Awk w przykładach, część pierwsza

Spis treści: 1. Wprowadzenie do wspaniałego języka o dziwnej nazwie ▾

1. Wprowadzenie do wspaniałego języka o dziwnej nazwie

W obronie awk

W tej serii artykułów czytelnik nauczy się wydajnie programować w Awk. Język ten nie ma szczególnie ładnej lub "modnej" nazwy, a nazwa jego wersji GNU, gawk, brzmi wprost dziwacznie. Osoby niezaznajomione z Awk mogą z tego powodu kojarzyć go z chaosem kodu dość przestarzałego, aby nawet najbardziej doświadczonego guru Uniksa doprowadzić na skraj szaleństwa (zmuszając go do ciągłego wykrzykiwania "kill -9!" w biegu do ekspresu do kawy).

Jasne, Awk nie ma zbyt dobrej nazwy. Ale to wspaniały język. Jego nazwa oznacza "Aho, Weinberger, Kernighan" - trzech spośród najsławniejszych informatyków. Został wyspecjalizowany do przetwarzania tekstu i przygotowania raportów, jednak zawiera wiele dobrze zaprojektowanych cech które umożliwiają poważne programowanie. Dodatkowo, w przeciwieństwie do niektórych innych języków, składnia Awk jest znajoma, zapożycza najlepsze cechy języków takich jak C, python i bash (aczkolwiek w zasadzie Awk powstało zarówno przed Pythonem jak i bashem). Awk jest jednym z tych języków, które po opanowaniu stają się kluczowym elementem każdego arsenału programistycznego.

Pierwszy awk

Listing 1.1: Pierwszy awk

```
$ awk '{ print }' /etc/passwd
```

Powinna się teraz ukazać zawartość pliku `/etc/passwd`. Teraz wyjaśnię, co uczynił Awk. Kiedy wywołaliśmy Awk, podaliśmy `/etc/passwd` jako jego wejście. Polecenie wykonało rozkaz print dla każdego wiersza `/etc/passwd`po kolei, a całe wyjście zostało przekazane na standardowe wyjście - co doprowadziło do wyniku identycznego z wykonaniem cat dla `/etc/passwd`.

Teraz z kolei nastąpi opis bloku { print }. W Awk, nawiasy klamrowe są wykorzystywane do wiązania ze sobą bloków kodu - podobnie jak w C. Wewnątrz takiego bloku mamy pojedyncze polecenie print. Występując samodzielnie powoduje ono wypisanie całego aktualnego wiersza.

Listing 1.2: Wypisywanie aktualnego wiersza; Wypisywanie pustych linii

```
$ awk '{ print $0 }' /etc/passwd
$ awk '{ print "" }' /etc/passwd
```

W Awk zmienna $0 reprezentuje cały aktualny wiesz, więc print oraz print $0 robią dokładnie to samo.

Listing 1.3: Wypełnianie ekranu odrobiną tekstu

```
$ awk '{ print "hiya" }' /etc/passwd
```

Przykłady demonstrujące selekcyjne wypisywanie pól

Listing 1.4: Wypisanie pierwszego i trzeciego pola dla separatora ':'

```
$ awk -F":" '{ print $1 $3 }' /etc/passwd
halt7
operator11
root0
shutdown6
sync5
bin1
....itd.
```

Listing 1.5: Wypisanie pierwszego i trzeciego pola oddzielonych znakiem spacji

```
$ awk -F":" '{ print $1 " " $3 }' /etc/passwd
```

Listing 1.6: Wypisanie formatowanych wierszy

```
$ awk -F":" '{ print "username: " $1 "\t\tuid:" $3 }' /etc/passwd
username: halt          uid:7
username: operator      uid:11
username: root          uid:0
username: shutdown      uid:6
username: sync          uid:5
username: bin           uid:1
....itd.
```

Skrypty zewnętrzne

Listing 1.7: Przykładowy skrypt

```
BEGIN { FS=":" }
{ print $1 }
```

Listing 1.8: Wykonywanie przykładowego skryptu

```
$ awk -f script.awk /etc/passwd
```

Różnicą pomiędzy tym i poprzednim rozwiązaniem jest to, jak oznaczyliśmy separator pól. W tym skrypcie separator został wybrany w samym kodzie (przez ustawienie zmiennej FS), podczas gdy w poprzednim przypadku FS zostało ustawione za pomocą opcji wiersza poleceń -F":". W ogólnym przypadku najlepiej jest wybierać separator wewnątrz skryptu, głównie dlatego, że dzięki temu nie trzeba pamiętać o argumencie dla polecenia. Zmienna FS zostanie dokładniej opisana dalej w tym artykule.

Bloki BEGIN oraz END

Zwykle awk wykonuje każdy blok danego skryptu raz dla każdego wiersza na wejściu. Podczas programowania występuje jednak wiele sytuacji, kiedy potrzebna jest inicjalizacja zanim Awk zacznie przetwarzać tekst. Ze względu na takie przypadki Awk pozwala na zdefiniowanie bloku BEGIN. Użyliśmy go w poprzednim przykładzie. Ponieważ blok ten jest wykonywany zanim Awk zacznie przetwarzać plik wejściowy to jest on świetnym miejscem na inicjalizację zmiennej FS (field separator - separator pól), wypisanie nagłówka lub zdefiniowanie innych zmiennych globalnych które wykorzystane będą w dalszej części programu.

Awk pozwala również na istnienie innego specjalnego bloku, nazywanego END. Awk wykonuje go po przetworzeniu wszystkich wierszy wejścia. W typowym przypadku blok END jest wykorzystywany do wykonania końcowych obliczeń lub wypisania podsumowania, które powinno pokazać się na końcu wyjścia.

Wyrażenia regularne

Listing 1.9: Wypisane zostaną tylko wiersze zgodne z wyrażeniem regularnym

```
/foo/ { print }
/[0-9]+\.[0-9]*/ { print }
```

Operatory

Listing 1.10: Jeżeli pierwsze pole jest równe...

```
$1 == "fred" { print $3 }
```

Listing 1.11: Jeżeli piąte pole spełnia wyrażenie regularne, wypisz trzecie

```
$5 ~ /root/ { print $3 }
```

Instrukcje warunkowe

Listing 1.12: Powyższy przykład z zastosowaniem instrukcji warunkowej if

```
{
    if ( $5 ~ /root/ ) {
        print $3
```

```
    }
}
```

Obydwa skrypty działają tak samo. W pierwszym przykładzie wyrażenie warunkowe zostało umieszczone na zewnątrz bloku, natomiast w drugim blok jest wykonywany dla każdego wiersza ale dzięki instrukcji if wykonuje polecenie print tylko w niektórych przypadkach. Obydwa sposoby są dostępne i można wybrać ten, który najlepiej pasuje do innych części skryptu.

Listing 1.13: Zagnieżdżanie instrukcji if

```
{
    if ( $1 == "foo" ) {
        if ( $2 == "foo" ) {
            print "uno"
        } else {
            print "one"
        }
    } else if ($1 == "bar" ) {
        print "two"
    } else {
        print "three"
    }
}
```

Listing 1.14: Wypisz, jeżeli wyrażenie regularne nie pasuje do wiersza

```
! /matchme/ { print $1 $3 $4 }
```

Listing 1.15: Powyższy przykład z wykorzystaniem instrukcji warunkowej if

```
{
    if ( $0 !~ /matchme/ ) {
        print $1 $3 $4
    }
}
```

Obydwa skrypty wypiszą tylko te wiersze, które nie zawierają sekwencji matchme. Ponownie, obydwa sposoby dają taki sam efekt i można wybrać dowolny.

Listing 1.16: Wypisanie wierszy zawierających pola pasujące do foo oraz bar

```
( $1 == "foo" ) && ( $2 == "bar" ) { print }
```

Ten przykład wypisze tylko te wiersze, w których pole pierwsze jest równe foo, a pole drugie jest równe bar.

Zmienne liczbowe!

Listing 1.17: Zmienne liczbowe!

```
BEGIN { x = 0 }
/^$/ { x = x + 1 }
END { print x }
```

W bloku BEGIN inicjalizujemy zmienną x jako zero. Następnie, za każdym razem, gdy Awk napotka pusty wiersz, wykona instrukcję x=x+1, zwiększając x. Po przetworzeniu wszystkich wierszy zostanie wykonany blok END, wypisując liczbę napotkanych pustych wierszy.

Zmienne łańcuchowe

Listing 1.18: Przykładowe pole

```
2,01
```

Listing 1.19: Przykład wyrażenia arytmetycznego

```
{ print ($1^2)+1 }
```

Po kilku eksperymentach można odkryć, że jeżeli dana zmienna nie zawiera prawidłowej liczby, to podczas wyliczania wyrażeń matematycznych Awk będzie ją traktował jako zero. Dla podanego przykładu otrzymamy wynik (2,01^2)+1 = 5,0401.

Mnóstwo operatorów

Kolejną zaletą Awk jest to, że zawiera pełen zestaw operatorów matematycznych. Poza standardowym dodawaniem, odejmowaniem, mnożeniem i dzieleniem, Awk pozwala na wykorzystanie zademonstrowanej już potęgi "^", operatora modulo (reszty) "%" oraz kilku przydatnych operatorów przypisania zapożyczonych z C.

Dostępne mamy także operatory dekrementacji i inkrementacji (w wersjach zarówno pre jak i post): i++, --foo oraz operatory przypisania z operacją na przykład dodawania czy mnożenia (a+=3, b*=2, c/=2.2, d-=6.2, a^=2, b%=4).

Separatory pól

Awk ma również własny zestaw zmiennych specjalnych. Część z nich pozwala na dopasowania sposobu, w jaki działa Awk, natomiast inne pozwalają uzyskać cenne informacje o wejściu. Jedną z tych zmiennych, FS, już wykorzystaliśmy. Jak już było powiedziane, pozwala ona określić jakiej sekwencji znaków Awk powinien oczekiwać pomiędzy polami. Kiedy wejściem było `/etc/passwd` FS było ustawione na ":". Wprawdzie to załatwiło sprawę, ale FS pozwala na nawet większą elastyczność.

Listing 1.20: Inny separator pól

```
FS="\t+"
```

Powyżej wykorzystaliśmy znak specjalny dla wyrażeń regularnych, "+", który oznacza "jeden lub więcej poprzednich znaków".

Listing 1.21: Ustawianie FS na spację

```
FS="[[:space:]+]"
```

To wprawdzie załatwia sprawę, ale jest niepotrzebne. Dlaczego? Bo domyślnie FS jest ustawione na pojedynczą spację, co Awk interpretuje jako "jedna lub więcej spacji". W tym konkretnym przypadku domyślne ustawienie FS jest dokładnie tym, co było potrzebne.

Listing 1.22: Przykładowy separator pól

```
FS="foo[0-9][0-9][0-9]"
```

Wskaźnik liczby pól

Listing 1.23: Jeżeli liczba pól większa od dwóch...

```
{
    if ( NF > 2 ) {
        print $1 " " $2 ":" $3
    }
}
```

Wskaźnik numeru wiersza

Listing 1.24: Dla wierszy począwszy od 11...

```
{
    if ( NR > 10 ) {
        print "ok, now for the real information!"
    }
}
```

Awk dostarcza dodatkowych zmiennych, które mogą zostać wykorzystane do wielu celów. Więcej z nich zostanie opisane w dalszych artykułach.

To już koniec wstępnego omówienia Awk. W dalszym ciągu serii poznamy bardziej zaawansowane cechy Awk, natomiast na koniec zbudujemy praktyczną aplikację Awk. W międzyczasie można skorzystać z wypisanych poniżej zasobów.

2. Zasoby

Przydatne linki

- Inne artykuły o Awk autorstwa Daniela na developerWorks: Common threads: Awk by example, Cześć druga oraz Część trzecia.
- Książka O'Reilly sed & awk, 2nd Edition jest doskonałym wyborem dla tych, którzy wolą papier.
- Konieczne należy odwiedzić FAQ grupy comp.lang.awk. Zawiera ono również mnóstwo dodatkowych linków związanych z Awk.
- Samouczek AWK Patricka Hartigana zawiera wiele przydatnych skryptów
- Podręcznik użytkownika GNU Awk zawiera mnóstwo użytecznych informacji.