Sed w przykładach, część trzecia

1. Wyższy poziom seda: reorganizacja danych, styl seda

Muskularny sed

W drugiej części artykułu pokazałem przykłady, w których przedstawiłem w jaki sposób działa sed, jednak niewielka cześć z tych skryptów robiła coś chociaż po części pożytecznego. W tym ostatnim artykule czas to zmienić i pokazać seda w dobrym świetle. Zaprezentuję kilka przykładów, w których nie tylko ukażę moc jaka w nim drzemie, ale również, że potrafi robić świetne i bardzo przydatne rzeczy. Dla przykładu, w drugiej części tego artykułu pokaże w jaki sposób zaprojektowałem skrypt seda, który konwertuje plik .QIF programu finansowego Intuit Quicken w ładnie sformatowany plik tekstowy. Przed wykonaniem tego skryptu przeglądniemy kilka mniej skomplikowanych przykładów.

Tłumaczenie/przesunięcie tekstu

Nasz pierwszy skrypt konwertuje tekst w formacie UNIX do formatu systemu DOS/Windows. Jak zapewne wiecie, pliki tekstowe systemu DOS/Windows posiadają znak CR powrotu karetki (carriage return) oraz LF, wysuw wiersza (line feed) na końcu każdej linii. Natomiast pliki tekstowe systemu UNIX posiadają jedynie line feed. Może okazać się, że przyjdzie kiedyś czas, że będziemy musieli przenieść część tekstów UNIX-a do systemu Windows, a ten skrypt wykona za nas niezbędną konwersję.

Listing 1.1: Konwersja formatu pomiędzy UNIX-em a Windowsem

```
$ sed -e 's/$/\r/' myunix.txt > mydos.txt
```

W skrypcie tym regularne wyrażenie '$' będzie odpowiadało końcowi linii, a oznaczenie '\r' powie sedowi, aby umieścił CR zaraz przed nim. Znak CR zostaje umieszczony przed znakiem LF, a kombinacja CR/LF kończy każdą linijkę. Należy jednak mieć na uwadze, że oznaczenie '\r' zostanie zastąpione znakiem CR tylko wtedy, gdy używamy GNU sed w wersji 3.02.80 lub nowszej. Jeżeli nie mamy jeszcze zainstalowanej wersji GNU sed 3.02.80, należy zobaczyć mój pierwszy artykuł o sedzie, aby zaczerpnąć informacji o tym jak zdobyć najnowszą jego wersję.

Nie mogę powiedzieć ile razy ściągnąłem przykładowe skrypty czy pliki źródłowe C tylko po to, aby sprawdzić czy są w formacie DOS/Windows. Z reguły większości programów nie przeszkadza format pliku DOS/Windows zakańczany znakami CR/LF, jednak części z nich tak. Najbardziej znanym jest bash, który nie przyjmuje danych gdy tylko rozpozna znak CR. Poniższe polecenie skonwertuje plik tekstowy z formatu DOS/Windows do formatu UNIX:

Listing 1.2: Konwersja kodu C z formatu Windows do UNIX

```
$ sed -e 's/.$//' mydos.txt > myunix.txt
```

Sposób działania tego skryptu jest prosty: zastępcze regularne wyrażenie oznacza ostatni znak w linii, którym jest znak CR. Nie zastępujemy go niczym, co oznacza, że zostaje on całkowicie skasowany w pliku wyjściowym. Jeżeli użyjemy tego skryptu i zauważymy, że każdy znak, każdej ostatniej linijki został skasowany, jako parametr podaliśmy plik tekstowy, który już był w formacie UNIX. Zatem nie ma potrzeby poddawać go działaniu tego skryptu.

Odwracanie linii

Teraz kolejny przydatny skrypt. Zadaniem tego skryptu jest odwracanie linii w tekście, podobnie do polecenia "tac", które jest obecne w większości dystrybucji systemu Linux. Nazwa "tac" może być nieco myląca, ponieważ "tac" nie odwraca pozycji znaków w linii (lewo i prawo), lecz częściej pozycję linii w pliku (góra i dół). Zilustrujemy korzystanie z polecenia "tac" na takim oto przykładzie:

Listing 1.3: Zawartość przykładowy plik

```
foo
bar
oni
```

....tworzy następujący plik wynikowy:

Listing 1.4: Zawartość pliku wynikowy

```
oni
```

```
bar
foo
```

Możemy uzyskać taki sam rezultat używając następującego skryptu seda:

Listing 1.5: To samo zadanie z wykorzystaniem skryptu

```
$ sed -e '1!G;h;$!d' forward.txt > backward.txt
```

Skrypt ten będzie przydatny, jeśli używamy systemu FreeBSD, który nie posiada komendy "tac". Dobrym pomysłem byłoby poznanie jak działa powyższy skrypt, zatem przeprowadźmy drobiazgową analizę.

Wytłumaczenie działania skryptu odwracającego tekst

Na początku skrypt ten zawiera trzy oddzielne komendy seda, oddzielone średnikami: '1!G', 'h' i '$!d'. Teraz czas na zrozumienie adresów użytych dla pierwszego i trzeciego polecenia. W pierwszym poleceniu, gdzie znajduje się wyrażenie '1!G', polecenie G powinno zostać zastosowane jedynie do pierwszej linii. Jednak dodatkowo znajduje się w tym poleceniu znak '!', który ma za zadanie negować adres. Oznacza to ni mniej ni więcej, że polecenie 'G' zostanie użyte do wszystkich linii oprócz pierwszej. Dla polecenia '$!d' mamy podobną sytuację. Jeżeli komenda miałaby postać '$d', komenda 'd' byłaby zatwierdzana jedynie do ostatniej linii w pliku (adres '$' jest prostym sposobem na określenie ostatniej linii). Jednak ze znakiem '!' polecenie w postaci '$!d' zastosuje komendę 'd' do wszystkich linii oprócz ostatniej. Teraz wystarczy zrozumieć co oznaczają poszczególne komendy.

Kiedy wykonamy nasz skrypt na pliku tekstowym przedstawionym powyżej, pierwszą komendą, która zostanie wykonana będzie komenda 'h'. Polecenie to mówi sedowi o skopiowaniu zawartości miejsca ze wzorem (bufor, w którym przetrzymywana jest aktualna linia, która jest przetwarzana) do miejsca przetrzymywania (bufor czasowy). Po tym zostaje wykonana komenda 'd', która kasuje wpis "foo" ze wzoru, więc nie jest on już wyświetlany po wykonaniu wszystkich poleceń na tej linii.

Teraz druga linia. Po tym jak wyrażenie "bar" zostaje odczytane, komenda 'G' zostaje wykonana, która ma za zadanie dodanie zawartości przestrzeni, w której są przetrzymywane dane ("foo\n") do przestrzeni wzorów ("bar\n"). Dzięki temu w naszej przestrzeni wzorów znajdzie się ciąg "bar\n\foo\n". Polecenie 'h' dodaje ten ciąg dla bezpieczeństwa z powrotem do przestrzeni przetrzymywania (hold space), natomiast 'd' kasuje linię z przestrzeni wzoru, tak aby nie była wyświetlana.

Dla ostatniej linii zawierającej "oni" powtarzane są te same kroki. Należy się spodziewać, że zawartość przestrzeni wzoru nie została skasowana (w związku z poleceniem '$!' przed 'd') i jej zawartość została wyświetlona w stdout.

Pora na bardziej zaawansowaną konwersję danych z wykorzystaniem seda.

Magia seda w QIF

W ciągu ostatnich kilku tygodni rozmyślałem nad zakupieniem egzemplarza programu Quicken do kontroli moich kont bankowych. Quicken jest bardzo przydatnym programem finansowym i na pewno wykonałby swoje zadanie znakomicie. Jednak po przemyśleniu wszystkiego, zdecydowałem, że mogę sam napisać program, który będzie kontrolował moją książkę wydatków. W końcu jestem deweloperem oprogramowania!

Opiekuję się małym programem (używającym awk) kontrolującym wydatki, który kalkuluje dochody i rozchody poprzez przetwarzanie tekstu zawierającego wszystkie moje transakcje. Po kilku modyfikacjach, poprawiłem program, abym mógł śledzić różne kategorie debetów i kredytów, w taki sam sposób jak to potrafi Quicken. Jednak chciałem dodać jeszcze jedną opcję do mojego programu. Niedawno zmieniłem bank, na taki który posiada interfejs obsługi przez Internet. Pewnego dnia zauważyłem, że istnieje możliwość ściągnięcia informacji na temat mojego konta ze strony banku w formacie .QIF programu Quicken. W bardzo krótkim czasie zdecydowałem, że byłoby bardzo dobrze, gdybym mógł konwertować te informacje do pliku tekstowego.

Opowieść o dwóch formatach

Przed poznaniem formatu QIF, spójrzmy jak wygląda format mojego pliku checkbook.txt:

Listing 1.6: Przykład pliku checkbook.txt

```
28 Aug 2000     food    -       -       Y     Supermarket             30.94
25 Aug 2000     watr    -       103     Y     Check 103               52.86
```

W moim pliku wszystkie pola oddzielone są od siebie jednym lub kilkoma tabulatorami, z jedną transakcją na linię. Następne pole po dacie pokazuje typ rozchodu (lub znak "-" gdy jest to element przychodu). Trzecie pole wskazuje typ przychodu (lub znak "-" gdy jest to element rozchodu). Następnie znajduje się pole sprawdzające numer (ponownie, gdy jest puste znajdziemy znak "-"), pole zrealizowania transakcji ("Y" lub "N"), komentarz oraz ilość dolarów. Teraz jesteśmy gotowi do przyjrzenia się formatowi .QIF. Gdy otworzyłem ściągnięty plik QIF w przeglądarce tekstu, oto co zobaczyłem:

Listing 1.7: Przykład pliku .QIF

```
!Type:Bank
D08/28/2000
T-8.15
N
PCHECKCARD SUPERMARKET
^
D08/28/2000
T-8.25
N
PCHECKCARD PUNJAB RESTAURANT
^
D08/28/2000
T-17.17
N
PCHECKCARD SUPERMARKET
```

Po przejrzeniu pliku, nie było trudnym zadaniem zrozumienie tego formatu. Pomijając pierwszą linijkę, format ma następujący wzór:

Listing 1.8: Format pliku

```
D<data>
T<kwota transakcji>
N<numer czeku>
P<opis>
(znak podziału pól)
^
```

Rozpoczęcie procesu

Kiedy napotkamy na jakieś trudności w podobnym projekcie realizowanym za pomocą seda, nie należy się zniechęcać. Sed pozwala stopniowo przekształcać dane do ich finalnej formy. W miarę postępów, możemy udoskonalać nasz skrypt, aż dane wynikowe będą wyglądały w taki sposób jaki sobie założyliśmy. Wcale nie musimy uzyskać takiego rezultatu za pierwszym razem.

Na początku stworzyłem plik `qiftrans.sed` i zacząłem przekształcać dane:

Listing 1.9: qiftrans.sed

```
1d
/^^/d
s/[[:cntrl:]]//g
```

Pierwsze polecenie '1d' usuwa pierwszą linię, natomiast drugie polecenie usuwa te brzydkie znaki '^' z pliku wyjściowego. Ostatnia linia usuwa wszystkie znaki kontrolne, które mogą znajdować się w pliku. Od kiedy mam do czynienia z obcymi formatami pliku, chcę wyeliminować ryzyko natknięcia się na jakiekolwiek znaki kontrolne. Jak na razie idzie nam dobrze. Teraz pora na dodanie kilku funkcji naszemu podstawowemu skryptowi:

Listing 1.10: Poprawiony podstawowy skrypt

```
1d
/^^/d
s/[[:cntrl:]]//g
/^D/ {
  s/^D\(.*\)/\1\tOUTY\tINNY\t/
        s/^01/Jan/
        s/^02/Feb/
        s/^03/Mar/
        s/^04/Apr/
        s/^05/May/
        s/^06/Jun/
```

```
        s/^07/Jul/
        s/^08/Aug/
        s/^09/Sep/
        s/^10/Oct/
        s/^11/Nov/
        s/^12/Dec/
        s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:
}
```

Najpierw dodaję adres '/^D/', aby sed rozpoczął przetwarzanie tylko wtedy gdy natknie się na pierwszy znak pola danych QIF, 'D'. Wszystkie polecenia w nawiasach klamrowych zostaną wykonane w porządku, gdy sed wczyta taką linię do swojej przestrzeni wzorów.

Pierwsza linia w nawiasach klamrowych przekształci się z takiej jak ta:

Listing 1.11: Pierwsza linia przed zmianą

```
D08/28/2000
```

w linię wyglądającą tak:

Listing 1.12: Pierwsza linia po zmianie

```
08/28/2000  OUTY  INNY
```

Oczywiście taki format nie jest jeszcze idealny, jednak nie ma się czym przejmować. Będziemy stopniowo usprawniać zawartość przestrzeni wzoru w miarę postępów. Kolejne 12 linii posiada efekt internetu przekształcenia danych do formatu trzech liter, z ostatnią linią usuwającą trzy ukośniki z danych. Zakańczamy taką linią:

Listing 1.13: Ostatnie spojrzenie na linię

```
Aug 28 2000  OUTY  INNY
```

Pola OUTY i INNY zostają podane jako zmienne i zostaną zastąpione później. Nie mogę ich jeszcze teraz określić, ponieważ w przypadku gdy kwota dolarów będzie ujemna będę chciał ustawić pola OUTY i INNY na "misc" i "-", jednak w przypadku gdy kwota dolarów będzie dodana, pola te będę chciał zamieć odpowiednio na "-" i "inco". Ponieważ kwota ta nie została jeszcze odczytana, muszę użyć zmiennych na pewien czas.

Doskonalenie

Teraz pora na dalsze udoskonalenie:

Listing 1.14: Dalsze udoskonalenia

```
1d
/^^/d
s/[[:cntrl:]]//g
/^D/ {
        s/^D\(.*\)/\1\tOUTY\tINNY\t/
        s/^01/Jan/
        s/^02/Feb/
        s/^03/Mar/
        s/^04/Apr/
        s/^05/May/
        s/^06/Jun/
        s/^07/Jul/
        s/^08/Aug/
        s/^09/Sep/
        s/^10/Oct/
        s/^11/Nov/
        s/^12/Dec/
        s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:
        N
        N
        N
        s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/
        s/NUMNUM/-/
        s/NUM\([0-9]*\)NUM/\1/
        s/\([0-9]\),/\1/
}
```

Kolejne siedem linii jest trochę bardziej skomplikowanych, więc opiszę je bardziej dokładnie. Na początku widzimy trzy polecenia 'N' w kolumnie. Polecenie 'N' mówi sedowi, aby odczytać następną linię na wyjściu oraz dodać ją do aktualnej przestrzeni wzoru. Trzy polecenia 'N' powodują dodanie kolejnych trzech linii do naszej aktualnej przestrzeni wzoru, więc nasza linia wygląda następująco:

Listing 1.15: Nowy wygląd linii

```
28 Aug 2000  OUTY  INNY  \nT-8.15\nN\nPCHECKCARD SUPERMARKET
```

Przestrzeń wzoru seda została lekko oszpecona, więc musimy usunąć ekstra dodane nowe linie oraz przeprowadzić kilka dodatkowych czynności. Aby tego dokonać użyjemy polecenia zamiany. Wzór, który chcemy uzyskać wygląda następująco:

Listing 1.16: Usuwanie dodatkowych linii oraz przeprowadzanie kilku poprawek

```
'\nT.*\nN.*\nP.*'
```

Zostanie to dopasowane do nowej linii, poprzedzonej przez 'T', poprzedzonego przez zero lub kilka znaków, poprzedzonych nową linią, poprzedzonej przez 'N', poprzedzonego przez dowolną liczbę znaków i nowych linii, poprzedzonych przez 'P', poprzedzonej dowolną liczbą znaków. Ufff! Regexp zaznaczy całą zawartość trzech linii, które właśnie dodaliśmy do przestrzeni wzorów. Jednak chcemy tylko przeformatować ten kawałek, a nie przemieszczać go całkowicie. Znak dolara, numer czeku (jeśli jest) oraz opis potrzebny do ponownego pojawienia się w naszym łańcuchu zastępczym. Aby tego dokonać, otaczamy "interesujące części" przy pomocy nawiasów z ukośnikami (backslashami), tak abyśmy mogli odnosić się do nich w naszym łańcuchu zastępczym (używając '\1', '\2' oraz '\3', aby powiedzieć sedowi gdzie je umieszczać). Poniżej znajduje się końcowa wersja polecenia:

Listing 1.17: Ostateczna wersja polecenia

```
s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/
```

Ta komenda przekształci naszą linię do takiej oto postaci:

Listing 1.18: Rezultat użycia powyższego polecenia

```
28 Aug 2000  OUTY  INNY  NUMNUM    Y     CHECKCARD SUPERMARKET AMT-8.15AMT
```

Podczas gdy będziemy poprawiać tę linię, znajdziemy parę rzeczy, które na pierwszy rzut oka wyglądają... interesująco. Pierwszą z nich jest śmieszny łańcuch znaków "NUMNUM" - czemu on służy? Dowiem się tego gdy przyglądniemy się dwóm kolejnym liniom ze skryptu seda, które zastępują wyrażenie "NUMNUM" znakami "-", w czasie gdy "NUM" <number>"NUM" zostanie zastąpiony przez <number>. Jak można zauważyć, otaczanie numeru czeku przez te głupie tagi pozwala nam dogodnie wpisywać "-", jeżeli dane pole jest puste.

Ostatnie poprawki

Ostatnia linia usuwa przecinek poprzedzający numer. Przekształca to kwotę dolarów z postaci "3,231.00" do "3231.00", którego sam używam. Przyszła pora, aby obejrzeć ostateczną, produkcyjną wersję skryptu:

Listing 1.19: Ostateczna produkcyjna wersja skryptu

```
1d
/^^/d
s/[[:cntrl:]]//g
/^D/ {
  s/^D\(.*\)/\1\tOUTY\tINNY\t/
  s/^01/Jan/
  s/^02/Feb/
  s/^03/Mar/
  s/^04/Apr/
  s/^05/May/
  s/^06/Jun/
  s/^07/Jul/
  s/^08/Aug/
  s/^09/Sep/
  s/^10/Oct/
  s/^11/Nov/
  s/^12/Dec/
  s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:
  N
  N
```

```
  N
  s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/
  s/NUMNUM/-/
  s/NUM\([0-9]*\)NUM/\1/
  s/\([0-9]\),/\1/
  /AMT-[0-9]*.[0-9]*AMT/b fixnegs
  s/AMT\(.*\)AMT/\1/
  s/OUTY/-/
  s/INNY/inco/
  b done
:fixnegs
  s/AMT-\(.*\)AMT/\1/
  s/OUTY/misc/
  s/INNY/-/
:done
}
```

Dodatkowe jedenaście linii używa zastępowania oraz funkcji rozgałęziania, aby dane wynikowe były jak najbardziej idealne. Chcemy się przyjrzeć najpierw tej linii:

Listing 1.20: Pierwsza linia wymagająca zainteresowania

```
        /AMT-[0-9]*.[0-9]*AMT/b fixnegs
```

Linia ta zawiera polecenie rozgałęzienia, który jest w formacie "/regexp/b label". Jeżeli nasza przestrzeń wzoru dopasuje regexp, sed rozgałęzi się do etykiety fixnegs. Powinniśmy łatwo dostrzec tą etykietę, gdyż pojawi się jako ":fixnegs" w kodzie. Jeżeli regexp nie pasuje, przetwarzanie będzie postępowało dalej w normalny sposób z kolejnymi poleceniami.

Gdy już rozumiemy działanie poleceń, pora przyjrzeć się rozgałęzieniom. Jeżeli przyglądniemy się regularnemu wyrażeniu rozgałęzienia, zauważymy, że zostanie on dopasowany do łańcucha 'AMT', poprzedzonego przez '-', poprzedzonego przez dowolną liczbę cyfr, przez '.', kolejny raz dowolną ilość cyfr i ponownie przez 'AMT'. Jestem pewny, że domyśliłeś się, że ten regexp radzi sobie szczególnie z ujemnymi kwotami dolarów. Wcześniej, otoczyliśmy kwotę dolarów przez łańcuch 'AMT', abyśmy mogli łatwiej go odnaleźć później. Ponieważ regexp oznacza jedynie kwotę dolarów zaczynających się przez '-' nasze rozgałęzienie zadziała, jedynie w wypadku gdy my będziemy sobie radzić z debetem. Jeżeli radzimy sobie z debetem, pole OUTY powinno zostać ustawione na 'misc', a INNY powinno zostać ustawione na '-'. Negatywny znak na przedzie debetu powinien zostać usunięty. Jeżeli prześledzimy kod, zauważymy, że tak dokładnie się dzieje. Jeżeli rozgałęzienie nie zostanie wykonane, pole OUTY zostaje zastąpione przez '-', a pole INNY przez 'inco'. Skończyliśmy! Nasza linia wynikowa jest teraz idealna:

Listing 1.21: Idealna linia wynikowa

```
28 Aug 2000  misc  -  -       Y     CHECKCARD SUPERMARKET  -8.15
```

Nie rozkojarzajmy się

Konwersja danych przy pomocy seda nie jest wcale taka trudna, gdy dobrze prześledzimy problem. Nie próbujmy wykonać wszystkiego za pomocą jednej komendy seda lub wszystkiego naraz. Zamiast tego, stopniowo dopracowujmy nasz projekt, aż do momentu, w którym będziemy zadowoleni z wyników. Sed oferuje wiele możliwości. Mam nadzieje, że zostałeś zaznajomiony z jego działaniem oraz że będziesz cały czas uczył się jego możliwości, aby stać się mistrzem seda!

- Przeczytaj inne artykuły seda z developerWorks: Wspólne wątki: Sed w przykładach, Część 1 i część 2.
- Sprawdź doskonale napisane przez Erica Pementa FAQ seda.
- Źródła seda można znaleźć na ftp://ftp.gnu.org/pub/gnu/sed.
- Eric Pement posiada również poręczną listę skryptów seda w jednej linijce, które każdy użytkownik aspirujący na mistrza seda powinien poznać.