Introdução

Awk é uma linguagem procedural (ou imperativa) interpretada. As suas vantagens em relação a C ou Pascal são:

1. se for necessário fazer frequentemente alterações em vários arquivos texto, onde quer que seja que certos padrões apareçam;
2. se for necessário extrair dados de algumas linhas de um arquivo enquanto o resto é descartado.

Ou seja, com awk é possível gerenciar tarefas simples de “reformatar dados” com apenas algumas linhas de código.

Com awk é possível: gerenciar pequenos banco de dados pessoais; gerar relatórios; validar dados; produzir índices, e fazer outras tarefas de preparação de documentos; fazer experimentos com algoritmos que podem ser adaptados posteriormente a outras linguagens de programação.

Função básica do **awk** é procurar por linhas (ou outras unidades de texto) em arquivos que possuem certos padrões especificados no programa; para cada **padrão** (pattern) deve haver uma ação associada, isto é, quando uma linha corresponde a um dos padrões, awk realiza a ação especificada naquela linha; depois, awk continua processando as linhas de entrada desta maneira até encontrar o fim do arquivo de entrada; o conjunto de comandos **padrão-ação** pode aparecer literalmente como um programa ou em um arquivo específico com a opção **-f**; arquivos de entrada são lidos em ordem; se não há arquivos, a entrada padrão é lida.

Programas em awk são diferentes dos programas na maioria das outras linguagens porque são data-driven, isto é, deve-se descrever os dados com os quais se quer trabalhar, e o que fazer quando encontrá-los. Em outras linguagens procedurais normalmente é mais difícil descrever claramente os dados que o programa irá processar. Por isso, programas awk são mais fáceis de escrever e ler.

Quando awk é executado, deve-se especificar um programa awk que diga o que ele tem que fazer. O programa consiste numa série de regras, sendo que cada uma especifica um padrão que deve ser procurado e uma ação que deve ser tomada quando o padrão é encontrado. Sintaticamente, uma regra consiste de um padrão seguido de uma ação que deve estar entre { e }. Cada regra é separada por uma nova linha. Então, um programa awk se parece com:

```
<padrão> { <ação> }
<padrão> { <ação> }
...
```

Executando um Programa awk

Existem várias maneiras de executar um programa awk.

1) Se o programa é pequeno, é mais fácil incluí-lo no comando de execução. Por exemplo:

```
awk '<Programa>' <Arq1Entrada> <Arq2Entrada> ...
```

<programa> consiste numa série de padrões e ações. Este comando é usado quando programas simples são escritos no momento que se deseja utilizá-los. Este formato de comando instrui o shell, ou interpretador de comandos, para iniciar o awk e usar o programa para processar registros nos arquivos de entrada. As aspas simples em torno do programa instruem o shell para: não interpretar os caracteres awk como caracteres especiais do shell; e tratar todo o programa como um único argumento para awk e permitir que o programa tenha mais do que uma linha.

Também é possível executar o awk sem arquivos de entrada. Neste caso, quando se digita a linha de comando:

```
awk '<programa> '
```

O awk aplica o programa à entrada padrão, o que significa que tudo que for digitado no terminal até que seja pressionado CTRL+d. O exemplo a seguir copia os caracteres digitados para a saída padrão.

```
awk '{ print }'
Este eh um exemplo
Este eh um exemplo
da utilizacao do awk!
da utilizacao do awk!
Fim!
Fim!
```

2) Quando o programa é extenso, é mais conveniente colocá-lo em um arquivo e executá-lo da seguinte maneira:

```
awk -f <ArqPrograma> <Arq1Entrada> <Arq2Entrada> ...
```

-f <ArqPrograma> intrui o awk a pegar o programa do arquivo. Qualquer nome pode ser usado no arquivo. Por exemplo, é possível colocar o programa:

```
BEGIN { print "Hello World!" }
```

No arquivo “exemplo”. O comando para execução é:

```
awk -f exemplo
```

Se não fosse utilizado um arquivo, o comando para execução seria:

```
awk "BEGIN { print \"Hello World! \" }"
```

A contra-barra (\) é necessária antes de cada aspas (“) interna devido às regras do shell.

Considerando programas executáveis, é possível escrever scripts em awk usando #!, que funciona em Linux e Unix derivados do Berkeley Unix, System V Release 4 ou 3. Este mecanismo não funciona com sistemas mais antigos. Por exemplo, o arquivo “exemplo” apresentado anteriormente pode ser alterado da seguinte maneira:

```
#! /bin/awk -f
```

```
BEGIN { print "Hello World!" }
```

Depois, basta transformá-lo em um executável usando o comando chmod e chamá-lo a partir do prompt.

```
chmod +x exemplo
```

Agora vamos mandar executar o nosso script

```
exemplo
Hello World
```

Isto é equivalente a usar o comando **awk -f** exemplo. A linha que começa com #! contém o caminho e o nome do interpretador que deve ser executado, e também comandos de inicialização opcionais que podem ser passados para o interpretador. O sistema operacional então executa o interpretador com os parâmetros fornecidos.

Comentários

Na linguagem awk, um comentário começa com o caracter #, e continua até o final da linha. O caracter # não precisa ser o primeiro caracter da linha. O arquivo exemplo pode ser alterado para inserir um comentário da seguinte maneira:

```
# Primeiro programa escrito na LP awk
# Este programa apenas exibe uma mensagem na tela
BEGIN { print "Hello World!" }
```

Exemplos

1) Exemplo simples

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

Este comando executa um programa awk simples que procura pela string “root” no arquivo de entrada ”/etc/passwd“. Quando uma linha que contém “root” é encontrada, ela é exibida na tela do computador, pois print $0, ou simplesmente print, significa imprimir a linha corrente. A / que aparece antes e depois de root indicam que root é o padrão a ser procurado. As aspas simples ao redor do programa notifica o shell para não interpretar os caracteres como caracteres especiais do shell. Este programa também poderia ser escrito em um arquivo, como por exemplo:

```
vim procura.awk
#! /bin/awk -f
BEGIN {
  print "Ocorrencias: "
}
```

```
#Padrão procurado
/root/ { print }
```

Agora vamos dar permissão para o nosso arquivo

```
chmod +x procura.awk
```

Neste caso a execução seria:

```
procura.awk /etc/passwd
```

Numa regra awk tanto o padrão como a ação podem ser omitidos, mas não ambos. Se o padrão é omitido, então a ação é executada para cada linha de entrada. Se a ação é omitida, a ação default é imprimir todas as linhas que correspondam ao padrão. Então, é possível eliminar a ação (o comando print) neste exemplo e o resultado seria o mesmo. Por exemplo:

```
awk '/root/' /etc/passwd
```

Porém, omitir o comando print e manter {}, faz com que uma ação “vazia” seja definida, isto é, nada é realizado. Por exemplo:

```
awk '/awk/ { }' texto.txt
```

2) Exemplo com duas regras O interpretador awk lê os arquivos de entrada linha a linha, e para cada uma ele testa o padrão especificado na(s) regra(s). Se vários padrões são encontrados, então várias ações são executadas na ordem em que aparecem no programa awk. Se nenhum padrão é encontrado, nenhuma ação é executada. Por exemplo, o seguinte programa awk contém duas regras:

```
awk '/0/ {print $1} /40/ {print $1}' /etc/passwd /etc/fstab
```

A primeira regra possui a string 0 como padrão e print $1 como ação. A segunda regra tem a string 40 como padrão e também print $1 como ação. As ações sempre devem estar entre { e }. O programa exibe cada linha que contém a string 0 ou 40. Se uma linha contém as duas strings, ela é exibida duas vezes, uma para cada regra.

Comandos X Linhas

Geralmente cada linha em um programa awk é um “separador” de comandos ou de regras, como por exemplo:

```
awk '/root/ { print }
>  /daemon/ { print }' /etc/passwd
```

Entretanto, awk irá ignorar novas linhas depois de um dos seguintes caracteres: , { ? : || && do else Uma nova linha em qualquer outro ponto é considerada o final de um comando.

Se o programador quiser “dividir” um único comando em duas linhas, basta colocar o caracter \ no

final da primeira linha. Este caracter no final da linha indica que o comando continua, e pode ser usado em qualquer parte de um comando, até no meio de uma string ou expressão. Por exemplo:

```
awk '/Esta expressao eh muito extensa, por isso continua\
                  na proxima linha/ { print } '
```

awk é uma linguagem “orientada a linha”. Cada ação de uma regra deve começar na mesma linha do padrão. Para ter o padrão e a ação em linhas separadas, deve-se usar \. Torna-se importante salientar que, além do uso deste caracter para “divisão” de um comando só ser permitido em arquivos de programas, ele não é aceito em todas as versões de awk.

Quando os comandos awk são “pequenos”, pode-se colocar mais do que um em uma única linha separados por ;. Por exemplo:

```
awk '/root/ { print };/daemon/ { print }' /etc/passwd
```

Exemplos de Programas de uma Linha

Muitos programas awk úteis possuem apenas uma ou duas linhas.

1) Programa que imprime o tamanho da maior linha.

```
awk '{ if (length($0) > max) max=length($0) }
 END { print max }' /etc/passwd
```

2) Programa que imprime todas as linhas que possuem mais do que 80 caracteres. A única regra possui uma expressão relacional e seu padrão, e não possui ação. Portanto, a ação default, imprimir o registro, é tomada.

```
awk 'length($0) > 80' /etc/inittab
```

3) Este programa imprime cada linha que possui pelo menos um campo. Esta é uma maneira simples de eliminar linhas em branco de um arquivo.

```
awk 'NF>0' /etc/fstab
```

4) Este programa imprime sete números randômicos de 0 a 100, inclusive.

```
awk 'BEGIN { for(i=1; i<=7; i++)
 print int(101*rand()) }'
```

5) Este programa imprime o número total de bytes usados pelos <arquivos>.

```
ls -lg /etc | awk '{x+= $4}; END {print "total bytes:" x}'
```

6) Este programa conta o número de linhas em um arquivo

```
awk 'END { print NR }' /etc/group
```

Expressões Regulares

Uma expressão regular é uma maneira de descrever um conjunto de strings. A expressão regular mais simples é uma seqüência de letras, números, ou ambos.

Uma expressão regular pode ser usada como um padrão quando colocada entre /. Neste caso ela é comparada com cada registro em todo o texto. Normalmente só é necessário encontrar uma parte do texto para ter sucesso. O seguinte exemplo imprime o segundo campo de cada registro que contém os três caracteres 'dev' em qualquer posição.

```
awk '/dev/ { print $2 }' /etc/fstab
```

Expressões regulares também podem ser usadas na correspondência de expressões. Estas expressões permitem especificar uma string a ser procurada que não precisa ser todo o registro de entrada. Os operadores ~ e !~ realizam comparações de expressões regulares. Expressões que usam estes operadores podem ser usadas como um padrão ou em comandos if, while, for e do. Por exemplo:

```
~ /<expressão_regular>/
```

Retorna verdadeiro se a expressão <exp> corresponder a <expressão_regular>.

Os próximos exemplos são usados para “selecionar” todos os registros de entrada que possuem a letra 'a' em algum lugar do primeiro campo.

```
awk '$1 ~ /a/' /etc/inittab
awk '{ if ($1 ~ /a/) print }' /etc/inittab
```

Quando uma expressão regular está entre /, ela é chamada de expressão regular constante (5.27 é uma constante numérica e “awk” é uma constante do tipo string).

Escape Sequences

Alguns caracteres não podem ser incluídos “literalmente” em expressões regulares constantes. Neste caso eles são representados com escape sequences, que são seqüências de caracteres que começam com \.

Isto é útil, por exemplo, para incluir ” numa constante do tipo string. Por exemplo:

```
awk 'BEGIN { print "Ele disse \"Oi!\" para ela." }'
```

O próprio caracter \ não pode ser incluído normalmente; deve-se colocar '\\' para que um '\' faça parte da string.

O caracter \ também é usado para representar caracteres que não podem ser exibidos, tais como tab

ou newline. Outras utilizações são apresentadas abaixo:

Caracter	Representa
\\	\
\/	/
\“	“
\a	Caracter de alerta (beep)
\t	tab horizontal
\v	tab vertical
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return

Operadores de Expressões Regulares

Expressões regulares podem ser combinadas com caracteres chamados de operadores de expressões regulares ou metacaracteres, apresentados a seguir.

- \ É usado para suprimir o significado especial de um caracter quando encontrado. Por exemplo: \$, corresponde ao caracter $
- **^** Corresponde ao início de uma string. Por exemplo: ^@chapter, corresponde a uma string que começa com '@chapter'.
- **$** É similar a ^, mas corresponde ao fim de uma string. Por exemplo: p$, corresponde a um registro que termina do 'p'.
- **.** O ponto corresponde a um único caracter, incluindo newline. Por exemplo: .P, corresponde a qualquer caracter seguido por P em uma string.
- **[..]** Esta é a chamada lista de caracteres. Corresponde aos caracteres que estão entre [e]. Por exemplo: [MVX], corresponde a qualquer um dos caracteres 'M', 'V ou 'X' em uma string. Um intervalo é indicado por um hífen (exemplo: [0-9]). Classe de caracter é uma notação especial para descrição de listas de caracteres que possuem um atributo específico. Uma classe de caracter consiste de: [: :]. Algumas classes de caracteres são:
- **[:alnum:]** caracteres alfanuméricos
- **[:alpha:]** caracteres alfabéticos
- **[:blank:]** espaço e tab
- **[:digit:]** caracteres numéricos
- **[:cntrl:]** caracteres de controle
- **[:print:]** caracteres que podem ser impressos(- caracteres de controle)
- **[:punct:]** caracteres de pontuação

Uma classe de equivalência é um nome para uma lista de caracteres que são equivalentes, que deve estar entre [= e =]. Por exemplo: e pode ser usado para representar e, é ou è. Neste caso, =e é uma expressão regular que corresponde a qualquer um destes caracteres (e,é,è).

- **[^..]** Corresponde a qualquer caracter exceto os listados. Por exemplo: [^0-9] corresponder a qualquer caracter que não é um dígito.
- | É um operador usado para especificar alternativas. Por exemplo: ^P|[0-9] corresponde a qualquer string que começa com P ou contém um dígito.
- **(..)** São usados para agrupar expressões regulares, assim como aritméticas.

- * Este símbolo significa que a expressão regular precedente pode ser repetida quantas vezes for necessário. Por exemplo: ph* corresponde a um p seguido de vários h.
- **+** É similar a *, a expressão precedente deve aparecer pelo menos uma vez. Isto significa que wh+y corresponde a why e whhy, mas não wy.
- **?** É similar a *, mas a expressão precedente deve aparecer uma ou nenhuma vez. Por exemplo: fe?d corresponde a fed e fd.
- **{n} {n,} {n,m}** Um ou dois números entre { e } denota uma expressão de intervalo. Por exemplo: wh{3}y corresponde a whhhy, mas não a why ou whhhhy; wh{3,5}y corresponde somente a whhhy, ou whhhhy ou whhhhhy; wh(2,}y corresponde a whhy ou whhhy e assim por diante.

Em expressões regulares os operadores *, + e ?, bem como { e }, possuem a maior precedência, seguidos pela concatenação e por |. Parênteses podem mudar como os operadores são agrupados.

Case-sensitivity

A correspondência de caracteres é case-sensitive. Isto significa que um w em uma expressão regular corresponde somente a w e não W. A maneira mais simples de solucionar este problema é usar uma lista de caracteres [Ww]. Entretanto isto pode tornar a leitura de expressões regulares mais difícil. Outra maneira é usar as funções tolower ou toupper. Por exemplo:

```
tolower($1) ~ /foo/ {...}
```

Converte o primeiro campo para minúsculas antes de fazer a comparação.

Outra maneira ainda é setar a variável **IGNORECASE** para um valor não zero, pois neste caso as operações com expressões regulares e string ignoram a diferença entre maiúsculas e minúsculas. Por exemplo:

```
x = "aB"
if (x ~ /ab/) ... # este teste irá falhar
IGNORECASE = 1
if (x ~ /ab/) ... # agora irá funcionar
```

Comandos de Controle em Ações

Comandos de controle tais como if e while, controlam o fluxo de execução em programas awk. A maioria deles são similares as comandos da linguagem C.

Comando if-else:

Sintaxe do comando

```
if (<condição>) { <comandos> }
if (<condição>) { <comandos> } else { <comandos> }
```

Exemplo do comando if-else:

```
if ( x % 2 == 0 )
   print "x eh par"
else
   print "x eh impar"
ou
if ( x % 2 == 0 ) print "x eh par"; else
  print "x eh impar"
```

Comando while:

Sintaxe do comando while

```
while (<condição>)
{
    <comandos>
}
```

Exemplo do comando while:

```
awk '{ i = 1
  while ( i <= 3 ) {
    print $i
    i++
   }
}' lista1.txt

**Comando do-while: **

Sintaxe do comando do-while
<sxh bash>
do {
  <comandos>
} while ( <condição> )
```

Exemplo do comando do-while:

```
awk '{ i = 1
  do {
    print $0
    i++
  } while ( i<= 10 )
}' lista1.txt
```

Comando for:

Sintaxe do comando for

```
for( <inicialização>; <condição>; <incremento> )
  <comandos>
```

Exemplo do comando for:

```
awk '{ for (i=1; i<=3; i++)
   print $i
     }' lista1.txt
```

Comando break: Força o término imediato do laço de onde é chamado (for, while ou do). Ele não tem significado quando usado fora do corpo de um laço.

Comando continue: Assim como o break, é usado somente dentro de um laço (for, while ou do). Ao invés de forçar o término do laço, este comando faz com que ocorra a próxima iteração do laço, pulando qualquer código intermediário.

Comando next: Força o awk a parar imediatamente de processar o registro corrente e passar para o próximo registro.

Comando nextfile: Similar ao comando next, ao invés de abandonar o processamento do registro corrente, o comando nextfile instrui o awk a parar de processar o arquivo corrente.

Comando exit: Faz com que o awk pare imediatamente de processar a regra corrente e pare de processar a entrada; qualquer entrada restante é ignorada.

Comandos print e printf

O print é um comando de saída simples (padrão). Já o printf é usado quando se deseja uma saída formatada.

O comando print exige simplesmente a especificação da lista de números, strings, variáveis, expressões awk e/ou campos do registro corrente (tal como $1), que devem ser exibidos, separados por vírgula. Eles são, então, exibidos separados por espaços em branco e seguidos por uma nova linha. Opcionalmente, a lista de números e strings pode ser colocada entre parênteses.

O print sem um item especificado, é equivalente a print $0, e imprime todo o registro corrente. Para imprimir uma linha em branco usa-se print ””, onde “” é a string vazia.

Exemplos do comando print:

```
awk 'BEGIN { print "line one \nline two \nline three" }'
awk '{ print $1 $2 }' lista1.txt
awk 'BEGIN { print "Campo1 Campo2"
  print "------ ------"
   }
{ print $1, " ", $2 }' lista1.txt
```

O default no comando print é separar os campos com um espaço em branco. Entretanto, qualquer seqüência de caracteres pode ser usada, basta inicializar a variável OFS (Output Field Separator). Da mesma maneira, é possível trocar a forma de separar os registros, cujo default é uma nova linha. Neste caso, deve-se inicializar a variável ORS (Output Record Separator). Por exemplo:

```
awk 'BEGIN { OFS = ";"; ORS = "\n\n" }
>          { print $1, $2 }' lista1.txt
Carla;3226.12.12

Ivan;3340.11.00

Maria;3223.78.21

Regis;3332.12.21
```

Para se ter um controle mais preciso da formatação, bem como para alinhar colunas de dados de uma maneira fácil, utiliza-se printf. Com printf é possível especificar a largura que será usada para cada item e várias maneiras de formatar números. Este comando é usado da seguinte maneira:

```
printf formato, item1, item2, ...
```

Onde o formato é muito semelhante ao ANSI C (ex.: %c, %s, %i, %4.3f). Exemplo da utilização do printf:

```
awk '{ printf "%-10s %s\n", $1, $2 }' lista1.txt
Carla 3226.12.12

Ivan 3340.11.00

Maria 3223.78.21

Regis 3332.12.21
```

Array em Awk

Um array é uma tabela de valores, chamados elementos. Os elementos de um array são distinguidos por índices. Índices em awk podem ser tanto números como strings, e cada array deve ter um nome (mesma sintaxe dos nomes de variáveis).

Arrays em awk são semelhantes às outras linguagens, mas existem algumas diferenças fundamentais. Em awk não é necessário especificar o tamanho de um array antes de começar a usá-lo. Adicionalmente, qualquer número ou string pode ser usado como indice do array, e não apenas números inteiros consecutivos.

Na verdade, Arrays em awk são associativos. Isto significa que cada array é uma coleção de pares: um índice, e o valor do elemento correspondente, como exemplificado abaixo:

```
Elemento 4        Valor 30
Elemento 2        Valor "awk"
Elemento 1        Valor 8
Elemento 3        Valor ""
```

Os pares acima estão fora de ordem porque a ordem é irrelevante.

Uma vantagem do array associativo é que novos pares podem ser adicionados em qualquer momento. Para ilustrar, no exemplo anterior é possível inserir o valor “numero 10” na décima posição:

```
Elemento 10     Valor "numero 10"
```

```
Elemento 4        Valor 30
Elemento 2        Valor "awk"
Elemento 1        Valor 8
Elemento 3        Valor ""
```

Desta maneira o array fica esparso, isto é, faltam alguns índices.

Outra conseqüência dos arrays associativos é que os índices não precisam ser números inteiros positivos. Por exemplo, logo abaixo é apresentado um array que traduz palavras do Inglês para o Francês:

```
Element "dog"      Valor "chien"
Element "cat"       Valor "chat"
Element "one"     Valor "un"
Element 1             Valor "un"
```

Um cuidado que deve ser tomado, é que letras maiúsculas são diferentes de minúsculas, e o valor de IGNORECASE não tem efeito. Deve-se usar a mesma string para armazenar e recuperar um elemento do array.

Quando awk cria um array, por exemplo com uma função, o array criado possuirá índices inteiros consecutivos que começam em um.

A principal maneira de usar um array é referenciar um dos seus elementos, da seguinte maneira:

```
nome_array[índice]
```

Se for usada uma referência para um elemento que não está armazenado, o valor resultante é “” (string nula).

É possível verificar se um elemento, num certo índice, existe em um array com a expressão:

```
índice in nome_array
```

Assim, pode-se testar quando um índice existe ou não. Se nome_array[índice] existe, é retornado o valor 1 (verdadeiro), caso contrário é retornado 0 (falso).

O exemplo abaixo ilustra a utilização de arrays.

```
vim array.awk
#! /bin/awk -f
# Comandos que serao executados antes de varrer o arquivo
BEGIN {
     print "Este programa ordena um arquivo cujas linhas sao numeradas!\n"
}
# Comandos que serao executados depois de varrer o arquivo
END {
     print "Arquivo Ordenado: \n"
     for (i=1; i<=max; i++)
       if (i in vetor) # para evitar linhas em branco
         print vetor[i]
```

```
     }
# Comandos que serao executados enquanto varre o arquivo
  {
   if ($1 > max)
           max = $1
         vetor[$1] = $0
  }
```

Para percorrer todos os elementos de um array cujos índices não são números inteiros em ordem crescente, awk possui um comando for “especial”:

```
for ( var in array )
   <comandos>
```

Este laço executa os comandos uma vez para índice do array que tenha sido previamente usado no programa, com a variável var “setada” para o índice. Exemplo:

```
#! /bin/awk -f
# Exemplo com array cujos índices não são números inteiros
# Armazena 1 para cada palavra que é usada pelo menos uma vez
{
# NF número de campos do registro de entrada corrente
for (i=1; i<=NF; i++)
   # armazena 1 para cada palavra que é usada pelo menos uma vez
  pal_usadas[$i] = 1
  }
# Comandos que serao executados depois de varrer o arquivo
  END {
   for (x in pal_usadas)
        if (length(x)>10) {
      ++num_pal_longas
      print x
         }
  print num_pal_longas, "palavras maiores que 10 caracteres!"
}
```

A ordem na qual os elementos do array são acessados por este comando é determinada pelo awk e não pode ser controlada ou alterada.

É possível remover um elemento individual de um array usando o comando delete:

```
delete nome_array[índice]
```

No exemplo abaixo todos os elementos do array são removidos:

```
for (i in vetor)
delete vetor[i]
```

Torna-se importante salientar que remover um elemento é diferente de atribuir um valor nulo. Por exemplo:

```
delete vetor[4]
if (4 in vetor)
   print "Esta mensagem nunca sera exibida."
   vetor[2] = ""
   if (2 in vetor)
   print "Esta mensagem sera exibida, apesar de nao ter conteudo."
```

Não é um erro remover um elemento que não existe, e todos os elementos podem ser removidos com um único comando:

```
delete vetor
```

Awk suporta arrays multidimensionais. Neste caso são usados mais do que um índice separados por vírgula. Também é possível testar quando um índice existe em um array multidimensional usando o mesmo operador in. Exemplo:

```
awk '{
if (max_nf < NF)
max_nf = NF
# NR é o número de registros de entrada processados
# desde o início da execução do programa
max_nr = NR
for (x=1; x<=NF; x++)
 vector[x, NR] = $x
}
END {
for (x=1; x<=max_nf; x++) {
for (y=max_nr; y>=1; --y)
  printf ("%s", vector[x, y])
  printf("\n")
   }
  }
}'
# Dada a entrada:
# 1 2 3 4 5 6
# 2 3 4 5 6 1
# 3 4 5 6 1 2
# 4 5 6 1 2 3
# O resultado é:
# 4 3 2 1
# 5 4 3 2
# 6 5 4 3
# 1 6 5 4
# 2 1 6 5
# 3 2 1 6
```

Funções Built-in

Funções Built-in são funções que estão sempre disponíveis para serem chamadas em programas awk. Para chamá-las, basta escrever o nome da função seguida pelos argumentos entre parênteses. Espaços em branco entre o nome da função e (são ignorados, mas recomenda-se que não seja deixado um espaço em branco). Algumas funções Built-in são apresentadas a seguir.

- **sqrt(x)**: retorna a raiz quadrada de x
- **int(x):** retorna o número inteiro mais próximo de x (trunca).
- **rand():** retorna um número randômico entre 0 e 1, mas não inclui 0 e 1. Pode-se obter números inteiros randômicos da seguinte maneira:

```
function randint(n) {
      return int(n* rand())
}
```

- **index(in, find):** procura na string “in” a primeira ocorrência da string “find” e retorna a posição onde foi encontrada dentro da string “in”. Exemplo:

```
awk 'BEGIN { print index("peanut", "an") }'
3
```

- **length():** retorna o número de caracteres da string.
- **split(string, array[, fieldsep]):** divide a string em partes separadas por fieldsep, e armazena as partes no array. O valor de fieldsep é uma expressão regular que descreve onde dividir a string. Se o valor de fieldsep é omitido, o valor de FS é usado. FS é o campo de entrada separador. Se o seu valor for a string nula (“”), então cada caracter torna-se um campo separador. O default é “ ”. split retorna o número de elementos criados. Exemplo:

```
split ("cul-de-sac", a, "-")
```

A função retorna 3 e o array terá o seguinte conteúdo:

```
a[1] = "cul"
a[2] = "de"
a[3] = "sac"
```

- **tolower(string):** retorna uma cópia da string com todos os caracteres minúsculos.
- **toupper(string):** retorna uma cópia da string com todos os caracteres maiúsculos.

AWK Para administradores Linux

A sintaxe dele é a seguinte:

```
awk 'padrão{ação}'
```

Vamos ao nosso primeiro exemplo:

Vamos mandar listar todas as linhas do arquivo /etc/passwd que contenham a string root

```
awk '/root/ { print }' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

Aqui foram nos mostradas todas as linhas com a string root mais eu quero somente a primeira e a terceira coluna, o awk por padrão usa o espaço para delimitar campos, com isso temos que passar um modificador para ele que é o -F, cada campo é identificado por uma variável $n onde n é o número do campo, o $0 identifica todos os campos.

```
awk -F ":" '/root/ { print $1 $3} ' /etc/passwd
root0
operator11
```

Note que mesmo nós deixamos separadas as variáveis $1 e $3 a saída apareceu ele juntos vamos ajustar isso agora inserindo entre o $1 e o $3 “ ”

```
awk -F ":" '/root/ { print $1 " " $3} ' /etc/passwd
root 0
operator 11
```

Agora ficou melhor a visualização, agora pense que precisamos identificar o que são esses campos, podemos fazer da seguinte forma

```
awk -F ":" '/root/ { print "Login:" $1 " ID:" $3} ' /etc/passwd
Login:root ID:0
Login:operator ID:11
```

Agora vamos fazer uma pesquisa condicional, vamos procurar as linhas que tenham o primeiro campo igual a root ou que contenham a string games

```
awk -F ":" '$1 == "root" || $1 ~ /games/' /etc/passwd
root:x:0:0:root:/root:/bin/bash
games:x:12:100:games:/usr/games:/sbin/nologin
```

Vamos ver uma lista dos operadores lógicos do awk:

Operador	**Nome**
==	Igual
!=	Diferente
<	Menor que
>	Maior que
<=	Menor ou igual
>=	Maior ou igual
~	Procura por uma expressão regular
!~	Que seja diferente da expressão regular
||	OU lógico
&&	E lógico

Operador	Nome
!	Negação lógica

Vamos ver algumas variáveis do awk

Variável	Descrição
NR	Contém o número do registro atual
NF	Contém o número de registros por campo
FS	Indica o separador de campos
RS	Indica o separador de registros
OFS	Indica o separador de campo do arquivo de saída
ORS	Indica o separador de registros do arquivo de saída

Vamos testar agora o separador de saída que podemos forçar com o OFS da seguinte forma

```
awk -F ":" '{OFS=" - "}{print $1,$6}' /etc/passwd
root - /root
bin - /bin
daemon - /sbin
adm - /var/adm
lp - /var/spool/lpd
sync - /sbin
shutdown - /sbin
halt - /sbin
mail - /var/spool/mail
uucp - /var/spool/uucp
operator - /root
games - /usr/games
gopher - /var/gopher
[...]
```

Agora vamos listar o nome dos grupos que tiverem como membro o usuário daemon

```
awk -F ":" '$4 ~  /daemon/ {print $1}' /etc/group
bin
daemon
adm
lp
haldaemon
```

Agora vamos mandar imprimir uma linha especifica por exemplo a linha 6 do arquivo /etc/fstab

```
awk 'NR==6' /etc/fstab
# Accessible filesystems, by reference, are maintained under '/dev/disk'
```

Agora vamos utilizar o awk para mandar imprimir o tamanho da maior linha de um arquivo, vamos utilizar o /etc/inittab como exemplo

```
awk '{ if (length($0) > max) max=length($0) } END { print max }'
/etc/inittab
```

```
78
```

Agora vamos mandar imprimir os usuários que tem o GID maior que 499

```
awk -F ":" '{if ($4 > 499) print $1}' /etc/passwd
nfsnobody
quintiliano
_tor
```

Agora vamos pegar o endereço ip da interface eth0

```
ifconfig eth0 | awk '/inet end/ {print $3}'
10.101.0.1
```

Agora vamos mandar inverter o endereço ip

```
ifconfig eth0 | awk '/inet end/ {print $3}' | awk -F "." '{ print $4 "." $3
"." $2 "." $1}'
1.0.101.10
```

Agora vamos pegar todos os endereços ips da máquina

```
ifconfig | awk '/inet end/ {print $3}'
10.101.0.1
127.0.0.1
```

Agora vamos pegar o nome e o endereço MAC de todas as interfaces que temos na máquina

```
ifconfig | awk '/eth[0-9]/ { print $1 " " $7} '
eth0 24:BA:E9:BC:5E:12
eth1 D8:4A:35:D7:4A:26
```

Agora vamos mandar imprimir as linhas do arquivo /etc/passwd e vamos numera-las

```
awk -F ":" '{ print NR " " $0 }' /etc/passwd
1 root:x:0:0:root:/root:/bin/bash
2 bin:x:1:1:bin:/bin:/sbin/nologin
3 daemon:x:2:2:daemon:/sbin:/sbin/nologin
4 adm:x:3:4:adm:/var/adm:/sbin/nologin
5 lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
6 sync:x:5:0:sync:/sbin:/bin/sync
7 shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
8 halt:x:7:0:halt:/sbin:/sbin/halt
9 mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
10 uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
11 operator:x:11:0:operator:/root:/sbin/nologin
12 games:x:12:100:games:/usr/games:/sbin/nologin
13 gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
[...]
```

Para contarmos a quantidade de linhas de um arquivo por exemplo /etc/passwd podemos fazer da seguinte forma.

```
awk 'END { print NR }' /etc/passwd
38
```

Agora pense que precisamos de uma determinada coluna em uma determinada linha podemos fazer isso da seguinte forma

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

Neste exemplo acima estamos obtendo o uid do usuário root que é o zero

Vamos ver agora como lemos somente uma determinada linha de um arquivo, vamos ler a linha 10 do /etc/passwd

```
awk 'NR == 30' /etc/passwd
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
```

Vamos localizar os processos no sistema que não são nem do usuário root e nem do nobody e que estão consumindo mais de 2% da memória

```
ps aux | awk '!/root|nobody/ { if ($4>2) { print $2,$11}}'
2672 /opt/google/chrome/chrome
3461 /opt/google/chrome/chrome
3481 /opt/google/chrome/chrome
5720 /opt/google/chrome/chrome
15664 /usr/lib/virtualbox/VirtualBox
18143 /usr/lib/virtualbox/VirtualBox
18272 /usr/lib/virtualbox/VirtualBox
```

Aqui o !/root|nobody/ é uma procurando por processos não sendo do usuário root ou do nobody depois temos $4>2 aonde o quarto campo é o % de memória depois disso mandamos imprimir o campo 2 que é o PID e o campo número 11 que é o aplicativo que o usuário está utilizando.

Agora vamos ver como podemos utilizar o awk para visualizarmos os logs do squid por exemplo. Vamos mostrar o endereço ip do usuário e o site que ele esta acessando

```
tail -f /var/log/squid3/access.log | awk '{print "IP: " $3 " Site: " $7}'
IP: 10.0.1.128 Site:
http://khm0.google.com.br/kh/v=123&src=app&x=390&y=572&z=10&s=Galile
IP: 10.0.1.128 Site:
http://mt0.google.com/vt/lyrs=h@203000000&hl=pt-BR&src=app&x=386&y=572&z=10&
s=Ga
IP: 10.0.1.128 Site:
http://mt0.google.com/vt/lyrs=h@203000000&hl=pt-BR&src=app&x=390&y=572&z=10&
s=Galile
IP: 10.0.1.128 Site:
http://khm0.google.com.br/kh/v=123&src=app&x=386&y=572&z=10&s=Ga
IP: 10.0.1.128 Site:
```

```
http://mt1.google.com/vt/lyrs=h@203000000&hl=pt-BR&src=app&x=389&y=572&z=10&
s=Gal
IP: 10.0.1.128 Site:
http://mt1.google.com/vt/lyrs=h@203000000&hl=pt-BR&src=app&x=387&y=572&z=10&
s=Galil
IP: 10.0.1.128 Site:
http://khm0.google.com.br/kh/v=123&src=app&x=388&y=572&z=10&s=
IP: 10.0.1.128 Site:
http://khm0.google.com.br/kh/v=123&src=app&x=386&y=572&z=10&s=Ga
IP: 10.0.1.128 Site:
http://mt0.google.com/vt/lyrs=h@203000000&hl=pt-BR&src=app&x=386&y=572&z=10&
s=Ga
IP: 10.0.1.128 Site:
http://mt0.google.com/vt/lyrs=h@203000000&hl=pt-BR&src=app&x=390&y=572&z=10&
s=Galile
IP: 10.0.1.128 Site:
http://khm0.google.com.br/kh/v=123&src=app&x=390&y=572&z=10&s=Galile
[...]
```

Vamos ver uma função para converter a data do squid que é um formato parecido com esse 1356713930.914 para o formato que utilizamos

```
echo "1356713930.914" | awk '{print strftime("%F %H:%M:%S",$1)}'
2012-12-28 14:58:50
```

Aqui ao invés de utilizar %Y%m%d utilizamos o %F que é equivalente, porém que eu precisar formatar a data com dd/mm/yyyy precisamos fazer da seguinte forma

```
echo "1356713930.914" | awk '{print strftime("%Y/%m/%d %H:%M:%S",$1)}'
2012/12/28 14:58:50
```

Agora se eu precisamos visualizar a data o ip do cliente e o site podemos fazer da seguinte forma

```
tail -f /var/log/squid3/access.log | awk '{print strftime("%F %H:%M:%S",$1)
" IP:"$3 " Site:" $7}'
2012-12-28 15:06:48 IP:10.0.0.215 Site:mail.google.com:443
2012-12-28 15:06:49 IP:10.0.0.213 Site:http://www.google.com.br/imghover?
2012-12-28 15:06:49 IP:10.0.0.213 Site:http://t1.gstatic.com/images?
2012-12-28 15:06:49 IP:10.0.0.213 Site:http://t0.gstatic.com/images?
2012-12-28 15:06:49 IP:10.0.0.213 Site:http://t1.gstatic.com/images?
2012-12-28 15:06:49 IP:10.0.0.213 Site:http://t3.gstatic.com/images?
2012-12-28 15:06:49 IP:10.0.0.213 Site:http://t0.gstatic.com/images?
2012-12-28 15:06:49 IP:10.0.0.213 Site:http://t0.gstatic.com/images?
2012-12-28 15:06:49 IP:10.0.0.213 Site:http://t3.gstatic.com/images?
2012-12-28 15:06:49 IP:10.0.0.213 Site:http://t0.gstatic.com/images?
[...]
```

Em alguns casos precisamos converter um determinado texto de maiúsculo para minúscula ou vice-versa.

Vamos converter um texto para maiúsculo

```
awk '{ print toupper($0)}' /etc/passwd
ROOT:X:0:0:ROOT:/ROOT:/BIN/BASH
DAEMON:X:1:1:DAEMON:/USR/SBIN:/BIN/SH
BIN:X:2:2:BIN:/BIN:/BIN/SH
SYS:X:3:3:SYS:/DEV:/BIN/SH
SYNC:X:4:65534:SYNC:/BIN:/BIN/SYNC
GAMES:X:5:60:GAMES:/USR/GAMES:/BIN/SH
MAN:X:6:12:MAN:/VAR/CACHE/MAN:/BIN/SH
LP:X:7:7:LP:/VAR/SPOOL/LPD:/BIN/SH
MAIL:X:8:8:MAIL:/VAR/MAIL:/BIN/SH
[...]
```

Agora pense que queremos converter um texto de maiúsculo para minúsculo

```
awk '{ print tolower($0)}' /tmp/passwdupper
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
```

Referências

1. http://cesarakg.freeshell.org/awk-1.html
2. http://cesarakg.freeshell.org/awk-2.html
3. http://cesarakg.freeshell.org/awk-3.html
4. http://www.inf.pucrs.br/~manssour/AWK/index.html