Narzędzia

AWK – opis języka z przykładami

Bogusław Lichoński
i Tomasz Przechlewski

Co to jest AWK?

Istnieją osoby, których nie trzeba przedstawiać żadnemu informatykowi. Knuth, Kernighan, Aho, Wirth i inni znani są nam wszystkim chociaż ze słyszenia. Dlatego wydaje nam się, że nie trzeba reklamować języka programowania utworzonego przez panów Alfreda **A**ho, Petera **W**einbergera i Briana **K**ernighana.

AWK – bo o nim mowa – powstał już w 1977 roku i naszym zdaniem nie jest wcale językiem archaicznym. Wręcz przeciwnie! Jeśli przyjmiemy, że ma służyć do konkretnych celów, to za chwilę okaże się, że może być narzędziem wprost niezastąpionym w codziennej pracy z plikami tekstowymi. W szczególności idealnie nadaje się do współpracy z TEX-em.

System UNIX wyposażony jest w szereg narzędzi wspomagających pracę użytkowników. AWK stał się jednym ze standardowych narzędzi tego systemu, choć implementacje AWK-a znaleźć można niemal na każdej platformie systemowej.

W jednym zdaniu można powiedzieć, że AWK służy do transformacji szeroko pojętych danych tekstowych. Istotą działania AWK-a jest przetwarzanie pliku lub plików wejściowych według zadanego zbioru reguł, generując pewien strumień danych wyjściowych, czy też plików wyjściowych.

Czy warto uczyć się języka AWK?

Wielką zaletą AWK-a jest jego przenośność. AWK jest (przynajmniej w swojej oryginalnej wersji) interpreterem, a więc źródła programów AWK-owych można z łatwością przenieść z UNIX-a na DOS-a lub gdziekolwiek indziej bez konieczności modyfikacji programu[1].

AWK jest językiem strukturalnym, a więc jego programy są czytelne i przejrzyste. Pod tym względem AWK przypomina język Pascal.

Cechą charakteryzującą programy AWK-owe jest ich zadziwiająca krótkość w stosunku do ilości wykonywanych zadań. Często zdarza się, że nasz program ma zaledwie 2–3 linijki, wykonanie jego trwa kilka sekund, a wykonuje on zadanie, które ręcznie wykonywane może być kilka godzin.

AWK i TEX

Plik źródłowy TEX-a jest plikiem tekstowym o określonej strukturze. Na przykład do wykonania zamian globalnych w naszym źródle wystarczy zwykły edytor tekstowy, co jednak zrobić jeżeli zamian zamierzamy dokonywać na wielu plikach i dokonujemy ich codziennie. Jeżeli nasz edytor nawet zapamiętuje kilka ostatnich zamian, to i tak jest to lista skończona!

Wykonywanie w kółko tych samych czynności jest nie tylko nużące, powoduje więcej błędów i pozbawia nas radości z napisania krótkiego programiku w AWK-u, dając w zamian niesmak, zmęczenie, brak poczucia własnej wartości.

Podstawy AWK-a

Każdy program języka AWK składa się z dowolnej ilości par

⟨*wzorzec*⟩ { ⟨*akcja*⟩ }

⟨*Wzorzec*⟩ jest wyrażeniem logicznym, które może być prawdziwe (wówczas wykonywana jest ⟨*akcja*⟩) lub fałszywe (⟨*akcja*⟩ nie jest wykonywana).

No tak – zapyta czytelnik – ale do czego ten ⟨*wzorzec*⟩ ma pasować? Odpowiedź na to pytanie wyjaśnia *istotę* działania języka AWK. Otóż standardowo ⟨*wzorzec*⟩ dopasowuje się po kolei do *każdej* linii pliku wejściowego.

Dla pewności przeczytajmy jeszcze raz poprzedni akapit i spójrzmy natychmiast na przykład (nazwijmy nasz pierwszy program `test1.awk`)!

```
$0=="TeX"  { print $0 }
```

i uruchommy AWK-a. Standardowo powinno wyglądać to tak[2]:

1: Zdanie to jest prawdziwe, jeżeli poruszamy się cały czas w obrębie tej samej implementacji AWK-a. Z reguły jednak narzecza AWK-a mają nieco inne nazwy od oryginału np. GAWK, mawk itp.

2: znak > symbolizuje prompt (znak zachęty) systemu operacyjnego

```
> awk -f test1.awk plik.we
```

Co się stanie? AWK sprawdzi czy w pliku wejściowym `plik.we` istnieją linie zawierające napis (i tylko napis) `TeX`, a następnie wypisze je do standardowego wyjścia, czyli na ekran monitora.

Taki będzie efekt, spójrzmy teraz nieco bliżej na powyższy przykład. Najpierw sprawy oczywiste: `$0=="TeX"` jest wzorcem, zaś `{ print $0 }` jest akcją. Akcja wykona się tylko wtedy gdy warunek `$0=="TeX"` jest prawdziwy, czyli gdy linijka w pliku wejściowym symbolizowana przez `$0` jest napisem `TeX`. Instrukcja `print $0` oznacza, że AWK wypisze żądaną linijkę.

Program w języku AWK może zawierać wiele par „⟨*wzorzec*⟩`{` ⟨*akcja*⟩ `}`”. Dla każdej linii pliku wejściowego obliczane są kolejne wzorce (w kolejności ich występowania w programie) i wykonywane akcje. Przykład 1 ilustruje program AWK-owy wykorzystujący 2 wzorce[3].

Zanim przejdziemy do bardziej szczegółowych informacji o języku wymienimy kilka podstawowych reguł składni AWK-a.

- Kolejne pary „⟨*wzorzec*⟩ `{` ⟨*akcja*⟩ `}`” muszą być oddzielone średnikami lub znakami nowej linii.
- Akcje mogą składać się z wielu poleceń, które muszą być oddzielone średnikami lub znakami nowej linii.
- Wzorzec lub akcja może zostać pominięty. W wypadku braku wzorca akcja zostaje wykonana dla każdej linii pliku wejściowego. Jeżeli pominiemy akcję, to AWK zastosuje akcję domyślną, czyli `{ print $0 }`.

W powyższym przykładzie AWK czyta dane z jednego pliku wejściowego (`plik.we`) ale w ogólności możemy uruchomić AWK-a w następujący sposób:

awk -f ⟨*program*⟩ ⟨*plik1*⟩ ⟨*plik2*⟩...

W takiej sytuacji AWK czyta po kolei linie z ⟨*pliku1*⟩, ⟨*pliku2*⟩ itd. dla wszystkich plików których nazwy podano w linii komend. Pliki te są modyfikowane wg programu z pliku ⟨*program*⟩.

3: Programy AWK-owe prezentowane we fragmentach tekstu rozpoczynających się słowem PRZYKŁAD, często zawierają konstrukcje języka *jeszcze nie omówione*. Jeżeli coś jest niezrozumiałe, czytaj dalej, a po lekturze całego artykułu wróć do do tego miejsca – wszystko powinno być jasne.

PRZYKŁAD 1 (Nelson H. F. Beebe)
Poniższy pomysłowy program przepisuje plik wejściowy zastępując kolejne puste linie, jedną pustą linią.

```
NF == 0 { nb++ }

NF > 0 { if (nb > 0) print "";
  nb = 0; print $0; }
```

AWK, gawk, nawk...

Istnieje wiele interpreterów AWK-a; w systemach UNIX-o podobnych, dostarczane razem z systemem, są one dziełem jego dostawcy. Istnieje też kilka wersji ogólnodostępnych, takich jak: GAWK – firmowany przez Free Software Foundation czy mawk Michaela Brennana.

Różne implementacje AWK-a *nie są* w 100% kompatybilne ze sobą. Ponadto wiele implementacji oferuje rozszerzenia w stosunku do standardu (za standard przyjmujemy opis z [1]). W niniejszym artykule przedstawimy standard AWK-a oraz rozszerzenia oferowane przez interpreter GAWK w wersji 3.0. Gorąco polecamy wszystkim tę implementację AWK-a, którą sami używamy od kilku lat.

Implementacja GNU interpetera AWK, powstała w 1986 r. dzięki pracy Paula Rubina i Jay Fenlason we współpracy z Richardem Stallmanem i Johnem Woodsem. Została ona gruntownie zmieniona w 1989 r. przez Davida Truemana i Arnolda Robbinsa. W styczniu 1996 r. ukazała się wersja 3.0 interpretera GAWK, zawierająca kilka interesujących rozszerzeń w stosunku do standardu. W dystrybucji znajduje się także ponad 300 stronicowy podręcznik ([4]), zawierający kompletny opis języka i wiele ciekawych przykładów.

UWAGI: Opis rozszerzeń jest specjalnie oznaczony w tekście *za pomocą pisma pochyłego.*

PRZEDSTAWIONE PRZYKŁADY SĄ PRZYGOTOWANE DO WYKONYWANIA W SYSTEMIE DOS. Użytkownicy UNIX-a, których zainteresuje nasz artykuł, nie powinni mieć większych problemów ze zmianą tych fragmentów programów, które są „DOS-owo zorientowane”.

Struktura pliku wejściowego

Dla AWK-a dane wejściowe składają się z *rekordów*, które rozdzielone są separatorami RS. Standardowo rekordem jest cała linia, czyli separatorem jest znak końca linii.

Rekordy podzielone są na *pola*, które rozdzielone są separatorami pól `FS`. Standardowo separatorem pól jest znak spacji lub tabulacji.

`RS` i `FS` są zmiennymi, a więc można im nadać wartość. Przykładowo, jeśli zmiennej `FS` nadamy wartość '`;`', to spacje i tabulatory nie będą separatorami pól lecz znaki '`;`'.

W akcjach i wzorcach, do wartości pól można odwoływać się zmiennymi postaci `$`⟨*nr-pola*⟩. Tak więc `$1` to pierwsze pole rekordu, `$2` drugie itd. `$0` oznacza cały rekord. Wbudowana zmienna `NF` przechowywuje liczbę pól bieżącego rekordu.

Wzorce

Wzorce służą do wyznaczenia tych linii tekstu, dla których wykonane mają być odpowiednie akcje. W ogólnym wypadku wzorzec może być kombinacją wyrażeń logicznych i wyrażeń regularnych. Ponieważ wzorce są wyrażeniami logicznymi, dozwolone są operatory logiczne: `&&`, `||`, `!` oraz nawiasy. Istnieją dwa specjalne wzorce o nazwie `BEGIN` i `END`. Oto ogólna specyfikacja wzorców:

Wzorce

`BEGIN{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywana *przed otwarciem* pliku wejściowego.

`END{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywana po zamknięciu pliku wejściowego.

⟨*wyrażenie*⟩`{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywana za każdym razem gdy wartość ⟨*wyrażenia*⟩ jest równa *prawda* tj. jest niezerowe (dla wyrażeń numerycznych) lub niepuste (dla napisów).

`/`⟨*wyrażenie-regularne*⟩`/{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywana za każdym razem gdy linia pliku wejściowego zawiera ciąg znaków pasujący do ⟨*wyrażenia-regularnego*⟩.

⟨*wzorzec-złożony*⟩`{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywany za każdym razem gdy linia czytanego przez AWK pliku zawiera ciąg znaków pasujący do ⟨*wzorca złożonego*⟩.

⟨*wzorzec1*⟩, ⟨*wzorzec2*⟩`{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywana dla wszystkich linii od linii zawierającej ⟨*wzorzec1*⟩ do linii zawierającej ⟨*wzorzec2*⟩ (łącznie z tymi liniami). ⟨*Wzorzec*⟩ oznacza ⟨*wyrażenie*⟩ bądź ⟨*wyrażenie-regularne*⟩.

Wzorce `BEGIN` i `END` nie mogą być częścią wzorca złożonego. Podobnie częścią wzorca złożonego nie może być wzorzec z przecinkiem.

BEGIN/END. Wzorzec `BEGIN` nie pasuje do żadnej linii z pliku wejściowego, a odpowiadająca mu akcja jest wykonywana przed przeczytaniem przez AWK-a pierwszego znaku z tego pliku. Podobnie instrukcje wzorca `END` wykonywane są po przeczytaniu wszystkich znaków pliku wejściowego. Możliwe jest umieszczenie wielu wzorców `BEGIN` i `END` w programie AWK-owym; są one wtedy wykonywane po kolei. Z reguły użytkownicy AWK-a umieszczają wzorce `BEGIN` na początku a `END` na końcu pliku. Dla AWK-a jest to absolutnie obojętne.

Jednym z najczęstszych sposobów użycia `BEGIN` jest zmiana domyślnego sposobu w jaki AWK dzieli linie czytanego pliku na pola. Wbudowana zmienna `FS` definiuje jaki znak jest separatorem pól w rekordzie. Domyślnie pola oddzielone są znakami spacji lub/i tabulacji. Taki sposób dzielenia rekordu odpowiada sytuacji kiedy `FS` jest nadana wartość równa spacji (`FS=" "`). Przy uruchomieniu programu zawierającego tylko wzorce `BEGIN` AWK nie oczekuje w linii komend nazwy żadnego pliku wejściowego, por. przykład 10, s. 21.

Wyrażenie. Wzorcami mogą być wyrażenia arytmetyczne lub napisowe. Odpowiednia ⟨*akcja*⟩ jest wykonywana za każdym razem gdy takie ⟨*wyrażenie*⟩ ma wartość różną od zera lub od napisu pustego. Przykłady: `$3/$2 >= 10`, `NR < 9` czy `"NY3"`.

Wzorzec regularny. Wzorzec regularny to wyrażenie regularne ujęte w parę znaków `/`. Podstawowe sposoby użycia wzorca regularnego to:

`/`*r*`/`
Pasuje do bieżącej linii pliku wejściowego jeżeli zawiera ona podnapis pasujący do wyrażenia regularnego *r*.

⟨*wyrażenie*⟩ `~ /`*r*`/`
Pasuje do napisu będącego wartością ⟨*wyrażenia*⟩ jeżeli zawiera on podnapis pasujący do wyrażenia regularnego *r*. Zapis `/`*r*`/` jest równoważny formie `$0 ~ /`*r*`/`.

⟨*wyrażenie*⟩ `!~ /`*r*`/`
Pasuje do napisu będącego wartością ⟨*wyrażenia*⟩ jeżeli *nie zawiera* on podnapisu pasującego do wyrażenia regularnego *r*.

Wzorzec złożony. Wzorzec złożony to wyrażenie złożone z wzorców i operatorów logicznych `||`, `&&`, `!`. Wzorzec złożony pasuje do bieżącej linii pliku wejściowego jeżeli wartością wyrażenia jest *prawda* (czyli jest niezerowa lub niepusta). Przykład:

```
$2 > 0.50 && $5 > 0.95
```

Wzorzec z przecinkiem. Pasuje do wszystkich linii, od linii pasującej do ⟨*wzorca1*⟩ do linii pasującej do ⟨*wzorca2*⟩ (łącznie z tymi liniami). Przykład:

```
/StartCharMetrics/,/EndCharMetrics/
```

Wyrażenia regularne

Używając AWK-a nie sposób pominąć wyrażeń regularnych, które stanowią o sile tego języka. Nie będziemy przytaczać ścisłej matematycznej definicji, gdyż zaciemnilibyśmy tylko bardzo intuicyjne i łatwe naszym zdaniem pojęcie.

Siła wyrażeń regularnych polega na możliwości stosowania uniwersalnych wzorców, które pasują (opisują) pewien zbiór napisów. Na pewno każdy z nas wydał polecenie swojemu systemowi operacyjnemu, w którym zawarty był znak `*`; na przykład:

```
> emacs *.tex
```

Jeśli nie zdarzyło Ci się wydać takiego polecenia, to już wyjaśniamy! emacs to popularny edytor tekstowy, zaś napis `*.tex` oznacza, że chcemy edytować *wszystkie* pliki z rozszerzeniem `.tex`[4] z bieżącego katalogu. Znak `*` jest właśnie wyrażeniem regularnym, które oznacza w tym wypadku dowolną (z dokładnością do ograniczeń naszego systemu operacyjnego) ilość i rodzaj znaków.

Wyrażenia regularne to wyrażenia umożliwiające specyfikowanie *klas* napisów. O napisie należącym do tej klasy mówimy, że *pasuje* do wyrażenia regularnego. Wyrażenia regularne są konstruowane z następujących elementów: „normalnych znaków” (wszystkie litery, cyfry, większość pozostałych znaków) oraz metaznaków `\`, `^`, `$`, `.`, `[`, `]`, `|`, `(`, `)`, `*`, `+`, `?`. Poniższa tabela przedstawia poszczególne elementy wyrażeń regularnych.

Wyrażenia regularne

Wyrażenie	Znaczenie
c	znak nie będący metaznakiem
`\`*c*	znak sterujący albo znak/metaznak *c*
`.`	dowolny znak
`^`	początek napisu
`$`	koniec napisu
`[`*ab*`...]`	dowolny ze znaków *a*, *b*...
`[^`*ab*`...]`	dowolny ze znaków oprócz *a*, *b*...
`[`*a-z*`]`	dowolny ze znaków z zakresu *a-z*
`[^`*a-z*`]`	dowolny ze znaków oprócz *a-z*
$r_1 \mid r_2$	r_1 lub r_2 (*r* oznacza wyrażenie regularne)
r`*`	zero lub więcej napisów pasujących do *r*
r`+`	jeden lub więcej napisów pasujących do *r*
r`?`	zero lub jeden napis pasujący do *r*
`(`*r*`)`	*r* (nawiasy służą do grupowania wyrażeń)

4: Czyli takie pliki, których nazwa kończy się znakami `.tex`.

Grupę znaków ujętą w nawisy klamrowe nazywamy *listą*. Do takiego wyrażenia regularnego pasuje jeden dowolny znak z listy. Zakres znaków to dwa znaki ujęte w nawiasy klamrowe oddzielone znakiem `-` (minus). Zakresy interpretowane są według kolejności wartości kodów ASCII jakie posiadają poszczególne znaki. Zatem specyfikacja `[0-9]` jest równoważna `[0123456789]`, zaś `[A-Da-d]` liście `[ABCDabcd]`.

Dopełnieniem listy/zakresu jest lista/zakres z poprzedzającym znakiem `^` (bezpośrednio po otwierającym nawiasie `[`). Przykładowo specyfikacja `[^0-9]` oznacza jeden dowolny znak *ale nie* cyfrę; `[^A-ZĄĆĘŁŃÓŚŹŻ]` dowolny znak nie będący dużą literą alfabetu.

Wewnątrz listy/zakresu wszystkie znaki oprócz `\`, `^`, `-` tracą swoje metaznaczenie. Przykładowo: `[...]` oznacza trzy kropki (a nie trzy dowolne znaki) zaś `^[^^]` wszystkie znaki *oprócz* znaku `^` na początku napisu.

Nawiasy okrągłe służą do grupowania. Przykładowo: `/(X|XX)(I|II|III)/` pasuje do następujących liczb XI, XII, XIII, XXI, XXII, XXIII.

Znaki sterujące, zapisujemy w konwencji języka C. Są to: `\a` (dzwonek, *alarm*), `\b` (znak cofnięcia, *backspace*), `\f` (znak końca strony, *form feed*), `\n` (przejście do nowego wiersza, *new line*), `\r` (*carriage return*), `\t` (znak tabulacji). Ponadto znak `\\` oznacza `\`, zaś każdy znak możemy zapisać przy pomocy kodu ósemkowego używając konwencji `\`⟨*cyfra*⟩⟨*cyfra*⟩⟨*cyfra*⟩.

PRZYKŁAD 2 (wyrażenia regularne)

`/^[ \t]*$/` pasuje do napisu składającego się tylko ze znaków spacji, tabulacji i napisu pustego;

`/^[^ \t]*$/` pasuje do wszystkich napisów oprócz składających się ze spacji, znaków tabulacji i pustych;

`/[+-]?[0-9]+[.]?[0-9]*/` pasuje do liczby rzeczywistej ze znakiem;

`/\\[A-Za-z]+/` pasuje do ciągu liter poprzedzonych znakiem `\` (np: komenda TEX-owa).

Wyrażenia

Podstawą składni wyrażeń AWK-a jest popularna składnia wyrażeń języka C. Składnia ta została w AWK wzbogacona o operacje tekstowe.

Elementami wyrażeń są: stałe, zmienne, operatory, funkcje wbudowane i definiowane przez użytkownika oraz elementy tablic asocjacyjnych. Omówimy je po kolei.

W AWK-u istnieją tylko dwa typy danych: liczbowy i napisowy. Zmienne liczbowe przechowują wartości zmiennopozycyjne, przy czym ich dokładność zależna jest od implementacji. Zmienne napisowe przechowują oczywiście ciągi znaków czyli napisy. Zmiennych nie deklaruje się. Typ zmiennej określony jest przez kontekst; w razie potrzeby zawsze dokonywana jest odpowiednia konwersja. Zmienna nie zainicjowana ma wartość zero lub `""` (pusty napis). Stałe liczbowe zapisujemy jak w C, tj. `3.1415` lub `1.333e-5`, stałe napisowe otacza się znakami `"`.

Istotny jest także sposób interpretacji wyrażeń numerycznych i tekstowych w operacjach logicznych. Fałsz odpowiada liczbie 0 i napisowi pustemu `""`, zaś prawda odpowiada wszystkim innym liczbom i napisom.

Zmienne wbudowane. Większość zmiennych została lub zostanie dokładnie omówiona przy okazji omawiania tych aspektów AWK-a, których dotyczą. Oto zestawienie wszystkich zmiennych:

Zmienne wbudowane

Zmienna	Opis znaczenia
ARGC	liczba argumentów wywołania programu
ARGV	tablica argumentów wywołania programu
ARGIND	*index w* ARGV *odpowiadający bieżącemu plikowi*
ENVIRON	*tablica zmiennych środowiskowych*
ERRNO	*napis z systemowym opisem błędu*
FIELDWIDTHS	*specyfikacja długości pól, por. s. 22*
FILENAME	nazwa bieżącego pliku wejściowego
FNR	numer bieżącego rekordu w bieżącym pliku
FS	separator pól
IGNORECASE	*przełącznik rozróżniania wysokości liter*
NF	liczba pól w bieżącym rekordzie
NR	liczba przeczytanych rekordów
OFMT	specyfikacja formatu dla liczb
OFS	separator pól na wyjściu, por. s. 23
ORS	separator rekordów na wyjściu, por. s. 23
RLENGTH	por. opis funkcji `match`, s. 16
RS	separator rekordów
RT	*napis pasujący do wyrażenia* RS*, por. 20*
RSTART	por. opis funkcji `match`, s. 16
SUBSEP	separator indeksów tablic, por. s. 19

`ENVIRON` *jest tablicą przechowującą wartości zmiennych środowiskowych. Indeksami są nazwy zmiennych, wartościami zaś napisy zawierające wartości tych zmiennych. Przykładowo:*

```
ENVIRON["TEXCONFIG"]
```

może zawierać, np: `.;\tex\dvips;\gslib\psfonts`

`AWKPATH` *– nazwa zmiennej środowiskowej zawierającej ścieżki dostępu do katalogów zawierających programy AWK-owe (czyli pliki* `*.awk`*) Jeżeli zmiennej* `AWKPATH` *nie nadano żadnej wartości (poleceniem* `SET` *w systemie* DOS*) to jest ona równa* `".;c:/lib/awk;c:/gnu/lib/awk"`.

`ERRNO` *udostępnia napis zawierający systemowy komunikat błędu, jeżeli przy wykonaniu funkcji* `getline` *lub* `close` *wystąpi błąd. Przykładowo, po wykonaniu:*

```
getline < "qq.qq.qq"
```

`ERRNO` *zawiera* `"No such file or directory"`.

`IGNORECASE` *określa czy AWK rozróżnia duże i małe litery przy porównywaniu napisów oraz wyrażeń regularnych. Jeżeli* `IGNORECASE` *jest niezerowe/niepuste, wtedy operatory* `~`, `!~` *oraz funkcje* `gensub`, `gsub`, `index`, `match`, `split` *oraz* `sub` *nie rozróżniają dużych i małych liter. Dotyczy to także wartości zmiennych wbudowanych* `RS` *i* `FS`*. GAWK począwszy od wersji 3.0 obsługuje normę ISO-8859-1 (tj. ISO Latin-1). Standard ten nie zawiera jednak większości polskich znaków diakrytycznych.*

`ARGIND` *przechowuje indeks pod którym, w tablicy* `ARGV` *znajduje się nazwa przeglądanego pliku. Zawsze jest prawdziwa równość* `FILENAME == ARGV[ARGIND]`.

UWAGI: Zmienna `FILENAME` zawiera nazwę bieżącego pliku wejściowego. Oznacza to, że w obrębie wzorców `BEGIN` i `END` wartość `FILENAME` jest nieokreślona.

Operatory arytmetyczne. Operatory arytmetyczne są naszym zdaniem łatwe i intuicyjne:

Operatory arytmetyczne

Postać wyrażenia	Opis znaczenia
*	iloczyn
+	suma
-	różnica
/	iloraz
%	modulo
^	potęga
++	inkrementacja
--	dekrementacja

Operatory napisowe. Napisy i zmienne napisowe można łączyć (konkatenować) przy pomocy „niewidocznego” operatora – po prostu należy umieścić napisy obok siebie. Przykładowo po wykonaniu:

```
y = "Ali"; x = y "ba" "ba"
```

zmienna `x` ma wartość `"Alibaba"`. Oprócz operacji konkatenacji AWK nie ma żadnych innych operatorów napisowych.

Operatory porównywania i operatory logiczne. Zapis i działanie operatorów w AWK-u w wypadku zmiennych typu liczbowego jest identyczny jak w języku C. Nowością AWK-a jest to, że mogą być także stosowane do napisów.

Operatory porównywania	
Operator	Opis znaczenia
`==`	równe
`!=`	różne
`<`	mniejsze
`<=`	mniejsze lub równe
`>`	większe
`>=`	większe lub równe

Napisy są porównywane w taki sposób, że najpierw porównywane są pierwsze znaki, potem drugie itd. Przykładowo: `"10"` jest mniejsze od `"9"`. Jeżeli jeden napis jest przedrostkiem drugiego to krótszy napis jest mniejszy od dłuższego, np. `Ali` jest mniejsze od `"Alibaba"`. Oczywiście prawdą jest: `"TeX" == "TeX"`.

Operatory logiczne	
Operator	Opis znaczenia
`&&`	suma logiczna
`||`	alternatywa
`!`	zaprzeczenie

Operatory związane z dopasowywaniem wyrażeń regularnych. Ostatnia grupa to operatory związane z dopasowywaniem wyrażeń regularnych, są one specyficzne dla języka AWK.

Mamy tylko dwa takie operatory `~` oraz `!~`. Umożliwiają one dopasowanie zmiennej do pewnego wyrażenia regularnego. Przykładowo `$1 ~ /TeX/` jest prawdziwe, gdy pierwsze pole rekordu zawiera napis `TeX`.

Samotnie pojawiające się na przykład w akcji wyrażenie `/TeX/` jest równoważne w AWK wyrażeniu `$0 ~ /TeX/`.

PRZYKŁAD 3
Zaimplementujmy funkcję, zwracającą 1 jeżeli rok jest przestępny, lub 0 dla lat nieprzestępnych. Algorytm cytujemy za [3], s. 121.

```
function leapyear(year) {
  return year %4 == 0 && year % 100 \
  != 0 || year%400 ==0; }

BEGIN {print leapyear(1996), leapyear(1806),
leapyear(1066)}
```

Wzorzec `BEGIN` jest oczywiście potrzebny tylko dla testowania funkcji. Jeżeli powyższy kod umieścimy np. w pliku `lyear.awk` to pisząc `gawk -f lyear.awk` otrzymamy na ekranie:

```
1 0 0
```

UWAGI: Bardzo długa instrukcja może zostać podzielona i zapisana w kilku linijkach. Znakiem kontynuacji jest `\`, bezpośrednio przed znakiem końca linii (por. drugą linijkę przykładu). Jeżeli linijka kończy się przecinkiem (por. linijka przedostatnia) to znak kontynuacji jest opcjonalny.

Przypisanie. Przypisanie oznaczane jest w AWK-u pojedynczym znakiem równości `=`. Podobnie jak w języku C operator ten nadaje zmiennej wartość i zwraca przypisaną wartość, stąd dozwolone są wyrażenia postaci `x = y = 1` lub `(x = y) <= 1`.

Z operatorem przypisania związane są operatory modyfikacji: `+=`, `-=`, `*=`, `/=`, `%=`, `/=` i `^=`. Przykładowo wyrażenie `x += y` jest tożsame z `x = x + y`, wyrażenie `x -= y` jest tożsame z `x = x - y` itd.

Operator warunkowy `?:`. Operator warunkowy `?:` posiada następującą składnię:

$\langle$*wyrażenie1*$\rangle$ `?` $\langle$*wyrażenie2*$\rangle$ `:` $\langle$*wyrażenie3*$\rangle$

Najpierw obliczane jest $\langle$*wyrażenie1*$\rangle$. Jeśli jest ono prawdziwe obliczane jest $\langle$*wyrażenie2*$\rangle$, w przeciwnym wypadku $\langle$*wyrażenie3*$\rangle$.

Poniższy program oblicza i drukuje odwrotność pierwszych pól wszystkich rekordów, sprawdzając czy `$1` nie jest równe zeru:

```
{print $1!=0 ? 1/$1 : "Zero w linii", NR;}
```

Arytmetyczne funkcje wbudowane

AWK oferuje inny zestaw funkcji wbudowanych niż język C. Funkcje wbudowane mogą być, bez żadnych ograniczeń, elementami wyrażeń. Oto lista takich funkcji (niech x, y będą pewnymi wyrażeniami):

Funkcje arytmetyczne	
Funkcja	Wartość funkcji
`atan2(y,x)`	arcus tangens z y/x w zakresia $-\pi$ do π
`cos(x)`	cosinus z x, x w radianach
`sin(x)`	sinus z x, x w radianach
`exp(x)`	eksponent, czyli funkcja wykładnicza e^x
`int(x)`	część całkowita z x
`log(x)`	logarytm z x przy podstawie e
`sqrt(x)`	pierwiastek kwadratowy z x
`rand()`	przypadkowa liczba z przedziału $\langle 0, 1)$
`srand(x)`	x jest wartością początkową dla generatora liczb pseudolosowych

Używając powyższych funkcji można uzyskać użyteczne liczby, na przykład π lub e; $\mathrm{atan2}(0, -1) = \pi$ oraz $\exp(1) = e$. Również uzyskanie logarytmu dziesiętnego nie jest problemem, jeśli zastosujemy wzór $\log(x)/\log(10)$.

Podstawienie

```
randint = int(n * rand()) + 1
```

spowoduje nadanie zmiennej randint wartości z przedziału $\langle 1, n\rangle$.

Napisowe funkcje wbudowane

Poniższe zestawienie zawiera funkcje AWK-a umożliwiające manipulowanie napisami. W zestawieniu $\langle r\rangle$ oznacza wyrażenie regularne, $\langle s\rangle$ i $\langle t\rangle$ napis.

Funkcje napisowe

gsub($\langle r\rangle$,$\langle s\rangle$,$\langle t\rangle$)
Zamienia wyrażenie regularne $\langle r\rangle$ na napis $\langle s\rangle$ w napisie $\langle t\rangle$. Zwracana jest liczba zamian. Jeżeli gsub wywołamy tylko z dwoma pierwszymi parametrami to zmiany dokonywane są w napisie $0. (tj. gsub($\langle r\rangle$,$\langle s\rangle$) jest równoważne gsub($\langle r\rangle$,$\langle s\rangle$,$0)).

gensub($\langle r\rangle$,$\langle s\rangle$,$\langle a\rangle$,$\langle t\rangle$)
Uogólniona funkcja gsub. *Zwraca zmieniony napis (nie modyfikuje oryginalnego napisu $\langle t\rangle$!). Zamienia wyrażenie regularne $\langle r\rangle$ w oparciu o napis $\langle s\rangle$, w napisie $\langle t\rangle$ (jeżeli nie ma $\langle t\rangle$, domyślnym argumentem jest* $0*). Argument $\langle a\rangle$ określa, który z kolei podnapis pasujący do wyrażenia $\langle r\rangle$ ma być wymieniony. Jeżeli $\langle a\rangle$ jest napisem rozpoczynającym się od* "g" *(lub* "G"*) to wymieniane są wszystkie napisy pasujące do $\langle r\rangle$. Funkcja* gensub *umożliwia wstawienie fragmentów wyrażenia regularnego w napisie $\langle s\rangle$. Jeżeli wyrażenie $\langle r\rangle$ podzielimy za pomocą nawiasów, (i) na części składowe to te składowe mogą później pojawić się w napisie $\langle s\rangle$ (oznaczamy je jako \\$\langle n\rangle$, gdzie $\langle n\rangle$ jest cyfrą od 1 do 9. Znaczenie tego jest takie, że napis pasujący do $\langle n\rangle$-tej składowej jest kopiowany, z tekstu $\langle t\rangle$ do tekstu zwracanego przez funkcję. W efekcie możliwe są wszelkiego rodzaju zmiany kontekstowe, por. przykład 5. Symbol* \0 *oznacza całe wyrażenie regularne $\langle r\rangle$*[5].
Poniższy przykład wyjaśnia znaczenie argumentu $\langle a\rangle$:

```
BEGIN{ t = "Alibababa";
  print gensub(/ba/, "BA", 2, t) }
```

otrzymamy: AlibaBAba

index($\langle s\rangle$,$\langle t\rangle$)
Zwraca numer pierwszego znaku napisu $\langle t\rangle$ w napisie $\langle s\rangle$. Jeżeli $\langle s\rangle$ nie zawiera $\langle t\rangle$ zwracana jest wartość zero. Pierwszy znak w napisie ma numer 1. Przykładowo:

```
index("Alibaba", "baba")
```

zwraca 4;

length($\langle s\rangle$)
Podaje długość napisu $\langle s\rangle$.

match($\langle s\rangle$,$\langle r\rangle$)
Jeżeli $\langle s\rangle$ zawiera podnapis pasujący do $\langle r\rangle$, to zwraca numer pierwszego znaku tego podnapisu; w przeciwnym razie zwracane jest 0. Ponadto nadawane są wartości zmiennym RSTART oraz RLENGTH. RSTART jest równe wartości zwracanej przez funkcję, RLENGTH jest równe długości podnapisu pasującego do $\langle r\rangle$.

split($\langle s\rangle$,$\langle a\rangle$,$\langle fs\rangle$)
Z napisu $\langle s\rangle$ tworzy tablicę napisów $\langle a\rangle$ w oparciu o znak separujący $\langle fs\rangle$. Jeżeli split wywołamy tylko z dwoma parametrami to znakiem separującym jest znak określony wartością zmiennej FS, czyli separator pól w rekordzie.

sprintf($\langle format\rangle$, lista-wyrażeń)
Zwraca napis, sformatowany według napisu $\langle format\rangle$, por. funkcja printf, s. 22.

sub($\langle r\rangle$,$\langle s\rangle$,$\langle t\rangle$)
Najdłuższy napis pasujący do wyrażenia regularnego $\langle r\rangle$ zamienia na napis $\langle s\rangle$ w napisie $\langle t\rangle$ (lub w $0 jeżeli wywołana jest tylko z dwoma pierwszymi parametrami – podobnie jak gsub). Zwracana jest liczba dokonanych zamian.

substr($\langle s\rangle$,$\langle p\rangle$,$\langle n\rangle$)
Zwraca napis wycięty z $\langle s\rangle$ począwszy od pozycji $\langle p\rangle$ o długości $\langle n\rangle$ znaków (lub do końca napisu $\langle s\rangle$, jeżeli ostatni argument jest pominięty). Przykładowo wykonanie instrukcji

```
print substr("Alibaba",4);
```

spowoduje wydrukowanie napisu „baba".

tolower($\langle s\rangle$)
Zwraca napis, w którym duże litery zostały zamienione na małe. W wypadku tekstów polskich funkcja ta ma ograniczone zastosowanie, nie zamieni bowiem liter z górnej połówki tabeli ASCII, gdzie znajdują się Ą, Ć, Ę, itd.

toupper($\langle s\rangle$)
Zwraca napis, w którym małe litery zostały zamienione na duże. Z „polskiego" punktu widzenia ma tę samą wadę co tolower.

PRZYKŁAD 4 (fragment spj.awk, pomysł M. Ryćko)

```
# wymień ⟨co⟩ w kontekście ⟨bef⟩ ⟨aft⟩ na ⟨na⟩
function exch (bef, co, aft, na) {
  while (match(para, bef co aft) > 0) {
    match(para, bef co aft);
    nowy = substr(para, 1, RSTART) na \
      substr(para, RSTART+RLENGTH-1);
    para = nowy;
  }
}
```

Powyższa funkcja[6] realizuje kontekstową zamianę frazy na frazę. Jest namiastką tego czego AWK-owi do tej pory brakowało (por. nast. przykład) – zamiany wyrażenia regularnego na *wyrażenie regularne*.

5: *W chwili pisania tego tekstu funkcja* gensub *jest zaimplementowana z błędem. Mianowicie, jeżeli w tekście $\langle t\rangle$ nie ma napisu pasującego do $\langle r\rangle$, to zwracana jest, zamiast niezmienionego napisu $\langle t\rangle$, liczba* 5.9976e-315*. Autorzy nie natknęli się na inne błędy, i w związku z tym wydaje się bezpieczne stosowanie tej funkcji połączone z testowaniem zwracanej wartości, np. można zdefiniować następującą funkcję* xgensub:

```
xgensub(r, s, a, t,     tmp){
   tmp = gensub(r, s, a, t);
   return tmp == 5.9976e-315 ? t : tmp
}
```

6: Porównaj punkt Funkcje dalej w tekście.

Zakładamy, że fraza ⟨*bef*⟩ jest jednoznakowa i poprzedza ⟨*co*⟩, zaś fraza ⟨*aft*⟩ następuje po ⟨*co*⟩ i także jest jednoznakowa. Przykład użycia:

```
{ exch("[0-9]","-","[0-9]", "--");
  print }
```

wymieni w całym tekście wszystkie frazy ⟨*cyfra*⟩-⟨*cyfra*⟩ na ⟨*cyfra*⟩--⟨*cyfra*⟩ czyli na przykład 1992-1993 zamieni na 1992--1993.

PRZYKŁAD 5 (GAWK3.0)
Poniższy programik wykorzystujący funkcję `gensub`*:*

```
{
 $0=gensub(/([0-9])-([0-9])/,"\\1--\\2","g",$0)
 print
}
```

robi to samo co funkcja `exch` *czyli zamienia w całym tekście wszystkie frazy* ⟨*cyfra*⟩-⟨*cyfra*⟩ *na* ⟨*cyfra*⟩--⟨*cyfra*⟩*, np.* `s.~1-11` *zmieni na* `s.~1--11`.

PRZYKŁAD 6 (zamiana małych liter na duże)
Poniższa funkcja jest odpowiednikiem funkcji `toupper`; ma tę zaletę, że „rozpoznaje” polskie znaki. Łatwo też daje się modyfikować dla różnych wariantów kodowania polskich znaków.

```
function upper(string,     i, j){
 newstring = ""
 for (i = 1; i <= length(string); i++){
   char = substr(string, i, 1)
   for (j = 1; j <= ALPHABET; j++){
     if (char == little[j]){
         char = big[j]
         j = ALPHABET + 1;
   } }
   newstring = newstring char;
 }
 return newstring
}

BEGIN{
 LOWER = "abcdefghijklmnopqrstuvwxyząćęłńóśźż";
 UPPER = "ABCDEFGHIJKLMNOPQRSTUVWXYZĄĆĘŁŃÓŚŹŻ";
 ALPHABET = length(LOWER);

 for (i = 1; i <= ALPHABET; i++){
   little[i] = substr(LOWER,i,1)
   big[i] = substr(UPPER,i,1)
 }
}
```

Dodajmy jeszcze następujący wzorzec `BEGIN`:

```
BEGIN { print upper("Tó jęśt tęśtć"); }
```

Uruchamiając AWK-a. Otrzymamy na ekranie:

```
TÓ JĘŚT TĘŚTĆ
```

Funkcje daty i czasu

Interpreter GAWK *od wersji* 3.0 *oferuje dwie funkcje dotyczące daty i czasu. Są to* `systime` *i* `strftime`*:*

`systime()`

Zwraca bieżący czas w sekundach jakie upłynęły od początku epoki. W standardzie POSIX jest to liczba sekund od 1 Stycznia 1970 r.

`strftime(`⟨*format*⟩`, `⟨*czas*⟩`)`

Zwraca napis zawierający ⟨*czas*⟩ *(liczba w takim samym formacie jak wartość zwracana przez* `systime`*) sformatowany według specyfikacji zawartych w napisie* ⟨*format*⟩*. Forma krótka* `strftime()` *oznacza użycie formatu* `"%a %b %d %H:%M:%S %Z %Y"` *oraz bieżącego czasu. Forma* `strftime(`⟨*format*⟩`)` *wypisuje bieżący czas według specyfikacji z* ⟨*formatu*⟩*.*

Specyfikacje przekształceń funkcji `strftime` *są zgodne ze standardem ANSI C. Każda specyfikacja składa się ze znaku „*`%`*” oraz następującego po nim znaku określającego typ konwersji (por. także funkcja* `printf`*, s. 22) Nie będziemy podawać pełnej listy znaków konwersji (por. [4], s. 149–151), ograniczymy się do najczęściej stosowanych:*

Znaki konwersji

Znak	Typ przekształcenia
d	dzień miesiąca (01–31)
H	godzina w zapisie 00–23
I	godzina w zapisie 01–12
j	dzień roku (001–366)
m	miesiąc (01-12)
M	minuta (00–59)
S	sekundy (00–61)
y	rok w zapisie dwucyfrowym (00-99)
Y	rok w zapisie czterocyfrowym (np. 1066)

Instrukcje sterujące

AWK pozwala nam grupować instrukcje, podejmować decyzje (konstrukcja `if-else`) oraz tworzyć pętle (instrukcje `for`, `while`). Składnia tych instrukcji pochodzi bezpośrednio z języka C.

Pojedyncza instrukcja może być zawsze zastąpiona listą instrukcji ujętych w nawiasy grupujące. Na liście instrukcje separowane są końcami linii lub średnikami. Znaki końca linii mogą pojawić się po dowolnym lewym i przed dowolnym prawym nawiasem grupującym.

Spójrzmy przykładowo na składnię instrukcji `if-else`

`if (`⟨*wyrażenie*⟩`)`
  ⟨*instrukcja1*⟩
`else`
  ⟨*instrukcja2*⟩

część `else` ⟨*instrukcja2*⟩ jest opcjonalna.

W celu uniknięcia dwuznaczności przyjęto, że każdy `else` jest w parze z bezpośrednio poprzedzającym go `if`-em; przykładowo

```
if (e1) if (e2) s=1; else s=2
```

tu `else` jest w parze z drugim `if`-em. (Średnik po `s=1` jest wymagany, gdyż `else` znalazł się w jednej linii z `if`-em.)

Instrukcje sterujące

`{` ⟨*instrukcje*⟩ `}`
grupowanie instrukcji

`if (`⟨*w*⟩`)` ⟨*instrukcja*⟩
jeśli wyrażenie ⟨*w*⟩ jest prawdziwe, wykonaj ⟨*instrukcję*⟩

`if (`⟨*w*⟩`)` ⟨*instrukcja1*⟩ `else` ⟨*instrukcja2*⟩
Wykonaj ⟨*instrukcję1*⟩ jeśli wyrażenie ⟨*w*⟩ jest prawdziwe, w wypadku przeciwnym ⟨*instrukcję2*⟩

`while (`⟨*w*⟩`)` ⟨*instrukcja*⟩
jeśli wyrażenie ⟨*w*⟩ jest prawdziwe, wykonaj ⟨*instrukcję*⟩ i powtórz

`for (`⟨*w1*⟩`;` ⟨*w2*⟩`;` ⟨*w3*⟩`)` ⟨*instrukcja*⟩
równoważne instrukcji:
⟨*w1*⟩`; while (`⟨*w2*⟩`) {`⟨*instrukcja*⟩`;` ⟨*w3*⟩`}`

`for (`⟨*zmienna*⟩ `in` ⟨*tablica*⟩`)` ⟨*instrukcja*⟩
⟨*instrukcja*⟩ jest wykonywana dla ⟨*zmiennej*⟩ przyjmującej kolejno wartości każdego elementu ⟨*tablicy*⟩

`do` ⟨*instrukcja*⟩ `while (`⟨*w*⟩`)`
wykonywana jest ⟨*instrukcja*⟩ jeśli wyrażenie ⟨*w*⟩ jest prawdziwe i powtórz

`break`
natychmiastowe wyjście z pętli `while`, `for`, `do`

`continue`
kontynuacja iteracji w pętlach `while`, `for`, `do`

`next`
rozpoczęcie następnej iteracji głównej pętli wejściowej[7]

`exit` ⟨*wyrażenie*⟩
sterowanie jest przekazywane bezpośrednio do akcji `END`. Jeśli komendy `exit` użyto w akcji `END`, program kończy definitywnie działanie. Opcjonalne ⟨*wyrażenie*⟩ zwracane jest jako status programu.

`nextfile`
zakończenie przeglądania bieżącego pliku i przejście do przeglądania następnego z podanych w linii komend, lub (dla ostatniego pliku) przejście do wzorca `END`. *W wyniku wykonania* `nextfile` *zmienia się wartość zmiennej* `FILENAME`, *wartością* `FNR` *staje się jeden oraz wartość* `ARGIND` *jest zwiększana o jeden.*

Instrukcja pusta. Jeśli w linii programu AWK-owego umieścimy samotny znak '`;`', to otrzymamy instrukcję pustą. Spójrzmy na poniższy program wykorzystujący taką instrukcję w pętli `for`; program drukuje wszystkie linie, które zawierają puste pole.

```
BEGIN { FS = "\t" }
  { for (i=1; i<=NF && $i-""; i++)
        ;
    if (i<=NF) { print }
  }
```

Tablice asocjacyjne

Tablice asocjacyjne to jedyny rodzaj tablic dostępny w AWK-u. Tablic nie trzeba deklarować, określać ich wymiarów czy typu elementów składowych. Utworzenie elementu tablicy następuje w chwili wykonania podstawienia, np. `a[44] = 3.14` lub `a[1]="Alibaba"`, albo innego odwołania do niego. Element nie zainicjowany jest równy zero lub równy napisowi pustemu. *Indeksy nie są liczbami ale napisami*. Użycie w kontekście indeksu liczby spowoduje jej konwersję do odpowiedniego napisu. Trzeba o tym pamiętać, np. `print a["01"]` nie spowoduje wydrukowania słowa `Alibaba` (tylko przypuszczalnie napis pusty) – napisy `"1"` oraz `"01"` są oczywiście różne, podczas gdy liczby nie.

Poniższy program zapamiętuje wszystkie słowa pliku wejściowego oraz liczbę ich wystąpień.

```
{for (i=1; i<=NF; i++) {ls[$i]++}}
```

Zwróćmy uwagę, że za każdym razem, gdy pojawia się nowy wyraz, AWK tworzy nowy element tablicy z wartością początkową 0. Następnie operator inkrementacji nadaje mu wartość 1. Każde następne pojawienie się tego wyrazu powoduje zwiększenie wartości już istniejącego elementu.

Powstaje problem jak dobrać się do poszczególnych elementów tablicy `ls`, skoro nie znamy wartości indeksów (wyrazów tekstu). Do tego celu służy specjalna forma pętli `for`:

`for (`⟨*zmienna*⟩ `in` ⟨*tablica*⟩`)`
  ⟨*instrukcja*⟩

⟨*zmienna*⟩ przyjmuje iteracyjnie wszystkie wartości indeksów ⟨*tablicy*⟩. Kolejność przeglądania tablicy nie jest ustalona i jest zależna od konkretnej implementacji AWK-a. Działanie pętli jest *nieokreślone* jeżeli wewnątrz pętli zostaną dodane kolejne elementy do ⟨*tablicy*⟩. Chcąc wydrukować listę słów z naszego przykładu możemy posłużyć się następującą akcją z wzorcem `END`:

```
END {for (word in ls)
  print word, ls[word]
}
```

Wyrażenie

7: Przez główną pętlę wejściową rozumiemy mechanizm AWK-a do analizowania rekord po rekordzie pliku wejściowego.

⟨*indeks*⟩ `in` ⟨*tablica*⟩

pozwala ustalić czy określony ⟨*indeks*⟩ występuje w ⟨*tablicy*⟩. Jeżeli występuje to wartością wyrażenia jest 1, w wypadku przeciwnym 0. Przykładowo, poniższa instrukcja sprawdza czy w tablicy `ls` wystąpiło słowo `TeX`:

```
if ("TeX" in ls) {print "OK!"}
else {print "KO!"}
```

delete. Element tablicy możemy usunąć za pomocą instrukcji:

`delete` ⟨*tablica*⟩`[`⟨*indeks*⟩`]`

przykładowo `delete ls["TeX"]` usuwa z tablicy `ls` element odpowiadający indeksowi `"TeX"`.

Tablice wielowymiarowe. Tablice wielowymiarowe są symulowane przez AWK-a za pomocą tablic jednowymiarowych. Z punktu widzenia użytkownika nie ma to wielkiego znaczenia. Przykładowo w wyniku działania poniższego fragmentu programu:

```
for (i=1; i<=10; i++)
  for (j=1; j<=10; j++)
     r[i,j] = rand();
```

zostanie utworzona tablica 100 elementów, do których możemy się odwoływać za pomocą par zmiennych indeksowanych postaci `i,j`. Wewnętrznie jednakże poszczególne elementy tablic są indeksowane za pomocą napisów postacji `i SUBSEP j`. Zmienna wbudowana `SUBSEP` przechowuje znak używany do oddzielenia indeksów składowych; standardową wartością tej zmiennej nie jest przecinek ale znak `"\034"`. Sposób testowania czy element `i,j` należy do tablicy nie zmienia się:

```
for ((i,j) in r) {...}
```

zaś w wypadku pętli `for` piszemy:

```
for (k in r) {print r[k]}
```

UWAGI: Elementy tablic nie mogą być tablicami.

Funkcje

AWK umożliwia definiowanie własnych funkcji. Definicja funkcji może być umieszczona w dowolnym miejscu programu, pomiędzy kolejnymi parami wzorzec-akcja. Funkcje są definiowane następująco:

`function` ⟨*nazwa*⟩ `(`⟨*lista-argumentów*⟩`) {`
⟨*lista-instrukcji*⟩
`}`

⟨*lista-argumentów*⟩ to ciąg oddzielonych przecinkami argumentów funkcji. Podczas wywołania funkcji, argumentom nadawane są odpowiednie wartości. Nazwy argumentów są lokalne dla funkcji; są one przekazywane przez wartość, z wyjątkiem tablic, które są przekazywane „przez referencję”. Argumenty funkcji są *jedynymi* zmiennymi lokalnymi w AWK-u.

⟨*Lista-instrukcji*⟩ może zawierać instrukcje `return` ⟨*wyrażenie*⟩. Wykonanie `return` polega na obliczeniu wartość ⟨*wyrażenia*⟩, a następnie przekazaniu tej wartości w miejsce wywołania funkcji (tzw. wartość zwracana przez funkcję). ⟨*Wyrażenie*⟩ jest opcjonalne – jeżeli go nie ma, instrukcja `return` jedynie przekazuje sterowanie do miejsca wywołania. Jeżeli wśród ⟨*listy-instrukcji*⟩ nie ma `return` to po wykonaniu ostatniej instrukcji (przed zamykającym nawiasem klamrowym) sterowanie jest przekazywane do miejsca wywołania, a wartość zwracana jest nieokreślona. Zilustrujmy to prostym przykładem funkcji `max` zwracającej większy ze swoich dwu argumentów ([1], s. 53):

```
function max(x, y) {
 return x > y ? x: y
}
```

Funkcje zdefiniowane za pomocą polecenia `function` mogą być użyte w dowolnym wyrażeniu, a także wewnątrz innych funkcji; dozwolona jest także rekursja (por. przykład 7). Przy wywołaniu funkcji *nie można* umieszczać odstępu pomiędzy jej nazwą a rozpoczynającym listę argumentów nawiasem `(`.

Jak już mówiliśmy *tylko* argumenty funci są zmiennymi lokalnymi. Wszystkie inne zmienne są *globalne*. Jeżeli chcemy aby AWK „widział” jakąś zmienną tylko lokalnie to jedyną metodą jest umieszczenie jej na liście parametrów przy definiowaniu funkcji. Po prostu nadmiarowe parametry umieszczamy na końcu listy. Nie będą one wykorzystywane do przekazywania wartości, lecz będą stanowić dodatkowe zmienne lokalne. Wywołanie funkcji z mniejszą od deklarowanej liczbą parametrów jest w AWK poprawne – wszystkie nadmiarowe parametry przyjmują wartości zerowe.

PRZYKŁAD 7
Następująca funkcja rekurencyjna odwraca napis poczynając od znaku `s` ([4], s. 151).

```
function rev(str, s) {
 if (s == 0)
    return ""
 return (substr(str, s, 1) rev(str, s-1))
}
```

Dla wypróbowania dopiszmy następujący prosty program:

```
BEGIN {print rev("Vrooom",length("Vrooom"))}
```

Zakładając, że funkcja `rev` i wzorzec `BEGIN` znajdują się w pliku `rev.awk` piszemy teraz `gawk -f rev.awk`. Na ekranie powinno się pojawić:

```
mooorV
```

Wejście

AWK może czytać dane wejściowe na kilka sposobów. Najprostszym jest uruchomienie go w standardowy sposób czyli, np:

```
gawk -f prog.awk plik.we
```

W takim kontekście, zgodnie z tym co już napisano we wstępie, AWK czyta plik `plik.we` linia po linii. Jeżeli nie podamy pliku wejściowego to AWK będzie czekał na strumień danych ze standardowego wejścia (klawiatury). Często jest to działanie niezamierzone – `^C`, `^break` czy `^Z` kończą działanie programu w takiej sytuacji.

Pola. Standardową wartością wbudowanej zmiennej FS jest `" "` (spacja – odstęp). W takiej sytuacji poszczególne pola są rozdzielone odstępami lub znakami tabulacji. Sposób rozdzielania pól można zmienić przypisując zmiennej FS odpowiedni napis. Jeżeli napis ten jest *dłuższy niż jeden znak* to AWK traktuje go jako *wyrażenie regularne*. Najdłuższe (*leftmost longest*) ciągi znaków, nie zachodzące na siebie (*non overlaping*), pasujące do tego wyrażenia regularnego będą odzielać poszczególne pola w bieżącej linii. Przykładowo deklaracja:

```
BEGIN {FS ="[,;:]"}
```

powoduje, że pola będą rozdzielane przecinkiem, średnikiem lub dwukropkiem. Kiedy wartością FS jest pojedynczy znak (inny od odstępu) to ten znak jest używany do rozdzielania pól.

UWAGI: Wartość zmiennej FS może zostać nadana także z poziomu uruchomienia AWK-a za pomocą przełącznika `-F`. Przyładowo zamiast powyższego wzorca BEGIN moglibyśmy napisać w linii komend (system DOS):

```
gawk -F[,;:] -f prog.awk plik.we
```

Rekordy. Wartość zmiennej RS przechowuje znak używany przez AWK-a do oddzielania poszczególnych rekordów. Standardowo rekordy są oddzielane znakami końca linii (co odpowiada przypisaniu `RS="\n"`). W ograniczony sposób możemy zmienić sposób w jaki AWK wyróżnia poszczególne rekordy, nadając odpowiednią wartość zmiennej RS. W opisie [1] separatorem rekordu może być tylko napis jednoznakowy lub napis pusty. Jeżeli `RS=""` (napis pusty) wtedy, separatorami rekordów są puste linie (jedna lub więcej).

PRZYKŁAD 8 Wstawianie tyld (podejście naiwne)
W zadaniu wstawienie tyld po spójnikach naturalnym wydaje się podejście, przy którym rekordem jest cały akapit (odpada problem spójników kończących linijkę):

```
BEGIN {RS=""; FS="\n"; }
{
  gsub(/[ \t]+i[ \t]*\n/,"\ni ");
  gsub(/\ni[ \t]+/,"\ni~");
  gsub(/[ \t]+i[ \t]+/," i~");
  itd. dla wszystkich spójników
  print;
}
```

Wadą tego podejścia jest to, że długie akapity mogą przekroczyć możliwości pamięciowe AWK-a.

Począwszy od wersji 3.0 GAWK-a, separator rekordu może być wyrażeniem regularnym. W takiej sytuacji, każdy napis pasujący do tego wyrażenia wyznacza koniec kolejnego rekordu (z tym, że napis ten nie jest częścią tego rekordu, podobnie jak w wypadku gdy separatorem jest pojedynczy znak.

Jeżeli `RS` *jest wyrażeniem regularnym wtedy zmienna* `RT` *przechowuje dla bieżącego rekordu, napis będący jego separatorem od rekordu następnego.*

PRZYKŁAD 9 (edytor potokowy)
Następujący program ([4], s. 240–241) jest AWK-ową implementacją edytora potokowego, tj. takiego programu, który czyta strumień danych, modyfikuje go i wysyła dalej. Poniższa implementacja wyróżnia się oryginalnością pomysłu. Sposób użycia jest następujący:

gawk -f awksed.awk ⟨*co*⟩ ⟨*naco*⟩ ⟨*plik1*⟩ ⟨*plik2*⟩...

Spowoduje zastąpienie frazy (wyrażenia regularnego) ⟨*co*⟩ *na napis* ⟨*naco*⟩ *(oba argumenty są wymagalne), w plikach* ⟨*plik1*⟩ ⟨*plik2*⟩.... *Jeżeli nie podamy listy plików dane będą czytane ze standardowego wejścia.*

```
# A. Robbins, arnold@gnu.ai.mit.edu
# Thanks to Michael Brennan for the idea
# August 1995

function usage() {
  print "usage: awksed pat repl files..." > "con"
  exit 1
}
BEGIN {
  # validate arguments
  if (ARGC < 3) { usage() }
  RS = ARGV[1]; ORS = ARGV[2]

  # don't use arguments as files
  ARGV[1] = ARGV[2] = ""
}
```

```
{
  if (RT == ""){ printf "%s", $0 }
  else { print }
}
```

Idea działania jest prosta: separatorem rekordów jest ⟨co⟩ a separatorem rekordów na wyjściu ⟨naco⟩. Problemem jest jedynie sytuacja, w której ostatni rekord nie kończy się napisem pasującym do `RS`. *Jeżeli plik nie kończy się napisem pasującym do* `RS` *to zmienna* `RT` *będzie równa napisowi pustemu. Stąd, warunek* `if (RT=="")...` *gwarantuje wydrukowanie całej zawartości pliku wejściowego.*

Drugi ciekawy fragment tego przykładu to przypisanie `ARGV[1] = ARGV[2] = ""`. *Chodzi o to, żeby AWK nie traktował napisów ⟨co⟩ i ⟨naco⟩ jako nazw plików wejściowych. Dokładne wyjaśnienie znaczenia tej linijki znajduje się w punkcie Argumenty wywołania programu, s. 24.*

getline. Funkcja `getline` umożliwia czytanie danych z bieżącego lub/i z innego pliku tekstowego albo z potoku generowanego przez inny program. Poniżej zestawiono różne formy użycia `getline`.

getline	
Postać instrukcji	Inicjalizowane zmienne
`getline`	`$0`, `NF`, `NR`, `FNR`
`getline` ⟨*z*⟩	⟨*z*⟩, `NR`, `FNR`
`getline <` ⟨*plik*⟩	`$0`, `NF`
`getline` ⟨*z*⟩ `<` ⟨*plik*⟩	⟨*z*⟩
⟨*program*⟩ `| getline`	`$0`, `NF`
⟨*program*⟩ `| getline` ⟨*z*⟩	⟨*z*⟩

⟨*plik*⟩ i ⟨*program*⟩ to zmienna lub stała napisowa zawierająca odpowiednio nazwę pliku lub programu z którego AWK ma czytać strumień danych.

Dwie pierwsze formy dotyczą czytania danych z bieżącego pliku, dwie następne z ⟨*pliku*⟩. Dwie ostatnie to wczytywanie danych ze strumienia generowanego przez inny ⟨*program*⟩. Funkcja `getline` zwraca wartość 1 jeżeli wczytana została następna linia tekstu (rekord), 0 jeżeli napotkano koniec pliku (strumienia) danych oraz −1 w wypadku napotkania błędu (np. otwarcia pliku). Jeżeli po słowie `getline` występuje zmienna ⟨*z*⟩ to wczytana linia jest dostępna jako wartość tej zmiennej, w innym wypadku jest dostępna jako wartość zmiennej $0. Przykładowo pętla:

`while (getline <` ⟨*plik*⟩ `> 0) {...}`

Jest „tradycyjną” techniką umożliwiającą przejrzenie całego ⟨*pliku*⟩. Poszczególne linie dostępne są w każdej iteracji pętli jako wartości zmiennej $0.

Podobnie wygląda czytanie danych z potoku. Przykładowo chcąc przekazać do AWK-a zawartość bieżącego katalogu możemy się posłużyć następującą pętlą (dla system DOS):

```
while ("dir/b *.*" | getline > 0) {...}
```

Pliki i potoki otwarte przez AWK-a są automatycznie zamykane z chwilą zakończenia działania programu. Jeżeli jednak musimy przeglądać wielokrotnie zawartość jakiegoś pliku w obrębie jednego programu AWK-owego to za każdym razem należy zamknąć czytany plik używając instrukcji `close`, np.

`close (`⟨*plik*⟩`); close("dir /b *.*")`

PRZYKŁAD 10
Poniższy program po uruchomieniu wyświetli listę wszystkich plików, których wielkość jest większa od `SIZE`.

```
BEGIN {
 flag = 1
 SIZE = 1000000
 while ("dir \\/s/a-d" | getline) {
  gsub(/\./,"");
  if ($NF ~/[0-90-9]:[0-90-9]/ && $(NF-2)> SIZE)
     {print $0; flag =0}
 }

 if (flag)
  {print "NO FILES BIGGER THAN", SIZE; }
}
```

UWAGI: Ponieważ DOS wyświetla wielkość plików z kropkami „księgowymi” przed każdymi trzema cyframi, tj. na przykład `200.124` pozbywamy się ich za pomocą funkcji `gsub`. Instrukcja `if` najpierw testuje czy wczytana linia zawiera ostatnie pole składające się z czterech cyfr z dwukropkiem po pierwszej parze (czas ostatniej modyfikacji pliku). Ta sztuczka pozwala „odcedzić” pola nie zawierające informacji o plikach (nagłówek listy i statystykę zbiorczą). Następnie sprawdzany jest warunek czy wielkość pola jest większa od `SIZE`. Pola liczymy od końca gdyż tylko wtedy możemy jednoznacznie ustalić, która kolumna zawiera wielkość pliku (druga od końca).

PRZYKŁAD 11
Poniższy funkcja pobiera bieżącą datę z komputera i udostępnia ją w trzyelementowej tablicy `CDATE`, której element `"year"` zawiera rok, element `"mon"` – miesiąc a element `"day"` – dzień.

```
function getdate(x,date) {
  "echo. | date" | getline x;
   gsub("-", " ", x);
   split(x, date, " ");
   CDATE["year"]= date[5];
   CDATE["mon"]= date[6];
   CDATE["day"]= date[7];
   return;
}
```

UWAGI: `"echo. | date"` z góry odpowiada DOS-owi na jego durne pytanie `Enter new date...` (inaczej program będzie czekał na naciśnięcie `enter`). Argumenty `x`, i `date` nie służą do przekazywania wartości, ale do uczynienia obu zmiennych lokalnymi (por. rozważania na ten temat w punkcie: Funkcje) – funkcję wywołujemy po prostu `getdate()` [8].

Pola o ustalonej długości. *Rekord może być także dzielony na pola o ustalonej długości. W tym celu zmiennej wbudowanej `FIELDWIDTHS` nadajemy wartość napisu zawierającego ciąg oddzielonych odstępami liczb. Każda liczba oznacza długość odpowiedniego pola w znakach. Jeżeli wartością zmiennej `FIELDWIDTHS` nie jest napis pusty, pola wyznaczane są w oparciu o specyfikację podaną w tej zmiennej, a nie w oparciu wartość separatora pól (czyli zmienną `FS`). Przypisanie wartości zmiennej `FS` (np. `FS=FS`) przywraca standardowy sposób wyznaczania pól.*

Zwróćmy uwagę, że GAWK nie dokonuje żadnego sprawdzenia poprawności specyfikacji podanej w napisie `FIELDWIDTHS`

PRZYKŁAD 12
Następujący program wyświetli listę wszystkich plików, których wielkość jest większa od `SIZE`. Lista plików generowana poleceniem `dir` ma zmienną liczbę pól w zależności od tego czy nazwa pliku ma czy nie ma rozszerzenie (pomijamy na razie nagłówek i katalogi). W przykładzie 10 liczyliśmy pola od końca. Posługując się zmienną `FIELDWIDTHS` możemy rozwiązać problem inaczej:

```
BEGIN {FIELDWIDTHS = "8 1 3 14 1 8 3 5";
BEGIN { SIZE = 1000000 }

{ gsub(/\./,"",$0); }

$0 !~ /<DIR>/ && $1 !~ /^ / && $4 > SIZE {
     printf "%s %s %d\n",  $1, $3, $4;
}
```

Program uruchamiamy w „egzotyczny”, jak na system DOS, sposób (zakładając, że powyższy kod znajduje się w `chksize.awk`):

```
dir | gawk -f chksize.awk
```

Instrukcje wyjścia – `print`/`printf`

Instrukcje `print` oraz `printf` służą do drukowania. Pierwsza z nich drukuje swoje argumenty zawsze według tego samego formatu, druga umożliwia precyzyjniejsze sterowanie postacią wypisywanych danych. Dane mogą być drukowane na ekran, do pliku lub do potoku.

printf. Składnia instrukcji `printf` jest niemalże identyczna z odpowiednią funkcją języka C. Ogólna postać tej instrukcji jest następująca:

`printf` ⟨*format*⟩, ⟨*arg1*⟩,⟨*arg2*⟩,...

lub

`printf (`⟨*format*⟩, ⟨*arg1*⟩,⟨*arg2*⟩,...`)`

Napis lub zmienna napisowa ⟨*format*⟩ określa sposób przekształcania i formatowania argumentów. Zawiera on dwa rodzaje obiektów: zwykłe znaki, kopiowane po prostu przy drukowaniu oraz specyfikacje przekształceń z których każda określa sposób przekształcenia i wypisania kolejnego argumentu funkcji `printf`. Specyfikacja ta rozpoczyna się od znaku % a kończy znakiem określającym typ konwersji. Pomiędzy tymi znakami możemy użyć ponadto następujących znaków modyfikujących:

- `-`, zawartość pola jest justowana do lewego krańca pola;
- ⟨*ciąg-cyfr*⟩, określający minimalny rozmiar pola (w znakach). Przekształcony argument będzie wpisany do pola o długości *co najmniej* równej ⟨*ciąg-cyfr*⟩. Jeżeli argument składa się z mniejszej liczby znaków, to będzie ono uzupełnione do długości minimalnej znakami odstępu. Jeżeli specyfikacja długości pola rozpoczyna się cyfrą 0, to pole będzie wypełniane nie znaczącymi zerami;
- `.`⟨*ciąg-cyfr*⟩, maksymalna szerokość pola (dla napisu) lub liczba cyfr po kropce dziesiętnej.

Oto lista znaków przekształceń i ich znaczenie:

Znaki konwersji

Znak	Typ przekształcenia
c	znak
d	liczba całkowita
e	liczba postaci `-d.ddddddE[+-]dd`
f	liczba postaci `-ddd.dddddd`
g	e lub f, w zależności od tego które jest krótsze przy czym nieznaczące zera nie są drukowane
o	liczba ósemkowa bez znaku
s	napis
x	liczba szestnastkowa bez znaku

Jeżeli znak występujący po % nie jest znakiem przekształcenia to jest on po prostu wypisany; zatem %% spowoduje wypisanie znaku %.

Poniższe zestawienie ilustruje działanie różnych specyfikacji. Aby można ocenić długości pól otoczono je znakami |, zaś spacje oznaczono znakiem ␣.

Przykłady specyfikacji

Specyfikacja	Argument	Wynik
%5d%%	33.33	|␣␣␣33%|
%c	33.33	|!|

8: Zwracamy uwagę, że program jest „nieodporny” na zmianę formatu daty, co w DOS-ie ma miejsce przy uaktywnieniu innej strony kodowej. Bardziej inteligentna wersja może rozpoznawać która liczba jest dniem, która miesiącem a która rokiem po zawartości drugiej linii drukowanej przez polecenie `date`, tj. tekstu: `Enter new date (yy-mm-dd)`.

`%d`	`33.33`	`|33|`
`%5d`	`33.33`	`|␣␣␣33|`
`%e`	`3.1415`	`|3.141500e+000|`
`%f`	`3.1415`	`|3.141500|`
`%8.3f`	`3.1415`	`|␣␣␣3.141|`
`%08.3f`	`3.1415`	`|0003.141|`
`%s`	`Alibaba`	`|Alibaba|`
`%9s`	`Alibaba`	`|␣␣Alibaba|`
`%-9s`	`Alibaba`	`|Alibaba␣␣|`
`%-.3s`	`Alibaba`	`|Ali|`
`%-9.3s`	`Alibaba`	`|Ali␣␣␣␣␣␣|`

print. Instrukcja `print` jest uproszczoną formą `printf` a jej działanie jest następujące: drukowane argumenty są oddzielane znakiem ustalonym jako wartość zmiennej wbudowanej `OFS` (standardowo `OFS=" "` – argumenty odzielone są znakiem odstępu); na końcu listy jest drukowany znak ustalonym jako wartość zmiennej wbudowanej `ORS` (standardowo `ORS="\n"` – każde kolejne `print` drukuje od nowego wiersza). Wszystkie argumenty są drukowane w oparciu o tę samą specyfikację, określoną poprzez wartość zmiennej `OFMT` (standardowo `OFMT="%.6g"`). Stąd, uruchamiając poniższy przykład:

```
BEGIN {OFS=":";ORS="->";
 print log(2), log(3); print log(5);
}
```

otrzymamy

```
0.693147:1.09861->1.60944->
```

Uwagi: `print` to skrót od `print $0` (a nie jak można by się spodziewać `print ORS`).

Drukowanie do plików. Zamiast na ekran (standardowe wyjście) instrukcje `printf`/`print` mogą przesłać dane do pliku. Służą do tego operatory `>` oraz `>>`, zaś instrukcja mają wówczas postać:

`printf` ⟨*format*⟩`,` ⟨*arg1*⟩`,... >` ⟨*plik*⟩
`print` ⟨*arg1*⟩`,... >` ⟨*plik*⟩

lub

`printf` ⟨*format*⟩`,` ⟨*arg1*⟩`,... >>` ⟨*plik*⟩
`print` ⟨*arg1*⟩`,... >>` ⟨*plik*⟩

⟨*plik*⟩ jest napisem lub zmienną typu napisowego zawierającą legalną (z punktu widzenia systemu operacyjnego) nazwę pliku. Przykładowo program:

```
{ printf "%s\n", $0 > $1 }
```

będzie działał doputy, dopóki wartość pierwszego pola w pliku wejściowym zawiera napis mogący być legalną nazwą pliku (w systemie UNIX byłby problem jeżeli `$1` zawierałoby dla którejś linii pliku np. frazę `Procter&Gamble`).

Uwagi: Instrukcja `printf "%d %d\n", $1, $2 > $3` spowoduje wydrukowanie `$1` i `$2` do pliku określonego jako zawartość pola `$3`, a nie `$1` oraz wyniku porówania wartości drugiego i trzeciego pola. Jeżeli chcemy osiągnąć to drugie to powinniśmy napisać: `printf "%d %d\n", $1, ($2 > $3)` albo `printf ("%d %d\n", $1, $2 >$3)`.

Operator `>` otwiera plik tylko raz (niszcząc poprzednią zawartość); kolejne instrukcje `printf` dodają tekst do tego pliku. Operator `>>` różni się od `>` tym, że przy otwarciu pliku poprzednia zawartość nie jest niszczona.

Drukowanie w potoku. Możliwe jest także drukowanie w potoku za pomocą instrukcji `printf` (`print`) o postaci:

`printf` ⟨*format*⟩`,` ⟨*arg1*⟩`,... |` ⟨*program*⟩
`print` ⟨*arg1*⟩`,... |` ⟨*program*⟩

⟨*program*⟩ jest napisem lub zmienną napisową zawierającą nazwą programu-odbiorcy strumienia danych drukowanych przez `printf` (`print`). Rozważmy prosty program drukujący z pliku wejściowego linie zawierające więcej niż 9 pól. Nic prostszego jak zapisać `NF > 9{print }`. Jednakże gdy liczba linii spełniających ten warunek jest duża to na ekranie zobaczymy tylko dwadzieścia parę ostatnich, a reszta mignie nam przed oczami. W systemie DOS możemy usunąć tę niedogodność pisząc[9]:

```
NF > 9 {print | "more"}
```

close. Instrukcja `close` służy do zamknięcia otwartego za pomocą operatorów `>`, `>>` i `|`. Jej ogólna postać jest następująca:

`close(`⟨*napis*⟩`)`

gdzie ⟨*napis*⟩ jest identyczny z napisem ⟨*plik*⟩ lub ⟨*program*⟩, za pomocą których uprzednio otwarto plik/potok. Instrukcja ta jest niezbędna w sytuacji gdy np. najpierw piszemy do pliku a następnie chcemy (w obrębie tego samego programu) czytać z niego dane.

fflush. *Instrukcja* `fflush` *o postaci*

`fflush(`⟨*napis*⟩`)`

9: Przykład trochę naciągany gdyż to samo można osiągnąć pisząc w linii komend `gawk "NF > 9 {print }" | more`.

opróżnia bufor związany z jakimś plikiem lub potokiem (poprzez ⟨napis⟩, identyczny z napisem ⟨plik⟩ lub ⟨program⟩ za pomocą których uprzednio otwarto plik/potok). Dopuszczalne są także dwie następujące postacie instrukcji: `fflush()` *oraz* `fflush("")`. *Pierwsza opróżnia bufor związany ze standardowym wyjściem, druga bufory związane z wszystkimi otwartymi w danej chwili plikami/potokami.*

system. Instrukcja system ma postać:

system(⟨*komenda*⟩)

wykonuje komendę systemową przekazaną jej jako wartość napisowego argumentu ⟨*komenda*⟩. Najczęściej funkcje printf/printf i operatory >, >> i | wystarczają do zrealizowania typowych zadań bez potrzeby uciekania się do komendy system.

Argumenty wywołania programu

Wartości argumentów wywołania programu są, podobnie jak w wypadku języka C, przechowywane we wbudowanej zmiennej tablicowej ARGV. Zmienna ARGC zaś zawiera liczbę argumentów. Przykładowo:

```
gawk -f 1.awk test.txt v=1 b
```

ARGC ma wartość 4, ARGV[0] ="gawk", ARGV[1] ="test.txt", ARGV[2] ="v=1", ARGV[3] ="b". Jak widzimy argumenty zdefiniowane za pomocą opcji -f nie są liczone (podobnie -v) ale za to ARCV[0] jest równa nazwie programu, który uruchomiliśmy (tu „gawk"). Poniższy program wypisuje wartość wszystkich argumentów wywołania:

```
BEGIN {for (i=0; i< ARGC; i++)
         {print ARGV[i]}
}
```

Zmienne ARGC *i* ARGV *można zmieniać wewnątrz programu. Po napotkaniu końca bieżącego pliku wejściowego* AWK *pobiera następny element tablicy* ARGV *jako nazwę następnego pliku wejściowego. Przykładowo:*

```
BEGIN{ ARGV[1]="test.txt";
  if(ARGC < 2) {ARGC = 2} }
```

spowoduje, że AWK, *zawsze będzie przeszukiwał jako pierwszy plik* „test.txt" *bez względu na to jaki (i czy w ogóle) podamy w linii komend.*

Aby usunąć nazwę pliku z tablicy ARGV *możemy albo nadać odpowiedniemu elementowi tablicy wartość napisu pustego, albo posłużyć się poleceniem* `delete`. *Napis „-" oznacza standardowe wejście. Tego typu manipulacje z reguły umieszczamy wewnątrz wzorca* BEGIN, *por. przykład 9.*

Ponieważ nie wiemy na ile omówione tutaj możliwości manipulowania zmiennymi ARGV *i* ARGC *są przenośne dla innych implementacji* AWK*-a na wszelki wypadek oznaczyliśmy je jako rozszerzenia* GAWK*-a pismem pochyłym.*

PRZYKŁAD 13
W przykładzie 10 wielkość pliku była zadeklarowana na stałe co jest pewną niedogodnością, poniższa modyfikacja pozbawiona jest już tej wady. Chcąc uzyskać listę plików np. większych od 500kb, wystarczy teraz napisać `gawk -f chkbig.awk 500` (gdzie `chkbig.awk` zawiera poniższy kod).

```
BEGIN {
 flag = 1
 if (ARGC > 0) {SIZE = ARGV[1] * 1000}
 else { SIZE = 1000000} # default

 dalej jak w przykładzie 10

}
```

Uruchamianie AWK-a. Do tej pory uruchamialiśmy AWK-a pisząc:

gawk -f ⟨*prog.awk*⟩ ⟨*plik.we*⟩ ⟨*plik.we*⟩...

Jest to sposób stosowany najczęściej, ale nie jedyny. Jeżeli program AWK-owy jest bardzo krótki można napisać po prostu[10]:

gawk "⟨*instrukcje*⟩" ⟨*plik.we*⟩

np.

gawk "NF != 5{print $0}" ⟨*plik.we*⟩

AWK można wywołać z kilkunastoma opcjami. Najważniejsze z nich to -f, -F oraz -v. Dwie pierwsze już omówiliśmy, ostatnia umożliwia nadanie zmiennej[11] wartości w momencie uruchomienia programu (z linii komend). Przykład 14 ilustruje sposób wykorzystania opcji -v.

UWAGI: Z czasem gdy dorobimy się biblioteki własnych funkcji AWK-owych, może powstać problem, jak w elegancki sposób dołączać ją do różnych plików, w taki sposób, jak umożliwia to instrukcja \input w TEX-u czy instrukcja #include w C? Okazuje się, że opcja -f nie musi wcale występować tylko raz:

```
gawk -f mylib.awk -f prog.awk plik.we
```

Gdzie plik mylib.awk zawiera bibliotekę naszych funkcji. Nie jest to rozwiązanie idealne, ale według wiedzy autorów, najlepsze z możliwych.

PRZYKŁAD 14
Przykład 10 można zmodyfikować w inny sposób niż ten zaprezentowany w przykładzie 13. Wystarczy, że

10: W systemie UNIX, zamiast cudzysłowów maszynowych używamy cudzysłowów pojedynczych, tj. gawk '⟨*instrukcje*⟩' ⟨*plik.we*⟩.

11: Ogólnie zmiennym, ponieważ możemy ją używać wielokrotnie.

Rys. Miejsce AWK w TEX-owym środowisku składu

przy uruchamianiu pliku chk_big.awk będzie nadawana odpowiednia wartość zmiennej SIZE (oczywiste zmiany w pliku chkbig.awk pozostawiamy czytelnikom), np:

```
gawk -vSIZE=500000 -f chkbig.awk
```

Jeszcze wygodniejsze będzie przygotowanie odpowiedniego skryptu bat, np. chkbig.bat:

```
@echo off
if %1.==. goto STANDARD
 gawk -vSIZE=%1000 -f chkbig.awk
 goto END
:STANDARD
 gawk -vSIZE=500000 -f chkbig.awk
:END
```

Żeby otrzymać listę plików większych od np. 600kb wystarczy teraz napisać w linii komend:

```
chkbig 600
```

Podsumowanie

W artykule przedstawiono opis standardu AWK, oraz rozszerzenia tego języka oferowane przez interpreter GAWK. Przykłady przedstawione w artykule oraz kilkanaście innych nie zaprezentowanych z braku miejsca są dostępne w archiwum GUST-u (ftp.GUST.org.pl). Tam też znaleźć można dystrybucję GAWK-a (zawiera [4]), oraz [2].

Bibliografia

[1] Aho A.V., Kernighan B.W., Weinberger P.J., *The AWK Programming Language*, Addison-Wesley 1988.

[2] Aho A.V., Kernighan B.W., Weinberger P.J., *AWK – A Pattern Scanning and Processing Language*, dostępny m.in. w ftp.GUST.org.pl jako plik postscriptowy, 1978.

[3] Kernighan B.W., Ritchie D.M., *Język C*, WNT 1988.

[4] Robbins A.D., *A User's Guide for GNU AWK*, version 3.0, January 1996.

Skorowidz

Hasła oznaczone gwiazdką oznaczają rozszerzenia standardu AWK-a, zaś oznaczone znakiem † funkcje zdefiniowane przez autorów.

◇ Bogusław Lichoński
ekosg@univ.gda.pl

◇ Tomasz Przechlewski
ekotp@univ.gda.pl