sed&awk

主要内容

1、 What sed?

2、 Sed Basic usage

3、 Sed Advance usage

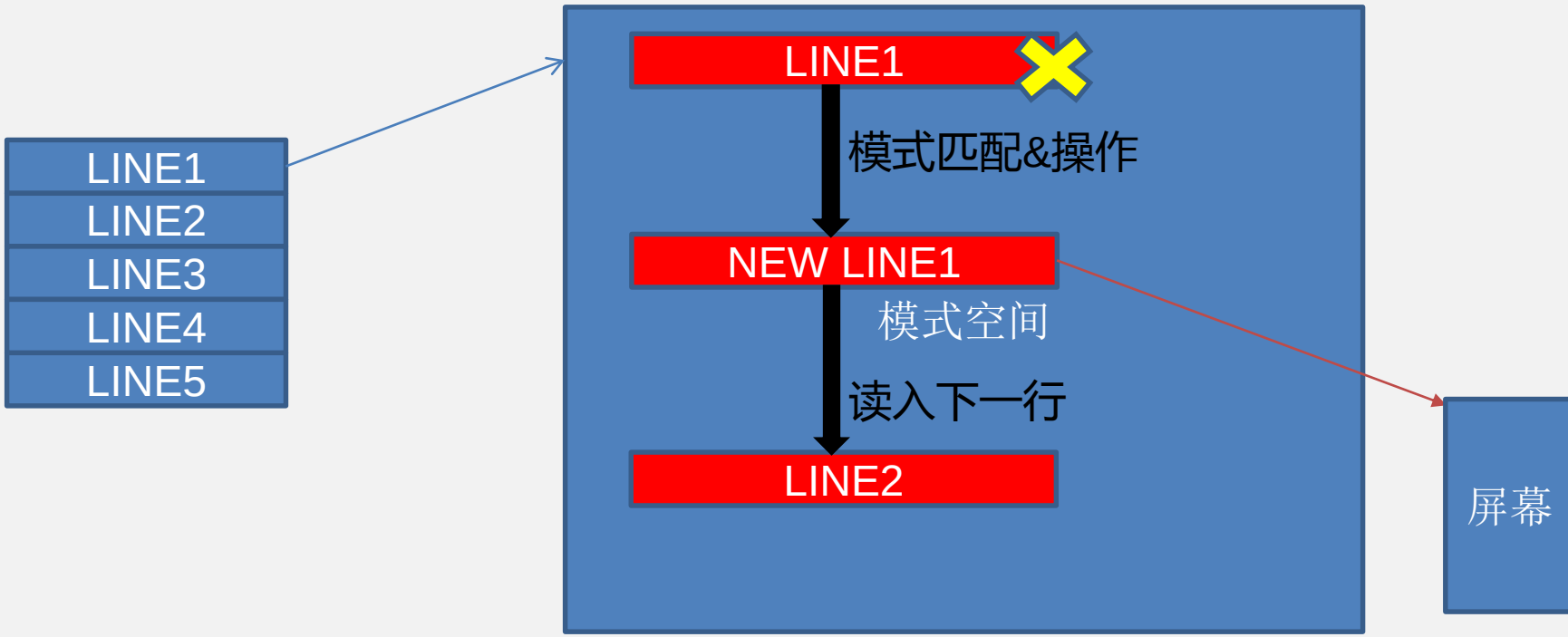
4、 What awk?

5、 Awk Basic

6、 Awk One Line script

What sed?

stream editor for filtering and transforming text



Sed Basic usage

sed options 'AddressCommand' file ....
options
Address
Command

Address:
1.Startline,Endline 匹配起始和结束
1,100
$ 表示最后一行
2./Pattern(RegExp)/ 匹配模式的行
/^root/
3./Pattern1/,/Pattern2/ 第一匹配到行pattern1，到第一次匹配到patter2

4.LineNumber 指定行

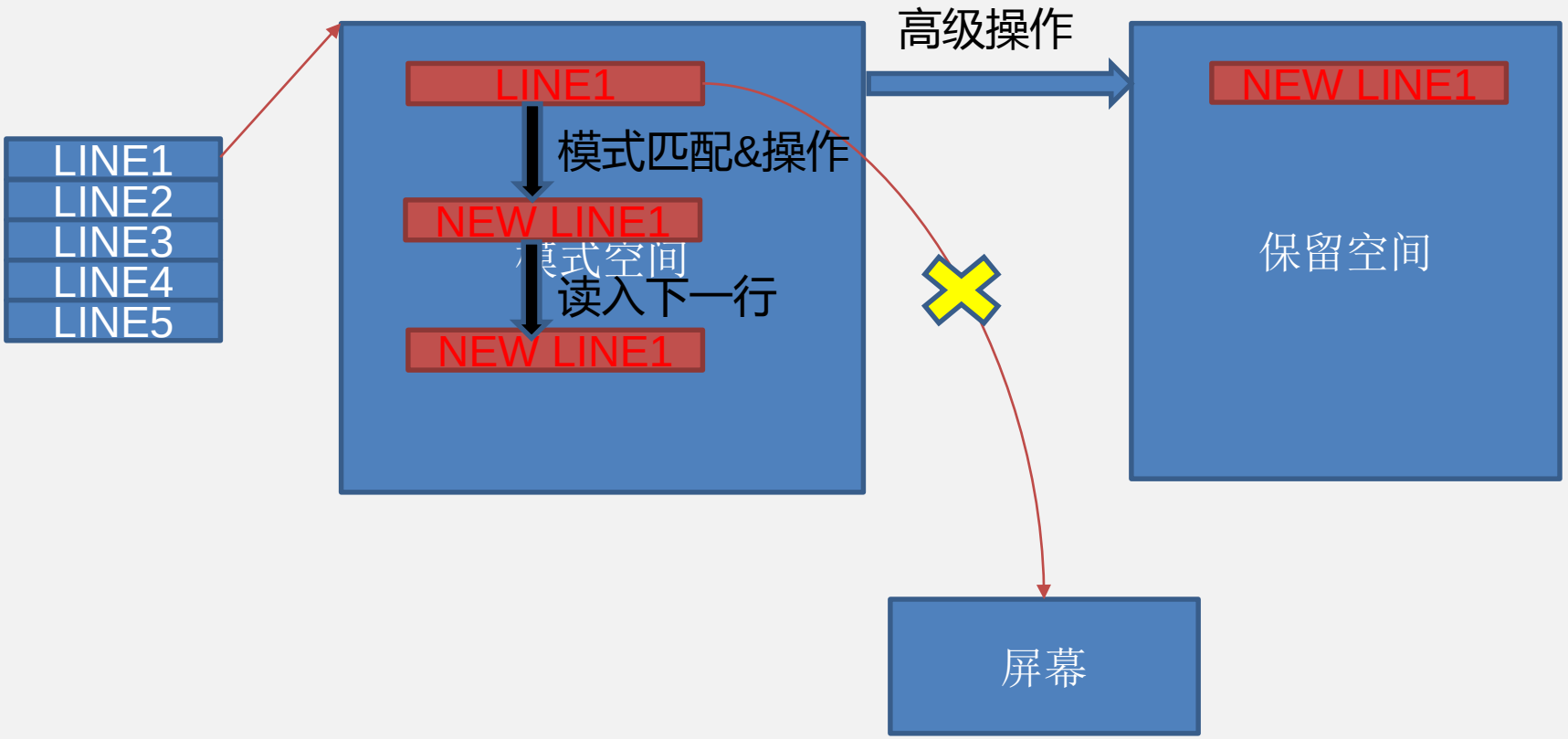
5.startline,+N
从startline开始，向后的N行

Sed Basic usage

Command:

- d: 删除模式空间中符合条件的行
- p: 输出模式空间中符合条件的行
- a \string: 在指定的行后面追加新行
- i \string: 在指定行的前面添加新行
- r FILE: 将FILE文件中的内容追加到符合条件的行后面
- w FILE: 将匹配的行保存到FILEL文件中
- s/pattern/string/: 查找并替换,默认只替换每行第一次匹配到的字符
  - 加修饰符:
    - g: 全局替换
    - i: 忽略字符大小写
- 分隔符可以是其他字符@,#
- 支持分组引用
- 注意: ;号用来实现多个命令

sed Advance usage



Sed Advance usage

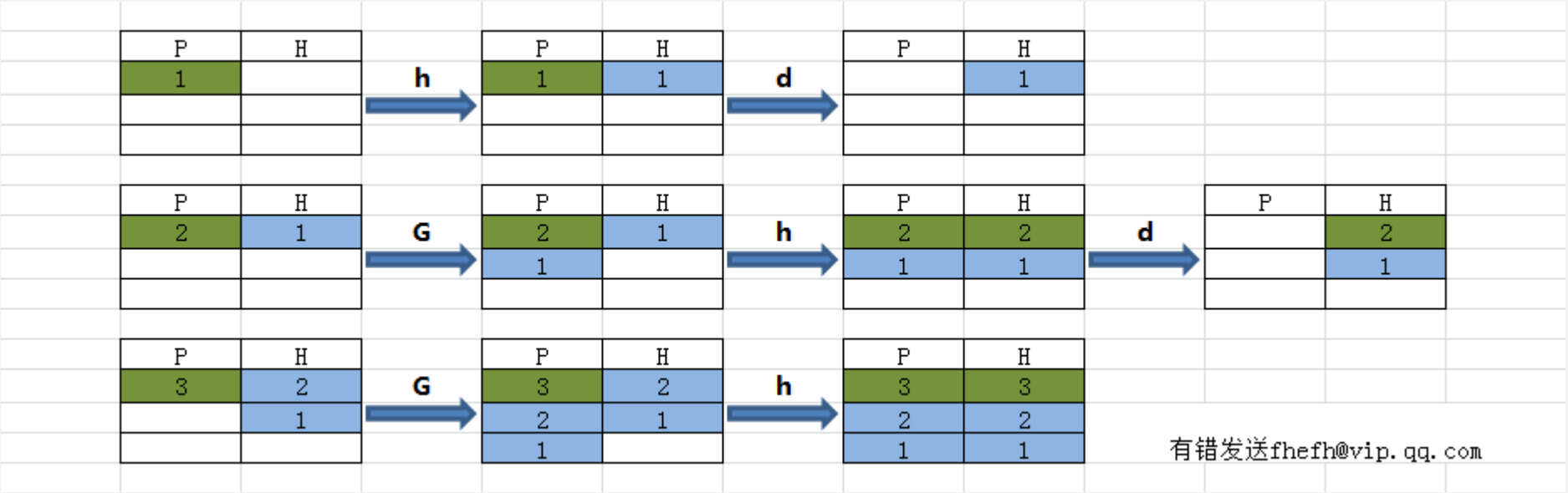
Command:

+ g : [address[,address]]g 将hold space中的内容拷贝到pattern space中，原来pattern space里的内容清除
+ G : [address[,address]]G 将hold space中的内容append到pattern space\n后
+ h : [address[,address]]h将pattern space中的内容拷贝到hold space中，原来的hold space里的内容被清除
+ H : [address[,address]]H 将pattern space中的内容append到hold space\n后
+ d : [address[,address]]d 删除pattern中的所有行，并读入下一新行到pattern中
+ D : [address[,address]]D 删除multiline pattern中的第一行，不读入下一行
+ x: 交换模式空间和Hold空间的内容

sed '1!G;h;$!d' filename

Sed Advance usage

sed '1!G;h;$!d' filename



Linux Hacker 不断提升自己逼格为己任

Sed Advance usage

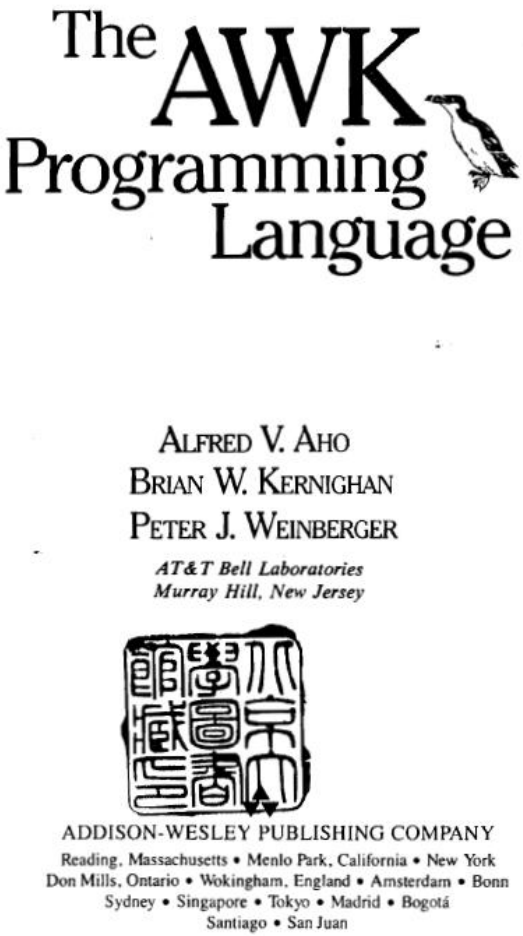
1. 在每一行后面增加空行
sed G
2. 删除空行
sed '/^$/d'
3. 在匹配到的行后插入一行
sed '/pattern/G'
4. 对文件的每一行进行编号
sed '=' /etc/passwd |sed 'N;s/\n/\t/'
5. 计算文本行数
sed -n '$=' /etc/passwd
6. 删除每行前导的空白字符
sed 's/^[[::space]]*//'
7. 删除每行的行尾的空白字符
sed 's/[[:space:]]*$//'
8. 倒置所有行(模拟tac)
sed '1!G;h;$!d'
sed -n '1!G;h;$p'
9. 两行连接成一行
sed 'N;s/\n/ /'
10. 删除文件顶部的所有空行
sed '/./,$!d'

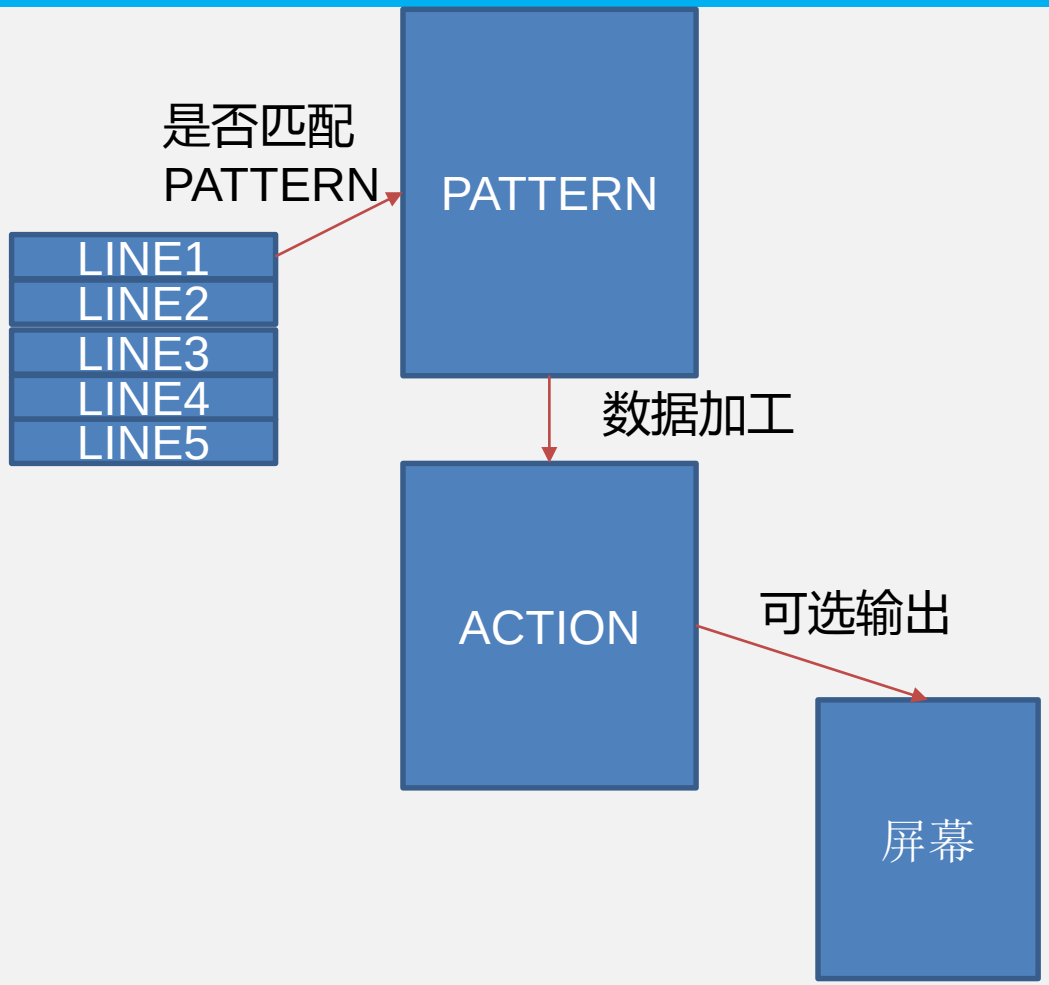
闲聊一会

What Awk

Pattern scanning and processig language
The report generator
Awk is a convenient and expressive programming language

What Awk


The AWK Programming Language
ALFRED V. AHO
BRIAN W. KERNIGHAN
PETER J. WEINBERGER
AT&T Bell Laboratories
Murray Hill, New Jersey
ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • Bogotá
Santiago • San Juan


是否匹配
PATTERN
PATTERN
LINE1
LINE2
LINE3
LINE4
LINE5
数据加工
ACTION
可选输出
屏幕

Awk Basic

awk [options] 'PATTERN { action }.....' file1 file2, ...

Variable内置变量
Pattern
action

Awk Basic

awk [options] 'PATTERN { action }.....' file1 file2, ...

Support C style Printf format output

Pattern:pattern模式

- $3 > 0 $3 == 0 $3 < 0
- $3 * $2 > 4
- $3 == "xxx"
- $3 > 0 && $2 < 1
- /^root/

Action:action

- normal combination output: example ptint $1,$3
- Computer: cnt = cnt + 1
- Handleing Text : number string convert to number
- String Concatenation: names = names $1 " "

Awk Basic

awk [options] 'PATTERN { action }.....' file1 file2, ...

Action:math func string func

built-in function: {print $1,length($1)}

Awk Basic

awk [options] 'PATTERN { action }.....' file1 file2, ...

Action:control-flow
- if else
- while
- for

Awk Basic

awk [options] 'PATTERN { action }.....' file1 file2, ...

Action:Array 关联数组 下标数组

Awk One Line script

awk [options] 'PATTERN { action }.....' file1 file2, ...

1. Print the total number of input lines:

```
END { print NR }
```

2. Print the tenth input line:

```
NR == 10
```

3. Print the last field of every input line:

```
{ print $NF }
```

4. Print the last field of the last input line:

```
    { field = $NF}
END { print field }
```

5. Print every input line with more than four fields:

```
NF > 4
```

6. Print every input line in which the last field is more than 4:

```
$NF > 4
```

7. Print the total number of fields in all input lines:

Awk One Line script

awk [options] 'PATTERN { action }.....' file1 file2, ...

9. Print the largest first field and the line that contains it (assumes some $1 is positive):

```
$1 > max { max = $1; maxline = $0 }
END      { print max, maxline }
```

10. Print every line that has at least one field:

```
NF > 0
```

11. Print every line longer than 80 characters:

```
length($0) > 80
```

12. Print the number of fields in every line followed by the line itself:

```
{ print NF, $0 }
```

13. Print the first two fields, in opposite order, of every line:

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

14. Exchange the first two fields of every line and then print the line:

```
{ temp = $1; $1 = $2; $2 = temp; print }
```

Awk One Line script

awk [options] 'PATTERN { action }.....' file1 file2, ...

18. Print the sums of the fields of every line:

```
{ sum = 0
  for (i = 1; i <= NF; i = i + 1) sum = sum + $i
  print sum
}
```

19. Add up all fields in all lines and print the sum:

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

20. Print every line after replacing each field by its absolute value:

```
{ for (i = 1; i <= NF; i = i + 1) if ($i < 0) $i = -$i
  print
}
```

谢谢！

TABLE 2-5. BUILT-IN VARIABLES

VARIABLE	MEANING	DEFAULT
ARGC	number of command-line arguments	-
ARGV	array of command-line arguments	-
FILENAME	name of current input file	-
FNR	record number in current file	-
FS	controls the input field separator	" "
NF	number of fields in current record	-
NR	number of records read so far	-
OFMT	output format for numbers	"%.6g"
OFS	output field separator	" "
ORS	output record separator	"\n"
RLENGTH	length of string matched by match function	-
RS	controls the input record separator	"\n"
RSTART	start of string matched by match function	-
SUBSEP	subscript separator	"\034"

Actions

The statements in actions can include:

expressions, with constants, variables, assignments, function calls, etc.

`print` *expression-list*

`printf(`*format*`,` *expression-list*`)`

`if (`*expression*`)` *statement*

`if (`*expression*`)` *statement* `else` *statement*

`while (`*expression*`)` *statement*

`for (`*expression*`;` *expression*`;` *expression*`)` *statement*

`for (`*variable* `in` *array*`)` *statement*

`do` *statement* `while (`*expression*`)`

`break`

`continue`

`next`

`exit`

`exit` *expression*

`{` *statements* `}`

TABLE 2-4. PATTERNS

PATTERN	EXAMPLE	MATCHES
BEGIN	`BEGIN`	before any input has been read
END	`END`	after all input has been read
expression	`$3 < 100`	lines in which third field is less than 100
string-matching	`/Asia/`	lines that contain `Asia`
compound	`$3 < 100 && $4 == "Asia"`	lines in which third field is less than 100 and fourth field is `Asia`
range	`NR==10, NR==20`	tenth to twentieth lines of input inclusive

String-Matching Patterns

1. */regexpr/*
   Matches when the current input line contains a substring matched by *regexpr*.

2. *expression* ~ */regexpr/*
   Matches if the string value of *expression* contains a substring matched by *regexpr*.

3. *expression* !~ */regexpr/*
   Matches if the string value of *expression* does not contain a substring matched by *regexpr*.

Any expression may be used in place of */regexpr/* in the context of ~ and !~.

Expressions

1. The primary expressions are:
   numeric and string constants, variables, fields, function calls, array elements.

2. These operators combine expressions:
   assignment operators = += -= *= /= %= ^=
   conditional expression operator ? :
   logical operators || (OR), && (AND), ! (NOT)
   matching operators ~ and !~
   relational operators < <= == != > >=
   concatenation (no explicit operator)
   arithmetic operators + - * / % ^
   unary + and -
   increment and decrement operators ++ and -- (prefix and postfix)
   parentheses for grouping

TABLE 2-6. BUILT-IN ARITHMETIC FUNCTIONS

FUNCTION	VALUE RETURNED
`atan2(`y`,`x`)`	arctangent of y/x in the range $-\pi$ to π
`cos(`x`)`	cosine of x, with x in radians
`exp(`x`)`	exponential function of x, e^x
`int(`x`)`	integer part of x; truncated towards 0 when $x > 0$
`log(`x`)`	natural (base e) logarithm of x
`rand()`	random number r, where $0 \leqslant r < 1$
`sin(`x`)`	sine of x, with x in radians
`sqrt(`x`)`	square root of x
`srand(`x`)`	x is new seed for `rand()`

TABLE 2-7. BUILT-IN STRING FUNCTIONS

FUNCTION	DESCRIPTION
`gsub(`*r*`,`*s*`)`	substitute *s* for *r* globally in `$0`, return number of substitutions made
`gsub(`*r*`,`*s*`,`*t*`)`	substitute *s* for *r* globally in string *t*, return number of substitutions made
`index(`*s*`,`*t*`)`	return first position of string *t* in *s*, or 0 if *t* is not present
`length(`*s*`)`	return number of characters in *s*
`match(`*s*`,`*r*`)`	test whether *s* contains a substring matched by *r*; return index or 0; sets `RSTART` and `RLENGTH`
`split(`*s*`,`*a*`)`	split *s* into array *a* on `FS`, return number of fields
`split(`*s*`,`*a*`,`*fs*`)`	split *s* into array *a* on field separator *fs*, return number of fields
`sprintf(`*fmt*`,`*expr-list*`)`	return *expr-list* formatted according to format string *fmt*
`sub(`*r*`,`*s*`)`	substitute *s* for the leftmost longest substring of `$0` matched by *r*; return number of substitutions made
`sub(`*r*`,`*s*`,`*t*`)`	substitute *s* for the leftmost longest substring of *t* matched by *r*; return number of substitutions made
`substr(`*s*`,`*p*`)`	return suffix of *s* starting at position *p*
`substr(`*s*`,`*p*`,`*n*`)`	return substring of *s* of length *n* starting at position *p*

Control-Flow Statements

`{` *statements* `}`
statement grouping

`if (`*expression*`)` *statement*
if *expression* is true, execute *statement*

`if (`*expression*`)` *statement*$_1$ `else` *statement*$_2$
if *expression* is true, execute *statement*$_1$ otherwise execute *statement*$_2$

`while (`*expression*`)` *statement*
if *expression* is true, execute *statement*, then repeat

`for (`*expression*$_1$`;` *expression*$_2$`;` *expression*$_3$`)` *statement*
equivalent to *expression*$_1$`;` `while (`*expression*$_2$`) {` *statement*`;` *expression*$_3$ `}`

`for (`*variable* `in` *array*`)` *statement*
execute *statement* with *variable* set to each subscript in *array* in turn

`do` *statement* `while (`*expression*`)`
execute *statement*; if *expression* is true, repeat

`break`
immediately leave innermost enclosing `while`, `for` or `do`

`continue`
start next iteration of innermost enclosing `while`, `for` or `do`

`next`
start next iteration of main input loop

`exit`
`exit` *expression*
go immediately to the `END` action; if within the `END` action, exit program entirely. Return *expression* as program status.

Output Statements

`print`
print $0 on standard output

`print` *expression*`,` *expression*`, ...`
print *expression*'s, separated by `OFS`, terminated by `ORS`

`print` *expression*`,` *expression*`, ...` `>`*filename*
print on file *filename* instead of standard output

`print` *expression*`,` *expression*`, ...` `>>`*filename*
append to file *filename* instead of overwriting previous contents

`print` *expression*`,` *expression*`, ...` `|` *command*
print to standard input of *command*

`printf(`*format*`,` *expression*`,` *expression*`, ...)`

`printf(`*format*`,` *expression*`,` *expression*`, ...)` `>`*filename*

`printf(`*format*`,` *expression*`,` *expression*`, ...)` `>>`*filename*

`printf(`*format*`,` *expression*`,` *expression*`, ...)` `|` *command*

`printf` statements are like `print` but the first argument specifies output format

`close(`*filename*`)`, `close(`*command*`)`
break connection between `print` and *filename* or *command*

`system(`*command*`)`
execute *command*; value is status return of *command*

The argument list of a `printf` statement does not need to be enclosed in parentheses. But if an expression in the argument list of a `print` or `printf` statement contains a relational operator, either the expression or the argument list must be enclosed in parentheses. Pipes and `system` may not be available on non-Unix systems.