AWK

Programovací jazyk textových manipulací

Obsah

Následující kapitolu jsme pro čtenáře Zpravodaje ÚVT MU vybrali z knihy připravované pro nakladatelství Grada Publishing. Nechť poslouží všem, kteří chtějí trošku více proniknout do ovládání operačního systému UNIX, i když **awk** nacházíme i v jiných systémech.

Awk je programovací jazyk pro práci s textem. V textově orientovaném UNIXu jej používáme také pro automatickou konstrukci příkazů, příkazových souborů a předzpracování vstupních dat. Název Awk pochází ze jmen jeho autorů: Alfred V. Aho, Peter J. Weinberger a Brian W. Kernighan. Interpretem tohoto jazyka je příkaz (program) **awk**. Jeho verzí se však vyskytuje více. Nejvíce funkcí má zřejmě implementace **awk** z projektu GNU (**gawk**) a jí je následující text věnován. Tato implementace má všechny rysy podle POSIX 1003.2 a navíc některá rozšíření.

1 Použití awk

Příkaz se spouští následujícím způsobem:

awk *program* [*soubory*]

Program můžeme také číst ze souboru, potom zadáme:

awk -f *soubor* [*soubory*]

Program **awk** čte řádky buď ze zadaných souborů nebo ze standardního vstupu. Výstup směřuje na standardní výstup.

2 Struktura programu

Program pro **awk** se tvoří posloupností příkazů ve tvaru:

vzorek **{** *akce* **}**
vzorek **{** *akce* **}**
atd.

V každém řádku čteném ze vstupu se hledá určený *vzorek*. Pokud se najde, provede se s řádkem zadaná *akce*. Poté, co se použijí všechny vzorky, přečte se ze vstupu další řádek a operace se opakují.

Jak *vzorek*, tak *akce* se smí vynechat. Nelze však vynechat obojí současně. Pokud není ke vzorku určena *akce*, potom se vyhovující řádek zkopíruje na výstup. Pro řádek vyhovující více vzorkům bude akce provedena vícekrát. Řádek nevyhovující žádnému ze zadaných vzorků se ignoruje.

Pokud vynecháme *vzorek*, potom se *akce* provede pro každý načtený řádek. Popis akce se musí uzavřít do složených závorek '**{ }**'. Tím se popis akce rozpozná od popisu vzorku.

3 Záznamy a položky

Vstup, který **awk** čte, dělíme do *záznamů* (records) ukončených *oddělovačem záznamu*. Implicitním oddělovačem záznamu je znak nového řádku. V tomto případě je záznamem jeden řádek. Číslo aktuálního záznamu **awk** udržuje v proměnné **NR**.

Každý zpracovávaný záznam se dělí do *položek* (field). Položky se implicitně oddělují bílým místem (mezera, tabulátor), lze však explicitně nastavit jinou hodnotu *oddělovače položek*. Na jednotlivé položky se odkazujeme **$1**, **$2** atd. Údaj za znakem dolar je číslo (ne jenom číslice). Identifikátorem **$0** se odkazujeme na celý záznam. Počet položek v aktuálním záznamu je uložen v proměnné **NF**.

Chceme-li změnit implicitní nastavení oddělovačů záznamů a položek, nastavíme novou hodnotu do proměnné: **RS** pro oddělovač záznamů a **FS** pro oddělovač položek. Obsah těchto proměnných můžeme změnit obecně na regulární výraz (v jiných verzích **awk** pouze

na libovolný jeden znak). Oddělovač položek můžeme také nastavit na příkazovém řádku při spouštění **awk** volbou **-F***c*, kde *c* je oddělovač položek.

Je-li oddělovač záznamů prázdný, potom se jako oddělovač chápe prázdný řádek na vstupu. Oddělovači položek potom jsou znaky mezera, tabulátor a nový řádek.

V proměnné **FILENAME** je uloženo jméno aktuálního vstupního souboru ('–' v případě standardního vstupu).

4 Výstup

Nejjednodušší program, který opíše standardní vstup na standardní výstup, je následující:

```
... | awk '{ print }' | ...
```

Vzorek jsme vynechali, a proto se akce **print** provede pro všechny vstupující řádky. Akce **print** bez parametrů opíše celý řádek na výstup. Užitečnější bude vybrat si určité položky a tyto vypsat, např. první dvě položky v opačném pořadí:

```
{ print $2, $1 }
```

Takový zápis akce na příkazovém řádku spouštějícím **awk** musí být nutně uzavřen do dvojice apostrofů, aby nedošlo k expanzi dvojic znaků **$1** a **$2** na poziční parametry shellu, ale aby se v nezměněné podobě předaly **awk**.

Položky oddělené v zápisu akce **print** čárkou se na výstupu oddělí aktuálně nastavenou hodnotou oddělovače položek. Položky oddělené pouze mezerou se spojí bez oddělovače. Vyzkoušejme si:

```
{ print $2 $1 }
```

Akce **print** umí vypisovat i obsahy proměnných a textové řetězce, např.:

```
{ print "Číslo záznamu=" NR, "Počet položek=" NF, $0 }
```

Takto zadaný program před kompletním záznamem vypíše číslo záznamu a počet položek v aktuálním záznamu.

Výstup můžeme rozdělit i do více výstupních souborů. Např. program:

```
{ print $1 >"soubor1"; print $2 >"soubor2" }
```

zapíše první položku do souboru `soubor1` a druhou položku do souboru `soubor2`. Lze použít i zápis >>. Potom se do souboru přidává za konec. Jméno souboru může být proměnná, obsah zapisovaný do souboru může být konstanta. Můžeme tedy napsat např.:

```
{ print "nesmysl" >>$2 }
```

Jako jméno souboru se použije obsah druhé položky (pozor, nesmí být prázdná). V tomto příkladě bude počet řádků v jednotlivých souborech znamenat četnost slov ve druhém poli.

Podobně lze výstup z akce **print** předat rourou procesu. Např. poslat poštou na adresu 'zaznamenej':

```
{ print | "mail zaznamenej" }
```

Pokud si tento příklad vyzkoušíme, zjistíme, že se procesu předá celý vstup naráz, tj. nepředává se po jednotlivých záznamech a navíc se předá až po načtení konce vstupu.

Pomocí proměnných **OFS** a **ORS** můžeme samostatně nastavit oddělovače pouze pro výstup. Obsahem **OFS** se oddělí vypisované položky a obsahem **ORS** se oddělí vypisované záznamy.

Jazyk **awk** také poskytuje možnost formátovaných výstupů pomocí akce **printf**. Tato akce se zapisuje následujícím způsobem:

printf *formát***,** *výraz***,** *výraz***,** ...

Popisovačem *formát* sdělíme strukturu výstupu a prostřednictvím nadefinované struktury vypíšeme jednotlivé *výrazy*. V popisovači *formát* se používá stejná syntaxe jako v jazyce C. Uveďme si ve stručnosti možnosti:

Do řetězce *formát* vkládáme text, který se má vypsat. Na místa, kde se má vypsat výsledek výrazu, vložíme popisovač ve tvaru:

%*příznaky šířka přesnost typ konverze*

Prvnímu popisovači (každý popisovač začíná vždy znakem **%**) se přiřadí výsledek prvního výrazu, druhému popisovači výsledek druhého výrazu atd. Jednotlivá pole popisovače mají tento význam (všechna pole vyjma *konverze* jsou nepovinná):

příznaky

se uvádějí žádný nebo více. Možné příznaky jsou:

-	Výstup bude zarovnán vlevo. Bez uvedení tohoto příznaku bude výstup zarovnán vpravo.
+	Výstup čísla se znaménkem bude znaménko vždy obsahovat. Bez uvedení tohoto příznaku se uvede pouze případné záporné znaménko.
mezer	Stejné jako příznak +, jenom se místo kladného znaménka vytiskne mezera.
a	Pokud se uvede + i mezera, potom + vyhraje.
#	Výstup se převede do alternativní podoby. Podrobnosti jsou uvedeny u popisu každé konverze.

šířka

Význam pole záleží na typu použité konverze.

přesnost

Přesnost uvádí, kolik číslic se má vypsat vpravo od desetinné tečky. Číslu odpovídající přesnosti musí předcházet tečka. Pokud se uvede tečka bez čísla, použije se hodnota 0. Přesnost lze zadat pouze s konverzemi **e**, **E**, **f**, **g** a **G**.

typ

Pole může obsahovat znaky **h**, **l** nebo **L**. Symbol **h** sděluje, že se argument před výstupem převede na formát **short**, typ **l** převede na formát **long int** a typ **L** na formát **long double**.

konverze

Konverze obsahuje jeden znak, který sděluje, jak se má *výraz* vytisknout.

Na místě *konverze* se smějí použít tyto symboly:

i nebo **d**
Předpokládá se celočíselný argument (**int**), který se interpretuje jako číslo se znaménkem. Pole *šířka* může obsahovat minimální počet znaků, na který se číslo vypíše. Implicitně je šířka 1. Význam příznaku # není definován.

o Celočíselný argument bez znaménka se vytiskne osmičkově. Význam pole *šířka* je stejný jako u konverze **i**. Příznak # zvýší šířku tak, aby první číslice byla nula.

u Celočíselný argument bez znaménka se vytiskne desítkově. Význam pole *šířka* je stejný jako u konverze **i**. Význam příznaku # není definován.

x Celočíselný argument bez znaménka se vytiskne šestnáctkově. Na místě číslic se použijí i znaky `abcdef`. Význam pole *šířka* je stejný jako u konverze **i**. Příznak # zvýší šířku tak, aby první číslice byla nula.

X Celočíselný argument bez znaménka se vytiskne šestnáctkově. Na místě číslic se použijí i znaky `ABCDEF`. Význam pole *šířka* je stejný jako u konverze **i**. Příznak # zvýší šířku tak, aby první číslice byla nula.

f Argument ve dvojnásobné přesnosti **double** se převede do tvaru '[-]*ddd*.*ddd*'. Pole *šířka* uvádí minimální počet vytisknutých znaků. Pole *přesnost* může za tečkou specifikovat počet desetinných míst. Po uvedení příznaku # se vypíše desetinná tečka, i když nenásleduje žádná desetinná číslice.

e Argument ve dvojnásobné přesnosti **double** se převede do tvaru '[-]*d*.*ddd*e*dd*'. Exponent budou vždy alespoň dvě číslice. Zobrazuje-li se hodnota 0, potom je i exponent nulový. Význam pole *přesnost* a příznaku # je stejný jako u konverze **f**.

E Stejné jako **e** s tím rozdílem, že se místo `e` vypíše `E`.

g Stejné jako **f** nebo **e**. Konverze **e** se použije tehdy, pokud exponent je menší než -4 nebo je větší než přesnost.

G Stejné jako **g** s tím rozdílem, že se místo `e` vypíše `E`.

c Celočíselný argument se převede na typ **unsigned char** a výsledný znak se vypíše (tj. ordinální hodnotu převede na ASCII znak). Význam pole *přesnost* a příznaku # je nedefinován.

s Předpokládá se, že argumentem je řetězec znaků (**char ***). Vypíší se znaky řetězce až po (ale bez) ukončovacího **null** znaku. Pole *šířka* představuje maximální počet znaků, které se vypíší. Význam příznaku # je nedefinován.

Akce **printf** negeneruje žádné výstupní oddělovače. Všechny se musejí specifikovat ve *formátu*. Uveďme si příklad použití:

```
{ printf "Průměr=%8.2f, počet pokusů=%10ld\n", $1, $2 }
```

První položka se vytiskne jako číslo v pohyblivé řádové čárce celkem na 8 znaků se dvěma číslicemi za desetinnou tečkou. Druhá položka se vytiskne jako long integer na 10 znaků. Znak **n** představuje nový řádek.

5 BEGIN a END

BEGIN a **END** jsou speciálními případy vzorků. Vzorek **BEGIN** specifikuje akci, která se má provést dříve, než se přečte první záznam vstupu. Naopak vzorek **END** popisuje akci, která se provede po zpracování posledního čteného záznamu. Tímto způsobem můžeme řídit zpracování před a po čtení záznamů.

Jako příklad uveďme nastavení specifického oddělovače položek a vytisknutí počtu načtených záznamů:

```
BEGIN { FS = ":" }
...zbytek programu...
END { print NR }
```

BEGIN musí být jako první vzorek (je-li uveden), **END** musí být posledním vzorkem.

6 Akce

Akce programu **awk** je posloupnost příkazů vzájemně oddělených novým řádkem nebo středníkem.

7 Proměnné, výrazy a přiřazení

Jazyk **awk** proměnné zpracovává podle kontextu: buď jako numerické hodnoty (v pohyblivé řádové čárce), nebo jako řetězce znaků. Řetězce se na numerické hodnoty převádějí podle potřeby. Potom např.

```
x = 1
```

je typicky numerický přiřazovací příkaz, ale v příkazu

```
x = "3" + "4"
```

se řetězce převedou na numerické hodnoty a proměnné `x` se přiřadí numerická hodnota 7. Řetězce, ze kterých nelze získat numerickou hodnotu, mají hodnotu 0.

Proměnná, které dříve nebyla přiřazena hodnota, má hodnotu nula. Interním proměnným **awk** se přiřazují hodnoty automaticky (viz dále). Proto např. program

```
    { s1 += $1; s2 += $2 }
END { print s1, s2 }
```

může k proměnné `s1` přičítat. Proměnná interpretující se jako řetězec bez přiřazené hodnoty obsahuje prázdný řetězec.

Potřebujeme-li se ujistit, že proměnná bude chápána jako numerická, přičteme k ní hodnotu 0. Potřebujeme-li naopak proměnnou interpretovat jako řetězec, připojme k ní prázdný řetězec, např.

```
b = 12 ""
```

I když se numerické proměnné zpracovávají a ukládají v pohyblivé řádové čárce, desetinná tečka a číslice za ní se vypisují jenom tehdy, pokud je desetinná část nenulová. Číslo se na řetězec konvertuje podle obsahu proměnné CONVFMT voláním **sprintf**. Řetězec se na číslo konvertuje voláním **atof**.

Příklad

Na závěr první části pojednání o **awk** uveďme ilustrační příklad: Chceme **awk** použít na převod výstupu příkazu `ls -l` do klasického DOSovského tvaru výpisu adresáře, tj. ze tvaru:

```
-rw-r--r-- 1 brandejs staff
             173741 May 16 21:10 text2.tex
```

do tvaru:

```
text2.tex              173741 May 16 21:10
```

Sestavíme-li kolonu:

```
ls -l | grep -v ^total | awk -f dirp
```

potom pro **awk**[1] potřebujeme následující program v souboru `dirp`:

```
BEGIN {print "User is " ENVIRON["LOGNAME"]; print }
  {printf "%-18s %10i %3s %2i %4s\n", $9, $5, $6, $7, $8;
   suma = suma + $5 }
END {printf "      %4i file(s)%11i bytes\n", NR, suma }
```

Příkazem `grep -v ^total` zrušíme nevýznamný řádek začínající slovem `total`. Pole `ENVIRON` popíšeme v pokračování.

8 Regulární výrazy

Na místě vzorku se mohou vyskytnout jak základní regulární výrazy (RE) podle definice v příkazu **ed**, tak i rozšířené regulární výrazy podle definice v příkazu **grep -E**. Regulární výraz se uzavírá do dvojice lomítek. Např. program

```
/L.*x/
```

vypíše všechny řádky, které obsahují nejprve znak `L` a potom `x`. Hledání vyhovujícího vzorku můžeme omezit např. na určitou položku. Např. program

```
$1 ~ /^[Ll].*x$/
```

vypíše ty řádky, jejichž první položka začíná písmenem `L` nebo `l` a končí písmenem `x`. Operátor `!~` vybere ten řádek, který vyhovující vzorek neobsahuje.

9 Relační výrazy

Jazyk **awk** povoluje použití relačních operátorů <, >, ==, != , >= a <=. Např. program

```
$2 > $1 + 99
```

vypíše ty řádky, jejichž druhá položka je alespoň o 100 větší než položka první. Aritmetické operátory se používají stejné jako v aritmetických expanzích shellu. Např. program

```
NF % 2 == 0
```

vypíše všechny řádky se sudým počtem položek. Relační operátory se mohou používat jak pro aritmetické srovnávání, tak i pro porovnávání řetězců. Proto např. program

```
$1 >= "s"
```

vybere ty řádky, jejichž první položka začíná znakem `s`, `t`, `u` atd.

10 Kombinace vzorků

Na místě vzorku se může vyskytnout i jejich kombinace spojená booleovskými operátory && (AND - logický součin), || (OR - logický součet) a ! (NOT - negace). Proto např. vzorek

```
$1 >= "l" && $1 < "o" && $1 !~ /^l.*x$/
```

způsobí výpis řádku, jehož první položka začíná písmenem `l` až `n` a zároveň položka nezačíná `l` a zároveň nekončí `x`. Operátory && a || se vyhodnocují zleva doprava. Vyhodnocování se zastaví v okamžiku, kdy je výsledek vzorku jasný.

11 Interval určený vzorky

Vzorek, kterým se vybírá akce, se může skládat ze dvou vzorků. Potom levý vzorek určuje první řádek a pravý vzorek určuje poslední řádek, pro který se akce provede. Např. vzorek

```
/start/,/stop/
```

vypíše všechny řádky od řádku vyhovujícího vzorku `/start/` po řádek vyhovující vzorku `/stop/`. Např. program

```
NR == 100, NR == 200
```

vypíše záznamy (řádky) 100 až 200.

12 Identifikátor položky jako proměnná

Identifikátory položek (**$1**, ...) požívají stejných vlastností jako proměnné. Mohou se používat jak v aritmetickém, tak i řetězcovém kontextu. Lze jim také přiřadit hodnotu. Proto lze napsat např.

```
{ $2 = NR; print }
```

nebo dokonce

```
{ $1 = $2 + $3; print $0 }
```

Odkazy na konkrétní položky se mohou také vyjádřit numerickým výrazem, např.

```
{ print $i, $(i+1), $(i+n) }
```

13 Příkazy pro řízení toku

Jazyk **awk** dovoluje použít příkazy pro řízení toku obvyklé u vyšších programovacích jazyků. Jde o tyto příkazové konstrukce:

```
if (podmínka) příkaz [ else příkaz ]
while (podmínka) příkaz
do příkaz while (podmínka)
for (výraz1; výraz2; výraz3) příkaz
for (proměnná in pole) příkaz
break
continue
next
delete pole[index]
exit [ výraz ]
{ příkaz[; příkaz ... ] }
```

Z výše uvedených konstrukcí můžeme vytvořit např. tyto příklady:

```
{ if ($3 > 1000)
     $3 = "moc velké"
  print
}
```

Následující příklad vytiskne vždy jednu položku na jeden řádek:

```
{ i = 1
while (i <= NF) {
    print $i
    ++i
}}
```

V příkladu jsme si ukázali, že na místě *příkazu* smí být i více příkazů uzavřených do složených závorek. Následující příkaz provede totéž:

```
{ for (i = 1; i <= NF; i++) print $i }
```

První výraz znamená počáteční přiřazení, druhý výraz představuje podmínku a třetí výraz se opakovaně provádí.

Příkaz **break** okamžitě ukončí provádění cyklu **while** nebo **for**. Příkaz **continue** přejde ihned na novou iteraci cyklu.

Příkazem **next** přejdeme na zpracování dalšího řádku (záznamu) vstupu. Příkaz **exit** je totéž, co načtení konce vstupu (souboru).

Do programu pro **awk** lze vkládat komentáře. Řádek s poznámkou musí začínat znakem **#**.

14 Pole

Pole (arrays) se v **awk** předem nedeklarují. Jako příklad použití pole uveďme

```
{ x[NR] = $0 }
```

Tímto programem načteme celý vstup do jednorozměrného pole a zpracujeme jej až v akci náležející speciálnímu vzorku **END**.

Prvky pole můžeme indexovat také nenumerickými hodnotami, např. řetězci. Ukažme si opět příklad použití

```
/modra/     { x["modra"]++ }
/cervena/   { x["cervena"]++ }
END         { print x["modra"],
              x["cervena"] }
```

Pole v **awk** mohou být obecně *n*-rozměrná. Jejich prvky se totiž ukládají tak, jak je tomu u asociativních pamětí. S prvkem je uložen i jeho index. V případě více prvkových položek se jednotlivé indexy od sebe oddělují oddělovačem \034 (tuto hodnotu lze změnit proměnnou SUBSEP, viz dále). Potom např.

```
i = "A"; j = "B"; k = "C"
x[i,j,k] = "hello, world\n"
```

do paměti uloží index ve tvaru A\034B\034C. V příkazu **for** můžeme použít zvláštní operátor **in** následovně:

for (*proměnná* **in** *pole***) ...**

Operátor **in** v příkazu **for** zajistí, že *proměnné* se budou postupně přiřazovat všechny prvky *pole*. Prvky jsou přiřazovány obecně v náhodném pořadí. Pokud obsah *proměnné* změníme nebo pokud současně zpřístupňujeme i jiné prvky pole, nastane zřejmě chaos. Je-li pole vícerozměrné, můžeme pro uložení všech indexů (jako řetězce) použít jednu proměnnou.

Operátor **in** můžeme použít i v příkazech **if** a **while** ke zjištění, zda element určitého indexu se v poli nachází. Můžeme napsat

```
if (hodnota in pole) print pole[hodnota]
```

V případě vícerozměrných položek používáme zápis např.

```
if (("A","B","C") in x)
   print x["A","B","C"]
```

Prvky pole rušíme příkazem **delete** např. následovně:

```
delete x["A","B","C"]
```

15 Funkce

Funkce nebyly do původního **awk** zahrnuty. Definují se tímto způsobem:

```
function jméno(seznam_parametrů) { příkazy }
```

Při volání funkce se formální parametry nahradí aktuálními. Pole se předávají odkazem, ostatní proměnné hodnotou.

Lokální proměnné se ve funkcích definují dost zvláštním způsobem. Je to zapříčiněno faktem, že původní **awk** pojem lokální proměnné nezná. Lokální proměnné se definují v rámci *seznamu_parametrů* tak, že se uvedou na konci a oddělí se více mezerami, např.:

```
function f(p, q,    a, b)
{ # proměnné a, b jsou lokální
                            ...  }
/abc/    { ... ; f(1, 2); ... }
```

Levá kulatá závorka musí následovat bezprostředně za jménem funkce (bez bílého místa). Tím se odstraní případné nejednoznačnosti syntaxe (možnost záměny s konkatenací).

Z funkce se smí volat jiná funkce a funkce smějí být rekurzivní. Na místě slova **function** se smí použít jenom **func** (rozšíření GNU verze).

16 Interní proměnné awk

ARGC
: Obsahuje počet argumentů zadaných na příkazovém řádku při spouštění **awk**. Volby se do argumentů nepočítají.

ARGIND
: Index do **ARGV** na právě zpracovávaný soubor.

ARGV
: Pole argumentů z příkazového řádku. Pole se indexuje od 0 do ARGC-1. Obsah tohoto pole můžeme měnit a tím řídíme výběr souborů ke zpracování.

CONVFMT
: Formát pro konverzi čísel, implicitně '`%.6g`'.

ENVIRON
: Pole obsahující kopii proměnných prostředí. Indexem je jméno proměnné; vyzkoušejte např. `ENVIRON["HOME"]`. Obsah jednotlivých prvků pole můžete měnit. Tato změna se však neprojeví na proměnných prostředí předávaných procesům, které jsou z **awk** spouštěny (např. symbolem |). Tato vlastnost se může v dalších verzích změnit.

ERRNO
: Tato proměnná obsahuje řetězec popisující chybu, která vznikla během poslední operace getline, přesměrování a zavření.

FILENAME
: Jméno právě zpracovávaného souboru nebo '-' pro standardní vstup. Proměnná FILENAME je uvnitř bloku BEGIN nedefinována.

FNR
: Číslo vstupního záznamu v rámci aktuálního souboru.

FS

Oddělovač položek na vstupu; implicitně mezera.

IGNORECASE
: Proměnná řídí zpracování malých a velkých písmen ve všech operacích s regulárními výrazy. Obsahuje-li proměnná IGNORECASE nenulovou hodnotu, potom se ignorují rozdíly mezi malými a velkými písmeny. Implicitně proměnná obsahuje nulu.

NF
: Počet položek aktuálně zpracovávaného záznamu.

NR
: Počet načtených záznamů.

OFMT
: Výstupní formát čísel, implicitně '`%.6g`'.

OFS
: Oddělovač položek na výstupu; implicitně mezera.

ORS
: Oddělovač záznamů na výstupu; implicitně nový řádek.

RS
: Oddělovač záznamů na vstupu; implicitně nový řádek. Z řetězce RS se akceptuje pouze první znak (což se může v budoucích verzích změnit). Pokud RS obsahuje prázdný řetězec, potom se záznamy oddělují prázdným řádkem a položky novým řádkem bez ohledu na nastavení FS.

RSTART
: Index prvního vyhovujícího znaku vraceného funkcí match(). Pokud žádný nevyhovuje, potom proměnná obsahuje 0.

RLENGTH
: Délka vyhovujícího řetězce vraceného funkcí match(). Pokud žádný nevyhovuje, potom proměnná obsahuje -1.

SUBSEP
: Znak oddělující jednotlivé indexy při ukládání indexu prvku vícerozměrného pole. Implicitně \034.

17 Vstupní a výstupní funkce

Následuje souhrnný přehled dosud uvedených i neuvedených vstupních a výstupních funkcí:

close(*soubor***)**
: Zavře soubor (nebo přesměrování).

getline
: Načte další záznam do $0, nastaví se NF, NR, FNR.

getline <*soubor*
: Načte další záznam do $0 ze souboru *soubor*, nastaví se NF, NR, FNR.

getline *var*
: Načte další záznam do *var*, nastaví se NF, NR, FNR.

getline *var* <*soubor*
: Načte další záznam do *var* ze souboru *soubor*, nastaví se NF, NR, FNR.

next
: Ukončí se zpracovávání běžného záznamu (řádku). Přečte se další záznam a zahájí se provádění prvního příkazu programu. Pokud se načte konec souboru, provede se blok END (je-li nějaký).

next file
: Ukončí se zpracovávání aktuálně čteného souboru. Další řádek se čte už z dalšího souboru. Obnoví se FILENAME, FNR se nastaví na 1. Přečte se další řádek a zahájí se provádění prvního příkazu programu. Pokud se načte konec souboru, provede se blok END (je-li nějaký).

print
: Vytiskne běžný záznam.

print *seznam_výrazů*
: Vytiskne uvedený seznam výrazů.

print *seznam_výrazů >soubor*
: Vytiskne uvedený seznam výrazů do souboru.

printf *formát*, *seznam_výrazů*
: Vytiskne uvedený seznam výrazů podle zadaného formátu.

printf *formát*, *seznam_výrazů >soubor*
: Vytiskne uvedený seznam výrazů podle zadaného formátu do souboru.

system(*příkaz_systému***)**
: Shell provede zadaný *příkaz_systému* a převezme se návratový kód.

18 Numerické funkce

V **awk** můžeme používat následující vestavěné aritmetické funkce.

atan2(*y***,** *x***)**
: Vypočte arkustangens *y*/*x* a vrátí hodnotu v radiánech.

cos(*výraz***)**
: Vrátí kosinus výrazu v radiánech.

exp(*výraz***)**
: Exponenciální funkce.

int(*výraz***)**
: Odřízne desetinnou část.

log(*výraz***)**
: Přirozený logaritmus.

rand()
: Náhodné číslo mezi 0 a 1.

sin(*výraz***)**
: Sinus.

sqrt(*výraz***)**
: Druhá odmocnina.

srand(*výraz***)**
: Zadaný *výraz* se použije na "rozmíchání" generátoru náhodných čísel. Není-li *výraz* uveden, potom se použije aktuální hodnota času. Funkce vrátí předchozí hodnotu výrazu.

19 Řetězcové funkce

gsub(*r*, *s*, *t***)**
: Každý podřetězec řetězce *t* vyhovující regulárnímu výrazu *r* se nahradí řetězcem *s*. Funkce vrací počet náhrad. Pokud nebyl zadán řetězec *t*, použije se $0.

index(*s*, *t***)**

Vrátí index podřetězce *t* v rámci řetězce *s* nebo 0, pokud se podřetězec nenašel.

length(*s***)**
Funkce vrátí délku řetězce *s* nebo řetězce $0, pokud jsme *s* nezadali.

match(*s***, *r*)**
Vrátí se pozice uvnitř řetězce *s*, od které začíná podřetězec vyhovující RE *r*. Nenajde-li se, vrátí se 0. Funkce rovněž nastavuje proměnné RSTART a RLENGTH.

split(*s***, *a*, *r*)**
Rozdělí řetězec *s* do jednorozměrného pole *a* podle RE *r*. Pokud *r* nezadáme, použije se obsah FS. Funkce v řetězci najde všechny výskyty oddělovače podle *r*. Tyto výskyty rozdělí řetězec na podřetězce a jednotlivé podřetězce se uloží do elementů pole *a* číslovaných od 1. Funkce vrátí počet podřetězců.

sprintf(*formát***, *výrazy*)**
Funkce vytiskne *výrazy* podle zadaného *formátu*. Vrátí se výsledný řetězec.

sub(*r*, *s*, *t***)**
Funkce se od **gsub()** liší tím, že se nahradí pouze první vyhovující řetězec.

substr(*s*, *i*, *n***)**
Funkce vrací *n*-znakový podřetězec řetězce *s* začínající na pozici *i*. Pokud *n* vynecháme, vezme se zbytek řetězce *s*.

tolower(*řetězec***)**
Vrací se *řetězec* se všemi velkými písmeny převedenými na malá písmena. Jiné znaky než velká písmena zůstanou beze změny.

toupper(*řetězec***)**
Vrací se *řetězec* se všemi malými písmeny převedenými na velká písmena. Jiné znaky než malá písmena zůstanou beze změny.

20 Časové funkce

Program GNU **awk** poskytuje možnost doplňovat produkovaný text informací o datu a čase.

systime()
Funkce předává aktuální čas v počtu sekund od 1.1.1970 (v POSIX kompatibilních systémech).

strftime(*formát***, *čas***)**
Funkce zformátuje údaj *čas* podle zadaného *formátu*. Hodnota *čas* musí být v takovém tvaru, jaký produkuje **systime()**. Pokud hodnotu *čas* vynecháme, uplatní se aktuální čas. Řetězec *formát* se zadává stejně jako ve stejnojmenném knihovním podprogramu.

Řetězec *formát* funkce **strftime()** obsahuje jak identifikaci časových údajů (začínají znakem **%**), tak i normální text - ten se opíše beze změn. Časové údaje jsou následující (uživatel by měl mít možnost konfigurovat údaje předávané touto funkcí podle národních zvyklostí):

%a Zkratka dne v týdnu.
%A Plné jméno dne v týdnu.
%b Zkratka názvu měsíce.
%B Plný název měsíce.
%c Preferovaný způsob zápisu data a času.
%d Den v měsíci (desítkově).
%H Hodina ve 24hodinovém cyklu (00 až 23).
%I Hodina ve 12hodinovém cyklu (01 až 12).

%j Den v roce (001 až 366).
%m Měsíc (01 až 12).
%M Minuta desítkově.
%p AM nebo PM.
%S Sekunda.
%U Týden v roce (desítkově). První týden začíná první nedělí v roce.
%W Týden v roce (desítkově). První týden začíná prvním pondělím v roce.
%w Den v týdnu (desítkově). Neděle je den 0.
%x Preferovaný způsob zápisu data bez času.
%X Preferovaný způsob zápisu času bez data.
%y Rok bez století (00 až 99).
%Y Rok včetně století.
%Z Jméno nebo zkratka časové zóny.
%% Znak procento.

Uveďme si příklad:

```
strftime("Dnes je %d. %m. %Y a máme %H:%M hodin.")
```

21 Řetězcové konstanty

Řetězcové konstanty uvnitř programu pro **awk** jsou řetězce uzavřené do dvojic uvozovek. Uvnitř těchto řetězcových konstant můžeme používat tyto *escape posloupnosti*:

\\ Znak obrácené lomítko.
a Znak 'zvonek' - pípne (alert).
b Backspace - návrat o znak zpět.
f Nová stránka (form feed).
n Nový řádek (new line).
r Návrat vozíku (carriage return).
t Horizontální tabulátor.
v Vertikální tabulátor.
x*hex* Znak zapsaný šestnáctkově, např. `\x1B` je znak ESCAPE.
oct Znak zapsaný osmičkově, např. `\033` je znak ESCAPE.
c Literál znaku *c*.

Výše uvedené escape posloupnosti se mohou použít i uvnitř jednoznakového RE reprezentujícího třídu znaků. Např. '`/[ \t\f\n\r\v]/`' vyhovuje všem znakům typu "bílé místo".

22 Speciální soubory

GNU verze **awk** při přesměrovávání výstupu příkazů print, printf a při čtení pomocí getline interně zpracovává následující speciální soubory. Tyto speciální soubory zprostředkují přístup k otevřeným popisovačům souborů zděděných od rodiče (zpravidla od shellu) nebo poskytnou informace o procesu.

/dev/pid
: Přečtením tohoto souboru obdržíme číslo aktuálně běžícího procesu ukončené znakem nového řádku.

/dev/ppid
: Přečtením tohoto souboru obdržíme číslo rodičovského procesu.

/dev/pgrpid
: Přečtením tohoto souboru získáme skupinové ID aktuálně běžícího procesu.

/dev/user
: Přečtením tohoto souboru obdržíme jeden záznam ukončený novým řádkem. Položky záznamu jsou odděleny mezerou. Položka $1 obsahuje uživatelské ID z volání getuid(), $2 obsahuje uživatelské ID z volání geteuid(), $3 obsahuje ID skupiny z getgid(), $4 je hodnota z getegid(). Případné další položky jsou hodnoty vrácené voláním getgroups().

/dev/stdin
: Standardní vstup.

/dev/stdout
: Standardní výstup.

/dev/stderr
: Standardní chybový výstup.

/dev/fd/*n*
: Soubor spojený s otevřeným popisovačem souboru číslo *n*.

Výstup na standardní chybový výstup můžeme poslat např. tímto příkazem:

```
print "Stala se chyba!" > "/dev/stderr"
```

23 Volby GNU awk

Identifikátory voleb podle POSIX začínají jedním znakem minus, volby podle GNU začínají dvěma minusy. V rámci jedné volby **-W** lze zadat více slovních voleb, stejně tak lze zadat i více voleb **-W** na jednom řádku.

-F*fs* nebo **--field-separator=***fs*
: Nastavení oddělovače položek na vstupu (tj. určení implicitní hodnoty proměnné FS).

-v *proměnná=hodnota* nebo **--assign=***proměnná=hodnota*
: Uvedené proměnné se přiřadí zadaná hodnota. Takto nastavené proměnné jsou k dispozici již během bloku BEGIN.

-f *soubor* nebo **--file=***soubor*
: Program se bude číst ze zadaného souboru. Těchto voleb může být na příkazovém řádku více.

-W compat nebo **--compat**
: Program GNU **awk** se spustí v režimu kompatibility s klasickým **awk**. Všechna GNU rozšíření jsou vypnuta.

-W copyleft nebo **-W copyright** nebo **--copyleft** nebo **--copyright**
: Na standardní chybový výstup se vypíše GNU copyright.

-W help nebo **-W usage** nebo **--help** nebo **--usage**
: Na standardní chybový výstup se vypíše krátká nápověda k programu.

-W lint nebo **--lint**

Program upozorní na ty programové konstrukce, které nemusejí jiné implementace **awk** zpracovat.

-W posix nebo **--posix**

Zapne se kompatibilní režim s následujícími omezeními:

- Nerozpoznají se escape posloupnosti *x*.
- Nerozpozná se synonymum **func** pro **function**.
- Na místo operátorů ^ a ^= se nesmějí použít operátory ** a **=.

-W source=*program* nebo **--source=***program*

Jako argument této volby se zadává text programu pro **awk**. Význam spočívá v možnosti číst v rámci jednoho spuštění **awk** program jak ze souboru (např. knihovnu funkcí), tak i z příkazového řádku (vlastní krátký program). V tomto případě nelze v rámci volby **-W** zadat volbu další.

-W version nebo **--version**

Na standardní chybový výstup předá informaci o verzi aktuálního **awk**.

--

Oznamuje konec zadávání voleb.

24 Proměnná AWKPATH

Do proměnné prostředí AWKPATH můžeme obvyklým způsobem nastavit seznam cest k adresářům, které se budou procházet při hledání programů (tj. uvedených za volbou **-f**). Pokud proměnná neexistuje, použije se implicitní hodnota:

```
.:/usr/lib/awk:/usr/local/lib/awk
```

25 GNU awk rozšíření

Na závěr shrneme rozšíření GNU verze **awk** oproti definici POSIX (viz též volba **-W compat**):

- Escape posloupnosti *x*
- Funkce systime() a strftime()
- Rozpoznávání jmen zvláštních souborů
- Proměnné ARGIND, ERRNO, AWKPATH a IGNORECASE
- Příkaz 'next file'

26 Ilustrační příklady

Příklad 1:

Mějme textovou databázi předmětů, ze kterých si student při zápisu do semestru vybírá. Jednotlivé položky oddělujeme dvojtečkou. Učitel je v databázi jednoznačně identifikován svým přihlašovacím jménem. Pokud je více než jeden učitel, pak jsou jejich přihlašovací jména oddělena čárkou. Databáze nechť má tento tvar:

```
P000:3:zk:Architektura počítačů:brandejs
P004:2:zk:UNIX:brandejs
I011:2:zk:Sémantiky programovacích jazyků:zlatuška
```

Úkol zní: vypsat obsah databáze v "čitelném" tvaru a převést přihlašovací jméno na příjmení a první písmeno křestního jména. Pro tento účel si vytvořme např. soubor `logins` s následujícím obsahem:

```
adelton:Jan:Pazdziora
brandejs:Michal:Brandejs
kas:Jan:Kasprzak
kron:David:Košťál
zlatuska:Jiří:Zlatuška
```

Pro převod přihlašovacího jména na příjmení pak použijeme jednorozměrné pole, kde indexem bude přihlašovací jméno a obsahem zkonstruovaná dvojice příjmení a křestního jména. Pole naplňujeme pouze jednou, a to v části BEGIN. Databáze předmětů nechť se načítá ze standardního vstupu.

```
... | awk -F':' 'BEGIN { while ( getline <"logins" == 1 ) {
logins[$1]=sprintf("%s %1.1s.", $3, $2) } }
{ pocet=split($5,loginy,",")
for (i=1; i <= pocet; i++) {
if (i == 1) last = logins[loginy[i]]
else last = last ", " logins[loginy[i]]; }
printf("%-5s %-50.50s %1s %2s %s\n",$1,$4,$2,$3,last) }'
```

Příklad 2:

Následujícím programem spočítáme počet všech různých slov v textu:

```
BEGIN { FS="[^a-zA-Z]" }
{ for (i=1;i<=NF;i++) words[$i] = "" }
END { delete words[""]
      # prázdné položky nepočítáme
      for (i in words) sum++
      print "V textu bylo " sum " různých slov"
    }
```

Příklad 3:

A poslední vzorový program vykonává totéž co příkaz **wc**:

```
{words+=NF; chars+=1+length($0)}
END {print NR,words,chars}
```