J.NAWROCKI, M. ANTCZAK, H. ĆWIEK, W. FROHMBERG, A. HOFFA, M. KIERZYNKA, S. WĄSIK

AWK - WPROWADZENIE

SKŁADNIA WYWOŁANIA AWK:

```
awk [-Fs] "program" [plik1 plik2...]      # komendy zapisywane są w linii poleceń
                                          # DOSa.
awk 'program{print "foo"}' plik1          # pojedyncze cudzysłowy otaczają argumenty
                                          # wywołania, które mogą zawierać podwójne
                                          # cudzysłowy.
    # Uwaga: Dopóki AWK akceptuje pojedyncze cudzysłowy znajdujące się wokół argumentów
    # podawanych z linii poleceń systemu operacyjnego, oznacza to, że ścieżki do plików, które
    # zawierają w sobie pojedyncze cudzysłowy nie są rozpoznawane przez AWK, pomimo nawet,
    # że są poprawnymi ścieżkami z punktu widzenia systemu operacyjnego. Aby AWK rozpoznał
    # plik foo'bar to podana nazwa musi być zapisana w następujący sposób foo'"'"'bar.

awk [-Fs] -f plik_źródłowy_programu [plik1 plik2...] # komendy zapisywane są w
                                                     # linii poleceń DOSa.
```

Jeżeli plik1 jest pominięty wtedy AWK zakłada, że analizowane dane są pobierane ze standardowego wejścia (konsola systemowa).
Argument wywołania oznaczony jako `-Fz` ustawia separator pól *FS* na znak *"z"*.

Plik źródłowy AWK składa się z następujących reguł przetwarzania:
"wzorzec {instrukcje}"

- Jeżeli *{instrukcje}* są pominięte, w regule przetwarzania, wtedy domyślnie jest wykonywane wypisanie wiersza na standardowe wyjście *{print $0}*.
- Jeżeli *"wzorzec"* jest pominięty w regule przetwarzania, wtedy każdy wiersz pliku wejściowego jest poddawany działaniu instrukcji zdefiniowanych w bloku *{instrukcje}*.

Pola są separowane najczęściej przez jedną lub więcej spacji lub tabulatorów: "pole1 pole2".
Jeżeli poniżej opisane przykłady poleceń znajdują się w pliku źródłowym, a nie są uruchamiane z linii poleceń, wtedy mogą zostać pominięte podwójne cudzysłowy.

PODSTAWOWE POLECENIA AWK:

`"NR == 5" plik` — wypisuje w rezultacie 5 wiersz (linia od góry) z pliku o nazwie `plik`.
Uwaga: *"=="* oznacza operator porównania.

`{FOO = 5}` — przypisanie do zmiennej FOO wartości "5".
Uwaga: pojedyncze *"="* oznacza operator przypisania.

`"$2 == 0 {print $1}"` — Jeżeli zawartość drugiego pola (pola są indeksowane od 1) jest równa 0 to w wyniku, na wyjściu, zostanie wypisana zawartość pola pierwszego.

`"$3 < 10"` — Jeżeli numeryczna zawartość trzeciego pola jest mniejsza od 10, wtedy wiersz

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

	zawierający to pole zostaje wypisany na wyjście. (numeryczne porównanie).
`'$3 < "10"'`	podczas porównywania łańcuchów wykorzystywane są pojedyncze cudzysłowy.
`-f pgmfile [$3 < "10"]`	wykorzystywane jest polecenie *"-f pgmfile"* podczas porównywania łańcuchów.
`"$3 ~ /regexp/"`	wypisanie wiersza, w którym trzecie pole spełnia opisane wyrażenie regularne */regexp/*.
`'$3 ~ "regexp"'`	*regexp* może wystąpić w łańcuchu oznaczonym dwoma cudzysłowami. Podwójne cudzysłowy mogą zastępować *„backlashes"* w wyrażeniach regularnych. Wyrażenie zapisane z wykorzystaniem cudzysłowów wymagają istnienia znaku dopasowania (~).
`"NF > 4"`	wypisanie wierszy, w których występuje 5 lub więcej pól.
`"$NF > 4"`	wypisanie wierszy, w których w ostatnim polu znajduje się wartość, co najmniej 5.
`"{print NF}"`	wypisuje na wyjściu ilość pól (wyrazów) znajduje się w każdym wierszu.
`"{print $NF}"`	wypisuje ostatnie pole dla każdego przetwarzanego wiersza.
`"/regexp/"`	wypisuje tylko te wiersze, które zawierają wyrażenie regularne *"regexp"*.
`"/text|file/"`	wiersze zawierające *"text"* lub *"file"* (wielkość liter ma znaczenie!)

`"/foo/`	`{print "za", NR}"`	błędnie zapisany argument zawierający wyrażenie regularne, który nie zadziała poprawnie podczas uruchomienia z linii poleceń!!
`'/foo/`	`{print "za", NR}'`	poprawnie zapisany atrybut!! Jeżeli analizowany wiersz będzie zawierał *"foo"*, wtedy na wyjściu zostanie wypisane słowo *„za"* i numer obecnego wiersza.

`"$3 ~ /B/ {print $2,$3}"`	Jeżeli trzecie pole zawiera *"B"*, wtedy na wyjście wypisywane jest drugie i trzecie pole.
`"$4 !~ /R/"`	wypisuje wiersze, których czwarte pole nie zawiera *"R"*.
`'$1=$1'`	Usuwa dodatkowe spacje pomiędzy polami i puste wiersze.
`'{$1=$1;print}'`	Usuwa dodatkowe spacje pomiędzy polami, zostawiając puste wiersze.
`'NF'`	Usuwa wszystkie puste wiersze.

AND(&&), OR(||), NOT(!)

`"$2 >= 4 || $3 <= 20"`	wypisuje wiersze, w których zawartość drugiego pola jest większa bądź równa 4 lub zawartość trzeciego pola jest mniejsza bądź równa 20.
`"NR > 5 && /with/"`	wypisuje wiersze, których numer jest, co najmniej równy 6 i które zawierają wzorzec *"with"*.
`"/x/ && NF > 2"`	wypisuje wiersze, które zawierają więcej niż dwa pola i ich zawartość zawiera wzorzec *"x"*.
`"$3/$2 != 5"`	wypisuje wiersze, dla których iloraz pola trzeciego przez drugie jest różny od 5. **Uwaga**: *„!="* operator nierówności zarówno dla liczb jak i łańcuchów.

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

`"$3 !~ /regexp/"` wypisuje wiersze, które w trzecim polu nie zawierają wyrażenia regularnego *„regexp”*.

`"!($3 == 2 && $1 ~ /foo/)"` wypisuje wiersze, które nie spełniają warunku zdefiniowanego w nawiasie (zawartość pola trzeciego musi być równa 2 i zawartość pola pierwszego musi zawierać wzorzec *„foo”*).

`"{print NF, $1, $NF}"` dla każdego wiersza wypisywana jest ilość jego pól, zawartość pierwszego pola i zawartość ostatniego pola.

`"{print NR, $0}"` wypisuje każdy wiersz poprzedzony prefiksem, który reprezentuje numer wiersza.

`'{print NR ": " $0}'` wypisuje każdy wiersz poprzedzony prefiksem, który reprezentuje numer wiersza, dwukropek i spację.

`"NR == 10, NR == 20"` wypisuje wiersze reprezentowane numerami od 10-20 włącznie.

`"/start/, /stop/"` wypisuje zawartość każdego wiersza pomiędzy wzorcem *"start"* i *"stop"*.

`"length($0) > 72"` wypisuje wszystkie wiersze, których długość jest większa niż 72 znaki.

`"{print $2, $1}"` następuje na wyjściu odwrócenie dwóch pierwszych pól, wszystkie pozostałe pola są pomijane.

`"{print substr($0,index($0,$3))}"` wypisuje zawartość wiersza od trzeciego pola do końca linii.

WYKORZYSTANIE KLAUZULI END{...}

Instrukcje w klauzuli END są uruchamiane po analizie wszystkich wierszy wejściowych. Jest wykorzystywana najczęściej do wypisywania końcowych statystyk.

```
1) END { print NR }      # w rezultacie wypisywana jest ilość analizowanych wierszy.
2) {s = s + $1 }         # wypisywana jest suma oraz średnia wszystkich liczb znajdujących się w polu
 END {print "sum is", s, "average is", s/NR}     # pierwszym dla każdego
                                                 # analizowanego wiersza.
3) {names=names $1 " " } # konwertuje całą zawartość pola 1 z wszystkich analizowanych
 END { print names }                  # wierszy konkatenując je w jedną linię, np.:
            +      Beth   4.00   0    #
 plik       |      Mary   3.75   0    # plik wejściowy jest konkatenowany do:
wejściowy   |      Kathy  4.00   10   # "Beth Mary Kathy Mark" na wyjściu
            +      Mark   5.00   30   #
4) { field = $NF }       # wypisuje ostatnie pole ostatniego analizowanego wiersza.
 END { print field }
```

PRINT, PRINTF: WYPISYWANIE WYRAŻEŃ, SPOSOBY FORMATOWANIA POLECENIA PRINT

```
print(expr1, expr2, ..., exprn)     # mogą zostać wykorzystane wszystkie możliwe
                                    # operatory: <, <=, ==, >, >=.
print                               # skrót dla instrukcji {print $0}.
print ""                            # wypisuje na wyjściu pusty wiersz.
printf(expr1,expr2,expr3,\n}        # dodaj znak nowej linii do polecenia „printf”.
print "expression" > "file name"          # rezultat wyrażenia może być zapisany do pliku
                                          # wyjściowego.
```

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

KONWERSJA FORMATU:

```
BEGIN{ RS="";     FS="\n";      # oznaczamy wejściowy separator wiersza za pomocą pustej
   ORS="\n"; OFS="," }          # linii i separator pola za pomocą znaku nowej linii.
{$1=$1; print }                 # następnie ustawiamy na wyjściu separator wiersza na znak
                                # nowej linii i separator pola na przecinek.
```

PARAGRAFY:

```
'BEGIN{RS="";ORS="\n\n"};/foo/'          # wypisuje każdy wiersz zawierający 'foo' w osobnym
                                         # paragrafie.
'BEGIN{RS="";ORS="\n\n"};/foo/&&/bar/'   # wymagane jest jednoczesne dopasowanie
                                         # obu wzorców
'BEGIN{RS="";ORS="\n\n"};/foo|bar/'      # wystarczy, że jeden ze wzorców zostanie
                                         # dopasowany.
```

PRZEKAZYWANE ZMIENNE:

```
gawk -v var="/regexp/" 'var{print "Here it is"}'      # zmienna "var" jest
                                                      # wyrażeniem regularnym
                                                      # "regexp", które może być
                                                      # wykorzystywane wewnątrz
                                                      # kodu programu AWK.
gawk -v var="regexp" '$0~var{print "Here it is"}'     # zmienna "var" jest
                                                      # łańcuchem otoczonym
                                                      # podwójnymi cudzysłowami.
gawk -v num=50 '$5 == num'    # zmienna "var" jest wartością numeryczną.
```

ZMIENNE WBUDOWANE:

ARGC	ilość argumentów przekazywanych z konsoli systemowej.
ARGV	tablica wartości argumentów przekazywanych z linii poleceń systemu (ARGV[0...ARVC-1]).
FILENAME	nazwa obecnie analizowanego pliku wejściowego.
FNR	ilość wszystkich rekordów w obecnie przetwarzanym pliku wejściowym.
FS	wejściowy separator pola (domyślnie spacja).
NF	ilość pól w wejściowym i obecnym wierszu.
NR	numer odpowiadający obecnemu, wejściowemu rekordowi liczony od początku pliku wejściowego.
OFMT	definicja wyjściowej reprezentacji formatu dla liczb (domyślnie "%.6g").
OFS	wyjściowy separator pola (domyślnie spacja).
ORS	wyjściowy separator wiersza (domyślnie znak nowej linii).
RLENGTH	długość łańcucha zwróconego w wyniku dopasowania wyrażenia regularnego.
RS	wejściowy separator rekordu (domyślnie znak nowej linii).
RSTART	początkowa pozycja łańcucha zwróconego przez dopasowanie.
SUBSEP	separator tablicy indeksów dolnych formularza [i,j,...] (domyślnie ^\).

SEKWENCJE WYJŚCIOWE:

\b	znak cofnięcia się (^H),
\n	znak nowej linii (DOS, CR/LF; Unix, LF),
\r	znak powrotu karetki,

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

`\t` znak tabulacji (^I),
`\ddd` wartość oktalna `*ddd*', gdzie `*ddd*' to cyfry od 1 do 3, z zakresu pomiędzy 0 i 7.
`\c` każdy inny znak jest dosłowny, np.: \" for " i \\ for \.

FUNKCJE ZNAKOWE W AWK:

W poniższych funkcjach wyróżniamy następujące terminy:

- `*r*' reprezentuje wyrażenie regularne *"regexp"*,
- `*s*' i `*t*' to łańcuchy,
- `*i*' i `*n*' to liczby całkowite,
- `*&*' w zastępowanym łańcuchu, w poleceniu *SUB* lub *GSUB*, jest zastępowany przez dopasowany łańcuch.

`gsub(r,s,t)` zastępuje wszystkie możliwe wyrażenia regularne *r,* łańcuchem *s*, znalezione w wejściowych danych *t*. W rezultacie zwrócona zostaje ilość udanych zastąpień. Jeżeli łańcuch *t* jest pominięty, wtedy domyślnie stosowany jest cały wiersz ($0).

`gensub(r,s,h,t)` zastępuje *h* pierwszych wyrażeń regularnych *r,* łańcuchem *s*, znalezionych w wejściowych danych *t*. Jeżeli *h = 'g'* to w wyniku uzyskamy postać funkcji `gsub(r,s,t)`. W rezultacie zwrócony zostaje skonwertowany wzorzec, a nie ilość udanych zastąpień. Jeżeli łańcuch *t* jest pominięty, wtedy domyślnie stosowany jest cały wiersz ($0).

`index(s,t)` zwrócona zostaje całkowita liczba reprezentująca indeks początku łańcucha *t* w danych *s*, lub 0 jeżeli łańcuch *s* nie zawiera łańcucha *t*.

`length(s)` zwrócona zostaje całkowita liczba reprezentująca długość łańcucha *s.*

`match(s,r)` zwrócona zostaje całkowita liczba reprezentująca indeks początku dopasowania łańcucha *r* w danych *s* lub 0, jeżeli dopasowanie nie występuje. W rezultacie ustawiane są wartości następujących zmiennych RSTART i RLENGTH.

`split(s,a,fs)` w rezultacie uzyskiwany zostaje podział łańcucha *s* na pola, przechowywany w tablicy *a*, zgodnie ze zdefiniowanym separatorem pól *fs*. W rezultacie zwrócona zostaje ilość pól, na którą nastąpił podział. Jeżeli szczegółowy separator pola *fs* jest pominięty, wtedy jest wykorzystywany domyślny separator pola FS.

`sprintf(fmt,expr-list)` zwrócona zostanie lista wyrażeń *expr-list* sformatowana zgodnie z definicją formatu *fmt*.

`sub(r,s,t)` zachowanie podobnie do funkcji `gsub`, różniący się tylko faktem, że tylko pierwsze dopasowanie jest zastępowane.

`substr(s,i,n)` zwrócony zostaje podłańcuch o długości *n* łańcucha wejściowego *s* rozpoczynający się od indeksu *i*. Jeżeli wartość *n* zostanie pominięta, wtedy zostanie zwrócony podłańcuch rozpoczynający się od indeksu *i*, a kończący się końcem obecnego wiersza.

FUNKCJE ARYTMETYCZNE:

`atan2(y,x)` arcustangens z y/x w radianach.
`cos(x)` cosinus (kąt w radianach).
`exp(n)` wykładnicza (n nie musi być liczbą całkowitą).
`int(x)` skrócenie liczby zmiennopozycyjnej do liczby całkowitej.
`log(x)` logarytm naturalny.
`rand()` pseudo-losowa liczba.
`sin(x)` sinus (kąt w radianach).

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

`sqrt(x)`	pierwiastek z *x*.
`srand(x)`	ustawia nową wartość ziarna wykorzystywanego przez generator liczb losowych; zwykle wykorzystywany jest aktualny czas, jeżeli *x* nie zostało zdefiniowane.

FUNKCJE DEFINIOWANE PRZEZ UŻYTKOWNIKA:

Postać funkcji jest podobna do języka C. Definicja funkcji składa się ze słowa kluczowego `function`, nazwy funkcji, nazw argumentów wejściowych i definicji ciała funkcji.

Przykład:

```
function add_three(number, temp) {
  temp = number + 3
  return temp
}
```

Wyrażenie może być wywołane w następujący sposób:

```
print add_three(36)      # wypisuje na wyjście 39
```

TABLICE ADRESOWANE ZAWARTOŚCIĄ (ASSOCIATIVE ARRAYS):

Przykład, którego zadaniem jest zliczenie częstotliwości (ilości) występowania słów w pliku wejściowym.

```
{ for (i=1; i<=NF; i++)
     words[$i]++ }
END { for (i in words)
    print i, words[i] }
```

DODATKOWE MATERIAŁY

- Zbiór książek do AWK dostępnych w celu darmowego pobrania - `http://www.computer-books.us/awk.php`,
- Internetowa dokumentacja do AWK - `http://www.cs.bell-labs.com/cm/cs/awkbook/index.html`,
- Zakładka *„materiały dla studentów”* dla przedmiotu *„języki formalne i kompilatory”* na stronie domowej dr inż. W. Complaka – `http://www.cs.put.poznan.pl/wcomplak`.

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.