Бесплатная электронная книга

учусь awk

Free unaffiliated eBook created from
Stack Overflow contributors.

#awk

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: awk

It is an unofficial and free awk ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official awk.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с awk

замечания

Название AWK происходит от последних инициалов его создателей Альфреда В. Ахо, Питера Дж. Вайнбергера и
Брайан У. Керниган.

Ресурсы

- Персональная страница Illumos AWK
- Персональная страница Plan9 AWK
- Руководство пользователя GNU AWK
- Язык программирования AWK

Версии

название	Первоначальная версия	Версия	Дата выхода
POSIX awk	1992	IEEE Std 1003.1, выпуск 2013	2013-04-19
Один True Awk или nawk или BWK awk	198x	-	2012-12-20
GNU awk или gawk	1986	4.1.3	2015-05-19

Examples

Привет, мир

Пример Hello world - это просто:

```
awk 'BEGIN {print "Hello world"}'
```

Самая основная программа `awk` состоит из истинного значения (обычно `1`) и делает `awk` echo его входным:

```
$ date | awk '1'
Mon Jul 25 11:12:05 CEST 2016
```

Поскольку «hello world» также является истинным значением, вы также можете сказать:

```
$ date | awk '"hello world"'
Mon Jul 25 11:12:05 CEST 2016
```

Однако ваше намерение становится намного яснее, если вы пишете

```
$ date | awk '{print}'
Mon Jul 25 11:12:05 CEST 2016
```

вместо.

Как запустить AWK-программы

Если программа короткая, вы можете включить ее в команду, которая запускает awk:

```
awk -F: '{print $1, $2}' /etc/passwd
```

В этом примере с помощью командной строки `-F:` мы советуем awk использовать: как разделитель полей ввода. Это то же самое, что и

```
awk 'BEGIN{FS=":"}{print $1,$2}' file
```

Альтернативно, мы можем сохранить весь awk-код в awk-файле и вызвать эту awk-программу следующим образом:

```
awk -f 'program.awk' input-file1 input-file2 ...
```

program.awk может быть любой многострочной программой, то есть:

```
# file print_fields.awk
BEGIN {print "this is a header"; FS=":"}
{print $1, $2}
END {print "that was it"}
```

И затем запустите его с помощью:

```
awk -f print_fields.awk /etc/passwd   #-f advises awk which program file to load
```

Или в целом:

```
awk -f program-file input-file1 input-file2 ...
```

Преимущество наличия программы в отдельном файле заключается в том, что вы можете написать программу с правильной идентификацией, чтобы иметь смысл, вы можете включать комментарии с # и т. Д.

AWK по примерам

AWK - это язык строковых манипуляций, используемый в основном в системах UNIX. Идея AWK заключалась в том, чтобы создать универсальный язык для использования при работе над файлами, который не был слишком сложным для понимания.

AWK имеет некоторые другие варианты, но основная концепция - то же самое, только с дополнительными функциями. Этими другими вариантами являются NAWK и GAWK. GAWK содержит все возможности обоих, в то время как NAWK на один шаг выше AWK, если хотите.

Самый простой способ думать о AWK - это рассмотреть, что он имеет две основные части. Шаблон и действие.

Вероятно, самый простой пример AWK: (См. Также: Hello World)

```
BEGIN {print "START"}
      {print        }
END   {print "STOP" }
```

Здесь ключевыми словами `BEGIN` и `END` являются шаблон, в то время как действие находится внутри {}. Этот пример был бы бесполезным, но для внесения изменений в полезную функцию потребовались бы незначительные изменения.

```
BEGIN {print "File\tAuthor"}
      {print $8, "\t", $3}
END {print " - DONE - "}
```

Здесь `\t` представляет символ Tab и используется для выравнивания границ выходной строки. $ 8 и $ 3 похожи на использование, которое видно в `Shell Scripts` , но вместо использования 3-го и 8-го аргументов он использует 3-й и 8-й столбцы строки ввода.

Таким образом, этот пример будет печатать: File Author в верхней строке, в то время как вторая строка связана с файловыми путями. $ 8 - это имя файла, 3 доллара - владелец (при просмотре пути к каталогу это будет более понятным). Наконец, нижняя строка будет печататься, как и следовало ожидать - DONE -

Кредит для вышеуказанного примера приведен на http://www.grymoire.com/Unix/Awk.html

Справочный файл

`coins.txt` от Грега Гебеля:

```
gold     1    1986  USA                American Eagle
gold     1    1908  Austria-Hungary    Franz Josef 100 Korona
silver  10    1981  USA                ingot
gold     1    1984  Switzerland        ingot
```

```
gold     1    1979  RSA               Krugerrand
gold     0.5  1981  RSA               Krugerrand
gold     0.1  1986  PRC               Panda
silver   1    1986  USA               Liberty dollar
gold     0.25 1986  USA               Liberty 5-dollar piece
silver   0.5  1986  USA               Liberty 50-cent piece
silver   1    1987  USA               Constitution dollar
gold     0.25 1987  USA               Constitution 5-dollar piece
gold     1    1988  Canada            Maple Leaf
```

Минимальная теория

Общий awk однострочный:

```
awk <awk program> <file>
```

или же:

```
<shell-command> | awk <awk program>
```

`<shell-command>` и `<file>` адресуются как *вход awk* .

`<awk program>` - это код, следующий за этим шаблоном (одиночный, не двойной, кавычки):

```
'BEGIN   {<init actions>};
 <cond1> {<program actions>};
 <cond2> {<program actions>};
 ...
 END  {<final actions>}'
```

где:

- `<condX>` условие чаще всего является регулярным выражением `/re/` , которое должно совпадать с строками ввода awk;
- `<* actions>` - это последовательности *операторов* , аналогичные командам оболочки, снабженные C-подобными конструкциями.

`` обрабатывается в соответствии со следующими правилами:

1. `BEGIN ...` и `END ...` являются необязательными и выполняются до или после обработки строк ввода awk.
2. Для каждой строки в входе awk, если условие `<condN>` является мясом, выполняется соответствующий блок `<program actions>` .
3. `{<program actions>}` умолчанию `{print $0}` .

Условия могут быть объединены со стандартными логическими операторами:

```
    /gold/ || /USA/ && !/1986/
```

где `&&` имеет приоритет над `||` ;

Оператор `print` . `print item1 item2` печатает элементы в STDOUT.
Элементы могут быть переменными (`X` , `$0`), строками («привет») или числами.
`item1, item2` сопоставляются со значением переменной `OFS` ;
`item1 item2` *justapoxed* ! Используйте `item1 " " item2` для пробелов или printf для получения дополнительных функций.

Переменные не нужны `$` , т.е.: `print myVar;`
В awk встроены следующие специальные переменные:

- `FS` : действует как разделитель полей для разделения строк ввода awk в полях. Я могу быть единственным персонажем, `FS="c"` ; пустая строка, `FS=""` (тогда каждый отдельный символ становится отдельным полем); регулярное выражение без косых черт, `FS="re"` ; `FS=" "` означает пробелы пробелов и вкладок и значение по умолчанию.
- `NF` : количество полей для чтения;
- `$1` , `$2` , ...: 1-ое поле, 2-е поле. и т. д. текущей входной линии,
- `$0` : текущая строка ввода;
- `NR` : текущий номер строки.
- `OFS` : строка для сортировки полей при печати.
- `ORS` : разделитель выходной записи, по умолчанию - новая строка.
- `RS` : разделитель строки ввода (записи). По умолчанию используется новая строка. Установить как `FS` .
- `IGNORECASE` : влияет на FS и RS при регулярном выражении;

Примеры

Фильтруйте линии по `gold` regexp и посчитайте их:

```
# awk 'BEGIN {print "Coins"} /gold/{i++; print $0}  END {print i " lines out of " NR}'
coins.txt
Coins
gold     1    1986  USA                American Eagle
gold     1    1908  Austria-Hungary    Franz Josef 100 Korona
gold     1    1984  Switzerland        ingot
gold     1    1979  RSA                Krugerrand
gold     0.5  1981  RSA                Krugerrand
gold     0.1  1986  PRC                Panda
gold     0.25 1986  USA                Liberty 5-dollar piece
gold     0.25 1987  USA                Constitution 5-dollar piece
gold     1    1988  Canada             Maple Leaf
9 lines out of 13
```

По умолчанию `print $0` действие и условие на основе внутренней переменной awk `NR` :

```
# awk 'BEGIN {print "First 3 coins"} NR<4' coins.txt
First 3 coins
gold     1    1986  USA                American Eagle
gold     1    1908  Austria-Hungary    Franz Josef 100 Korona
silver  10    1981  USA                ingot
```

Форматирование с помощью C-style `printf`:

```
# awk '{printf ("%s \t %3.2f\n", $1, $2)}' coins.txt
gold     1.00
gold     1.00
silver   10.00
gold     1.00
gold     1.00
gold     0.50
gold     0.10
silver   1.00
gold     0.25
silver   0.50
silver   1.00
gold     0.25
gold     1.00
```

Примеры условий

```
awk 'NR % 6'            # prints all lines except those divisible by 6
awk 'NR > 5'            # prints from line 6 onwards (like tail -n +6, or sed '1,5d')
awk '$2 == "foo"'       # prints lines where the second field is "foo"
awk '$2 ~ /re/'         # prints lines where the 2nd field mateches the regex /re/
awk 'NF >= 6'           # prints lines with 6 or more fields
awk '/foo/ && !/bar/'   # prints lines that match /foo/ but not /bar/
awk '/foo/ || /bar/'    # prints lines that match /foo/ or /bar/ (like grep -e 'foo' -e 'bar')
awk '/foo/,/bar/'       # prints from line matching /foo/ to line matching /bar/, inclusive
awk 'NF'                # prints only nonempty lines (or: removes empty lines, where NF==0)
awk 'NF--'              # removes last field and prints the line
```

Добавив действие `{...}` можно напечатать определенное поле, а не всю строку, например:

```
awk '$2 ~ /re/{print $3 " " $4}'
```

печатает третье и четвертое поле линий, где второе поле выполняет регулярное выражение / re /.

Некоторые строковые функции

Функция `substr()` :

```
# awk '{print substr($3,3) " " substr($4,1,3)}'
86 USA
08 Aus
81 USA
84 Swi
```

```
79 RSA
81 RSA
86 PRC
86 USA
86 USA
86 USA
87 USA
87 USA
88 Can
```

`match(s, r [, arr])` возвращает позицию в `s` где выполняется регулярное выражение `r` и устанавливает значения `RSTART` и `RLENGTH` . Если аргумент `arr` предоставляется, он возвращает массив `arr` где элементы устанавливаются в согласованное подвыражение в скобках. Сопоставления 0-го элемента `arr` устанавливаются во все регулярное выражение. Также выражения `arr[n, "start"]` и `arr[n, "length"]` предоставляют начальную позицию и длину каждой подстроки.

Более строковые функции:

```
sub(/regexp/, "newstring"[, target])
gsub(/regexp/, "newstring"[, target])
toupper("string")
tolower("string")
```

Заявления

Простым утверждением часто бывает одно из следующего:

```
variable = expression
print [ expression-list ]
printf format [ , expression-list ]
next # skip remaining patterns on this input line
exit # skip the rest of the input
```

Если `stat1` и `stat2` являются `stat2` , следующие утверждения также:

```
{stat}

{stat1;  stat2}

{stat1
stat2}

if ( conditional ) statement [ else statement ]
```

Следующими стандартными конструкциями типа C являются:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break    # usual C meaning
```

```
continue # usual C meaning
```

Цикл C-стиля для печати элемента описания переменной длины, начиная с поля 4:

```
# awk '{out=""; for(i=4;i<=NF;i++){out=out" "$i}; print out}' coins.txt
USA American Eagle
Austria-Hungary Franz Josef 100 Korona
USA ingot
Switzerland ingot
RSA Krugerrand
RSA Krugerrand
PRC Panda
USA Liberty dollar
USA Liberty 5-dollar piece
USA Liberty 50-cent piece
USA Constitution dollar
USA Constitution 5-dollar piece
Canada Maple Leaf
```

Обратите внимание, что `i` инициализируется равным 0.

Если условия и вычисления применяются к нумеру:

```
# awk '/gold/ {if($3<1980) print $0 "$" 425*$2}' coins.txt
gold     1    1908  Austria-Hungary     Franz Josef 100 Korona      $425
gold     1    1979  RSA                 Krugerrand                  $425
```

AWK исполняемый скрипт

```
#!/usr/bin/gawk -f
# This is a comment
(pattern) {action}
...
```

Передача переменных оболочки

```
# var="hello"
# awk -v x="$var" 'BEGIN {print x}'
hello
```

Прочитайте Начало работы с awk онлайн: https://riptutorial.com/ru/awk/topic/937/начало-работы-с-awk

глава 2: Встроенные переменные

Examples

FS - полевой разделитель

Используется awk для разделения каждой записи на несколько полей:

```
echo "a-b-c
d-e-f" | awk 'BEGIN {FS="-"} {print $2}'
```

приведет к:

```
b
e
```

Переменная `FS` также может быть установлена с использованием опции `-F` :

```
echo "a-b-c
d-e-f" | awk -F '-' '{print $2}'
```

По умолчанию поля разделяются пробелами (пробелы и табуляции), а несколько пробелов и вкладок подсчитываются как один разделитель.

RS - записывающий разделитель

Используется awk для разделения ввода на несколько записей. Например:

```
echo "a b c|d e f" | awk 'BEGIN {RS="|"} {print $0}'
```

производит:

```
a b c
d e f
```

По умолчанию разделитель записи является символом новой строки.

Аналогично: echo "abc | de f" | awk 'BEGIN {RS = "|"} {print $ 2}'

производит:

```
b
e
```

OFS - выходной полевой разделитель

Используется awk для разделения полей, выводимых оператором `print` . Например:

```
echo "a b c
d e f" | awk 'BEGIN {OFS="-"} {print $2, $3}'
```

производит:

```
b-c
e-f
```

Значение по умолчанию , строка, состоящая из одного пробела.

ORS - выходной разделитель записи

Используется awk для разделения записей и выводится в конце каждого оператора `print` . Например:

```
echo "a b c
d e f" | awk 'BEGIN {ORS="|"} {print $2, $3}'
```

производит:

```
b c|e f
```

Значение по умолчанию - `\n` (символ новой строки).

ARGV, ARGC - массив аргументов командной строки

Аргументы командной строки, переданные awk, хранятся во внутреннем массиве `ARGV` элементов `ARGC` . Первым элементом массива является имя программы. Например:

```
awk 'BEGIN {
   for (i = 0; i < ARGC; ++i) {
      printf "ARGV[%d]=\"%s\"\n", i, ARGV[i]
   }
}' arg1 arg2 arg3
```

производит:

```
ARGV[0]="awk"
ARGV[1]="arg1"
ARGV[2]="arg2"
ARGV[3]="arg3"
```

FS - полевой разделитель

Переменная `FS` используется для установки *разделителя полей ввода* . В `awk` пространство

и вкладка действуют как разделители полей по умолчанию. Соответствующее значение поля можно получить через `$1` , `$2` , `$3` ... и так далее.

```
awk -F'=' '{print $1}' file
```

- `-F` - опция командной строки для установки разделителя полей ввода.

```
awk 'BEGIN { FS="=" } { print $1 }' file
```

OFS - выходной полевой разделитель

Эта переменная используется для установки *разделителя выходных полей,* который по умолчанию является пространством.

```
awk -F'=' 'BEGIN { OFS=":" } { print $1 }' file
```

Пример:

```
$ cat file.csv
col1,col2,col3,col4
col1,col2,col3
col1,col2
col1
col1,col2,col3,col4,col5

$ awk -F',' 'BEGIN { OFS="|" } { $1=$1 } 1' file.csv
col1|col2|col3|col4
col1|col2|col3
col1|col2
col1
col1|col2|col3|col4|col5
```

Присвоение `$1` к `$1` в `$1=$1` изменяет поле (в этом случае `$1`), и это приводит к `awk` восстанавливающему запись `$0` . Восстановление записи заменяет разделители `FS` на `OFS` .

RS - входной разделитель записи

Эта переменная используется для установки разделителя входных записей, по умолчанию - новой строки.

```
awk 'BEGIN{RS=","} {print $0}' file
```

ORS - выходной разделитель записи

Эта переменная используется для установки разделителя выходной записи, по умолчанию - новой строки.

```
awk 'BEGIN{ORS=","} {print $0}' file
```

NF - количество полей

Эта переменная даст вам общее количество полей в текущей входной записи.

```
awk -F',' '{print NF}' file.csv
```

Пример:

```
$ cat file.csv
col1,col2,col3,col4
col1,col2,col3
col1,col2
col1
col1,col2,col3,col4,col5

$ awk -F',' '{print NF}' file.csv
4
3
2
1
5
```

NR - общее количество записей

Будет предоставлено общее количество записей, обработанных в текущем экземпляре `awk` .

```
cat > file1
suicidesquad
harley quinn
joker
deadshot

cat > file2
avengers
ironman
captainamerica
hulk

awk '{print NR}' file1 file2
1
2
3
4
5
6
7
8
```

В этом экземпляре было обработано всего 8 записей.

FNR - количество записей в файле

Предоставляет общее количество записей, обрабатываемых экземпляром `awk` относительно

файлов `awk` , обрабатывает

```
cat > file1
suicidesquad
harley quinn
joker
deadshot

cat > file2
avengers
ironman
captainamerica
hulk

awk '{print FNR}' file1 file2
1
2
3
4
1
2
3
4
```

Каждый файл имел по 4 строки каждый, поэтому всякий раз, когда `awk` сталкивался, `EOF FNR` был сброшен на 0.

NF - количество полей

Предоставляет количество столбцов или полей в каждой записи (запись соответствует каждой строке). Каждая строка демаркируется `RS` которая по умолчанию соответствует новой строке.

```
cat > file1
Harley Quinn Loves Joker
Batman Loves Wonder Woman
Superman is not dead
Why is everything I type four fielded!?

awk '{print NF}' file1
4
4
4
7
```

`FS` (где-то там) по умолчанию используется вкладка или пробел. Так что Harley, Quinn, Loves, Joker считаются столбцами. Случай имеет место для следующих двух строк, но последняя строка содержит 7 пробелов, что означает 7 столбцов.

FNR - обрабатываемый текущий номер записи

FNR содержит номер обрабатываемой строки входного файла. В этом примере вы увидите awk, начиная с 1, когда начнете обрабатывать второй файл.

Пример с одним файлом

```
$ cat file1
AAAA
BBBB
CCCC
$ awk '{ print FNR }' file1
1
2
3
```

Пример с двумя файлами

```
$ cat file1
AAAA
BBBB
CCCC
$ cat file2
WWWW
XXXX
YYYY
ZZZZ
$ awk '{ print FNR, FILENAME, $0 }' file1 file2
1 file1 AAAA
2 file1 BBBB
3 file1 CCCC
1 file2 WWWW
2 file2 XXXX
3 file2 YYYY
4 file2 ZZZZ
```

Расширенный пример с двумя файлами

`FNR` может использоваться для определения того, обрабатывает ли awk первый файл, поскольку `NR==FNR` является истинным только для первого файла. Например, если мы хотим объединить записи из файлов `file1` и `file2` в их `FNR` :

```
$ awk 'NR==FNR { a[FNR]=$0; next } (FNR in a) { print FNR, a[FNR], $1 }' file1 file2
1 AAAA WWWW
2 BBBB XXXX
3 CCCC YYYY
```

Запись `ZZZZ` из `file2` отсутствует, поскольку `FNR` имеет другое максимальное значение для `file1` и `file2` и нет соединения для разных `FNR` .

Прочитайте Встроенные переменные онлайн: https://riptutorial.com/ru/awk/topic/3227/встроенные-переменные

глава 3: Встроенные функции

Examples

Длина ([строка])

Возвращает количество символов данной *строки*

Соображения

- Если вместо строки указывается String, результатом будет длина строки, представляющей заданный номер. Т.е. если мы выполним `length(12345)` результат будет таким же, как `length("12345")` , то есть *5*
- Если значение не задано, результатом будет длина обрабатываемой фактической строки, то есть `length($0)`
- Он может использоваться внутри шаблона или внутри кодовых блоков.

Примеры

Вот несколько примеров, демонстрирующих, как работает `length()`

```
$ cat file
AAAAA
BBBB
CCCC
DDDD
EEEE
```

Внутри шаблона

Фильтрация всех строк длиной более 4 символов

```
$ awk ' length($0) > 4 ' file
AAAAA
```

Внутри кодового блока

Распечатает размер текущей строки

```
$ awk '{ print length($0) }' file
5
4
```

```
4
4
4
```

Без данных

Распечатает размер текущей строки

```
$ awk '{ print length }' file
5
4
4
4
4
```

Распечатает размер текущей строки

```
$ awk '{ print length() }' file
5
4
4
4
4
```

Число, заданное вместо строки

Будет напечатан размер строки, представляющей номер

```
$ awk '{ print length(12345) }' file
5
5
5
5
5
```

Исправлена строчка

Будет напечатан размер строки

```
$ awk '{ print length("12345") }' file
5
5
5
5
5
```

Прочитайте Встроенные функции онлайн: https://riptutorial.com/ru/awk/topic/4922/встроенные-функции

глава 4: Манипуляция строк

Examples

Извлечение определенных строк из текстового файла

Предположим, что у нас есть файл

```
cat -n lorem_ipsum.txt
 1    Lorem Ipsum is simply dummy text of the printing and typesetting industry.
 2    Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an
unknown printer took a galley of type and scrambled it to make a type specimen book.
 3    It has survived not only five centuries, but also the leap into electronic typesetting,
remaining essentially unchanged.
 4    It was popularised in the 1960s with the release of Letraset sheets containing Lorem
Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker
including versions of Lorem Ipsum
```

Мы хотим извлечь строки 2 и 3 из этого файла

```
awk 'NR==2,NR==3' lorem_ipsum.txt
```

Это напечатает строки 2 и 3:

```
 2    Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an
unknown printer took a galley of type and scrambled it to make a type specimen book.
 3    It has survived not only five centuries, but also the leap into electronic typesetting,
remaining essentially unchanged.
```

Извлечение определенного столбца / поля из определенной строки

Если у вас есть следующий файл данных

```
cat data.csv
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

возможно, вам нужно прочитать четвертый столбец третьей строки, это будет «24»,

```
awk 'NR==3 { print $4 }' data.csv
```

дает

    24

Модификация строк «на лету» (например, для исправления строк Windows)

Если файл может содержать окончание строк в Windows или Unix (или даже смесь обоих), то предполагаемая замена текста может не работать должным образом.

Образец:

```
$ echo -e 'Entry 1\nEntry 2.1\tEntry 2.2\r\nEntry 3\r\n\r\n' \
> | awk -F'\t' '$1 != "" { print $1 }' \
> | hexdump -c
0000000   E   n   t   r   y       1  \n   E   n   t   r   y       2   .
0000010   1  \n   E   n   t   r   y       3  \r  \n  \r  \n
000001d
```

Это можно легко устранить с помощью дополнительного правила, которое вставлено в начале awk-скрипта:

```
/\r$/ { $0 = substr($0, 1, length($0) - 1) }
```

Поскольку действие не заканчивается `next` , следующие правила применяются, как и прежде.

Пример (с исправлением строк):

```
$ echo -e 'Entry 1\nEntry 2.1\tEntry 2.2\r\nEntry 3\r\n\r\n' \
> | awk -F'\t' '/\r$/ { $0 = substr($0, 1, length($0) - 1) } $1 != "" { print $1 }' \
> | hexdump -c
0000000   E   n   t   r   y       1  \n   E   n   t   r   y       2   .
0000010   1  \n   E   n   t   r   y       3  \n
000001a
```

Прочитайте Манипуляция строк онлайн: https://riptutorial.com/ru/awk/topic/3947/манипуляция-строк

глава 5: Массивы

Examples

Основы массива

Создание нового массива несколько запутанно, так как нет реального идентификатора для массива в awk. Таким образом, массив не может быть инициализирован с помощью нашего AWK-кода.

Массив в awk является ассоциативным, что означает, что любая строка или число могут быть ключом. Это означает, что массив больше похож на словарь пар ключ-значение, карту и т. Д. С другой стороны, массивы не имеют максимального размера.

Создание массива в AWK очень просто, поскольку вы берете имя переменной, правильный ключ и назначаете его переменной. Это означает, что когда выполняется следующий код, мы уже создали массив с именем `myArray` :

```
BEGIN {
    myArray["key"] = "value"
}
```

Мы не обязаны создавать массивы только в начале. Допустим, у нас есть следующий поток ввода:

```
A b c
D e f
G h i
```

И выполните следующий код с этим:

```
{
    myOtherArray[$1] = $2 "-" $3
}
# The array will look like this:
# myOtherArray["A"] = "b-c"
# myOtherArray["D"] = "e-f"
# myOtherArray["G"] = "h-i"
```

Когда массив заполняется парами значений ключа, можно получить значение только с помощью ключа. Это означает, что если мы используем ключ `"A"` в `myOtherArray` мы получаем `"bc"` .

```
END {
    print(myOtherArray["A"])
}
```

У нас также есть возможность прокручивать каждую клавишу, чтобы получить каждое значение. Проникновение через каждый ключ массива - это простая вещь, но она имеет дело с падением: она несортирована. Следующий цикл похож на цикл for-each, который извлекает ключ:

```
END {
    for (key in myOtherArray) {
        print "myOtherArray[\"" key "\"] = " myOtherArray[key]
    }
}
# Outputs (literal strings):
myOtherArray["A"] = "b-c"
myOtherArray["D"] = "e-f"
myOtherArray["G"] = "h-i"
```

Прочитайте Массивы онлайн: https://riptutorial.com/ru/awk/topic/7209/массивы

глава 6: Обработка двух файлов

Examples

Проверка совпадающих полей в двух файлах

Учитывая эти два файла CSV:

```
$ cat file1
1,line1
2,line2
3,line3
4,line4
$ cat file2
1,line3
2,line4
3,line5
4,line6
```

Чтобы напечатать эти строки в `file2` , второй столбец которого встречается также в первом файле, мы можем сказать:

```
$ awk -F, 'FNR==NR {lines[$2]; next} $2 in lines' file1 file2
1,line3
2,line4
```

Здесь `lines[]` содержат массив, который заполняется при чтении `file1` содержимым второго поля каждой строки.

Затем условие `$2 in lines` проверяет для каждой строки в `file2` , если в массиве существует 2-е поле. Если это так, условие равно True, а `awk` выполняет свое действие по умолчанию, состоящее в печати полной строки.

Если требуется только одно поле для печати, это может быть выражением:

```
$ awk -F, 'FNR==NR {lines[$2]; next} $2 in lines {print $1}' file1 file2
1
2
```

Печать переменных awk при чтении двух файлов

Надеюсь, что этот пример поможет каждому понять, как изменяются внутренние переменные awk, такие как NR, FNR и т. Д., Когда awk обрабатывает два файла.

```
awk '{print "NR:",NR,"FNR:",FNR,"fname:",FILENAME,"Field1:",$1}' file1 file2
NR: 1 FNR: 1 fname: file1 Field1: f1d1
NR: 2 FNR: 2 fname: file1 Field1: f1d5
```

```
NR: 3 FNR: 3 fname: file1 Field1: f1d9
NR: 4 FNR: 1 fname: file2 Field1: f2d1
NR: 5 FNR: 2 fname: file2 Field1: f2d5
NR: 6 FNR: 3 fname: file2 Field1: f2d9
```

Где file1 и file2 выглядят следующим образом:

```
$ cat file1
f1d1 f1d2 f1d3 f1d4

$ cat file2
f2d1 f2d2 f2d3 f2d4
```

Обратите внимание, что значение `NR` продолжает увеличиваться среди всех файлов, тогда как `FNR` сбрасывается на каждый файл. Вот почему выражение `NR==FNR` всегда относится к первому файлу, поданному в awk, поскольку только в первом файле возможно иметь `NR` равный `FNR` .

Прочитайте Обработка двух файлов онлайн: https://riptutorial.com/ru/awk/topic/4486/обработка-двух-файлов

глава 7: переменные

Examples

Назначение переменной командной строки

Чтобы назначить переменные из командной строки, можно использовать `-v` :

```
$ awk -v myvar="hello" 'BEGIN {print myvar}'
hello
```

Обратите внимание, что вокруг знака равенства нет пробелов.

Это позволяет использовать переменные оболочки:

```
$ shell_var="hello"
$ awk -v myvar="$shell_var" 'BEGIN {print myvar}'
hello
```

Кроме того, это позволяет устанавливать встроенные переменные, которые управляют `awk` :

См. Пример с `FS` (разделитель полей):

```
$ cat file
1,2;3,4
$ awk -v FS="," '{print $2}' file
2;3
$ awk -v FS=";" '{print $2}' file
3,4
```

Или с `OFS` (разделитель выходного поля):

```
$ echo "2 3" | awk -v OFS="--" '{print $1, $2}'
2--3
$ echo "2 3" | awk -v OFS="+" '{print $1, $2}'
2+3
```

Передача параметров программе с использованием опции -v

Опция `-v` за которой следует назначение *переменной variable = value,* может использоваться для передачи параметров в **awk-** программу. Это иллюстрируется нижеприведенной программой **наказания** , задачей которой является запись *отсчета* времени на предложение «Я не буду говорить в классе» на стандартном выходе. В следующем примере используется значение 100, которое очень популярно среди учителей:

```
awk -v count=100 'BEGIN {
```

```
  for(i = 1; i <= count; ++i) {
    print("I shall not talk in class.")
  }
  exit
}'
```

Можно передать несколько параметров с повторным использованием флага `-v` :

```
awk -v count=100 -v "sentence=I shall not talk in class." 'BEGIN {
  for(i = 1; i <= count; ++i) {
    print(sentence)
  }
  exit
}'
```

Нет встроенной поддержки параметров массива или списка, их нужно обрабатывать вручную. Классический подход для передачи параметра списка заключается в объединении списка с использованием разделителя, популярными являются `:` `|` или `,` . Функция *split* затем позволяет восстановить список как массив **awk** :

```
awk -v 'serialised_list=a:b:c:d:e:f' 'BEGIN {
  list_sz = split(serialised_list, list, ":")
  for(i = 1; i <= list_sz; ++i) {
    printf("list: %d: %s\n", i, list[i])
  }
  exit
}'
```

Вывод этой **awk-** программы

```
list: 1: a
list: 2: b
list: 3: c
list: 4: d
list: 5: e
list: 6: f
```

Иногда удобнее восстанавливать элементы списка как ключи массива **awk** , так как это позволяет легко проверять членство. Например, следующая программа печатает каждую строку, первое слово которой не принадлежит к фиксированному списку исключений:

```
awk -v 'serialised_exception_list=apple:pear:cherry' 'BEGIN {
  _list_sz = split(serialised_exception_list, _list, ":")
  for(i = 1; i <= _list_sz; ++i) {
    exception[_list[i]]
  }
}

! ($1 in exception) { print }' <<EOF
apple Apples are yummy, I like them.
pineapple Do you like pineapple?
EOF
```

Вывод этой программы

```
pineapple Do you like pineapple?
```

В качестве последнего примера мы покажем, как обернуть программу **наказания** в сценарий оболочки, поскольку это иллюстрирует, как сценарий оболочки передает параметры вспомогательному **awk-** скрипту:

```
#!/bin/sh

usage()
{
   cat <<EOF
Usage: punishment [-c COUNT][-s SENTENCE]
 Prepare your punishments for you
EOF
}

punishment_count='100'
punishment_sentence='I shall not talk in class.'
while getopts "c:hs:" OPTION; do
  case "${OPTION}" in
    c) punishment_count="${OPTARG}";;
    s) punishment_sentence="${OPTARG}";;
    h) usage; exit 0;;
    *) usage; exit 64;;
  esac
done

awk -v "count=${punishment_count}" -v "sentence=${punishment_sentence}" 'BEGIN {
  for(i = 1; i <= count; ++i) {
    print(sentence)
  }
  exit
}'
```

Местные переменные

Язык **awk** напрямую не поддерживает переменные, локальные для функций. Однако легко эмулировать их, добавляя дополнительные аргументы в функции. Традиционно префикс этих переменных обозначается _ чтобы указать, что они не являются фактическими параметрами.

Мы проиллюстрируем эту технику с помощью определения функции `single_quote` которая добавляет одинарные кавычки вокруг строки:

```
# single_quote(TEXT)
#  Return a string made of TEXT surrounded by single quotes

function single_quote(text, _quote) {
  _quote = sprintf("%c", 39)
  return sprintf("%s%s%s", _quote, text, _quote);
}
```

Более простой подход использования `sprintf("'%s'", text)` приводит к практическим проблемам, поскольку **awk-** скрипты обычно передаются как одинарные кавычки для **awk-** программы.

Аргументы присваивания

Аргументы присваивания появляются в конце `awk` вызова, в той же области, что и переменные файла, и назначения `-v` назначения аргументов должны соответствовать следующему регулярному выражению. (предполагая локаль POSIX)

```
^[[:alpha:]_][[:alnum:]_]*=
```

В следующем примере предполагается файл `file` содержащий следующее: `1 2 3` (выделение пробела)

```
$ awk '{$1=$1}1' file OFS=, file OFS=- file
1 2 3
1,2,3
1-2-3
```

Прочитайте переменные онлайн: https://riptutorial.com/ru/awk/topic/1403/переменные

глава 8: Полезные однострочные - вычисление среднего значения из CSV и т. д.

Examples

Надежная обработка табличных данных (CSV и др.)

Обработка табличных данных с помощью **awk** очень просто, при условии, что вход правильно отформатирован. Большинство программных продуктов, использующих табличные данные, используют специфические функции этого семейства форматов, а **awk-** программы, обрабатывающие табличные данные, часто специфичны для данных, созданных конкретным программным обеспечением. Если требуется более общее или надежное решение, большинство популярных языков предоставляют библиотеки, содержащие множество функций, найденных в табличных данных:

- необязательные имена столбцов в первой строке
- смесь значений котируемых и некотируемых столбцов
- различные разделители
- локализованные форматы для плавающих чисел

Хотя определенно можно обрабатывать все эти функции чисто и в целом с помощью **awk,** это, вероятно, не стоит усилий.

Обмен двумя столбцами в табличных данных

При использовании файла `;` как разделитель столбцов. Перенос первого и второго столбцов осуществляется

```
awk -F';' -v 'OFS=;' '{ swap = $2; $2 = $1; $1 = swap; print }'
```

Вычислить среднее значение значений в столбце из табличных данных

При использовании файла `;` как разделитель столбцов. Мы вычисляем среднее значение значений во втором столбце со следующей программой, предоставленный вход представляет собой список классов студенческой группы:

```
awk -F';' '{ sum += $2 } END { print(sum / NR) }' <<EOF
Alice;2
Victor;1
Barbara;1
```

```
Casper;4
Deborah;0
Ernest;1
Fabiola;4
Giuseppe;4
EOF
```

Результат этой программы - `2.125` .

Помните, что `NR` содержит номер обрабатываемой линии, в блоке `END` он удерживает общее количество строк в файле.

Помните, что во многих приложениях *(мониторинг, статистика)* медиана является гораздо более полезной информацией. См. Соответствующий пример.

Выбор определенных столбцов в табличных данных

Мы предполагаем использование файла; как разделитель столбцов. Для выбора определенного набора столбцов требуется только оператор *печати* . Например, следующая программа выбирает столбцы 3, 4 и 7 из своего ввода:

```
awk -F';' -v 'OFS=;' '{ print $3, $4, $7 }'
```

Как обычно, можно более тщательно выбирать строки для печати. Следующая программа выбирает столбцы 3, 4 и 7 из своего ввода, когда первое поле - это `Alice` или `Bob` :

```
awk -F';' -v 'OFS=;' '($1 == "Alice") || ($1 == "Bob") { print $3, $4, $7 }'
```

Вычислить медиану значений в столбце из табличных данных

При использовании файла ; как разделитель столбцов. Мы вычисляем медиану значений во втором столбце со следующей программой, написанной для **GNU awk** .
Предоставленный вход - это список классов студенческой группы:

```
gawk -F';' '{ sample[NR] = $2 }
 END {
   asort(sample);
   if(NR % 2 == 1) {
     print(sample[int(NR/2) + 1])
   } else {
     print(sample[NR/2])
   }
}' <<EOF
Alice;2
Victor;1
Barbara;1
Casper;4
Deborah;0
Ernest;1
Fabiola;4
```

```
Giuseppe;4
EOF
```

Результат этой программы: `1` .

Помните, что `NR` содержит номер обрабатываемой линии, в блоке `END` он удерживает общее количество строк в файле.

Многие реализации **awk** не имеют функции для сортировки массивов, поэтому их необходимо определить до того, как будет использован код выше.

Выбор набора линий между двумя шаблонами

Сравнение шаблонов может эффективно использоваться с `awk` поскольку оно контролирует действия, которые следуют за ним, т.е. `{ pattern } { action }` . Одним из интересных применений сопоставления шаблонов является выбор нескольких между двумя шаблонами в файле say `patternA` и `patternB`

```
$ awk '/patternA/,/patternB/' file
```

Предположим, что содержимое моего файла выглядит следующим образом, и я хочу извлечь строки только между указанным выше шаблоном:

```
$ cat file
This is line - 1
This is line - 2
patternA
This is line - 3
This is line - 4
This is line - 5
patternB
This is line - 6

$ awk '/patternA/,/patternB/' file
patternA
This is line - 3
This is line - 4
This is line - 5
patternB
```

Вышеупомянутая команда не выполняет никаких конкретных `{ action }` кроме как для печати строк, но любые конкретные действия в подмножестве строк могут применяться с помощью блока действий (`{}`).

Прочитайте Полезные однострочные - вычисление среднего значения из CSV и т. Д. онлайн: https://riptutorial.com/ru/awk/topic/3331/полезные-однострочные---вычисление-среднего-значения-из-csv-и-т--д-

глава 9: поля

Examples

Заполняющие поля

```
awk '{ for(f=1; f<=NF; f++) { print $f; } }' file
```

Прочитайте поля онлайн: https://riptutorial.com/ru/awk/topic/6245/поля

глава 10: Узоры

Examples

Шаблоны регулярных выражений

Шаблоны могут быть указаны как регулярные выражения:

```
/regular expression/ {action}
```

Например:

```
echo "user@hostname.com
not an email" | awk '/[^@]+@.+/ {print}'
```

Производит:

```
user@hostname.com
```

Обратите внимание, что действие, состоящее только из оператора `print` может быть полностью опущено. Вышеприведенный пример эквивалентен:

```
echo "user@hostname.com
not an email" | awk '/[^@]+@.+/'
```

Прочитайте Узоры онлайн: https://riptutorial.com/ru/awk/topic/3475/узоры

глава 11: Функции манипуляции строками

Синтаксис

- индекс (большой, маленький)
- длина или длина ()
- Длина (строка)
- match (string, regex)
- split (строка, массив, разделитель)
- split (строка, массив)
- sprintf (формат, ...)
- sub (regex, subst, string)
- sub (regex, subst)
- gsub (regex, subst)
- gsub (regex, subst, string)
- substr (строка, начало, конец)
- substr (строка, начало)
- ToLower (строка)
- ToUpper (строка)

параметры

параметр	подробности
большой	Строка, отсканированная для «маленького».
конец	Индекс, по которому заканчивается подстрока.
формат	Строка формата `printf` .
немного	Строка для сканирования в «большом».
регулярное выражение	Расширенное регулярное выражение .
Начните	Индекс, с которого начинается подстрока.
строка	Строка.
Подст	Строка для замены для согласованной части.

Examples

Преобразование строки в верхний регистр

Функция `toupper` преобразует строку в верхний регистр (заглавные буквы). Например:

```
BEGIN {
    greeting = "hello"
    loud_greeting = toupper(greeting)
    print loud_greeting
}
```

Этот код выдает «HELLO» при запуске.

Конкатенация строк

Конкатенация строк выполняется просто путем написания выражений рядом друг с другом без какого-либо оператора. Например:

```
BEGIN {
   user = "root"
   print "Hello "user "!"
}
```

будет печатать: `Hello root!`

Обратите внимание, что выражения не должны разделяться пробелами.

Вычисление хеша строки

Хотя реализация одного из стандартных алгоритмов хэширования в **awk** , вероятно, является утомительной задачей, определение *хеш-* функции, которая может использоваться как дескриптор текстовых документов, гораздо более сговорчива. Практическая ситуация, когда такая функция полезна, заключается в том, чтобы назначить короткие идентификаторы элементам, указанным в их описании, например, тестовые примеры, так что короткий идентификатор может быть указан как ссылка на элемент пользователем, а не на его длинное описание.

Хэш- функции необходимо преобразовать символы в числовые коды, которые выполняются с помощью таблицы поиска, инициализированной в начале скрипта. Затем *хэш-* функция вычисляется с использованием модулярных арифметических преобразований, очень классического подхода к вычислению хешей.

Для демонстрационных целей мы добавляем правило, чтобы украшать строки ввода хешем, но это правило не требуется для использования функции:

```
BEGIN{
  for(n=0;n<256;n++) {
    ord[sprintf("%c",n)] = n
  }
```

```
}

function hash(text, _prime, _modulo, _ax, _chars, _i)
{
  _prime = 104729;
  _modulo = 1048576;
  _ax = 0;
  split(text, _chars, "");
  for (_i=1; _i <= length(text); _i++) {
    _ax = (_ax * _prime + ord[_chars[_i]]) % _modulo;
  };
  return sprintf("%05x", _ax)
}

# Rule to demonstrate the function
#  These comments and the following line are not relevant
#  to the definition of the hash function but illustrate
#  its use.

{ printf("%s|%s\n", hash($0), $0) }
```

Мы сохраняем программу выше в файл `hash.awk` и демонстрируем ее в кратком списке классических английских книг:

```
awk -f hash.awk <<EOF
Wuthering Heights
Jane Eyre
Pride and Prejudice
The Mayor of Casterbridge
The Great Gatsby
David Copperfield
Great Expectations
The Return of the Soldier
Alice's Adventures in Wonderland
Animal Farm
EOF
```

Выход

```
6d6b1|Wuthering Heights
7539b|Jane Eyre
d8fba|Pride and Prejudice
fae95|The Mayor of Casterbridge
17fae|The Great Gatsby
c0005|David Copperfield
7492a|Great Expectations
12871|The Return of the Soldier
c3ab6|Alice's Adventures in Wonderland
46dc0|Animal Farm
```

При применении к каждой из 6948 непустых строк моего любимого романа эта хэш-функция не вызывает никакого столкновения.

Преобразование строки в нижний регистр

AWK часто используется для управления целыми файлами, содержащими список строк. Скажем, файл *awk_test_file.txt* содержит:

```
First String
Second String
Third String
```

Чтобы преобразовать все строки в нижний регистр:

```
awk '{ print tolower($0) }' awk_test_file.txt
```

Это приведет к:

```
first string
second string
third string
```

Подстановка текста строки

Функция SUB позволяет заменить текст внутри awk

    sub (регулярное выражение, замена, цель)

где regexp может быть полным регулярным выражением

```
$ cat file
AAAAA
BBBB
CCCC
DDDD
EEEE
FFFF
GGGG
$ awk '{sub("AAA","XXX", $0); print}' file
XXXAA
BBBB
CCCC
DDDD
EEEE
FFFF
GGGG
```

Выделение подстроки

`GNU` awk поддерживает функцию извлечения подстроки для возврата последовательности символов фиксированной длины из основной строки. Синтаксис

```
*substr(string, start [, length ])*
```

где, `string` - это `string` источника и `start` отметка начала позиции подстроки, которую вы

хотите, чтобы извлечение выполнялось для необязательных символов длины `length` . Если длина не указана, извлечение выполняется до конца строки.

Первый символ строки рассматривается как символ номер один.

```
awk '
BEGIN {
    testString = "MyTESTstring"
    substring  =  substr(testString, 3, 4)    # Start at character 3 for a length of 4
characters
    print substring
}'
```

выведет подстроку `TEST` .

```
awk '
BEGIN {
    testString = "MyTESTstring"
    substring  =  substr(testString, 3)    # Start at character 3 till end of the string
    print substring
}'
```

это извлекает подстроку из позиции символа 3 в конец всей строки, возвращая `TESTstring`

Заметка:-

- Если для `start` задано отрицательное значение, `GNU` awk печатает всю строку, а если `length` задана ненулевым значением, то поведение `GNU` awk возвращает `null` строку, и поведение варьируется в разных реализациях `awk` .

Прочитайте Функции манипуляции строками онлайн: https://riptutorial.com/ru/awk/topic/2284/функции-манипуляции-строками

глава 12: Шаблоны и действия

Examples

Вступление

`awk` состоит из шаблонов и действий, заключенных в фигурные скобки, которые должны быть приняты, если шаблон совпадает. Самый простой шаблон - пустой шаблон, который соответствует любой записи. Самое основное действие - это пустое действие, эквивалентное `{ print }` , которое, в свою очередь, эквивалентно `{ print $0 }` . Если оба шаблона и действие пустые, `awk` просто ничего не сделает.

Следующая программа просто повторит ввод, например:

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

Поскольку `{ print }` является действием по умолчанию, и поскольку истинное значение соответствует любой записи, эта программа может быть переписана как:

```
awk '1' /etc/passwd
```

Наиболее распространенным типом шаблона является, вероятно, регулярное выражение, заключенное в косые черты. Следующая программа будет печатать все записи, содержащие по крайней мере два последующих появления буквы `o` , например:

```
awk '/oo+/ { print }' /etc/passwd
```

Однако вы можете использовать произвольные выражения как шаблоны. Следующая программа печатает имена (полевые) пользователей в группе нуль (поле четыре), например:

```
awk -F: '$4 == 0 { print $1 }' /etc/passwd
```

Вместо того, чтобы точно соответствовать, вы также можете соответствовать регулярному выражению. Следующая программа печатает имена всех пользователей в группе с хотя бы одним нулем в своем идентификаторе группы:

```
awk -F: '$4 ~ /0/ { print $1 }' /etc/passwd
```

Линии фильтров по длине

Этот шаблон позволит вам фильтровать строки в зависимости от их длины

```
$cat file
AAAAA
BBBB
CCCC
DDDD
EEEE

$awk 'length($0) > 4 { print $0 }' file
AAAAA
$
```

Во всяком случае, шаблон позволит выполнить следующий блок кода, тогда как **действие по умолчанию для AWK печатает текущую строку** `{print}` , мы увидим тот же результат при выполнении этого:

```
$awk 'length($0) > 4 ' file
AAAAA
```

Прочитайте Шаблоны и действия онлайн: https://riptutorial.com/ru/awk/topic/3987/шаблоны-и-действия

кредиты

S. No	Главы	Contributors
1	Начало работы с awk	Andrea, antonio, AstraSerg, Community, David, fedorqui, George Vasiliou, kdhp, Michael Vehrs
2	Встроенные переменные	Alejandro Teixeira Muñoz, Andrzej Pronobis, FoldedChromatin, George Vasiliou, James Brown, sat
3	Встроенные функции	Alejandro Teixeira Muñoz, Armali
4	Манипуляция строк	pacomet, Scheff, UNagaswamy
5	Массивы	engineercoding
6	Обработка двух файлов	fedorqui, George Vasiliou
7	переменные	Andrzej Pronobis, anishsane, fedorqui, karakfa, kdhp, Michael Le Barbier Grünewald, Michael Vehrs
8	Полезные однострочные - вычисление среднего значения из CSV и т. Д.	Chris Seymour, Inian, karakfa, Michael Le Barbier Grünewald
9	поля	chaos
10	Узоры	Andrzej Pronobis
11	Функции манипуляции строками	Alejandro Teixeira Muñoz, Andrzej Pronobis, AstraSerg, fedorqui, Inian, kdhp, Michael Le Barbier Grünewald, schot, Thor
12	Шаблоны и действия	Alejandro Teixeira Muñoz, Michael Vehrs