A AWK SUMMARY

This appendix contains a summary of the awk language. In syntactic rules, components enclosed in brackets [...] are optional.

Command-line

`awk [-Fs] 'program' optional list of filenames`
`awk [-Fs] -f progfile optional list of filenames`

The option `-Fs` sets the field separator variable `FS` to *s*. If there are no filenames, the standard input is read. A filename can be of the form *var=text*, in which case it is treated as an assignment of *text* to the variable *var*, performed at the time when that argument would be accessed as a file.

AWK programs

An awk program is a sequence of pattern-action statements and function definitions. A pattern-action statement has the form:

pattern { *action* }

An omitted pattern matches all input lines; an omitted action prints a matched line.

A function definition has the form:

`function` *name(parameter-list)* { *statement* }

Pattern-action statements and function definitions are separated by newlines or semicolons and can be intermixed.

Patterns

`BEGIN`
`END`
expression
/regular expression/
pattern && *pattern*
pattern || *pattern*
!*pattern*
(*pattern*)
pattern, *pattern*

The last pattern is a range pattern, which cannot be part of another pattern. Similarly, `BEGIN` and `END` do not combine with other patterns.

Actions

An action is a sequence of statements of the following kinds:

```
break
continue
delete array-element
do statement while (expression)
exit [expression]
expression
if (expression) statement [else statement]
input-output statement
for (expression; expression; expression) statement
for (variable in array) statement
next
return [expression]
while (expression) statement
{ statements }
```

A semicolon by itself denotes the empty statement. In an `if-else` statement, the first *statement* must be terminated by a semicolon or enclosed in braces if it appears on the same line as `else`. Similarly, in a `do` statement, *statement* must be terminated by a semicolon or enclosed in braces if it appears on the same line as `while`.

Program format

Statements are separated by newlines or semicolons or both. Blank lines may be inserted before or after any statement, pattern-action statement, or function definition. Spaces and tabs may be inserted around operators and operands. A long statement may be broken by a backslash. In addition, a statement may be broken without a backslash after a comma, left brace, `&&`, `||`, `do`, `else`, and the right parenthesis in an `if` or `for` statement. A comment beginning with # can be put at the end of any line.

Input-output

`close(`*expr*`)`	close file or pipe denoted by *expr*
`getline`	set `$0` from next input record; set `NF`, `NR`, `FNR`
`getline <`*file*	set `$0` from next record of *file*; set `NF`
`getline` *var*	set *var* from next input record; set `NR`, `FNR`
`getline` *var* `<`*file*	set *var* from next record of *file*
`print`	print current record
`print` *expr-list*	print expressions in *expr-list*
`print` *expr-list* `>`*file*	print expressions on *file*
`printf` *fmt, expr-list*	format and print
`printf` *fmt, expr-list* `>`*file*	format and print on *file*
`system(`*cmd-line*`)`	execute command *cmd-line*, return status

The *expr-list* following `print` and the *fmt, expr-list* following `printf` may be parenthesized. In `print` and `printf`, `>>`*file* appends to the *file*, and `|` *command* writes on a pipe. Similarly, *command* `|` `getline` pipes into `getline`. The function `getline` returns 0 on end of file, and −1 on error.

Printf format conversions

These conversions are recognized in `printf` and `sprintf` statements.

`%c`	ASCII character
`%d`	decimal number
`%e`	`[-]d.dddddd E[+-]dd`
`%f`	`[-]ddd.dddddd`
`%g`	`e` or `f` conversion, whichever is shorter, with nonsignificant zeros suppressed
`%o`	unsigned octal number
`%s`	string
`%x`	unsigned hexadecimal number
`%%`	print a %; no argument is converted

Additional parameters may lie between the % and the control letter:

`-`	left-justify expression in its field
width	pad field to this width as needed; leading 0 pads with zeros
.prec	maximum string width or digits to right of decimal point

Built-in variables

The following built-in variables can be used in any expression:

`ARGC`	number of command-line arguments
`ARGV`	array of command-line arguments (`ARGV[0..ARGC-1]`)
`FILENAME`	name of current input file
`FNR`	input record number in current file
`FS`	input field separator (default blank)
`NF`	number of fields in current input record
`NR`	input record number since beginning
`OFMT`	output format for numbers (default `"%.6g"`)
`OFS`	output field separator (default blank)
`ORS`	output record separator (default newline)
`RLENGTH`	length of string matched by regular expression in `match`
`RS`	input record separator (default newline)
`RSTART`	beginning position of string matched by `match`
`SUBSEP`	separator for array subscripts of form [*i*,*j*,...] (default `"\034"`)

`ARGC` and `ARGV` include the name of the invoking program (usually `awk`) but not the program arguments or options. `RSTART` is also the value returned by `match`.

The current input record is named `$0`. The fields in the current input record are named `$1`, `$2`, ..., `$NF`.

Built-in string functions

In the following string functions, *s* and *t* represent strings, *r* a regular expression, and *i* and *n* integers.

An `&` in the replacement string *s* in `sub` and `gsub` is replaced by the matched string; `\&` yields a literal ampersand.

`gsub(r,s,t)`	globally substitute *s* for each substring of *t* matched by *r*, return number of substitutions; if *t* is omitted, `$0` is used
`index(s,t)`	return the index of *t* in *s*, or 0 if *s* does not contain *t*
`length(s)`	return the length of *s*
`match(s,r)`	return index of where *s* matches *r* or 0 if there is no match; set `RSTART` and `RLENGTH`
`split(s,a,fs)`	split *s* into array *a* on *fs*, return number of fields; if *fs* is omitted, `FS` is used in its place
`sprintf(`*fmt, expr-list*`)`	return *expr-list* formatted according to *fmt*
`sub(r,s,t)`	like `gsub` except only the first matched substring is replaced
`substr(s,i,n)`	return the *n*-character substring of *s* starting at *i*; if *n* is omitted, return the suffix of *s* starting at *i*

Built-in arithmetic functions

`atan2(y,x)`	arctangent of y/x in radians in the range $-\pi$ to π
`cos(x)`	cosine (angle in radians)
`exp(x)`	exponential e^x
`int(x)`	truncate to integer
`log(x)`	natural logarithm
`rand()`	pseudo-random number r, $0 \leqslant r < 1$
`sin(x)`	sine (angle in radians)
`sqrt(x)`	square root
`srand(x)`	set new seed for random number generator; uses time of day if no *x* given

Expression operators (increasing in precedence)

Expressions may be combined with the following operators:

`= += -= *= /= %= ^=`	assignment
`?:`	conditional expression
`||`	logical OR
`&&`	logical AND
`in`	array membership
`~ !~`	regular expression match, negated match
`< <= > >= != ==`	relationals
	string concatenation (no explicit operator)
`+ -`	add, subtract
`* / %`	multiply, divide, mod
`+ - !`	unary plus, unary minus, logical NOT
`^`	exponentiation
`++ --`	increment, decrement (prefix and postfix)
`$`	field

All operators are left associative, except assignment, `?:`, and `^`, which are right associative. Any expression may be parenthesized.

Regular expressions

The regular expression metacharacters are

```
\ ^ $ . [ ] | ( ) * + ?
```

The following table summarizes regular expressions and the strings they match:

c	matches the nonmetacharacter *c*
c	matches the escape sequence or literal character *c*
^	matches the beginning of a string
$	matches the end of a string
.	matches any single character
[*abc...*]	character class: matches any of *abc...*
[^*abc...*]	negated class: matches any single character but *abc...*
$r_1	r_2$	alternation: matches any string matched by r_1 or r_2
$(r_1)(r_2)$	concatenation: matches xy where r_1 matches x and r_2 matches y
(*r*)*	matches zero or more consecutive strings matched by *r*
(*r*)+	matches one or more consecutive strings matched by *r*
(*r*)?	matches the null string or one string matched by *r*
(*r*)	grouping: matches the same strings as *r*

The operators are listed in increasing precedence. Redundant parentheses in regular expressions may be omitted as long as the precedence of operators is respected.

Escape sequences

These sequences have special meanings in strings and regular expressions.

\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tab
ddd	octal value *ddd*, where *ddd* is 1 to 3 digits between 0 and 7
c	any other character *c* literally, e.g., \" for " and \\ for \

Limits

Any particular implementation of awk enforces some limits. Here are typical values:

100 fields
3000 characters per input record
3000 characters per output record
1024 characters per field
3000 characters per `printf` string
400 characters maximum literal string
400 characters in character class
15 open files
1 pipe
double-precision floating point

Numbers are limited to what can be represented on the local machine, e.g., $10^{-38}..10^{38}$; numbers outside this range will have string values only.

Initialization, comparison, and type coercion

Each variable and field can potentially be a string or a number or both at any time. When a variable is set by an assignment

```
var = expr
```

its type is set to that of the expression. ("Assignment" includes +=, -=, etc.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in `v1 = v2`, then the type of `v1` is set to that of `v2`.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by subterfuges such as

```
expr + 0
```

and to string by

```
expr ""
```

(i.e., concatenation with a null string). The numeric value of an arbitrary string is the numeric value of its numeric prefix.

Uninitialized variables have the numeric value 0 and the string value "". Accordingly, if `x` is uninitialized,

```
if (x) ...
```

is false, and

```
if (!x) ...
if (x == 0) ...
if (x == "") ...
```

are all true. But note that

```
if (x == "0") ...
```

is false.

The type of a field is determined by context when possible; for example,

```
$1++
```

implies that `$1` must be coerced to numeric if necessary, and

```
$1 = $1 "," $2
```

implies that `$1` and `$2` will be coerced to strings if necessary.

In contexts where types cannot be reliably determined, e.g.,

```
if ($1 == $2) ...
```

the type of each field is determined on input. All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value ""; they are not numeric. Nonexistent fields (i.e., fields past `NF`) and `$0` for blank lines are treated this way too.

As it is for fields, so it is for array elements created by `split`.

Mentioning a variable in an expression causes it to exist, with the values 0 and "" as described above. Thus, if `arr[i]` does not currently exist,

```
if (arr[i] == "") ...
```

causes it to exist with the value "" and thus the `if` is satisfied. The test

```
if (i in arr) ...
```

determines if `arr[i]` exists without the side effect of creating it.