny, za to znak `]` wciąż jest, bo oznacza koniec konstrukcji. Aby więc powiedzieć "pojedynczy znak `[` lub `]`" możemy użyć alternatywy:

```
\[|\]
```

lub nawiasów kwadratowych:

```
[[\]]
```

druga wersja jest niewygodna w odczycie ("nawiasy wewnątrz nawiasów"), ale jest poprawna.

Nawiasy kwadratowe mają swoją "odwrotną formę", w przypadku której w miejsce nawiasów dopasuje się pojedynczy dowolny znak OPRÓCZ tych podanych w nawiasach:

```
[^abc]
```

Do tego regexu dopasuje się każdy pojedynczy znak poza `a`, `b` i `c`. Za takie zachowanie odpowiada znak `^`, jeśli zostanie użyty *od razu* za nawiasem otwierającym. Jeśli wtawimy ten znak w innym miejscu, to zostanie użyta "normalna" wersja nawiasów, a znak `^` nie jest już specjalny:

```
[a^bc]
```

powyższy regex oznacza pojedynczy znak: `a`, `b`, `c` lub `^`. Aby więc użyć `^` dosłownie wewnątrz nawiasów należy albo umieścić go gdzie indziej niż na początku, albo poprzedzić go backslashem. Tak więc, jeśli chcemy znaleźć dowolny znak poza `^`, to użyjemy:

```
[^^]
```

Pierwszy znak `^` oznacza, że mamy do czynienia z "odwrotną" wersją nawiasów, a drugi oznacza, że tego właśnie znaku nie szukamy.

Kolejnym znakiem "częściowo specjalnym" dla nawiasów kwadratowych jest minus (łącznik), czyli znak `-`. Normalnie jest traktowany dosłownie, ale nie gdy oznacza "zasięg" znaków. Przykładowo poniższy regex:

```
[b-f]
```

oznacza "pojedynczy znak z przedziału od `b` do `f` z tymi znakami włącznie". Ta cecha jest bardzo przydatna. Na przykład dozwolona nazwa zmiennej (identyfikator) języka `C` składa się z małych liter, dużych liter, cyfr i podkreślnika[2]. Taki znak może zostać opisany jako:

```
[a-zA-Z0-9_]
```

Zauważmy brak spacji (które były by potraktowane dosłownie!).

Składnia "od-do" może sprawiać czasami problemy. Jest to związane z próbami autorów `awk`a w kwestii automatycznego dostosowania się regexów do aktualnie ustawianego języka. Celem było uzyskanie np. wzorca "dowolna polska litera":

```
[a-ząćęłńóśźż]
```

za pomocą zwykłego:

```
[a-z]
```

o ile język ustawiony jest na polski. Niestety, próba ta spowodowała, że zasięgi "od-do" mogą czasami nie działać (np. do `[a-z]` dopasuje się nie tylko mała litera). Jeśli stwierdzamy ten problem, to należy albo zrezygnować ze składni "od-do", albo przed wywołaniem `awk`a ustawić odpowiednią zmienną `bash`ową:

```
LC_LANG=c
```

Kolejną bardzo użyteczną cechą nawiasów kwadratowych są tzw. klasy znaków. Problematyczny w nich jest fakt, że wykorzystują one *drugi* zestaw nawiasów kwadratowych, co może być mylące. Przykładowo zamiast używać zasięgu:

```
a-z
```

możemy użyć odpowiedniej klasy `lower`:

[2]Pomijając pierwszy znak, który nie może być cyfrą.

```
[:lower:]
```

Istotne jest, że obie konsturkcje działają *tylko* w nawiasach kwadratowych:

```
[a-z] lub [[:lower:]]
```

Możemy użyć więcej niż jednej klasy. Przykładowo, możemy użyć klasy `alnum`:

```
[[:alnum:]]
```

co daje podobny efekt, jak użycie trzech klas `lower`, `upper` i `digit` w jednym "zewnętrznym" nawiasie kwadratowym:

```
[[:lower:][:upper:][:digit:]]
```

Chociaż równie dobrze możemy użyć zasięgów:

```
[a-zA-Z0-9]
```

Efektem ubocznym jest fakt, że gdy podajemy klasę, to znaki `[`, `]` oraz `:` stają się specjalne. Przykładowo:

```
[[:]       # błąd, niezamknięta definicja klasy
[[::]]     # błąd, zła nazwa klasy
```

aby uzyskać efekt "pojedynczy znak `[` lub `:`" należało użyć backlasha:

```
[\[\:]
```

Na zakończenie tematu nawiasów kwadratowych pokażemy, że możliwe jest użycie wszystkich powyższych technik naraz:

```
[a-z[:digit:] !]
```

co oznacza "jeden znak, który jest małą literą, cyfrą, spacją lub wykrzyknikiem".

# 2 Określenie krotności

Do tej pory mówiliśmy o pojedynczych znakach lub grupach znakach o określonej długości. Np. 3 litery `a` pod rząd mogą zostać określone tak:

```
aaa
```

Jeśli chcemy powiedzieć 3 razy ciąg `ab` pod rząd, to możemy użyć regexa:

```
ababab
```

Podobnie, dla 3 powtórzeń "znak `a` lub znak `b`" możemy użyć:

```
[ab][ab][ab]      # lub (a|b)(a|b)(a|b)
```

Do ostatniego przykładu dopasują się tekstu `aaa`, `bbb`, `aab` czy `bab`. Wszystkie powyższe przykłady zadziałają. Jednak co, gdy chcemy określić bardziej ogólne zależności, jak "co najmniej 1 wystąpienie" lub gdy powtarzane wyrażenie jest długie i wklejanie go jest niewygodne?

Rozwiązaniem jest użycie szeregu mechanizmów, które pozwalają określić krotność danego danego znaku lub całej grupy (ujętej w nawiasy okrągłe). Pierwszym z takich znaków jest pytajnik, który oznacza, że poprzedni znak/grupa może wystąpić raz lub nie wystąpić w ogóle (element opcjonalny):

```
a?       # znak 'a' lub bez znaku
abc?     # ciąg 'ab' lub 'abc'
a(bc)?   # ciąg 'a lub 'abc'
```

Kolejnym znakiem jest plus `+`, który oznacza co najmniej jedno wystąpienie (1 lub więcej):

```
a+                        # ciąg złożonych z samych 'a'
(0|1|2|3|4|5|6|7|8|9)+    # ciąg cyfr
[0-9]+                    # ciąg cyfr
[0-9][0-9]+               # ciąg co najmniej dwóch cyfr
([0-9][0-9])+             # ciąg z parzystą liczbą cyfr
```

Zauważmy, że w pierwszym i trzecim przypadku "zwielokrotnianym" elementem jest jeden znak, więc nie ma potrzeby dodawania nawiasów okrągłych. W przypadku drugim nawiasy są podane ze względu na priorytety "operatorów" `+` oraz `|`. Ostatnie dwa przypadki obrazują wpływ rozmieszczenia nawiasów na interpretację wyrażenia.

Kolejnym możliwym operatorem jest gwiazdka `*`, która oznacza "zero lub więcej wystąpień". Na laboratorium często kusi użycie wyrażenia:

```
.*
```

co oznacza "dowolną liczbę (także zero) dowolnych znaków". W praktyce takie wyrażenie jest zwykle zbyt "liberalne" i często oznacza błąd w rozumowaniu.

Przydatne do określania częstości wystąpień są "wyrażenia przedziałowe":

```
a{n}      # dokładnie n-elementowy ciąg liter 'a'
a{n,m}    # od n- do m-elementowy ciąg liter 'a'
a{n,}     # co najmniej n-elementowy ciąg liter 'a'
```

Od razu zauważmy, że nie ma wersji "co najwyżej n-elementowy" czyli `a{,n}`. Ponadto, w tym kontekście znaki {, } oraz , są specjalne. Uwaga! Składnia wyrażeń przedziałowych może nie być domyślnie dostępna, jednak można ją włączyć podając opcję `--re-interval` podczas wywołania `awka`[3]

Dalsze informacje dotyczące wyrażeń regularnych zostaną wyjaśnione już w kontekście ich użycia w `awk`u.

[3]Można podać też opcję `--posix`, ale wprowadza ona więcej zmian.

# 3 Regexy a `awk`

Zajmiemy się teraz sposobami definicji i użycia regexów z `awk` oraz opiszemy niewyjaśnione jeszcze znaki specjalne.

Pierwszym miejsce, w którym możemy chcieć użyć regexów jest definicja separatorów wejściowych RS i FS (pamiętamy, że separatory wyjściowe ORS i OFS to zwykły tekst, nie regexy!). Przykładowo, jeśli nasze rekordy oddzielone są "ciągiem jednego lub więcej znaku hasha (`#`)", jak poniżej:

```
przykładowy##tekst#z#hashami####jako###separatory
```

to wystarczy przypisać odpowiedni regex do zmiennej RS jako tekst:

```
RS="#+"
```

Oczywiście należy dokonać to w akcji ze wzorcem `BEGIN`, zanim `awk` rozpocznie podział na rekordy. Zauważmy, że regex "zdefiniowany" poprzez cudzysłowy sprawia, że cudzysłów wewnątrz regexa będzie musiał być poprzedzony backslashem, pomimo, że nie jest specjalny dla samego regexa.

Innym częstym wykorzystaniem regexów są wzorce. Przykładowo:

```
/regex/ { print $0 }
```

wydrukuje tylko rekordy, które "spełniają" podany regex. Jest w tym jednak kilka pułapek. Po pierwsze powyższy zapis wykorzystuje lewy ukośnik `/` do oznaczenia początku i końca wyrażenia regularnego. Oznacza to, że chociaż w samym regexie lewy ukośnik nie jest znakiem specjalnym, to jednak trzeba go w tym kontekście poprzedzić backslashem, jeśli chcemy go użyć w regexie:

```
/\//     # regex "szukajacy" doslownego ukosnika
```

Po drugie należy zwrócić uwagę na to jaki łańcuch tekstowy zostanie przeszukany. Przedstawiony powyżej przykład tak naprawdę jest równoważny:

```
$0 ~ /regex/ { print $0 }
```

Znak $\sim$, to operator dopasowania. Po jego prawej stronie jest regex, a po lewej przeszukiwany tekst. Oczywiście zamiast $0 możemy użyć zmiennej lub pola:

```
$3 ~ /regex/     # "przeszukanie" pola
var ~ /regex/    # "przeszukanie" zmiennej
```

Istnieje też odwrotny operator `!`$\sim$ oznaczający "brak dopasowania".

Trzecią pułapką jest zrozumienie operatora dopasowania. Wiemy, że poniższy regex:

```
/a/
```

będzie poszukiwał jedynie pojedynczej litery `a`. Spodziewamy się więc, że poniższy kod:

```
/a/ { print $0 }
```

wydrukuje wyłącznie rekordy składające się z samego `a`. Nie jest to jednak prawda. Dzieje się tak, bo operator $\sim$ zwraca prawdę, gdy GDZIEKOLWIEK w przeszukiwanym tekście znajduje się podciąg, który pasuje do wzorca. Innymi słowy, chociaż sam regex definiuje pojedynczy znak `a`, to wystarczy, że znak ten pojawi się gdziekolwiek w poszukiwanym tekście (w powyższym przykładzie w całym rekordzie), by dopasowanie nastąpiło.

Co jeśli chcemy znaleźć tylko rekordy zawierające jedną literę `a` i nic więcej? Użyjemy w tym celu nieomówionych do tej pory znaków `^` oraz `$`. Pierwszy z nich dopasuje się do *początku* przeszukiwanego tekstu, a drugi do jego *końca*. Tak więc, by wydrukować rekordy zawierające tylko pojedynczą literę `a` powinniśmy użyć:

```
/^a$/ { print $0 }
```

Możemy też "wykryć" pusty rekord poprzez użycie tych znaków bezpośrednio po sobie.

```
/^$/
```

Oczywiście wynika z tego, że znaki `^` oraz `$` są specjalne.

To nie koniec pułapek. Zauważmy, że używaliśmy do tej pory dwóch form regexów: w ukośnikach (`/regex/`) i w cudzysłowach (`"regex"`). Niestety, te formy nie są w pełni równoważne. Zwykle używamy pierwszej – druga działa głównie przy przypisaniu do zmiennej oraz w funkcji `match()`. Pierwszy problem brzmi: jak użyć regexa zapisanego w zmiennej? Intuicyjne użycie:

```
BEGIN { var="regex" }
/var/ { print $0 }
```

nie zadziała – `awk` jako regexa użyje tekstu `"var"` zamiast zawartości tej zmiennej. Poprawne wykonanie wymaga użycia wprost operatora dopasowania i pominięcia ukośników:

```
BEGIN { var="regex" }
$0 ~ var { print $0 }
```

Druga różnica pomiędzy regexami w ukośnikach i w cudzysłowach, jest specyfika działania cudzysłowów. Działają one podobnie jak w języku `C/C++`, co oznacza, że niektóre znaki mogą zostać potraktowane specjalnie *zanim* zostaną zrozumiane jako regex. Wyjaśnimy to na przykładzie. Załóżmy, że chcemy stworzyć regex, które dopasuje się do dosłownej kropki. Użycie wprost:

```
/./ { print $0 }
```

nie zadziała, bo kropka oznacza "dowolny znak". Musimy więc ją poprzedzić backslashem:

```
/\./ { print $0 }
```

ta wersja zadziała poprawnie. Jeśli jednak użyjemy wersji z cudzysłowami i zmienną:

```
BEGIN { var="\." }
$0 ~ var { print $0 }
```

to kod przestanie działać. Dzieje się tak dlatego, że backslash jest interpretowany przez łańcuch tekstowy. `\n` zamieni się więc na znak nowej linii, `\t` na tabulator, a `\.` na... zwykłą kropkę. Nam jednak potrzebna jest kropka poprzedzona backslashem. Musimy więc uzyskać backslash w łańcuchu tekstowym. Uzyskujemy to dodając... drugi backslash:

```
BEGIN { var="\\." }
$0 ~ var { print $0 }
```

Interpretacja łańcucha "zje" pierwszy backslash, dając w wyniku `\.`. Interpretacja regexa "zje" drugi backslash i nada kropce pożądane znaczenia. Analogicznie, jeśli chcemy dopasować się do "dosłownego backslasha" to musimy użyć:

```
/\\/
```

w wersji z ukośnikami lub:

```
"\\\\"
```

w wersji z cudzysłowami.

# 4 Funkcje `match` oraz `sub`

Regexy mogą być także wykorzystywane do bardziej zaawansowanego "dopasowania" (funkcja `match`) oraz do zastępowania jednego tekstu drugim (rodzina funkcji `sub`).

Ogólnie cel `match` jest podobny do operatora $\sim$ – sprawdzenie czy istnieje dopasowanie do danego regexa w danym tekście. Jeśli dopasowania nie ma, to funkcja `match` zwraca wartość 0. Jeśli dopasowanie nastąpiło, to funkcja zwraca indeks w poszukiwanym łańcuchu, w którym zaczyna się dopasowany tekst[4]. Wykonanie `match` ustawia też zmienne `RSTART` i `RLENGTH` opisujące gdzie w łańcuchu nastąpiło dopasowanie i jaką ma długość.

To jednak nie koniec. Funkcja `match` może dodatkowo otrzymać nazwę tablicy. Po dopasowaniu w indeksie zerowym tablicy (np. `a[0]`) znajdzie się cały dopasowany tekst. Ważniejsze jest jednak co innego – jeśli w naszym regexie użyliśmy okrągłych nawiasów do określenia grup, to w kolejnych (`a[1]`, `a[2]` itp.) indeksach tablicy znajdą się fragmenty dopasowanego tekstu, które dopasowały się do *poszczególnych grup*. Jest to bardzo silny mechanizm pozwalający "wyłuskać" części dopasowanego łańcuchu, by można było je łatwo wyświetlać/modyfikować. Przykładowo, gdy użyjemy `match`a:

```
match( $0 , /(a|b)([0-9]+)/ , tab )
```

[4]Uwaga! `awk` indeksuje pozycje w łańcuchu od 1! Litera `c` w łańcuchu `abc` jest na pozycji 3, nie 2!

a rekord (czyli $0) zawiera:

```
jakis tekst b2001
```

to w rezultacie w tablicy `tab` znajdzie się:

```
tab[0] = "b2001"
tab[1] = "b"
tab[2] = "2001"
```

Dodatkowo, dla każdej grupy ustawione zostaną dodatkowe 2 elementy tablicy, które przechowują odpowiednik zmiennych `RSTART` i `RLENGTH` dla tej właśnie grupy.

Funkcja `sub` jest podstawową funkcją do podmiany tekstu. Funkcja zamienia pierwsze dopasowanie do podanego regexa w podanym tekście (domyślnie tekstem tym jest `$0`) na podany tekst. Przykładowo wywołanie:

```
sub( /c+/ , "d" , $1 )
```

zamieni pierwszy napotkany "ciąg liter `c`" (`/c+/`) w polu pierwszym (`$1`) na pojedynczą literę `d`. Zauważmy, że najpierw podajemy regex, a na końcu tekst do przeszukania. Funkcje z rodziny `sub` mają taką kolejność, podczas gdy `match` i operator $\sim$ po lewej mają tekst, a po prawej regex.

Funkcja `sub` zamienia *pierwsze* wystąpienie poszukiwanego regexa i *zmienia* oryginalny tekst (w naszym przypadku zmienia bezpośrednio pole pierwsze, czyli `$1`). Ponadto `sub` zwraca liczbę dokonanych podmian (czyli 0 lub 1).

To jednak nie koniec możliwości `sub`. Załóżmy, że chcemy zamienić pierwszą napotkaną liczbę całkowitą na liczbę z przecinkiem (np. zamienić `2` na `2.00`). Innymi słowy, podając tekst do "zamiany" musimy wiedzieć co dokładnie się dopasowało. Jak tego dokonać? Służy do tego znak ampersandu (`&`). Przykładowo:

```
sub( /[0-9]+/ , "&.00" , $1 )
```

zamieni pierwszy napotkany ciąg cyfr na ten sam ciąg cyfr z doklejonym tekstem `".00"`. Znak `&` może być użyty w zamienianym tekście wielokrotnie. Problem pojawia się, gdy chcemy w zamieniamyn tekście uzyskać dosłowny znak `&`. Używamy do tego backslasha, ale sposób jego użycia nie jest trywialny. Problem wynika z dwóch faktów. Po pierwsze, zamieniamy łańcuch podajemy w cudzysłowach, więc backslashe są traktowane specjalnie – dwa backslashe pod rząd zostaną przekazane do `sub` jako jeden backslash. Drugi problem wynika ze sposobu w jaki `sub` interpretuje znaki `&` oraz \. Sposób ten zmieniał się w czasie i może być zależny od tego czy podano do `awk` opcję `--posix`. W jednej z wersji niemożliwe było uzyskanie "dosłownego backslasha po którym następuje dopasowany tekst". W rezultacie backslasha w zamienianym tekście w funkcji `sub` należy używać z ostrożnością.

Ponieważ funkcja `sub` naprawdę zmienia podany tekst, to możemy używać jej wielokrotnie w pętli, by zamieniać kolejne wystąpienia (dopasowania) tego

samego regexa. Są jednak sytuacje, w których to nie wystarczy. W podanym powyżej przypadku zamieniliśmy `"2"` na `"2.00"`. Jednak w kolejnym "przejściu" `"2.00'` zostanie zamienione na `"2.00.00"`, czyli nie o to o co nam chodziło. Oczywiście możemy wycinać już zamienione fragment z tekstu (`match` powie nam na której pozycji nastąpiło dopasowanie i jakie jest długie), po czym możemy "wyciąć" tekst za pomocą `substr`. Jednak istnieje prostsze wyjście.

Rozwiązaniem jest użycie funkcji `gsub`, która działa identycznie do `sub`, ale zamienia wszystkie znalezione dopasowania w przeszukiwanym łańcuchu:

```
gsub( /[0-9]+/ , "&.00" , $1 )
```

Znaki `&` oraz \ działają podobnie (ale niekoniecznie identycznie) jak dla `sub`. Oczywiście jeśli `gsub` zamieni na raz 3 wystąpienia, to za każdym razem w miejsce `&` zostanie podstawione dopasowanie do *aktualnego* wystąpienia. `gsub` zwraca liczbę dokonanych podmian podobnie do `sub`, z tym, że może dokonać dowolnej liczby podmian.

Ostatnią funkcją do omówienia jest `gensub`, która jest ogólniejsza i oferująca więcej możliwości niż `sub` i `gsub`. Funkcja ta jest rozszerzeniem `GNU` i jest dostępna tylko w `gawk`, jednakże w wielu systemach `awk` i tak jest linkiem symbolicznym do `gawk`. Tak też powinno być w laboratorium. Funkcja ta jednak przestanie działać jeśli podamy do `(g)awk` opcje `--traditional` lub `--posix`.

Przejdźmy teraz do samej funkcji. Ma ona trzy cechy charakterystyczne. Po pierwsze `gensub` NIE modyfikuje wskazanego tekstu, a jedynie ZWRACA zmodyfikowaną kopię tekstu. Jeśli chcemy cyklicznie przetwarzać jakiś tekst za pomocą `gensub` trzeba najpierw przypisać wynik funkcji do poszukiwanego tekstu:

```
tekst = gensub( ... , tekst )
```

bo inaczej znajdziemy się w nieskończonej pętli.

Drugą różnicą jest dodatkowy argument podawany jako trzeci. Jest to tekst. Jeśli jego zawartością jest `"g"` lub `"G"`, to `gensub` zachowa się podobnie do `gsub` – zamieni wszystkie napotkane wystąpienia regexa. Jeśli jednak trzeci argument jest liczbą (np. `3`), to `gensub` zamieni tylko to (tutaje trzecie) wystąpienie regexa.

Trzecią różnicą są dodatkowe opcje wpisania dopasowania regexa do zamienianego tekstu. Znak `&` działa podobnie jak poprzednio, ale identyczną rolę pełni też dwuznak `\0`. Żeby `gensub` zobaczył ten dwuznak, to należy go umieścić w tekście jako `\\0`. Najważniejsze jest jednak to, że `gensub` rozpoznaje także dwuznaki od `\1` do `\9` (także tutaj będzie potrzebny podwójny backlasha). W miejsce `\n` zostanie umieszczone dopasowanie do n-tego nawiasu regexa. Przykładowo jeśli chcemy zamienić wszystkie wystąpienia w tekście na zasadzie zamiany `"2,56 PLN"` na `"2 zl i 56 gr"`, to możemy napisać:

```
gensub( /([0-9]+),([0-9]+) PLN/ , "\\1 zl i \\2 gr" , "g" ,
$0 )
```