The AWK Manual

Edition 1.0
December 1995

Diane Barlow Close
Arnold D. Robbins
Paul H. Rubin
Richard Stallman
Piet van Oostrum

Copyright © 1989, 1991, 1992, 1993 Free Software Foundation, Inc.

This is Edition 1.0 of *The AWK Manual*,
for the new implementation of AWK (sometimes called nawk).

Notice: This work is derived from the original `gawk` manual. Adaptions for NAWK made by Piet van Oostrum, Dec. 1995, July 1998.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Foundation.

Preface

If you are like many computer users, you would frequently like to make changes in various text files wherever certain patterns appear, or extract data from parts of certain lines while discarding the rest. To write a program to do this in a language such as C or Pascal is a time-consuming inconvenience that may take many lines of code. The job may be easier with `awk`.

The `awk` utility interprets a special-purpose programming language that makes it possible to handle simple data-reformatting jobs easily with just a few lines of code.

This manual teaches you what `awk` does and how you can use `awk` effectively. You should already be familiar with basic system commands such as `ls`. Using `awk` you can:

- manage small, personal databases
- generate reports
- validate data
- produce indexes, and perform other document preparation tasks
- even experiment with algorithms that can be adapted later to other computer languages

This manual has the difficult task of being both tutorial and reference. If you are a novice, feel free to skip over details that seem too complex. You should also ignore the many cross references; they are for the expert user, and for the on-line Info version of the manual.

History of `awk`

The name `awk` comes from the initials of its designers: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan. The original version of `awk` was written in 1977. In 1985 a new version made the programming language more powerful, introducing user-defined functions, multiple input streams, and computed regular expressions. This new version became generally available with System V Release 3.1. The version in System V Release 4 added some new features and also cleaned up the behavior in some of the “dark corners” of the language. The specification for `awk` in the POSIX Command Language and Utilities standard further clarified the language.

We need to thank many people for their assistance in producing this manual. Jay Fenlason contributed many ideas and sample programs. Richard Mlynarik and Robert J. Chassell gave helpful comments on early drafts of this manual. The paper *A Supplemental Document for* `awk` by John W. Pierce of the Chemistry Department at UC San Diego, pinpointed several issues relevant both to `awk` implementation and to this manual, that would otherwise have escaped us. David Trueman, Pat Rankin, and Michal Jaegermann also contributed sections of the manual.

The following people provided many helpful comments on this edition of the manual: Rick Adams, Michael Brennan, Rich Burridge, Diane Close, Christopher (“Topher”) Eliot, Michael Lijewski, Pat Rankin, Miriam Robbins, and Michal Jaegermann. Robert J. Chassell provided much valuable advice on the use of Texinfo.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),

EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

1 Using this Manual

The term `awk` refers to a particular program, and to the language you use to tell this program what to do. When we need to be careful, we call the program "the `awk` utility" and the language "the `awk` language." The term `gawk` refers to a version of `awk` developed as part the GNU project. The purpose of this manual is to explain both the `awk` language and how to run the `awk` utility.

The term `awk` *program* refers to a program written by you in the `awk` programming language.

See Chapter 2 [Getting Started with `awk`], page 13, for the bare essentials you need to know to start using `awk`.

Some useful "one-liners" are included to give you a feel for the `awk` language (see Chapter 5 [Useful "One-liners"], page 45).

A sample `awk` program has been provided for you (see Appendix B [Sample Program], page 119).

If you find terms that you aren't familiar with, try looking them up in the glossary (see Appendix C [Glossary], page 121).

The entire `awk` language is summarized for quick reference in Appendix A [`awk` Summary], page 107. Look there if you just need to refresh your memory about a particular feature.

Most of the time complete `awk` programs are used as examples, but in some of the more advanced sections, only the part of the `awk` program that illustrates the concept being described is shown.

1.1 Data Files for the Examples

Many of the examples in this manual take their input from two sample data files. The first, called '`BBS-list`', represents a list of computer bulletin board systems together with information about those systems. The second data file, called '`inventory-shipped`', contains information about shipments on a monthly basis. Each line of these files is one *record*.

In the file '`BBS-list`', each record contains the name of a computer bulletin board, its phone number, the board's baud rate, and a code for the number of hours it is operational. An '`A`' in the last column means the board operates 24 hours a day. A '`B`' in the last column means the board operates evening and weekend hours, only. A '`C`' means the board operates only on weekends.

```
aardvark     555-5553     1200/300          B
alpo-net     555-3412     2400/1200/300     A
barfly       555-7685     1200/300          A
bites        555-1675     2400/1200/300     A
camelot      555-0542     300               C
core         555-2912     1200/300          C
fooey        555-1234     2400/1200/300     B
foot         555-6699     1200/300          B
macfoo       555-6480     1200/300          A
sdace        555-3430     2400/1200/300     A
```

```
sabafoo       555-2127     1200/300          C
```

The second data file, called '`inventory-shipped`', represents information about shipments during the year. Each record contains the month of the year, the number of green crates shipped, the number of red boxes shipped, the number of orange bags shipped, and the number of blue packages shipped, respectively. There are 16 entries, covering the 12 months of one year and 4 months of the next year.

```
Jan  13  25  15 115
Feb  15  32  24 226
Mar  15  24  34 228
Apr  31  52  63 420
May  16  34  29 208
Jun  31  42  75 492
Jul  24  34  67 436
Aug  15  34  47 316
Sep  13  55  37 277
Oct  29  54  68 525
Nov  20  87  82 577
Dec  17  35  61 401

Jan  21  36  64 620
Feb  26  58  80 652
Mar  24  75  70 495
Apr  21  70  74 514
```

2 Getting Started with awk

The basic function of `awk` is to search files for lines (or other units of text) that contain certain patterns. When a line matches one of the patterns, `awk` performs specified actions on that line. `awk` keeps processing input lines in this way until the end of the input file is reached.

When you run `awk`, you specify an `awk` *program* which tells `awk` what to do. The program consists of a series of *rules*. (It may also contain *function definitions*, but that is an advanced feature, so we will ignore it for now. See Chapter 12 [User-defined Functions], page 95.) Each rule specifies one pattern to search for, and one action to perform when that pattern is found.

Syntactically, a rule consists of a pattern followed by an action. The action is enclosed in curly braces to separate it from the pattern. Rules are usually separated by newlines. Therefore, an `awk` program looks like this:

```
pattern { action }
pattern { action }
...
```

2.1 A Very Simple Example

The following command runs a simple `awk` program that searches the input file '`BBS-list`' for the string of characters: '`foo`'. (A string of characters is usually called, a *string*. The term *string* is perhaps based on similar usage in English, such as "a string of pearls," or, "a string of cars in a train.")

```
awk '/foo/ { print $0 }' BBS-list
```

When lines containing '`foo`' are found, they are printed, because '`print $0`' means print the current line. (Just '`print`' by itself means the same thing, so we could have written that instead.)

You will notice that slashes, '`/`', surround the string '`foo`' in the actual `awk` program. The slashes indicate that '`foo`' is a pattern to search for. This type of pattern is called a *regular expression*, and is covered in more detail later (see Section 6.2 [Regular Expressions as Patterns], page 47). There are single-quotes around the `awk` program so that the shell won't interpret any of it as special shell characters.

Here is what this program prints:

```
fooey        555-1234     2400/1200/300     B
foot         555-6699     1200/300          B
macfoo       555-6480     1200/300          A
sabafoo      555-2127     1200/300          C
```

In an `awk` rule, either the pattern or the action can be omitted, but not both. If the pattern is omitted, then the action is performed for *every* input line. If the action is omitted, the default action is to print all lines that match the pattern.

Thus, we could leave out the action (the `print` statement and the curly braces) in the above example, and the result would be the same: all lines matching the pattern '`foo`' would be printed. By comparison, omitting the `print` statement but retaining the curly braces makes an empty action that does nothing; then no lines would be printed.

2.2 An Example with Two Rules

The `awk` utility reads the input files one line at a time. For each line, `awk` tries the patterns of each of the rules. If several patterns match then several actions are run, in the order in which they appear in the `awk` program. If no patterns match, then no actions are run.

After processing all the rules (perhaps none) that match the line, `awk` reads the next line (however, see Section 9.7 [The `next` Statement], page 78). This continues until the end of the file is reached.

For example, the `awk` program:

```
/12/  { print $0 }
/21/  { print $0 }
```

contains two rules. The first rule has the string '`12`' as the pattern and '`print $0`' as the action. The second rule has the string '`21`' as the pattern and also has '`print $0`' as the action. Each rule's action is enclosed in its own pair of braces.

This `awk` program prints every line that contains the string '`12`' *or* the string '`21`'. If a line contains both strings, it is printed twice, once by each rule.

If we run this program on our two sample data files, '`BBS-list`' and '`inventory-shipped`', as shown here:

```
awk '/12/ { print $0 }
     /21/ { print $0 }' BBS-list inventory-shipped
```

we get the following output:

```
aardvark     555-5553     1200/300          B
alpo-net     555-3412     2400/1200/300     A
barfly       555-7685     1200/300          A
bites        555-1675     2400/1200/300     A
core         555-2912     1200/300          C
fooey        555-1234     2400/1200/300     B
foot         555-6699     1200/300          B
macfoo       555-6480     1200/300          A
sdace        555-3430     2400/1200/300     A
sabafoo      555-2127     1200/300          C
sabafoo      555-2127     1200/300          C
Jan  21  36  64 620
Apr  21  70  74 514
```

Note how the line in ‘BBS-list’ beginning with ‘sabafoo’ was printed twice, once for each rule.

2.3 A More Complex Example

Here is an example to give you an idea of what typical awk programs do. This example shows how awk can be used to summarize, select, and rearrange the output of another utility. It uses features that haven’t been covered yet, so don’t worry if you don’t understand all the details.

```
ls -l | awk '$5 == "Nov" { sum += $4 }
             END { print sum }'
```

This command prints the total number of bytes in all the files in the current directory that were last modified in November (of any year). (In the C shell you would need to type a semicolon and then a backslash at the end of the first line; in a POSIX-compliant shell, such as the Bourne shell or the Bourne-Again shell, you can type the example as shown.)

The ‘ls -l’ part of this example is a command that gives you a listing of the files in a directory, including file size and date. Its output looks like this:

```
-rw-r--r--  1 close        1933 Nov  7 13:05 Makefile
-rw-r--r--  1 close       10809 Nov  7 13:03 awk.h
-rw-r--r--  1 close         983 Apr 13 12:14 awk.tab.h
-rw-r--r--  1 close       31869 Jun 15 12:20 awk.y
-rw-r--r--  1 close       22414 Nov  7 13:03 awk1.c
-rw-r--r--  1 close       37455 Nov  7 13:03 awk2.c
-rw-r--r--  1 close       27511 Dec  9 13:07 awk3.c
-rw-r--r--  1 close        7989 Nov  7 13:03 awk4.c
```

The first field contains read-write permissions, the second field contains the number of links to the file, and the third field identifies the owner of the file. The fourth field contains the size of the file in bytes. The fifth, sixth, and seventh fields contain the month, day, and time, respectively, that the file was last modified. Finally, the eighth field contains the name of the file.

The $5 == "Nov" in our awk program is an expression that tests whether the fifth field of the output from ‘ls -l’ matches the string ‘Nov’. Each time a line has the string ‘Nov’ in its fifth field, the action ‘{ sum += $4 }’ is performed. This adds the fourth field (the file size) to the variable sum. As a result, when awk has finished reading all the input lines, sum is the sum of the sizes of files whose lines matched the pattern. (This works because awk variables are automatically initialized to zero.)

After the last line of output from ls has been processed, the END rule is executed, and the value of sum is printed. In this example, the value of sum would be 80600.

These more advanced awk techniques are covered in later sections (see Chapter 7 [Overview of Actions], page 55). Before you can move on to more advanced awk programming, you have to know how awk interprets your input and displays your output. By manipulating fields and using print statements, you can produce some very useful and spectacular looking reports.

2.4 How to Run `awk` Programs

There are several ways to run an `awk` program. If the program is short, it is easiest to include it in the command that runs `awk`, like this:

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

where *program* consists of a series of patterns and actions, as described earlier.

When the program is long, it is usually more convenient to put it in a file and run it with a command like this:

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

2.4.1 One-shot Throw-away `awk` Programs

Once you are familiar with `awk`, you will often type simple programs at the moment you want to use them. Then you can write the program as the first argument of the `awk` command, like this:

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

where *program* consists of a series of *patterns* and *actions*, as described earlier.

This command format instructs the shell to start `awk` and use the *program* to process records in the input file(s). There are single quotes around *program* so that the shell doesn't interpret any `awk` characters as special shell characters. They also cause the shell to treat all of *program* as a single argument for `awk` and allow *program* to be more than one line long.

This format is also useful for running short or medium-sized `awk` programs from shell scripts, because it avoids the need for a separate file for the `awk` program. A self-contained shell script is more reliable since there are no other files to misplace.

2.4.2 Running `awk` without Input Files

You can also run `awk` without any input files. If you type the command line:

```
awk 'program'
```

then `awk` applies the *program* to the *standard input*, which usually means whatever you type on the terminal. This continues until you indicate end-of-file by typing *`Control-d`*.

For example, if you execute this command:

```
awk '/th/'
```

whatever you type next is taken as data for that `awk` program. If you go on to type the following data:

```
Kathy
Ben
Tom
Beth
Seth
Karen
Thomas
Control-d
```

then `awk` prints this output:

```
Kathy
Beth
Seth
```

as matching the pattern '`th`'. Notice that it did not recognize '`Thomas`' as matching the pattern. The `awk` language is *case sensitive*, and matches patterns exactly.

2.4.3 Running Long Programs

Sometimes your `awk` programs can be very long. In this case it is more convenient to put the program into a separate file. To tell `awk` to use that file for its program, you type:

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

The '`-f`' instructs the `awk` utility to get the `awk` program from the file *source-file*. Any file name can be used for *source-file*. For example, you could put the program:

```
/th/
```

into the file '`th-prog`'. Then this command:

```
awk -f th-prog
```

does the same thing as this one:

```
awk '/th/'
```

which was explained earlier (see Section 2.4.2 [Running `awk` without Input Files], page 16). Note that you don't usually need single quotes around the file name that you specify with '`-f`', because most file names don't contain any of the shell's special characters. Notice that in '`th-prog`', the `awk` program did not have single quotes around it. The quotes are only needed for programs that are provided on the `awk` command line.

If you want to identify your `awk` program files clearly as such, you can add the extension '`.awk`' to the file name. This doesn't affect the execution of the `awk` program, but it does make "housekeeping" easier.

2.4.4 Executable `awk` Programs

Once you have learned `awk`, you may want to write self-contained `awk` scripts, using the '`#!`' script mechanism. You can do this on many Unix systems[1] (and someday on GNU).

For example, you could create a text file named '`hello`', containing the following (where '`BEGIN`' is a feature we have not yet discussed):

```
#! /bin/awk -f

# a sample awk program
BEGIN    { print "hello, world" }
```

After making this file executable (with the `chmod` command), you can simply type:

```
hello
```

at the shell, and the system will arrange to run `awk`[2] as if you had typed:

```
awk -f hello
```

Self-contained `awk` scripts are useful when you want to write a program which users can invoke without knowing that the program is written in `awk`.

If your system does not support the '`#!`' mechanism, you can get a similar effect using a regular shell script. It would look something like this:

```
: The colon makes sure this script is executed by the Bourne shell.
awk 'program' "$@"
```

Using this technique, it is *vital* to enclose the *program* in single quotes to protect it from interpretation by the shell. If you omit the quotes, only a shell wizard can predict the results.

The '`"$@"`' causes the shell to forward all the command line arguments to the `awk` program, without interpretation. The first line, which starts with a colon, is used so that this shell script will work even if invoked by a user who uses the C shell.

2.5 Comments in `awk` Programs

A *comment* is some text that is included in a program for the sake of human readers, and that is not really part of the program. Comments can explain what the program does, and how it works.

[1] The '`#!`' mechanism works on Unix systems derived from Berkeley Unix, System V Release 4, and some System V Release 3 systems.

[2] The line beginning with '`#!`' lists the full pathname of an interpreter to be run, and an optional initial command line argument to pass to that interpreter. The operating system then runs the interpreter with the given argument and the full argument list of the executed program. The first argument in the list is the full pathname of the `awk` program. The rest of the argument list will either be options to `awk`, or data files, or both.

Nearly all programming languages have provisions for comments, because programs are typically hard to understand without their extra help.

In the `awk` language, a comment starts with the sharp sign character, '`#`', and continues to the end of the line. The `awk` language ignores the rest of a line following a sharp sign. For example, we could have put the following into '`th-prog`':

```
# This program finds records containing the pattern 'th'.  This is how
# you continue comments on additional lines.
/th/
```

You can put comment lines into keyboard-composed throw-away `awk` programs also, but this usually isn't very useful; the purpose of a comment is to help you or another person understand the program at a later time.

2.6 awk Statements versus Lines

Most often, each line in an `awk` program is a separate statement or separate rule, like this:

```
awk '/12/  { print $0 }
     /21/  { print $0 }' BBS-list inventory-shipped
```

But sometimes statements can be more than one line, and lines can contain several statements. You can split a statement into multiple lines by inserting a newline after any of the following:

```
,    {    ?    :    ||    &&    do    else
```

A newline at any other point is considered the end of the statement. (Splitting lines after '`?`' and '`:`' is a minor `gawk` extension. The '`?`' and '`:`' referred to here is the three operand conditional expression described in Section 8.11 [Conditional Expressions], page 69.)

If you would like to split a single statement into two lines at a point where a newline would terminate it, you can *continue* it by ending the first line with a backslash character, '\'. This is allowed absolutely anywhere in the statement, even in the middle of a string or regular expression. For example:

```
awk '/This program is too long, so continue it\
 on the next line/ { print $1 }'
```

We have generally not used backslash continuation in the sample programs in this manual. Since in `awk` there is no limit on the length of a line, it is never strictly necessary; it just makes programs prettier. We have preferred to make them even more pretty by keeping the statements short. Backslash continuation is most useful when your `awk` program is in a separate source file, instead of typed in on the command line. You should also note that many `awk` implementations are more picky about where you may use backslash continuation. For maximal portability of your `awk` programs, it is best not to split your lines in the middle of a regular expression or a string.

Warning: backslash continuation does not work as described above with the C shell. Continuation with backslash works for `awk` programs in files, and also for one-shot programs *provided* you are using a POSIX-compliant shell, such as the Bourne shell or the Bourne-again shell. But the C shell used on Berkeley Unix behaves differently! There, you must use two backslashes in a row, followed by a newline.

When `awk` statements within one rule are short, you might want to put more than one of them on a line. You do this by separating the statements with a semicolon, '`;`'. This also applies to the rules themselves. Thus, the previous program could have been written:

```
/12/ { print $0 } ; /21/ { print $0 }
```

Note: the requirement that rules on the same line must be separated with a semicolon is a recent change in the `awk` language; it was done for consistency with the treatment of statements within an action.

2.7 When to Use `awk`

You might wonder how `awk` might be useful for you. Using additional utility programs, more advanced patterns, field separators, arithmetic statements, and other selection criteria, you can produce much more complex output. The `awk` language is very useful for producing reports from large amounts of raw data, such as summarizing information from the output of other utility programs like `ls`. (See Section 2.3 [A More Complex Example], page 15.)

Programs written with `awk` are usually much smaller than they would be in other languages. This makes `awk` programs easy to compose and use. Often `awk` programs can be quickly composed at your terminal, used once, and thrown away. Since `awk` programs are interpreted, you can avoid the usually lengthy edit-compile-test-debug cycle of software development.

Complex programs have been written in `awk`, including a complete retargetable assembler for 8-bit microprocessors (see Appendix C [Glossary], page 121, for more information) and a microcode assembler for a special purpose Prolog computer. However, `awk`'s capabilities are strained by tasks of such complexity.

If you find yourself writing `awk` scripts of more than, say, a few hundred lines, you might consider using a different programming language. Emacs Lisp is a good choice if you need sophisticated string or pattern matching capabilities. The shell is also good at string and pattern matching; in addition, it allows powerful use of the system utilities. More conventional languages, such as C, C++, and Lisp, offer better facilities for system programming and for managing the complexity of large programs. Programs in these languages may require more lines of source code than the equivalent `awk` programs, but they are easier to maintain and usually run more efficiently.

3 Reading Input Files

In the typical awk program, all input is read either from the standard input (by default the keyboard, but often a pipe from another command) or from files whose names you specify on the awk command line. If you specify input files, awk reads them in order, reading all the data from one before going on to the next. The name of the current input file can be found in the built-in variable FILENAME (see Chapter 13 [Built-in Variables], page 101).

The input is read in units called records, and processed by the rules one record at a time. By default, each record is one line. Each record is split automatically into fields, to make it more convenient for a rule to work on its parts.

On rare occasions you will need to use the getline command, which can do explicit input from any number of files (see Section 3.7 [Explicit Input with getline], page 30).

3.1 How Input is Split into Records

The awk language divides its input into records and fields. Records are separated by a character called the *record separator*. By default, the record separator is the newline character, defining a record to be a single line of text.

Sometimes you may want to use a different character to separate your records. You can use a different character by changing the built-in variable RS. The value of RS is a string that says how to separate records; the default value is "\n", the string containing just a newline character. This is why records are, by default, single lines.

RS can have any string as its value, but only the first character of the string is used as the record separator. The other characters are ignored. RS is exceptional in this regard; awk uses the full value of all its other built-in variables.

You can change the value of RS in the awk program with the assignment operator, '=' (see Section 8.7 [Assignment Expressions], page 64). The new record-separator character should be enclosed in quotation marks to make a string constant. Often the right time to do this is at the beginning of execution, before any input has been processed, so that the very first record will be read with the proper separator. To do this, use the special BEGIN pattern (see Section 6.7 [BEGIN and END Special Patterns], page 53). For example:

```
awk 'BEGIN { RS = "/" } ; { print $0 }' BBS-list
```

changes the value of RS to "/", before reading any input. This is a string whose first character is a slash; as a result, records are separated by slashes. Then the input file is read, and the second rule in the awk program (the action with no pattern) prints each record. Since each print statement adds a newline at the end of its output, the effect of this awk program is to copy the input with each slash changed to a newline.

Another way to change the record separator is on the command line, using the variable-assignment feature (see Chapter 14 [Invoking awk], page 105).

```
awk '{ print $0 }' RS="/" BBS-list
```

This sets `RS` to '`/`' before processing '`BBS-list`'.

Reaching the end of an input file terminates the current input record, even if the last character in the file is not the character in `RS`.

The empty string, `""` (a string of no characters), has a special meaning as the value of `RS`: it means that records are separated only by blank lines. See Section 3.6 [Multiple-Line Records], page 29, for more details.

The `awk` utility keeps track of the number of records that have been read so far from the current input file. This value is stored in a built-in variable called `FNR`. It is reset to zero when a new file is started. Another built-in variable, `NR`, is the total number of input records read so far from all files. It starts at zero but is never automatically reset to zero.

If you change the value of `RS` in the middle of an `awk` run, the new value is used to delimit subsequent records, but the record currently being processed (and records already processed) are not affected.

3.2 Examining Fields

When `awk` reads an input record, the record is automatically separated or *parsed* by the interpreter into chunks called *fields*. By default, fields are separated by whitespace, like words in a line. Whitespace in `awk` means any string of one or more spaces and/or tabs; other characters such as newline, formfeed, and so on, that are considered whitespace by other languages are *not* considered whitespace by `awk`.

The purpose of fields is to make it more convenient for you to refer to these pieces of the record. You don't have to use them—you can operate on the whole record if you wish—but fields are what make simple `awk` programs so powerful.

To refer to a field in an `awk` program, you use a dollar-sign, '`$`', followed by the number of the field you want. Thus, `$1` refers to the first field, `$2` to the second, and so on. For example, suppose the following is a line of input:

```
This seems like a pretty nice example.
```

Here the first field, or `$1`, is '`This`'; the second field, or `$2`, is '`seems`'; and so on. Note that the last field, `$7`, is '`example.`'. Because there is no space between the '`e`' and the '`.`', the period is considered part of the seventh field.

No matter how many fields there are, the last field in a record can be represented by `$NF`. So, in the example above, `$NF` would be the same as `$7`, which is '`example.`'. Why this works is explained below (see Section 3.3 [Non-constant Field Numbers], page 23). If you try to refer to a field beyond the last one, such as `$8` when the record has only 7 fields, you get the empty string.

Plain `NF`, with no '`$`', is a built-in variable whose value is the number of fields in the current record.

`$0`, which looks like an attempt to refer to the zeroth field, is a special case: it represents the whole input record. This is what you would use if you weren't interested in fields.

Here are some more examples:

```
awk '$1 ~ /foo/ { print $0 }' BBS-list
```

This example prints each record in the file '`BBS-list`' whose first field contains the string '`foo`'. The operator '`~`' is called a *matching operator* (see Section 8.5 [Comparison Expressions], page 62); it tests whether a string (here, the field `$1`) matches a given regular expression.

By contrast, the following example:

```
awk '/foo/ { print $1, $NF }' BBS-list
```

looks for '`foo`' in *the entire record* and prints the first field and the last field for each input record containing a match.

3.3 Non-constant Field Numbers

The number of a field does not need to be a constant. Any expression in the `awk` language can be used after a '`$`' to refer to a field. The value of the expression specifies the field number. If the value is a string, rather than a number, it is converted to a number. Consider this example:

```
awk '{ print $NR }'
```

Recall that `NR` is the number of records read so far: 1 in the first record, 2 in the second, etc. So this example prints the first field of the first record, the second field of the second record, and so on. For the twentieth record, field number 20 is printed; most likely, the record has fewer than 20 fields, so this prints a blank line.

Here is another example of using expressions as field numbers:

```
awk '{ print $(2*2) }' BBS-list
```

The `awk` language must evaluate the expression `(2*2)` and use its value as the number of the field to print. The '`*`' sign represents multiplication, so the expression `2*2` evaluates to 4. The parentheses are used so that the multiplication is done before the '`$`' operation; they are necessary whenever there is a binary operator in the field-number expression. This example, then, prints the hours of operation (the fourth field) for every line of the file '`BBS-list`'.

If the field number you compute is zero, you get the entire record. Thus, `$(2-2)` has the same value as `$0`. Negative field numbers are not allowed.

The number of fields in the current record is stored in the built-in variable `NF` (see Chapter 13 [Built-in Variables], page 101). The expression `$NF` is not a special feature: it is the direct consequence of evaluating `NF` and using its value as a field number.

3.4 Changing the Contents of a Field

You can change the contents of a field as seen by `awk` within an `awk` program; this changes what `awk` perceives as the current input record. (The actual input is untouched: `awk` never modifies the input file.)

Consider this example:

```
awk '{ $3 = $2 - 10; print $2, $3 }' inventory-shipped
```

The '`-`' sign represents subtraction, so this program reassigns field three, `$3`, to be the value of field two minus ten, `$2 - 10`. (See Section 8.3 [Arithmetic Operators], page 60.) Then field two, and the new value for field three, are printed.

In order for this to work, the text in field `$2` must make sense as a number; the string of characters must be converted to a number in order for the computer to do arithmetic on it. The number resulting from the subtraction is converted back to a string of characters which then becomes field three. See Section 8.9 [Conversion of Strings and Numbers], page 67.

When you change the value of a field (as perceived by `awk`), the text of the input record is recalculated to contain the new field where the old one was. Therefore, `$0` changes to reflect the altered field. Thus,

```
awk '{ $2 = $2 - 10; print $0 }' inventory-shipped
```

prints a copy of the input file, with 10 subtracted from the second field of each line.

You can also assign contents to fields that are out of range. For example:

```
awk '{ $6 = ($5 + $4 + $3 + $2) ; print $6 }' inventory-shipped
```

We've just created `$6`, whose value is the sum of fields `$2`, `$3`, `$4`, and `$5`. The '`+`' sign represents addition. For the file '`inventory-shipped`', `$6` represents the total number of parcels shipped for a particular month.

Creating a new field changes the internal `awk` copy of the current input record—the value of `$0`. Thus, if you do '`print $0`' after adding a field, the record printed includes the new field, with the appropriate number of field separators between it and the previously existing fields.

This recomputation affects and is affected by several features not yet discussed, in particular, the *output field separator*, `OFS`, which is used to separate the fields (see Section 4.3 [Output Separators], page 37), and `NF` (the number of fields; see Section 3.2 [Examining Fields], page 22). For example, the value of `NF` is set to the number of the highest field you create.

Note, however, that merely *referencing* an out-of-range field does *not* change the value of either `$0` or `NF`. Referencing an out-of-range field merely produces a null string. For example:

```
if ($(NF+1) != "")
    print "can't happen"
else
```

```
        print "everything is normal"
```

should print '`everything is normal`', because `NF+1` is certain to be out of range. (See Section 9.1 [The `if` Statement], page 73, for more information about `awk`'s `if-else` statements.)

It is important to note that assigning to a field will change the value of `$0`, but will not change the value of `NF`, even when you assign the null string to a field. For example:

```
echo a b c d | awk '{ OFS = ":"; $2 = "" ; print ; print NF }'
```

prints

```
a::c:d
4
```

The field is still there, it just has an empty value. You can tell because there are two colons in a row.

3.5 Specifying how Fields are Separated

(This section is rather long; it describes one of the most fundamental operations in `awk`. If you are a novice with `awk`, we recommend that you re-read this section after you have studied the section on regular expressions, Section 6.2 [Regular Expressions as Patterns], page 47.)

The way `awk` splits an input record into fields is controlled by the *field separator*, which is a single character or a regular expression. `awk` scans the input record for matches for the separator; the fields themselves are the text between the matches. For example, if the field separator is '`oo`', then the following line:

```
moo goo gai pan
```

would be split into three fields: '`m`', '` g`' and '` gai pan`'.

The field separator is represented by the built-in variable `FS`. Shell programmers take note! `awk` does not use the name `IFS` which is used by the shell.

You can change the value of `FS` in the `awk` program with the assignment operator, '`=`' (see Section 8.7 [Assignment Expressions], page 64). Often the right time to do this is at the beginning of execution, before any input has been processed, so that the very first record will be read with the proper separator. To do this, use the special `BEGIN` pattern (see Section 6.7 [`BEGIN` and `END` Special Patterns], page 53). For example, here we set the value of `FS` to the string `","`:

```
awk 'BEGIN { FS = "," } ; { print $2 }'
```

Given the input line,

```
John Q. Smith, 29 Oak St., Walamazoo, MI 42139
```

this `awk` program extracts the string ‘` 29 Oak St.`’.

Sometimes your input data will contain separator characters that don’t separate fields the way you thought they would. For instance, the person’s name in the example we’ve been using might have a title or suffix attached, such as ‘`John Q. Smith, LXIX`’. From input containing such a name:

```
John Q. Smith, LXIX, 29 Oak St., Walamazoo, MI 42139
```

the previous sample program would extract ‘` LXIX`’, instead of ‘` 29 Oak St.`’. If you were expecting the program to print the address, you would be surprised. So choose your data layout and separator characters carefully to prevent such problems.

As you know, by default, fields are separated by whitespace sequences (spaces and tabs), not by single spaces: two spaces in a row do not delimit an empty field. The default value of the field separator is a string `" "` containing a single space. If this value were interpreted in the usual way, each space character would separate fields, so two spaces in a row would make an empty field between them. The reason this does not happen is that a single space as the value of `FS` is a special case: it is taken to specify the default manner of delimiting fields.

If `FS` is any other single character, such as `","`, then each occurrence of that character separates two fields. Two consecutive occurrences delimit an empty field. If the character occurs at the beginning or the end of the line, that too delimits an empty field. The space character is the only single character which does not follow these rules.

More generally, the value of `FS` may be a string containing any regular expression. Then each match in the record for the regular expression separates fields. For example, the assignment:

```
FS = ", \t"
```

makes every area of an input line that consists of a comma followed by a space and a tab, into a field separator. (‘`\t`’ stands for a tab.)

For a less trivial example of a regular expression, suppose you want single spaces to separate fields the way single commas were used above. You can set `FS` to `"[ ]"`. This regular expression matches a single space and nothing else.

`FS` can be set on the command line. You use the ‘`-F`’ argument to do so. For example:

```
awk -F, 'program' input-files
```

sets `FS` to be the ‘`,`’ character. Notice that the argument uses a capital ‘`F`’. Contrast this with ‘`-f`’, which specifies a file containing an `awk` program. Case is significant in command options: the ‘`-F`’ and ‘`-f`’ options have nothing to do with each other. You can use both options at the same time to set the `FS` argument *and* get an `awk` program from a file.

The value used for the argument to ‘`-F`’ is processed in exactly the same way as assignments to the built-in variable `FS`. This means that if the field separator contains special characters, they must be escaped appropriately. For example, to use a ‘\’ as the field separator, you would have to type:

```
# same as FS = "\\"
awk -F\\\\ '...' files ...
```

Since '\' is used for quoting in the shell, awk will see '-F\\'. Then awk processes the '\\' for escape characters (see Section 8.1 [Constant Expressions], page 57), finally yielding a single '\' to be used for the field separator.

As a special case, in compatibility mode (see Chapter 14 [Invoking awk], page 105), if the argument to '-F' is 't', then FS is set to the tab character. (This is because if you type '-F\t', without the quotes, at the shell, the '\' gets deleted, so awk figures that you really want your fields to be separated with tabs, and not 't's. Use '-v FS="t"' on the command line if you really do want to separate your fields with 't's.)

For example, let's use an awk program file called 'baud.awk' that contains the pattern /300/, and the action 'print $1'. Here is the program:

```
/300/   { print $1 }
```

Let's also set FS to be the '-' character, and run the program on the file 'BBS-list'. The following command prints a list of the names of the bulletin boards that operate at 300 baud and the first three digits of their phone numbers:

```
awk -F- -f baud.awk BBS-list
```

It produces this output:

```
aardvark     555
alpo
barfly       555
bites        555
camelot      555
core         555
fooey        555
foot         555
macfoo       555
sdace        555
sabafoo      555
```

Note the second line of output. If you check the original file, you will see that the second line looked like this:

```
alpo-net     555-3412     2400/1200/300     A
```

The '-' as part of the system's name was used as the field separator, instead of the '-' in the phone number that was originally intended. This demonstrates why you have to be careful in choosing your field and record separators.

The following program searches the system password file, and prints the entries for users who have no password:

```
awk -F: '$2 == ""' /etc/passwd
```

Here we use the '`-F`' option on the command line to set the field separator. Note that fields in '`/etc/passwd`' are separated by colons. The second field represents a user's encrypted password, but if the field is empty, that user has no password.

According to the POSIX standard, `awk` is supposed to behave as if each record is split into fields at the time that it is read. In particular, this means that you can change the value of `FS` after a record is read, but before any of the fields are referenced. The value of the fields (i.e. how they were split) should reflect the old value of `FS`, not the new one.

However, many implementations of `awk` do not do this. Instead, they defer splitting the fields until a field reference actually happens, using the *current* value of `FS`! This behavior can be difficult to diagnose. The following example illustrates the results of the two methods. (The `sed` command prints just the first line of '`/etc/passwd`'.)

```
sed 1q /etc/passwd | awk '{ FS = ":" ; print $1 }'
```

will usually print

```
root
```

on an incorrect implementation of `awk`, while `gawk` will print something like

```
root:nSijPlPhZZwgE:0:0:Root:/:
```

There is an important difference between the two cases of '`FS = " "`' (a single blank) and '`FS = "[ \t]+"`' (which is a regular expression matching one or more blanks or tabs). For both values of `FS`, fields are separated by runs of blanks and/or tabs. However, when the value of `FS` is `" "`, `awk` will strip leading and trailing whitespace from the record, and then decide where the fields are.

For example, the following expression prints '`b`':

```
echo ' a b c d ' | awk '{ print $2 }'
```

However, the following prints '`a`':

```
echo ' a b c d ' | awk 'BEGIN { FS = "[ \t]+" } ; { print $2 }'
```

In this case, the first field is null.

The stripping of leading and trailing whitespace also comes into play whenever `$0` is recomputed. For instance, this pipeline

```
echo '   a b c d' | awk '{ print; $2 = $2; print }'
```

produces this output:

```
   a b c d
```

```
     a b c d
```

The first `print` statement prints the record as it was read, with leading whitespace intact. The assignment to `$2` rebuilds `$0` by concatenating `$1` through `$NF` together, separated by the value of `OFS`. Since the leading whitespace was ignored when finding `$1`, it is not part of the new `$0`. Finally, the last `print` statement prints the new `$0`.

The following table summarizes how fields are split, based on the value of `FS`.

`FS == " "` Fields are separated by runs of whitespace. Leading and trailing whitespace are ignored. This is the default.

`FS ==` *any single character*
Fields are separated by each occurrence of the character. Multiple successive occurrences delimit empty fields, as do leading and trailing occurrences.

`FS ==` *regexp*
Fields are separated by occurrences of characters that match *regexp*. Leading and trailing matches of *regexp* delimit empty fields.

3.6 Multiple-Line Records

In some data bases, a single line cannot conveniently hold all the information in one entry. In such cases, you can use multi-line records.

The first step in doing this is to choose your data format: when records are not defined as single lines, how do you want to define them? What should separate records?

One technique is to use an unusual character or string to separate records. For example, you could use the formfeed character (written `\f` in `awk`, as in C) to separate them, making each record a page of the file. To do this, just set the variable `RS` to `"\f"` (a string containing the formfeed character). Any other character could equally well be used, as long as it won't be part of the data in a record.

Another technique is to have blank lines separate records. By a special dispensation, a null string as the value of `RS` indicates that records are separated by one or more blank lines. If you set `RS` to the null string, a record always ends at the first blank line encountered. And the next record doesn't start until the first nonblank line that follows—no matter how many blank lines appear in a row, they are considered one record-separator. (End of file is also considered a record separator.)

The second step is to separate the fields in the record. One way to do this is to put each field on a separate line: to do this, just set the variable `FS` to the string `"\n"`. (This simple regular expression matches a single newline.)

Another way to separate fields is to divide each of the lines into fields in the normal manner. This happens by default as a result of a special feature: when `RS` is set to the null string, the newline character *always* acts as a field separator. This is in addition to whatever field separations result from `FS`.

The original motivation for this special exception was probably so that you get useful behavior in the default case (i.e., `FS == " "`). This feature can be a problem if you really don't want the newline character to separate fields, since there is no way to prevent it. However, you can work around this by using the `split` function to break up the record manually (see Section 11.3 [Built-in Functions for String Manipulation], page 90).

3.7 Explicit Input with `getline`

So far we have been getting our input files from `awk`'s main input stream—either the standard input (usually your terminal) or the files specified on the command line. The `awk` language has a special built-in command called `getline` that can be used to read input under your explicit control.

This command is quite complex and should *not* be used by beginners. It is covered here because this is the chapter on input. The examples that follow the explanation of the `getline` command include material that has not been covered yet. Therefore, come back and study the `getline` command *after* you have reviewed the rest of this manual and have a good knowledge of how `awk` works.

`getline` returns 1 if it finds a record, and 0 if the end of the file is encountered. If there is some error in getting a record, such as a file that cannot be opened, then `getline` returns −1.

In the following examples, *command* stands for a string value that represents a shell command.

`getline` The `getline` command can be used without arguments to read input from the current input file. All it does in this case is read the next input record and split it up into fields. This is useful if you've finished processing the current record, but you want to do some special processing *right now* on the next record. Here's an example:

```
awk '{
     if (t = index($0, "/*")) {
          if (t > 1)
               tmp = substr($0, 1, t - 1)
          else
               tmp = ""
          u = index(substr($0, t + 2), "*/")
          while (u == 0) {
               getline
               t = -1
               u = index($0, "*/")
          }
          if (u <= length($0) - 2)
               $0 = tmp substr($0, t + u + 3)
          else
               $0 = tmp
     }
     print $0
}'
```

This `awk` program deletes all C-style comments, '`/* ... */`', from the input. By replacing the '`print $0`' with other statements, you could perform more complicated processing on the decommented input, like searching for matches of a regular expression. (This program has a subtle problem—can you spot it?)

This form of the `getline` command sets `NF` (the number of fields; see Section 3.2 [Examining Fields], page 22), `NR` (the number of records read so far; see Section 3.1 [How Input is Split into Records], page 21), `FNR` (the number of records read from this input file), and the value of `$0`.

Note: the new value of `$0` is used in testing the patterns of any subsequent rules. The original value of `$0` that triggered the rule which executed `getline` is lost. By contrast, the `next` statement reads a new record but immediately begins processing it normally, starting with the first rule in the program. See Section 9.7 [The `next` Statement], page 78.

`getline` *var*

This form of `getline` reads a record into the variable *var*. This is useful when you want your program to read the next record from the current input file, but you don't want to subject the record to the normal input processing.

For example, suppose the next line is a comment, or a special string, and you want to read it, but you must make certain that it won't trigger any rules. This version of `getline` allows you to read that line and store it in a variable so that the main read-a-line-and-check-each-rule loop of `awk` never sees it.

The following example swaps every two lines of input. For example, given:

```
wan
tew
free
phore
```

it outputs:

```
tew
wan
phore
free
```

Here's the program:

```
awk '{
     if ((getline tmp) > 0) {
          print tmp
          print $0
     } else
          print $0
}'
```

The `getline` function used in this way sets only the variables `NR` and `FNR` (and of course, *var*). The record is not split into fields, so the values of the fields (including `$0`) and the value of `NF` do not change.

`getline <` *file*

This form of the `getline` function takes its input from the file *file*. Here *file* is a string-valued expression that specifies the file name. '`<` *file*' is called a *redirection* since it directs input to come from a different place.

This form is useful if you want to read your input from a particular file, instead of from the main input stream. For example, the following program reads its input record from the file '`foo.input`' when it encounters a first field with a value equal to 10 in the current input file.

```
awk '{
    if ($1 == 10) {
         getline < "foo.input"
```

```
                print
            } else
                print
        }'
```

Since the main input stream is not used, the values of `NR` and `FNR` are not changed. But the record read is split into fields in the normal manner, so the values of `$0` and other fields are changed. So is the value of `NF`.

This does not cause the record to be tested against all the patterns in the `awk` program, in the way that would happen if the record were read normally by the main processing loop of `awk`. However the new record is tested against any subsequent rules, just as when `getline` is used without a redirection.

`getline` *var* `<` *file*

This form of the `getline` function takes its input from the file *file* and puts it in the variable *var*. As above, *file* is a string-valued expression that specifies the file from which to read.

In this version of `getline`, none of the built-in variables are changed, and the record is not split into fields. The only variable changed is *var*.

For example, the following program copies all the input files to the output, except for records that say '`@include` *filename*'. Such a record is replaced by the contents of the file *filename*.

```
awk '{
     if (NF == 2 && $1 == "@include") {
          while ((getline line < $2) > 0)
               print line
          close($2)
     } else
          print
}'
```

Note here how the name of the extra input file is not built into the program; it is taken from the data, from the second field on the '`@include`' line.

The `close` function is called to ensure that if two identical '`@include`' lines appear in the input, the entire specified file is included twice. See Section 3.8 [Closing Input Files and Pipes], page 33.

One deficiency of this program is that it does not process nested '`@include`' statements the way a true macro preprocessor would.

command `| getline`

You can *pipe* the output of a command into `getline`. A pipe is simply a way to link the output of one program to the input of another. In this case, the string *command* is run as a shell command and its output is piped into `awk` to be used as input. This form of `getline` reads one record from the pipe.

For example, the following program copies input to output, except for lines that begin with '`@execute`', which are replaced by the output produced by running the rest of the line as a shell command:

```
awk '{
     if ($1 == "@execute") {
          tmp = substr($0, 10)
          while ((tmp | getline) > 0)
               print
          close(tmp)
     } else
```

```
                    print
          }'
```

The `close` function is called to ensure that if two identical '`@execute`' lines appear in the input, the command is run for each one. See Section 3.8 [Closing Input Files and Pipes], page 33.

Given the input:

```
          foo
          bar
          baz
          @execute who
          bletch
```

the program might produce:

```
          foo
          bar
          baz
          hack      ttyv0   Jul 13 14:22
          hack      ttyp0   Jul 13 14:23     (gnu:0)
          hack      ttyp1   Jul 13 14:23     (gnu:0)
          hack      ttyp2   Jul 13 14:23     (gnu:0)
          hack      ttyp3   Jul 13 14:23     (gnu:0)
          bletch
```

Notice that this program ran the command `who` and printed the result. (If you try this program yourself, you will get different results, showing you who is logged in on your system.)

This variation of `getline` splits the record into fields, sets the value of `NF` and recomputes the value of `$0`. The values of `NR` and `FNR` are not changed.

command `| getline` *var*

The output of the command *command* is sent through a pipe to `getline` and into the variable *var*. For example, the following program reads the current date and time into the variable `current_time`, using the `date` utility, and then prints it.

```
          awk 'BEGIN {
               "date" | getline current_time
               close("date")
               print "Report printed on " current_time
          }'
```

In this version of `getline`, none of the built-in variables are changed, and the record is not split into fields.

3.8 Closing Input Files and Pipes

If the same file name or the same shell command is used with `getline` more than once during the execution of an `awk` program, the file is opened (or the command is executed) only the first time. At that time, the first record of input is read from that file or command. The next time the same file or command is used in `getline`, another record is read from it, and so on.

This implies that if you want to start reading the same file again from the beginning, or if you want to rerun a shell command (rather than reading more output from the command), you must take special steps. What you must do is use the `close` function, as follows:

```
close(filename)
```

or

```
close(command)
```

The argument *filename* or *command* can be any expression. Its value must exactly equal the string that was used to open the file or start the command—for example, if you open a pipe with this:

```
"sort -r names" | getline foo
```

then you must close it with this:

```
close("sort -r names")
```

Once this function call is executed, the next `getline` from that file or command will reopen the file or rerun the command.

`close` returns a value of zero if the close succeeded. Otherwise, the value will be non-zero.

4 Printing Output

One of the most common things that actions do is to output or *print* some or all of the input. For simple output, use the `print` statement. For fancier formatting use the `printf` statement. Both are described in this chapter.

4.1 The `print` Statement

The `print` statement does output with simple, standardized formatting. You specify only the strings or numbers to be printed, in a list separated by commas. They are output, separated by single spaces, followed by a newline. The statement looks like this:

```
print item1, item2, ...
```

The entire list of items may optionally be enclosed in parentheses. The parentheses are necessary if any of the item expressions uses a relational operator; otherwise it could be confused with a redirection (see Section 4.6 [Redirecting Output of `print` and `printf`], page 42). The relational operators are '`==`', '`!=`', '`<`', '`>`', '`>=`', '`<=`', '`~`' and '`!~`' (see Section 8.5 [Comparison Expressions], page 62).

The items printed can be constant strings or numbers, fields of the current record (such as `$1`), variables, or any `awk` expressions. The `print` statement is completely general for computing *what* values to print. With two exceptions, you cannot specify *how* to print them—how many columns, whether to use exponential notation or not, and so on. (See Section 4.3 [Output Separators], page 37, and Section 4.4 [Controlling Numeric Output with `print`], page 37.) For that, you need the `printf` statement (see Section 4.5 [Using `printf` Statements for Fancier Printing], page 38).

The simple statement '`print`' with no items is equivalent to '`print $0`': it prints the entire current record. To print a blank line, use '`print ""`', where `""` is the null, or empty, string.

To print a fixed piece of text, use a string constant such as `"Hello there"` as one item. If you forget to use the double-quote characters, your text will be taken as an `awk` expression, and you will probably get an error. Keep in mind that a space is printed between any two items.

Most often, each `print` statement makes one line of output. But it isn't limited to one line. If an item value is a string that contains a newline, the newline is output along with the rest of the string. A single `print` can make any number of lines this way.

4.2 Examples of `print` Statements

Here is an example of printing a string that contains embedded newlines:

```
awk 'BEGIN { print "line one\nline two\nline three" }'
```

produces output like this:

```
line one
line two
line three
```

Here is an example that prints the first two fields of each input record, with a space between them:

```
awk '{ print $1, $2 }' inventory-shipped
```

Its output looks like this:

```
Jan 13
Feb 15
Mar 15
...
```

A common mistake in using the `print` statement is to omit the comma between two items. This often has the effect of making the items run together in the output, with no space. The reason for this is that juxtaposing two string expressions in `awk` means to concatenate them. For example, without the comma:

```
awk '{ print $1 $2 }' inventory-shipped
```

prints:

```
Jan13
Feb15
Mar15
...
```

Neither example's output makes much sense to someone unfamiliar with the file '`inventory-shipped`'. A heading line at the beginning would make it clearer. Let's add some headings to our table of months (`$1`) and green crates shipped (`$2`). We do this using the `BEGIN` pattern (see Section 6.7 [`BEGIN` and `END` Special Patterns], page 53) to force the headings to be printed only once:

```
awk 'BEGIN {  print "Month Crates"
              print "----- ------" }
           {  print $1, $2 }' inventory-shipped
```

Did you already guess what happens? This program prints the following:

```
Month Crates
----- ------
Jan 13
Feb 15
Mar 15
...
```

The headings and the table data don't line up! We can fix this by printing some spaces between the two fields:

```
awk 'BEGIN { print "Month Crates"
             print "----- ------" }
           { print $1, "     ", $2 }' inventory-shipped
```

You can imagine that this way of lining up columns can get pretty complicated when you have many columns to fix. Counting spaces for two or three columns can be simple, but more than this and you can get "lost" quite easily. This is why the `printf` statement was created (see Section 4.5 [Using `printf` Statements for Fancier Printing], page 38); one of its specialties is lining up columns of data.

4.3 Output Separators

As mentioned previously, a `print` statement contains a list of items, separated by commas. In the output, the items are normally separated by single spaces. But they do not have to be spaces; a single space is only the default. You can specify any string of characters to use as the *output field separator* by setting the built-in variable `OFS`. The initial value of this variable is the string `" "`, that is, just a single space.

The output from an entire `print` statement is called an *output record*. Each `print` statement outputs one output record and then outputs a string called the *output record separator*. The built-in variable `ORS` specifies this string. The initial value of the variable is the string `"\n"` containing a newline character; thus, normally each `print` statement makes a separate line.

You can change how output fields and records are separated by assigning new values to the variables `OFS` and/or `ORS`. The usual place to do this is in the `BEGIN` rule (see Section 6.7 [`BEGIN` and `END` Special Patterns], page 53), so that it happens before any input is processed. You may also do this with assignments on the command line, before the names of your input files.

The following example prints the first and second fields of each input record separated by a semicolon, with a blank line added after each line:

```
awk 'BEGIN { OFS = ";"; ORS = "\n\n" }
           { print $1, $2 }'  BBS-list
```

If the value of `ORS` does not contain a newline, all your output will be run together on a single line, unless you output newlines some other way.

4.4 Controlling Numeric Output with `print`

When you use the `print` statement to print numeric values, `awk` internally converts the number to a string of characters, and prints that string. `awk` uses the `sprintf` function to do this conversion. For now, it suffices to say that the `sprintf` function accepts a *format specification* that tells it how to format numbers (or strings), and that there are a number of different ways that numbers can be formatted. The different format specifications are discussed more fully in Section 4.5 [Using `printf` Statements for Fancier Printing], page 38.

The built-in variable `OFMT` contains the default format specification that `print` uses with `sprintf` when it wants to convert a number to a string for printing. By supplying different format specifications as the value of `OFMT`, you can change how `print` will print your numbers. As a brief example:

```
awk 'BEGIN { OFMT = "%d"  # print numbers as integers
             print 17.23 }'
```

will print '`17`'.

4.5 Using `printf` Statements for Fancier Printing

If you want more precise control over the output format than `print` gives you, use `printf`. With `printf` you can specify the width to use for each item, and you can specify various stylistic choices for numbers (such as what radix to use, whether to print an exponent, whether to print a sign, and how many digits to print after the decimal point). You do this by specifying a string, called the *format string*, which controls how and where to print the other arguments.

4.5.1 Introduction to the `printf` Statement

The `printf` statement looks like this:

`printf` *format*`,` *item1*`,` *item2*`,` ...

The entire list of arguments may optionally be enclosed in parentheses. The parentheses are necessary if any of the item expressions uses a relational operator; otherwise it could be confused with a redirection (see Section 4.6 [Redirecting Output of `print` and `printf`], page 42). The relational operators are '`==`', '`!=`', '`<`', '`>`', '`>=`', '`<=`', '`~`' and '`!~`' (see Section 8.5 [Comparison Expressions], page 62).

The difference between `printf` and `print` is the argument *format*. This is an expression whose value is taken as a string; it specifies how to output each of the other arguments. It is called the *format string*.

The format string is the same as in the ANSI C library function `printf`. Most of *format* is text to be output verbatim. Scattered among this text are *format specifiers*, one per item. Each format specifier says to output the next item at that place in the format.

The `printf` statement does not automatically append a newline to its output. It outputs only what the format specifies. So if you want a newline, you must include one in the format. The output separator variables `OFS` and `ORS` have no effect on `printf` statements.

4.5.2 Format-Control Letters

A format specifier starts with the character '`%`' and ends with a *format-control letter*; it tells the `printf` statement how to output one item. (If you actually want to output a '`%`', write '`%%`'.) The format-control letter specifies what kind of value to print. The rest of the format specifier is made up of optional *modifiers* which are parameters such as the field width to use.

Here is a list of the format-control letters:

'`c`'
: This prints a number as an ASCII character. Thus, '`printf "%c", 65`' outputs the letter '`A`'. The output for a string value is the first character of the string.

'`d`'
: This prints a decimal integer.

'`i`'
: This also prints a decimal integer.

'`e`'
: This prints a number in scientific (exponential) notation. For example,

```
printf "%4.3e", 1950
```

prints '`1.950e+03`', with a total of four significant figures of which three follow the decimal point. The '`4.3`' are *modifiers*, discussed below.

'`f`'
: This prints a number in floating point notation.

'`g`'
: This prints a number in either scientific notation or floating point notation, whichever uses fewer characters.

'`o`'
: This prints an unsigned octal integer.

'`s`'
: This prints a string.

'`x`'
: This prints an unsigned hexadecimal integer.

'`X`'
: This prints an unsigned hexadecimal integer. However, for the values 10 through 15, it uses the letters '`A`' through '`F`' instead of '`a`' through '`f`'.

'`%`'
: This isn't really a format-control letter, but it does have a meaning when used after a '`%`': the sequence '`%%`' outputs one '`%`'. It does not consume an argument.

4.5.3 Modifiers for `printf` Formats

A format specification can also include *modifiers* that can control how much of the item's value is printed and how much space it gets. The modifiers come between the '`%`' and the format-control letter. Here are the possible modifiers, in the order in which they may appear:

'`-`'
: The minus sign, used before the width modifier, says to left-justify the argument within its specified width. Normally the argument is printed right-justified in the specified width. Thus,

```
printf "%-4s", "foo"
```

prints '`foo `'.

'*width*'
: This is a number representing the desired width of a field. Inserting any number between the '`%`' sign and the format control character forces the field to be expanded to this width. The default way to do this is to pad with spaces on the left. For example,

```
printf "%4s", "foo"
```

prints '` foo`'.

The value of *width* is a minimum width, not a maximum. If the item value requires more than *width* characters, it can be as wide as necessary. Thus,

```
printf "%4s", "foobar"
```

prints '`foobar`'.

Preceding the *width* with a minus sign causes the output to be padded with spaces on the right, instead of on the left.

'*.prec*' This is a number that specifies the precision to use when printing. This specifies the number of digits you want printed to the right of the decimal point. For a string, it specifies the maximum number of characters from the string that should be printed.

The C library `printf`'s dynamic *width* and *prec* capability (for example, `"%*.*s"`) is supported. Instead of supplying explicit *width* and/or *prec* values in the format string, you pass them in the argument list. For example:

```
w = 5
p = 3
s = "abcdefg"
printf "<%*.*s>\n", w, p, s
```

is exactly equivalent to

```
s = "abcdefg"
printf "<%5.3s>\n", s
```

Both programs output '`<••abc>`'. (We have used the bullet symbol "•" to represent a space, to clearly show you that there are two spaces in the output.)

Earlier versions of `awk` did not support this capability. You may simulate it by using concatenation to build up the format string, like so:

```
w = 5
p = 3
s = "abcdefg"
printf "<%" w "." p "s>\n", s
```

This is not particularly easy to read, however.

4.5.4 Examples of Using `printf`

Here is how to use `printf` to make an aligned table:

```
awk '{ printf "%-10s %s\n", $1, $2 }' BBS-list
```

prints the names of bulletin boards (`$1`) of the file '`BBS-list`' as a string of 10 characters, left justified. It also prints the phone numbers (`$2`) afterward on the line. This produces an aligned two-column table of names and phone numbers:

```
aardvark   555-5553
alpo-net   555-3412
barfly     555-7685
bites      555-1675
camelot    555-0542
core       555-2912
fooey      555-1234
foot       555-6699
macfoo     555-6480
sdace      555-3430
sabafoo    555-2127
```

Did you notice that we did not specify that the phone numbers be printed as numbers? They had to be printed as strings because the numbers are separated by a dash. This dash would be interpreted as a minus sign if we had tried to print the phone numbers as numbers. This would have led to some pretty confusing results.

We did not specify a width for the phone numbers because they are the last things on their lines. We don't need to put spaces after them.

We could make our table look even nicer by adding headings to the tops of the columns. To do this, use the `BEGIN` pattern (see Section 6.7 [`BEGIN` and `END` Special Patterns], page 53) to force the header to be printed only once, at the beginning of the `awk` program:

```
awk 'BEGIN { print "Name      Number"
             print "----      ------" }
     { printf "%-10s %s\n", $1, $2 }' BBS-list
```

Did you notice that we mixed `print` and `printf` statements in the above example? We could have used just `printf` statements to get the same results:

```
awk 'BEGIN { printf "%-10s %s\n", "Name", "Number"
             printf "%-10s %s\n", "----", "------" }
     { printf "%-10s %s\n", $1, $2 }' BBS-list
```

By outputting each column heading with the same format specification used for the elements of the column, we have made sure that the headings are aligned just like the columns.

The fact that the same format specification is used three times can be emphasized by storing it in a variable, like this:

```
awk 'BEGIN { format = "%-10s %s\n"
             printf format, "Name", "Number"
             printf format, "----", "------" }
     { printf format, $1, $2 }' BBS-list
```

See if you can use the `printf` statement to line up the headings and table data for our '`inventory-shipped`' example covered earlier in the section on the `print` statement (see Section 4.1 [The `print` Statement], page 35).

4.6 Redirecting Output of `print` and `printf`

So far we have been dealing only with output that prints to the standard output, usually your terminal. Both `print` and `printf` can also send their output to other places. This is called *redirection*.

A redirection appears after the `print` or `printf` statement. Redirections in `awk` are written just like redirections in shell commands, except that they are written inside the `awk` program.

4.6.1 Redirecting Output to Files and Pipes

Here are the three forms of output redirection. They are all shown for the `print` statement, but they work identically for `printf` also.

`print` *items* `>` *output-file*

This type of redirection prints the items onto the output file *output-file*. The file name *output-file* can be any expression. Its value is changed to a string and then used as a file name (see Chapter 8 [Expressions as Action Statements], page 57).

When this type of redirection is used, the *output-file* is erased before the first output is written to it. Subsequent writes do not erase *output-file*, but append to it. If *output-file* does not exist, then it is created.

For example, here is how one `awk` program can write a list of BBS names to a file '`name-list`' and a list of phone numbers to a file '`phone-list`'. Each output file contains one name or number per line.

```
awk '{ print $2 > "phone-list"
       print $1 > "name-list" }' BBS-list
```

`print` *items* `>>` *output-file*

This type of redirection prints the items onto the output file *output-file*. The difference between this and the single-'`>`' redirection is that the old contents (if any) of *output-file* are not erased. Instead, the `awk` output is appended to the file.

`print` *items* `|` *command*

It is also possible to send output through a *pipe* instead of into a file. This type of redirection opens a pipe to *command* and writes the values of *items* through this pipe, to another process created to execute *command*.

The redirection argument *command* is actually an `awk` expression. Its value is converted to a string, whose contents give the shell command to be run.

For example, this produces two files, one unsorted list of BBS names and one list sorted in reverse alphabetical order:

```
awk '{ print $1 > "names.unsorted"
       print $1 | "sort -r > names.sorted" }' BBS-list
```

Here the unsorted list is written with an ordinary redirection while the sorted list is written by piping through the `sort` utility.

Here is an example that uses redirection to mail a message to a mailing list '`bug-system`'. This might be useful when trouble is encountered in an `awk` script run periodically for system maintenance.

```
report = "mail bug-system"
print "Awk script failed:", $0 | report
print "at record number", FNR, "of", FILENAME  | report
```

```
          close(report)
```

We call the `close` function here because it's a good idea to close the pipe as soon as all the intended output has been sent to it. See Section 4.6.2 [Closing Output Files and Pipes], page 43, for more information on this. This example also illustrates the use of a variable to represent a *file* or *command*: it is not necessary to always use a string constant. Using a variable is generally a good idea, since `awk` requires you to spell the string value identically every time.

Redirecting output using '`>`', '`>>`', or '`|`' asks the system to open a file or pipe only if the particular *file* or *command* you've specified has not already been written to by your program, or if it has been closed since it was last written to.

4.6.2 Closing Output Files and Pipes

When a file or pipe is opened, the file name or command associated with it is remembered by `awk` and subsequent writes to the same file or command are appended to the previous writes. The file or pipe stays open until `awk` exits. This is usually convenient.

Sometimes there is a reason to close an output file or pipe earlier than that. To do this, use the `close` function, as follows:

```
close(filename)
```

or

```
close(command)
```

The argument *filename* or *command* can be any expression. Its value must exactly equal the string used to open the file or pipe to begin with—for example, if you open a pipe with this:

```
print $1 | "sort -r > names.sorted"
```

then you must close it with this:

```
close("sort -r > names.sorted")
```

Here are some reasons why you might need to close an output file:

- To write a file and read it back later on in the same `awk` program. Close the file when you are finished writing it; then you can start reading it with `getline` (see Section 3.7 [Explicit Input with `getline`], page 30).
- To write numerous files, successively, in the same `awk` program. If you don't close the files, eventually you may exceed a system limit on the number of open files in one process. So close each one when you are finished writing it.
- To make a command finish. When you redirect output through a pipe, the command reading the pipe normally continues to try to read input as long as the pipe is open. Often this means the command cannot really do its work until the pipe is closed. For example, if you redirect output to the `mail` program, the message is not actually sent until the pipe is closed.

- To run the same program a second time, with the same arguments. This is not the same thing as giving more input to the first run!

  For example, suppose you pipe output to the `mail` program. If you output several lines redirected to this pipe without closing it, they make a single message of several lines. By contrast, if you close the pipe after each line of output, then each line makes a separate message.

`close` returns a value of zero if the close succeeded. Otherwise, the value will be non-zero.

4.7 Standard I/O Streams

Running programs conventionally have three input and output streams already available to them for reading and writing. These are known as the *standard input*, *standard output*, and *standard error output*. These streams are, by default, terminal input and output, but they are often redirected with the shell, via the '`<`', '`<<`', '`>`', '`>>`', '`>&`' and '`|`' operators. Standard error is used only for writing error messages; the reason we have two separate streams, standard output and standard error, is so that they can be redirected separately.

In other implementations of `awk`, the only way to write an error message to standard error in an `awk` program is as follows:

```
print "Serious error detected!\n" | "cat 1>&2"
```

This works by opening a pipeline to a shell command which can access the standard error stream which it inherits from the `awk` process. This is far from elegant, and is also inefficient, since it requires a separate process. So people writing `awk` programs have often neglected to do this. Instead, they have sent the error messages to the terminal, like this:

```
NF != 4 {
   printf("line %d skipped: doesn't have 4 fields\n", FNR) > "/dev/tty"
}
```

This has the same effect most of the time, but not always: although the standard error stream is usually the terminal, it can be redirected, and when that happens, writing to the terminal is not correct. In fact, if `awk` is run from a background job, it may not have a terminal at all. Then opening '`/dev/tty`' will fail.

5 Useful “One-liners”

Useful `awk` programs are often short, just a line or two. Here is a collection of useful, short programs to get you started. Some of these programs contain constructs that haven’t been covered yet. The description of the program will give you a good idea of what is going on, but please read the rest of the manual to become an `awk` expert!

```
awk '{ if (NF > max) max = NF }
     END { print max }'
```
This program prints the maximum number of fields on any input line.

```
awk 'length($0) > 80'
```
This program prints every line longer than 80 characters. The sole rule has a relational expression as its pattern, and has no action (so the default action, printing the record, is used).

```
awk 'NF > 0'
```
This program prints every line that has at least one field. This is an easy way to delete blank lines from a file (or rather, to create a new file similar to the old file but from which the blank lines have been deleted).

```
awk '{ if (NF > 0) print }'
```
This program also prints every line that has at least one field. Here we allow the rule to match every line, then decide in the action whether to print.

```
awk 'BEGIN { for (i = 1; i <= 7; i++)
                 print int(101 * rand()) }'
```
This program prints 7 random numbers from 0 to 100, inclusive.

`ls -l` *files* `| awk '{ x += $4 } ; END { print "total bytes: " x }'`

This program prints the total number of bytes used by *files*.

`expand` *file* `| awk '{ if (x < length()) x = length() }`
`                    END { print "maximum line length is " x }'`

This program prints the maximum line length of *file*. The input is piped through the `expand` program to change tabs into spaces, so the widths compared are actually the right-margin columns.

```
awk 'BEGIN { FS = ":" }
     { print $1 | "sort" }' /etc/passwd
```
This program prints a sorted list of the login names of all users.

```
awk '{ nlines++ }
     END { print nlines }'
```
This programs counts lines in a file.

```
awk 'END { print NR }'
```
This program also counts lines in a file, but lets `awk` do the work.

```
awk '{ print NR, $0 }'
```
This program adds line numbers to all its input files, similar to ‘`cat -n`’.

6 Patterns

Patterns in `awk` control the execution of rules: a rule is executed when its pattern matches the current input record. This chapter tells all about how to write patterns.

6.1 Kinds of Patterns

Here is a summary of the types of patterns supported in `awk`.

`/`*regular expression*`/`
: A regular expression as a pattern. It matches when the text of the input record fits the regular expression. (See Section 6.2 [Regular Expressions as Patterns], page 47.)

expression
: A single expression. It matches when its value, converted to a number, is nonzero (if a number) or nonnull (if a string). (See Section 6.5 [Expressions as Patterns], page 52.)

pat1`,` *pat2*
: A pair of patterns separated by a comma, specifying a range of records. (See Section 6.6 [Specifying Record Ranges with Patterns], page 53.)

`BEGIN`
`END`
: Special patterns to supply start-up or clean-up information to `awk`. (See Section 6.7 [`BEGIN` and `END` Special Patterns], page 53.)

null
: The empty pattern matches every input record. (See Section 6.8 [The Empty Pattern], page 54.)

6.2 Regular Expressions as Patterns

A *regular expression*, or *regexp*, is a way of describing a class of strings. A regular expression enclosed in slashes (‘`/`’) is an `awk` pattern that matches every input record whose text belongs to that class.

The simplest regular expression is a sequence of letters, numbers, or both. Such a regexp matches any string that contains that sequence. Thus, the regexp ‘`foo`’ matches any string containing ‘`foo`’. Therefore, the pattern `/foo/` matches any input record containing ‘`foo`’. Other kinds of regexps let you specify more complicated classes of strings.

6.2.1 How to Use Regular Expressions

A regular expression can be used as a pattern by enclosing it in slashes. Then the regular expression is matched against the entire text of each record. (Normally, it only needs to match some part of the text in order to succeed.) For example, this prints the second field of each record that contains ‘`foo`’ anywhere:

```
    awk '/foo/ { print $2 }' BBS-list
```

Regular expressions can also be used in comparison expressions. Then you can specify the string to match against; it need not be the entire current input record. These comparison expressions can be used as patterns or in `if`, `while`, `for`, and `do` statements.

exp `~ /`*regexp*`/`
: This is true if the expression *exp* (taken as a character string) is matched by *regexp*. The following example matches, or selects, all input records with the upper-case letter '`J`' somewhere in the first field:

```
awk '$1 ~ /J/' inventory-shipped
```

So does this:

```
awk '{ if ($1 ~ /J/) print }' inventory-shipped
```

exp `!~ /`*regexp*`/`
: This is true if the expression *exp* (taken as a character string) is *not* matched by *regexp*. The following example matches, or selects, all input records whose first field *does not* contain the upper-case letter '`J`':

```
awk '$1 !~ /J/' inventory-shipped
```

The right hand side of a '`~`' or '`!~`' operator need not be a constant regexp (i.e., a string of characters between slashes). It may be any expression. The expression is evaluated, and converted if necessary to a string; the contents of the string are used as the regexp. A regexp that is computed in this way is called a *dynamic regexp*. For example:

```
identifier_regexp = "[A-Za-z_][A-Za-z_0-9]+"
$0 ~ identifier_regexp
```

sets `identifier_regexp` to a regexp that describes `awk` variable names, and tests if the input record matches this regexp.

6.2.2 Regular Expression Operators

You can combine regular expressions with the following characters, called *regular expression operators*, or *metacharacters*, to increase the power and versatility of regular expressions.

Here is a table of metacharacters. All characters not listed in the table stand for themselves.

`^`
: This matches the beginning of the string or the beginning of a line within the string. For example:

```
^@chapter
```

matches the '`@chapter`' at the beginning of a string, and can be used to identify chapter beginnings in Texinfo source files.

`$`
: This is similar to '`^`', but it matches only at the end of a string or the end of a line within the string. For example:

```
p$
```

matches a record that ends with a '`p`'.

`.`
: This matches any single character except a newline. For example:

```
.P
```

matches any single character followed by a '`P`' in a string. Using concatenation we can make regular expressions like '`U.A`', which matches any three-character sequence that begins with '`U`' and ends with '`A`'.

`[...]` This is called a *character set*. It matches any one of the characters that are enclosed in the square brackets. For example:

```
[MVX]
```

matches any one of the characters '`M`', '`V`', or '`X`' in a string.

Ranges of characters are indicated by using a hyphen between the beginning and ending characters, and enclosing the whole thing in brackets. For example:

```
[0-9]
```

matches any digit.

To include the character '`\`', '`]`', '`-`' or '`^`' in a character set, put a '`\`' in front of it. For example:

```
[d\]]
```

matches either '`d`', or '`]`'.

This treatment of '`\`' is compatible with other `awk` implementations, and is also mandated by the POSIX Command Language and Utilities standard. The regular expressions in `awk` are a superset of the POSIX specification for Extended Regular Expressions (EREs). POSIX EREs are based on the regular expressions accepted by the traditional `egrep` utility.

In `egrep` syntax, backslash is not syntactically special within square brackets. This means that special tricks have to be used to represent the characters '`]`', '`-`' and '`^`' as members of a character set.

In `egrep` syntax, to match '`-`', write it as '`---`', which is a range containing only '`-`'. You may also give '`-`' as the first or last character in the set. To match '`^`', put it anywhere except as the first character of a set. To match a '`]`', make it the first character in the set. For example:

```
[]d^]
```

matches either '`]`', '`d`' or '`^`'.

`[^ ...]` This is a *complemented character set*. The first character after the '`[`' *must* be a '`^`'. It matches any characters *except* those in the square brackets (or newline). For example:

```
[^0-9]
```

matches any character that is not a digit.

`|` This is the *alternation operator* and it is used to specify alternatives. For example:

```
^P|[0-9]
```

matches any string that matches either '`^P`' or '`[0-9]`'. This means it matches any string that contains a digit or starts with '`P`'.

The alternation applies to the largest possible regexps on either side.

`(...)` Parentheses are used for grouping in regular expressions as in arithmetic. They can be used to concatenate regular expressions containing the alternation operator, '`|`'.

`*` This symbol means that the preceding regular expression is to be repeated as many times as possible to find a match. For example:

```
ph*
```

applies the '`*`' symbol to the preceding '`h`' and looks for matches to one '`p`' followed by any number of '`h`'s. This will also match just '`p`' if no '`h`'s are present.

The '`*`' repeats the *smallest* possible preceding expression. (Use parentheses if you wish to repeat a larger expression.) It finds as many repetitions as possible. For example:

```
awk '/\(c[ad][ad]*r x\)/ { print }' sample
```

prints every record in the input containing a string of the form '`(car x)`', '`(cdr x)`', '`(cadr x)`', and so on.

`+` This symbol is similar to '`*`', but the preceding expression must be matched at least once. This means that:

```
wh+y
```

would match '`why`' and '`whhy`' but not '`wy`', whereas '`wh*y`' would match all three of these strings. This is a simpler way of writing the last '`*`' example:

```
awk '/\(c[ad]+r x\)/ { print }' sample
```

`?` This symbol is similar to '`*`', but the preceding expression can be matched once or not at all. For example:

```
fe?d
```

will match '`fed`' and '`fd`', but nothing else.

`\` This is used to suppress the special meaning of a character when matching. For example:

```
\$
```

matches the character '`$`'.

The escape sequences used for string constants (see Section 8.1 [Constant Expressions], page 57) are valid in regular expressions as well; they are also introduced by a '`\`'.

In regular expressions, the '`*`', '`+`', and '`?`' operators have the highest precedence, followed by concatenation, and finally by '`|`'. As in arithmetic, parentheses can change how operators are grouped.

6.2.3 Case-sensitivity in Matching

Case is normally significant in regular expressions, both when matching ordinary characters (i.e., not metacharacters), and inside character sets. Thus a '`w`' in a regular expression matches only a lower case '`w`' and not an upper case '`W`'.

The simplest way to do a case-independent match is to use a character set: '`[Ww]`'. However, this can be cumbersome if you need to use it often; and it can make the regular expressions harder for humans to read. There are two other alternatives that you might prefer.

One way to do a case-insensitive match at a particular point in the program is to convert the data to a single case, using the `tolower` or `toupper` built-in string functions (which we haven't discussed yet; see Section 11.3 [Built-in Functions for String Manipulation], page 90). For example:

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

converts the first field to lower case before matching against it.

```
x = "aB"
if (x ~ /ab/) ...   # this test will fail
```

6.3 Comparison Expressions as Patterns

Comparison patterns test relationships such as equality between two strings or numbers. They are a special case of expression patterns (see Section 6.5 [Expressions as Patterns], page 52). They are written with *relational operators*, which are a superset of those in C. Here is a table of them:

x `<` *y*	True if *x* is less than *y*.
x `<=` *y*	True if *x* is less than or equal to *y*.
x `>` *y*	True if *x* is greater than *y*.
x `>=` *y*	True if *x* is greater than or equal to *y*.
x `==` *y*	True if *x* is equal to *y*.
x `!=` *y*	True if *x* is not equal to *y*.
x `~` *y*	True if *x* matches the regular expression described by *y*.
x `!~` *y*	True if *x* does not match the regular expression described by *y*.

The operands of a relational operator are compared as numbers if they are both numbers. Otherwise they are converted to, and compared as, strings (see Section 8.9 [Conversion of Strings and Numbers], page 67, for the detailed rules). Strings are compared by comparing the first character of each, then the second character of each, and so on, until there is a difference. If the two strings are equal until the shorter one runs out, the shorter one is considered to be less than the longer one. Thus, `"10"` is less than `"9"`, and `"abc"` is less than `"abcd"`.

The left operand of the '`~`' and '`!~`' operators is a string. The right operand is either a constant regular expression enclosed in slashes (`/`*regexp*`/`), or any expression, whose string value is used as a dynamic regular expression (see Section 6.2.1 [How to Use Regular Expressions], page 47).

The following example prints the second field of each input record whose first field is precisely '`foo`'.

```
awk '$1 == "foo" { print $2 }' BBS-list
```

Contrast this with the following regular expression match, which would accept any record with a first field that contains '`foo`':

```
awk '$1 ~ "foo" { print $2 }' BBS-list
```

or, equivalently, this one:

```
awk '$1 ~ /foo/ { print $2 }' BBS-list
```

6.4 Boolean Operators and Patterns

A *boolean pattern* is an expression which combines other patterns using the *boolean operators* "or" ('`||`'), "and" ('`&&`'), and "not" ('`!`'). Whether the boolean pattern matches an input record depends on whether its subpatterns match.

For example, the following command prints all records in the input file 'BBS-list' that contain both '2400' and 'foo'.

```
awk '/2400/ && /foo/' BBS-list
```

The following command prints all records in the input file 'BBS-list' that contain *either* '2400' or 'foo', or both.

```
awk '/2400/ || /foo/' BBS-list
```

The following command prints all records in the input file 'BBS-list' that do *not* contain the string 'foo'.

```
awk '! /foo/' BBS-list
```

Note that boolean patterns are a special case of expression patterns (see Section 6.5 [Expressions as Patterns], page 52); they are expressions that use the boolean operators. See Section 8.6 [Boolean Expressions], page 64, for complete information on the boolean operators.

The subpatterns of a boolean pattern can be constant regular expressions, comparisons, or any other `awk` expressions. Range patterns are not expressions, so they cannot appear inside boolean patterns. Likewise, the special patterns `BEGIN` and `END`, which never match any input record, are not expressions and cannot appear inside boolean patterns.

6.5 Expressions as Patterns

Any `awk` expression is also valid as an `awk` pattern. Then the pattern "matches" if the expression's value is nonzero (if a number) or nonnull (if a string).

The expression is reevaluated each time the rule is tested against a new input record. If the expression uses fields such as `$1`, the value depends directly on the new input record's text; otherwise, it depends only on what has happened so far in the execution of the `awk` program, but that may still be useful.

Comparison patterns are actually a special case of this. For example, the expression `$5 == "foo"` has the value 1 when the value of `$5` equals `"foo"`, and 0 otherwise; therefore, this expression as a pattern matches when the two values are equal.

Boolean patterns are also special cases of expression patterns.

A constant regexp as a pattern is also a special case of an expression pattern. `/foo/` as an expression has the value 1 if 'foo' appears in the current input record; thus, as a pattern, `/foo/` matches any record containing 'foo'.

Other implementations of `awk` that are not yet POSIX compliant are less general than `gawk`: they allow comparison expressions, and boolean combinations thereof (optionally with parentheses), but not necessarily other kinds of expressions.

6.6 Specifying Record Ranges with Patterns

A *range pattern* is made of two patterns separated by a comma, of the form *begpat*, *endpat*. It matches ranges of consecutive input records. The first pattern *begpat* controls where the range begins, and the second one *endpat* controls where it ends. For example,

```
awk '$1 == "on", $1 == "off"'
```

prints every record between '`on`'/'`off`' pairs, inclusive.

A range pattern starts out by matching *begpat* against every input record; when a record matches *begpat*, the range pattern becomes *turned on*. The range pattern matches this record. As long as it stays turned on, it automatically matches every input record read. It also matches *endpat* against every input record; when that succeeds, the range pattern is turned off again for the following record. Now it goes back to checking *begpat* against each record.

The record that turns on the range pattern and the one that turns it off both match the range pattern. If you don't want to operate on these records, you can write `if` statements in the rule's action to distinguish them.

It is possible for a pattern to be turned both on and off by the same record, if both conditions are satisfied by that record. Then the action is executed for just that record.

6.7 `BEGIN` and `END` Special Patterns

`BEGIN` and `END` are special patterns. They are not used to match input records. Rather, they are used for supplying start-up or clean-up information to your `awk` script. A `BEGIN` rule is executed, once, before the first input record has been read. An `END` rule is executed, once, after all the input has been read. For example:

```
awk 'BEGIN { print "Analysis of `foo'" }
     /foo/ { ++foobar }
     END   { print "`foo' appears " foobar " times." }' BBS-list
```

This program finds the number of records in the input file '`BBS-list`' that contain the string '`foo`'. The `BEGIN` rule prints a title for the report. There is no need to use the `BEGIN` rule to initialize the counter `foobar` to zero, as `awk` does this for us automatically (see Section 8.2 [Variables], page 59).

The second rule increments the variable `foobar` every time a record containing the pattern '`foo`' is read. The `END` rule prints the value of `foobar` at the end of the run.

The special patterns `BEGIN` and `END` cannot be used in ranges or with boolean operators (indeed, they cannot be used with any operators).

An `awk` program may have multiple `BEGIN` and/or `END` rules. They are executed in the order they appear, all the `BEGIN` rules at start-up and all the `END` rules at termination.

Multiple BEGIN and END sections are useful for writing library functions, since each library can have its own BEGIN or END rule to do its own initialization and/or cleanup. Note that the order in which library functions are named on the command line controls the order in which their BEGIN and END rules are executed. Therefore you have to be careful to write such rules in library files so that the order in which they are executed doesn't matter. See Chapter 14 [Invoking awk], page 105, for more information on using library functions.

If an awk program only has a BEGIN rule, and no other rules, then the program exits after the BEGIN rule has been run. (Older versions of awk used to keep reading and ignoring input until end of file was seen.) However, if an END rule exists as well, then the input will be read, even if there are no other rules in the program. This is necessary in case the END rule checks the NR variable.

BEGIN and END rules must have actions; there is no default action for these rules since there is no current record when they run.

6.8 The Empty Pattern

An empty pattern is considered to match *every* input record. For example, the program:

```
awk '{ print $1 }' BBS-list
```

prints the first field of every record.

7 Overview of Actions

An `awk` program or script consists of a series of rules and function definitions, interspersed. (Functions are described later. See Chapter 12 [User-defined Functions], page 95.)

A rule contains a pattern and an action, either of which may be omitted. The purpose of the *action* is to tell `awk` what to do once a match for the pattern is found. Thus, the entire program looks somewhat like this:

```
[pattern] [{ action }]
[pattern] [{ action }]
...
function name (args) { ... }
...
```

An action consists of one or more `awk` *statements*, enclosed in curly braces ('`{`' and '`}`'). Each statement specifies one thing to be done. The statements are separated by newlines or semicolons.

The curly braces around an action must be used even if the action contains only one statement, or even if it contains no statements at all. However, if you omit the action entirely, omit the curly braces as well. (An omitted action is equivalent to '`{ print $0 }`'.)

Here are the kinds of statements supported in `awk`:

- Expressions, which can call functions or assign values to variables (see Chapter 8 [Expressions as Action Statements], page 57). Executing this kind of statement simply computes the value of the expression and then ignores it. This is useful when the expression has side effects (see Section 8.7 [Assignment Expressions], page 64).
- Control statements, which specify the control flow of `awk` programs. The `awk` language gives you C-like constructs (`if`, `for`, `while`, and so on) as well as a few special ones (see Chapter 9 [Control Statements in Actions], page 73).
- Compound statements, which consist of one or more statements enclosed in curly braces. A compound statement is used in order to put several statements together in the body of an `if`, `while`, `do` or `for` statement.
- Input control, using the `getline` command (see Section 3.7 [Explicit Input with `getline`], page 30), and the `next` statement (see Section 9.7 [The `next` Statement], page 78).
- Output statements, `print` and `printf`. See Chapter 4 [Printing Output], page 35.
- Deletion statements, for deleting array elements. See Section 10.6 [The `delete` Statement], page 85.

The next two chapters cover in detail expressions and control statements, respectively. We go on to treat arrays and built-in functions, both of which are used in expressions. Then we proceed to discuss how to define your own functions.

8 Expressions as Action Statements

Expressions are the basic building block of `awk` actions. An expression evaluates to a value, which you can print, test, store in a variable or pass to a function. But beyond that, an expression can assign a new value to a variable or a field, with an assignment operator.

An expression can serve as a statement on its own. Most other kinds of statements contain one or more expressions which specify data to be operated on. As in other languages, expressions in `awk` include variables, array references, constants, and function calls, as well as combinations of these with various operators.

8.1 Constant Expressions

The simplest type of expression is the *constant*, which always has the same value. There are three types of constants: numeric constants, string constants, and regular expression constants.

A *numeric constant* stands for a number. This number can be an integer, a decimal fraction, or a number in scientific (exponential) notation. Note that all numeric values are represented within `awk` in double-precision floating point. Here are some examples of numeric constants, which all have the same value:

```
105
1.05e+2
1050e-1
```

A string constant consists of a sequence of characters enclosed in double-quote marks. For example:

```
"parrot"
```

represents the string whose contents are '`parrot`'. Strings in `gawk` can be of any length and they can contain all the possible 8-bit ASCII characters including ASCII NUL. Other `awk` implementations may have difficulty with some character codes.

Some characters cannot be included literally in a string constant. You represent them instead with *escape sequences*, which are character sequences beginning with a backslash ('`\`').

One use of an escape sequence is to include a double-quote character in a string constant. Since a plain double-quote would end the string, you must use '`\"`' to represent a single double-quote character as a part of the string. The backslash character itself is another character that cannot be included normally; you write '`\\`' to put one backslash in the string. Thus, the string whose contents are the two characters '`"\`' must be written `"\"\\"`.

Another use of backslash is to represent unprintable characters such as newline. While there is nothing to stop you from writing most of these characters directly in a string constant, they may look ugly.

Here is a table of all the escape sequences used in `awk`:

`\\`	Represents a literal backslash, '`\`'.
`\a`	Represents the "alert" character, control-g, ASCII code 7.
`\b`	Represents a backspace, control-h, ASCII code 8.
`\f`	Represents a formfeed, control-l, ASCII code 12.
`\n`	Represents a newline, control-j, ASCII code 10.
`\r`	Represents a carriage return, control-m, ASCII code 13.
`\t`	Represents a horizontal tab, control-i, ASCII code 9.
`\v`	Represents a vertical tab, control-k, ASCII code 11.
`\`*nnn*	Represents the octal value *nnn*, where *nnn* are one to three digits between 0 and 7. For example, the code for the ASCII ESC (escape) character is '`\033`'.
`\x`*hh*...	Represents the hexadecimal value *hh*, where *hh* are hexadecimal digits ('`0`' through '`9`' and either '`A`' through '`F`' or '`a`' through '`f`'). Like the same construct in ANSI C, the escape sequence continues until the first non-hexadecimal digit is seen. However, using more than two hexadecimal digits produces undefined results. (The '`\x`' escape sequence is not allowed in POSIX `awk`.)

A *constant regexp* is a regular expression description enclosed in slashes, such as `/^beginning and end$/`. Most regexps used in `awk` programs are constant, but the '`~`' and '`!~`' operators can also match computed or "dynamic" regexps (see Section 6.2.1 [How to Use Regular Expressions], page 47).

Constant regexps may be used like simple expressions. When a constant regexp is not on the right hand side of the '`~`' or '`!~`' operators, it has the same meaning as if it appeared in a pattern, i.e. '`($0 ~ /foo/)`' (see Section 6.5 [Expressions as Patterns], page 52). This means that the two code segments,

```
if ($0 ~ /barfly/ || $0 ~ /camelot/)
    print "found"
```

and

```
if (/barfly/ || /camelot/)
    print "found"
```

are exactly equivalent. One rather bizarre consequence of this rule is that the following boolean expression is legal, but does not do what the user intended:

```
if (/foo/ ~ $1) print "found foo"
```

This code is "obviously" testing `$1` for a match against the regexp `/foo/`. But in fact, the expression `(/foo/ ~ $1)` actually means `(($0 ~ /foo/) ~ $1)`. In other words, first match the input record against the regexp `/foo/`. The result will be either a 0 or a 1, depending upon the success or failure of the match. Then match that result against the first field in the record.

Another consequence of this rule is that the assignment statement

```
matches = /foo/
```

will assign either 0 or 1 to the variable `matches`, depending upon the contents of the current input record.

Constant regular expressions are also used as the first argument for the `sub` and `gsub` functions (see Section 11.3 [Built-in Functions for String Manipulation], page 90).

This feature of the language was never well documented until the POSIX specification.

You may be wondering, when is

```
$1 ~ /foo/ { ... }
```

preferable to

```
$1 ~ "foo" { ... }
```

Since the right-hand sides of both '`~`' operators are constants, it is more efficient to use the '`/foo/`' form: `awk` can note that you have supplied a regexp and store it internally in a form that makes pattern matching more efficient. In the second form, `awk` must first convert the string into this internal form, and then perform the pattern matching. The first form is also better style; it shows clearly that you intend a regexp match.

8.2 Variables

Variables let you give names to values and refer to them later. You have already seen variables in many of the examples. The name of a variable must be a sequence of letters, digits and underscores, but it may not begin with a digit. Case is significant in variable names; `a` and `A` are distinct variables.

A variable name is a valid expression by itself; it represents the variable's current value. Variables are given new values with *assignment operators* and *increment operators*. See Section 8.7 [Assignment Expressions], page 64.

A few variables have special built-in meanings, such as `FS`, the field separator, and `NF`, the number of fields in the current input record. See Chapter 13 [Built-in Variables], page 101, for a list of them. These built-in variables can be used and assigned just like all other variables, but their values are also used or changed automatically by `awk`. Each built-in variable's name is made entirely of upper case letters.

Variables in `awk` can be assigned either numeric or string values. By default, variables are initialized to the null string, which is effectively zero if converted to a number. There is no need to "initialize" each variable explicitly in `awk`, the way you would in C or most other traditional languages.

8.2.1 Assigning Variables on the Command Line

You can set any `awk` variable by including a *variable assignment* among the arguments on the command line when you invoke `awk` (see Chapter 14 [Invoking `awk`], page 105). Such an assignment has this form:

variable=*text*

With it, you can set a variable either at the beginning of the `awk` run or in between input files.

If you precede the assignment with the '`-v`' option, like this:

`-v` *variable*=*text*

then the variable is set at the very beginning, before even the `BEGIN` rules are run. The '`-v`' option and its assignment must precede all the file name arguments, as well as the program text.

Otherwise, the variable assignment is performed at a time determined by its position among the input file arguments: after the processing of the preceding input file argument. For example:

```
awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list
```

prints the value of field number `n` for all input records. Before the first file is read, the command line sets the variable `n` equal to 4. This causes the fourth field to be printed in lines from the file '`inventory-shipped`'. After the first file has finished, but before the second file is started, `n` is set to 2, so that the second field is printed in lines from '`BBS-list`'.

Command line arguments are made available for explicit examination by the `awk` program in an array named `ARGV` (see Chapter 13 [Built-in Variables], page 101).

`awk` processes the values of command line assignments for escape sequences (see Section 8.1 [Constant Expressions], page 57).

8.3 Arithmetic Operators

The `awk` language uses the common arithmetic operators when evaluating expressions. All of these arithmetic operators follow normal precedence rules, and work as you would expect them to. This example divides field three by field four, adds field two, stores the result into field one, and prints the resulting altered input record:

```
awk '{ $1 = $2 + $3 / $4; print }' inventory-shipped
```

The arithmetic operators in `awk` are:

x + *y*	Addition.
x - *y*	Subtraction.
- *x*	Negation.

`+ x` Unary plus. No real effect on the expression.

`x * y` Multiplication.

`x / y` Division. Since all numbers in `awk` are double-precision floating point, the result is not rounded to an integer: `3 / 4` has the value 0.75.

`x % y` Remainder. The quotient is rounded toward zero to an integer, multiplied by *y* and this result is subtracted from *x*. This operation is sometimes known as “trunc-mod.” The following relation always holds:

```
b * int(a / b) + (a % b) == a
```

One possibly undesirable effect of this definition of remainder is that *x* `%` *y* is negative if *x* is negative. Thus,

```
-17 % 8 = -1
```

In other `awk` implementations, the signedness of the remainder may be machine dependent.

`x ^ y`
`x ** y` Exponentiation: *x* raised to the *y* power. `2 ^ 3` has the value 8. The character sequence ‘`**`’ is equivalent to ‘`^`’. (The POSIX standard only specifies the use of ‘`^`’ for exponentiation.)

8.4 String Concatenation

There is only one string operation: concatenation. It does not have a specific operator to represent it. Instead, concatenation is performed by writing expressions next to one another, with no operator. For example:

```
awk '{ print "Field number one: " $1 }' BBS-list
```

produces, for the first record in ‘`BBS-list`’:

```
Field number one: aardvark
```

Without the space in the string constant after the ‘`:`’, the line would run together. For example:

```
awk '{ print "Field number one:" $1 }' BBS-list
```

produces, for the first record in ‘`BBS-list`’:

```
Field number one:aardvark
```

Since string concatenation does not have an explicit operator, it is often necessary to insure that it happens where you want it to by enclosing the items to be concatenated in parentheses. For example, the following code fragment does not concatenate `file` and `name` as you might expect:

```
file = "file"
name = "name"
print "something meaningful" > file name
```

It is necessary to use the following:

```
print "something meaningful" > (file name)
```

We recommend you use parentheses around concatenation in all but the most common contexts (such as in the right-hand operand of '`=`').

8.5 Comparison Expressions

Comparison expressions compare strings or numbers for relationships such as equality. They are written using *relational operators*, which are a superset of those in C. Here is a table of them:

x `<` *y*	True if *x* is less than *y*.
x `<=` *y*	True if *x* is less than or equal to *y*.
x `>` *y*	True if *x* is greater than *y*.
x `>=` *y*	True if *x* is greater than or equal to *y*.
x `==` *y*	True if *x* is equal to *y*.
x `!=` *y*	True if *x* is not equal to *y*.
x `~` *y*	True if the string *x* matches the regexp denoted by *y*.
x `!~` *y*	True if the string *x* does not match the regexp denoted by *y*.
subscript `in` *array*	True if array *array* has an element with the subscript *subscript*.

Comparison expressions have the value 1 if true and 0 if false.

The rules `gawk` uses for performing comparisons are based on those in draft 11.2 of the POSIX standard. The POSIX standard introduced the concept of a *numeric string*, which is simply a string that looks like a number, for example, `" +2"`.

When performing a relational operation, `gawk` considers the type of an operand to be the type it received on its last *assignment*, rather than the type of its last *use* (see Section 8.10 [Numeric and String Values], page 68). This type is *unknown* when the operand is from an "external" source: field variables, command line arguments, array elements resulting from a `split` operation, and the value of an `ENVIRON` element. In this case only, if the operand is a numeric string, then it is considered to be of both string type and numeric type. If at least one operand of a comparison is of string type only, then a string comparison is performed. Any numeric operand will be converted to a string using the value of `CONVFMT` (see Section 8.9 [Conversion of Strings and Numbers], page 67). If one operand of a comparison is numeric, and the other operand is either numeric or both numeric and string, then `awk` does a numeric comparison. If both operands have both types, then the comparison is numeric. Strings are compared by comparing the first character of each, then the second character of each, and so on. Thus `"10"` is less than `"9"`. If there are two strings where one is a prefix of the other, the shorter string is less than the longer one. Thus `"abc"` is less than `"abcd"`.

Here are some sample expressions, how `awk` compares them, and what the result of the comparison is.

`1.5 <= 2.0`
numeric comparison (true)

`"abc" >= "xyz"`
: string comparison (false)

`1.5 != " +2"`
: string comparison (true)

`"1e2" < "3"`
: string comparison (true)

`a = 2; b = "2"`
`a == b`
: string comparison (true)

```
echo 1e2 3 | awk '{ print ($1 < $2) ? "true" : "false" }'
```

prints '`false`' since both `$1` and `$2` are numeric strings and thus have both string and numeric types, thus dictating a numeric comparison.

The purpose of the comparison rules and the use of numeric strings is to attempt to produce the behavior that is "least surprising," while still "doing the right thing."

String comparisons and regular expression comparisons are very different. For example,

```
$1 == "foo"
```

has the value of 1, or is true, if the first field of the current input record is precisely '`foo`'. By contrast,

```
$1 ~ /foo/
```

has the value 1 if the first field contains '`foo`', such as '`foobar`'.

The right hand operand of the '`~`' and '`!~`' operators may be either a constant regexp (`/`...`/`), or it may be an ordinary expression, in which case the value of the expression as a string is a dynamic regexp (see Section 6.2.1 [How to Use Regular Expressions], page 47).

In very recent implementations of `awk`, a constant regular expression in slashes by itself is also an expression. The regexp `/`*regexp*`/` is an abbreviation for this comparison expression:

```
$0 ~ /regexp/
```

In some contexts it may be necessary to write parentheses around the regexp to avoid confusing the `awk` parser. For example, `(/x/ - /y/) > threshold` is not allowed, but `((/x/) - (/y/)) > threshold` parses properly.

One special place where `/foo/` is *not* an abbreviation for `$0 ~ /foo/` is when it is the right-hand operand of '`~`' or '`!~`'! See Section 8.1 [Constant Expressions], page 57, where this is discussed in more detail.

8.6 Boolean Expressions

A *boolean expression* is a combination of comparison expressions or matching expressions, using the boolean operators "or" ('`||`'), "and" ('`&&`'), and "not" ('`!`'), along with parentheses to control nesting. The truth of the boolean expression is computed by combining the truth values of the component expressions.

Boolean expressions can be used wherever comparison and matching expressions can be used. They can be used in `if`, `while do` and `for` statements. They have numeric values (1 if true, 0 if false), which come into play if the result of the boolean expression is stored in a variable, or used in arithmetic.

In addition, every boolean expression is also a valid boolean pattern, so you can use it as a pattern to control the execution of rules.

Here are descriptions of the three boolean operators, with an example of each. It may be instructive to compare these examples with the analogous examples of boolean patterns (see Section 6.4 [Boolean Operators and Patterns], page 51), which use the same boolean operators in patterns instead of expressions.

boolean1 `&&` *boolean2*
: True if both *boolean1* and *boolean2* are true. For example, the following statement prints the current input record if it contains both '`2400`' and '`foo`'.

```
if ($0 ~ /2400/ && $0 ~ /foo/) print
```

The subexpression *boolean2* is evaluated only if *boolean1* is true. This can make a difference when *boolean2* contains expressions that have side effects: in the case of `$0 ~ /foo/ && ($2 == bar++)`, the variable `bar` is not incremented if there is no '`foo`' in the record.

boolean1 `||` *boolean2*
: True if at least one of *boolean1* or *boolean2* is true. For example, the following command prints all records in the input file '`BBS-list`' that contain *either* '`2400`' or '`foo`', or both.

```
awk '{ if ($0 ~ /2400/ || $0 ~ /foo/) print }' BBS-list
```

The subexpression *boolean2* is evaluated only if *boolean1* is false. This can make a difference when *boolean2* contains expressions that have side effects.

`!`*boolean*
: True if *boolean* is false. For example, the following program prints all records in the input file '`BBS-list`' that do *not* contain the string '`foo`'.

```
awk '{ if (! ($0 ~ /foo/)) print }' BBS-list
```

8.7 Assignment Expressions

An *assignment* is an expression that stores a new value into a variable. For example, let's assign the value 1 to the variable `z`:

```
z = 1
```

After this expression is executed, the variable `z` has the value 1. Whatever old value `z` had before the assignment is forgotten.

Assignments can store string values also. For example, this would store the value `"this food is good"` in the variable `message`:

```
thing = "food"
predicate = "good"
message = "this " thing " is " predicate
```

(This also illustrates concatenation of strings.)

The '`=`' sign is called an *assignment operator*. It is the simplest assignment operator because the value of the right-hand operand is stored unchanged.

Most operators (addition, concatenation, and so on) have no effect except to compute a value. If you ignore the value, you might as well not use the operator. An assignment operator is different; it does produce a value, but even if you ignore the value, the assignment still makes itself felt through the alteration of the variable. We call this a *side effect*.

The left-hand operand of an assignment need not be a variable (see Section 8.2 [Variables], page 59); it can also be a field (see Section 3.4 [Changing the Contents of a Field], page 24) or an array element (see Chapter 10 [Arrays in `awk`], page 81). These are all called *lvalues*, which means they can appear on the left-hand side of an assignment operator. The right-hand operand may be any expression; it produces the new value which the assignment stores in the specified variable, field or array element.

It is important to note that variables do *not* have permanent types. The type of a variable is simply the type of whatever value it happens to hold at the moment. In the following program fragment, the variable `foo` has a numeric value at first, and a string value later on:

```
foo = 1
print foo
foo = "bar"
print foo
```

When the second assignment gives `foo` a string value, the fact that it previously had a numeric value is forgotten.

An assignment is an expression, so it has a value: the same value that is assigned. Thus, `z = 1` as an expression has the value 1. One consequence of this is that you can write multiple assignments together:

```
x = y = z = 0
```

stores the value 0 in all three variables. It does this because the value of `z = 0`, which is 0, is stored into `y`, and then the value of `y = z = 0`, which is 0, is stored into `x`.

You can use an assignment anywhere an expression is called for. For example, it is valid to write `x != (y = 1)` to set `y` to 1 and then test whether `x` equals 1. But this style tends to make programs hard to read; except in a one-shot program, you should rewrite it to get rid of such nesting of assignments. This is never very hard.

Aside from '`=`', there are several other assignment operators that do arithmetic with the old value of the variable. For example, the operator '`+=`' computes a new value by adding the right-hand value to the old value of the variable. Thus, the following assignment adds 5 to the value of `foo`:

```
foo += 5
```

This is precisely equivalent to the following:

```
foo = foo + 5
```

Use whichever one makes the meaning of your program clearer.

Here is a table of the arithmetic assignment operators. In each case, the right-hand operand is an expression whose value is converted to a number.

lvalue `+=` *increment*
: Adds *increment* to the value of *lvalue* to make the new value of *lvalue*.

lvalue `-=` *decrement*
: Subtracts *decrement* from the value of *lvalue*.

lvalue `*=` *coefficient*
: Multiplies the value of *lvalue* by *coefficient*.

lvalue `/=` *quotient*
: Divides the value of *lvalue* by *quotient*.

lvalue `%=` *modulus*
: Sets *lvalue* to its remainder by *modulus*.

lvalue `^=` *power*
lvalue `**=` *power*
: Raises *lvalue* to the power *power*. (Only the `^=` operator is specified by POSIX.)

8.8 Increment Operators

Increment operators increase or decrease the value of a variable by 1. You could do the same thing with an assignment operator, so the increment operators add no power to the `awk` language; but they are convenient abbreviations for something very common.

The operator to add 1 is written '`++`'. It can be used to increment a variable either before or after taking its value.

To pre-increment a variable *v*, write `++`*v*. This adds 1 to the value of *v* and that new value is also the value of this expression. The assignment expression *v* `+= 1` is completely equivalent.

Writing the '`++`' after the variable specifies post-increment. This increments the variable value just the same; the difference is that the value of the increment expression itself is the variable's *old* value. Thus, if `foo` has the value 4, then the expression `foo++` has the value 4, but it changes the value of `foo` to 5.

The post-increment `foo++` is nearly equivalent to writing `(foo += 1) - 1`. It is not perfectly equivalent because all numbers in `awk` are floating point: in floating point, `foo + 1 - 1` does not necessarily equal `foo`. But the difference is minute as long as you stick to numbers that are fairly small (less than a trillion).

Any lvalue can be incremented. Fields and array elements are incremented just like variables. (Use '`$(i++)`' when you wish to do a field reference and a variable increment at the same time. The parentheses are necessary because of the precedence of the field reference operator, '`$`'.)

The decrement operator '`--`' works just like '`++`' except that it subtracts 1 instead of adding. Like '`++`', it can be used before the lvalue to pre-decrement or after it to post-decrement.

Here is a summary of increment and decrement expressions.

`++`*lvalue*
: This expression increments *lvalue* and the new value becomes the value of this expression.

lvalue`++`
: This expression causes the contents of *lvalue* to be incremented. The value of the expression is the *old* value of *lvalue*.

`--`*lvalue*
: Like `++`*lvalue*, but instead of adding, it subtracts. It decrements *lvalue* and delivers the value that results.

lvalue`--`
: Like *lvalue*`++`, but instead of adding, it subtracts. It decrements *lvalue*. The value of the expression is the *old* value of *lvalue*.

8.9 Conversion of Strings and Numbers

Strings are converted to numbers, and numbers to strings, if the context of the `awk` program demands it. For example, if the value of either `foo` or `bar` in the expression `foo + bar` happens to be a string, it is converted to a number before the addition is performed. If numeric values appear in string concatenation, they are converted to strings. Consider this:

```
two = 2; three = 3
print (two three) + 4
```

This eventually prints the (numeric) value 27. The numeric values of the variables `two` and `three` are converted to strings and concatenated together, and the resulting string is converted back to the number 23, to which 4 is then added.

If, for some reason, you need to force a number to be converted to a string, concatenate the null string with that number. To force a string to be converted to a number, add zero to that string.

A string is converted to a number by interpreting a numeric prefix of the string as numerals: `"2.5"` converts to 2.5, `"1e3"` converts to 1000, and `"25fix"` has a numeric value of 25. Strings that can't be interpreted as valid numbers are converted to zero.

The exact manner in which numbers are converted into strings is controlled by the `awk` built-in variable `CONVFMT` (see Chapter 13 [Built-in Variables], page 101). Numbers are converted using a special version of the `sprintf` function (see Chapter 11 [Built-in Functions], page 89) with `CONVFMT` as the format specifier.

CONVFMT's default value is "%.6g", which prints a value with at least six significant digits. For some applications you will want to change it to specify more precision. Double precision on most modern machines gives you 16 or 17 decimal digits of precision.

Strange results can happen if you set CONVFMT to a string that doesn't tell sprintf how to format floating point numbers in a useful way. For example, if you forget the '%' in the format, all numbers will be converted to the same constant string.

As a special case, if a number is an integer, then the result of converting it to a string is *always* an integer, no matter what the value of CONVFMT may be. Given the following code fragment:

```
CONVFMT = "%2.2f"
a = 12
b = a ""
```

b has the value "12", not "12.00".

Prior to the POSIX standard, awk specified that the value of OFMT was used for converting numbers to strings. OFMT specifies the output format to use when printing numbers with print. CONVFMT was introduced in order to separate the semantics of conversions from the semantics of printing. Both CONVFMT and OFMT have the same default value: "%.6g". In the vast majority of cases, old awk programs will not change their behavior. However, this use of OFMT is something to keep in mind if you must port your program to other implementations of awk; we recommend that instead of changing your programs, you just port gawk itself!

8.10 Numeric and String Values

Through most of this manual, we present awk values (such as constants, fields, or variables) as *either* numbers *or* strings. This is a convenient way to think about them, since typically they are used in only one way, or the other.

In truth though, awk values can be *both* string and numeric, at the same time. Internally, awk represents values with a string, a (floating point) number, and an indication that one, the other, or both representations of the value are valid.

Keeping track of both kinds of values is important for execution efficiency: a variable can acquire a string value the first time it is used as a string, and then that string value can be used until the variable is assigned a new value. Thus, if a variable with only a numeric value is used in several concatenations in a row, it only has to be given a string representation once. The numeric value remains valid, so that no conversion back to a number is necessary if the variable is later used in an arithmetic expression.

Tracking both kinds of values is also important for precise numerical calculations. Consider the following:

```
a = 123.321
CONVFMT = "%3.1f"
b = a " is a number"
c = a + 1.654
```

The variable a receives a string value in the concatenation and assignment to b. The string value of a is "123.3". If the numeric value was lost when it was converted to a string, then the numeric use of a in the last statement would lose information. c would be assigned the value 124.954 instead of 124.975. Such errors accumulate rapidly, and very adversely affect numeric computations.

Once a numeric value acquires a corresponding string value, it stays valid until a new assignment is made. If CONVFMT (see Section 8.9 [Conversion of Strings and Numbers], page 67) changes in the meantime, the old string value will still be used. For example:

```
BEGIN {
    CONVFMT = "%2.2f"
    a = 123.456
    b = a ""                # force ‘a’ to have string value too
    printf "a = %s\n", a
    CONVFMT = "%.6g"
    printf "a = %s\n", a
    a += 0                  # make ‘a’ numeric only again
    printf "a = %s\n", a    # use ‘a’ as string
}
```

This program prints ‘a = 123.46’ twice, and then prints ‘a = 123.456’.

See Section 8.9 [Conversion of Strings and Numbers], page 67, for the rules that specify how string values are made from numeric values.

8.11 Conditional Expressions

A *conditional expression* is a special kind of expression with three operands. It allows you to use one expression's value to select one of two other expressions.

The conditional expression looks the same as in the C language:

selector ? *if-true-exp* : *if-false-exp*

There are three subexpressions. The first, *selector*, is always computed first. If it is "true" (not zero and not null) then *if-true-exp* is computed next and its value becomes the value of the whole expression. Otherwise, *if-false-exp* is computed next and its value becomes the value of the whole expression.

For example, this expression produces the absolute value of x:

```
x > 0 ? x : -x
```

Each time the conditional expression is computed, exactly one of *if-true-exp* and *if-false-exp* is computed; the other is ignored. This is important when the expressions contain side effects. For example, this conditional expression examines element i of either array a or array b, and increments i.

```
x == y ? a[i++] : b[i++]
```

This is guaranteed to increment `i` exactly once, because each time one or the other of the two increment expressions is executed, and the other is not.

8.12 Function Calls

A *function* is a name for a particular calculation. Because it has a name, you can ask for it by name at any point in the program. For example, the function `sqrt` computes the square root of a number.

A fixed set of functions are *built-in*, which means they are available in every `awk` program. The `sqrt` function is one of these. See Chapter 11 [Built-in Functions], page 89, for a list of built-in functions and their descriptions. In addition, you can define your own functions in the program for use elsewhere in the same program. See Chapter 12 [User-defined Functions], page 95, for how to do this.

The way to use a function is with a *function call* expression, which consists of the function name followed by a list of *arguments* in parentheses. The arguments are expressions which give the raw materials for the calculation that the function will do. When there is more than one argument, they are separated by commas. If there are no arguments, write just '`()`' after the function name. Here are some examples:

```
sqrt(x^2 + y^2)        # One argument
atan2(y, x)            # Two arguments
rand()                 # No arguments
```

Do not put any space between the function name and the open-parenthesis! A user-defined function name looks just like the name of a variable, and space would make the expression look like concatenation of a variable with an expression inside parentheses. Space before the parenthesis is harmless with built-in functions, but it is best not to get into the habit of using space to avoid mistakes with user-defined functions.

Each function expects a particular number of arguments. For example, the `sqrt` function must be called with a single argument, the number to take the square root of:

```
sqrt(argument)
```

Some of the built-in functions allow you to omit the final argument. If you do so, they use a reasonable default. See Chapter 11 [Built-in Functions], page 89, for full details. If arguments are omitted in calls to user-defined functions, then those arguments are treated as local variables, initialized to the null string (see Chapter 12 [User-defined Functions], page 95).

Like every other expression, the function call has a value, which is computed by the function based on the arguments you give it. In this example, the value of `sqrt(`*argument*`)` is the square root of the argument. A function can also have side effects, such as assigning the values of certain variables or doing I/O.

Here is a command to read numbers, one number per line, and print the square root of each one:

```
     awk '{ print "The square root of", $1, "is", sqrt($1) }'
```

8.13 Operator Precedence (How Operators Nest)

Operator precedence determines how operators are grouped, when different operators appear close by in one expression. For example, '`*`' has higher precedence than '`+`'; thus, `a + b * c` means to multiply `b` and `c`, and then add `a` to the product (i.e., `a + (b * c)`).

You can overrule the precedence of the operators by using parentheses. You can think of the precedence rules as saying where the parentheses are assumed if you do not write parentheses yourself. In fact, it is wise to always use parentheses whenever you have an unusual combination of operators, because other people who read the program may not remember what the precedence is in this case. You might forget, too; then you could make a mistake. Explicit parentheses will help prevent any such mistake.

When operators of equal precedence are used together, the leftmost operator groups first, except for the assignment, conditional and exponentiation operators, which group in the opposite order. Thus, `a - b + c` groups as `(a - b) + c`; `a = b = c` groups as `a = (b = c)`.

The precedence of prefix unary operators does not matter as long as only unary operators are involved, because there is only one way to parse them—innermost first. Thus, `$++i` means `$(++i)` and `++$x` means `++($x)`. However, when another operator follows the operand, then the precedence of the unary operators can matter. Thus, `$x^2` means `($x)^2`, but `-x^2` means `-(x^2)`, because '`-`' has lower precedence than '`^`' while '`$`' has higher precedence.

Here is a table of the operators of `awk`, in order of increasing precedence:

assignment
: '`=`', '`+=`', '`-=`', '`*=`', '`/=`', '`%=`', '`^=`', '`**=`'. These operators group right-to-left. (The '`**=`' operator is not specified by POSIX.)

conditional
: '`?:`'. This operator groups right-to-left.

logical "or".
: '`||`'.

logical "and".
: '`&&`'.

array membership
: '`in`'.

matching
: '`~`', '`!~`'.

relational, and redirection
: The relational operators and the redirections have the same precedence level. Characters such as '`>`' serve both as relationals and as redirections; the context distinguishes between the two meanings.

  The relational operators are '`<`', '`<=`', '`==`', '`!=`', '`>=`' and '`>`'.

  The I/O redirection operators are '`<`', '`>`', '`>>`' and '`|`'.

  Note that I/O redirection operators in `print` and `printf` statements belong to the statement level, not to expressions. The redirection does not produce an expression which could be the operand of another operator. As a result, it does not make sense to use a redirection operator near another operator of lower precedence, without parentheses. Such combinations, for example '`print foo > a ? b : c`', result in syntax errors.

concatenation
No special token is used to indicate concatenation. The operands are simply written side by side.

add, subtract
'`+`', '`-`'.

multiply, divide, mod
'`*`', '`/`', '`%`'.

unary plus, minus, "not"
'`+`', '`-`', '`!`'.

exponentiation
'`^`', '`**`'. These operators group right-to-left. (The '`**`' operator is not specified by POSIX.)

increment, decrement
'`++`', '`--`'.

field '`$`'.

9 Control Statements in Actions

Control statements such as `if`, `while`, and so on control the flow of execution in `awk` programs. Most of the control statements in `awk` are patterned on similar statements in C.

All the control statements start with special keywords such as `if` and `while`, to distinguish them from simple expressions.

Many control statements contain other statements; for example, the `if` statement contains another statement which may or may not be executed. The contained statement is called the *body*. If you want to include more than one statement in the body, group them into a single compound statement with curly braces, separating them with newlines or semicolons.

9.1 The `if` Statement

The `if-else` statement is `awk`'s decision-making statement. It looks like this:

`if (`*condition*`)` *then-body* [`else` *else-body*]

condition is an expression that controls what the rest of the statement will do. If *condition* is true, *then-body* is executed; otherwise, *else-body* is executed (assuming that the `else` clause is present). The `else` part of the statement is optional. The condition is considered false if its value is zero or the null string, and true otherwise.

Here is an example:

```
if (x % 2 == 0)
    print "x is even"
else
    print "x is odd"
```

In this example, if the expression `x % 2 == 0` is true (that is, the value of `x` is divisible by 2), then the first `print` statement is executed, otherwise the second `print` statement is performed.

If the `else` appears on the same line as *then-body*, and *then-body* is not a compound statement (i.e., not surrounded by curly braces), then a semicolon must separate *then-body* from `else`. To illustrate this, let's rewrite the previous example:

```
awk '{ if (x % 2 == 0) print "x is even"; else
        print "x is odd" }'
```

If you forget the '`;`', `awk` won't be able to parse the statement, and you will get a syntax error.

We would not actually write this example this way, because a human reader might fail to see the `else` if it were not the first thing on its line.

9.2 The `while` Statement

In programming, a *loop* means a part of a program that is (or at least can be) executed two or more times in succession.

The `while` statement is the simplest looping statement in `awk`. It repeatedly executes a statement as long as a condition is true. It looks like this:

```
while (condition)
  body
```

Here *body* is a statement that we call the *body* of the loop, and *condition* is an expression that controls how long the loop keeps running.

The first thing the `while` statement does is test *condition*. If *condition* is true, it executes the statement *body*. (*condition* is true when the value is not zero and not a null string.) After *body* has been executed, *condition* is tested again, and if it is still true, *body* is executed again. This process repeats until *condition* is no longer true. If *condition* is initially false, the body of the loop is never executed.

This example prints the first three fields of each record, one per line.

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

Here the body of the loop is a compound statement enclosed in braces, containing two statements.

The loop works like this: first, the value of `i` is set to 1. Then, the `while` tests whether `i` is less than or equal to three. This is the case when `i` equals one, so the `i`-th field is printed. Then the `i++` increments the value of `i` and the loop repeats. The loop terminates when `i` reaches 4.

As you can see, a newline is not required between the condition and the body; but using one makes the program clearer unless the body is a compound statement or is very simple. The newline after the open-brace that begins the compound statement is not required either, but the program would be hard to read without it.

9.3 The `do-while` Statement

The `do` loop is a variation of the `while` looping statement. The `do` loop executes the *body* once, then repeats *body* as long as *condition* is true. It looks like this:

```
do
  body
while (condition)
```

Even if *condition* is false at the start, *body* is executed at least once (and only once, unless executing *body* makes *condition* true). Contrast this with the corresponding `while` statement:

```
while (condition)
  body
```

This statement does not execute *body* even once if *condition* is false to begin with.

Here is an example of a `do` statement:

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

prints each input record ten times. It isn't a very realistic example, since in this case an ordinary `while` would do just as well. But this reflects actual experience; there is only occasionally a real use for a `do` statement.

9.4 The `for` Statement

The `for` statement makes it more convenient to count iterations of a loop. The general form of the `for` statement looks like this:

```
for (initialization; condition; increment)
  body
```

This statement starts by executing *initialization*. Then, as long as *condition* is true, it repeatedly executes *body* and then *increment*. Typically *initialization* sets a variable to either zero or one, *increment* adds 1 to it, and *condition* compares it against the desired number of iterations.

Here is an example of a `for` statement:

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

This prints the first three fields of each input record, one field per line.

In the `for` statement, *body* stands for any statement, but *initialization*, *condition* and *increment* are just expressions. You cannot set more than one variable in the *initialization* part unless you use a multiple assignment statement such as `x = y = 0`, which is possible only if all the initial values are equal. (But you can initialize additional variables by writing their assignments as separate statements preceding the `for` loop.)

The same is true of the *increment* part; to increment additional variables, you must write separate statements at the end of the loop. The C compound expression, using C's comma operator, would be useful in this context, but it is not supported in `awk`.

Most often, *increment* is an increment expression, as in the example above. But this is not required; it can be any expression whatever. For example, this statement prints all the powers of 2 between 1 and 100:

```
for (i = 1; i <= 100; i *= 2)
  print i
```

Any of the three expressions in the parentheses following the `for` may be omitted if there is nothing to be done there. Thus, '`for (;x > 0;)`' is equivalent to '`while (x > 0)`'. If the *condition* is omitted, it is treated as *true*, effectively yielding an *infinite loop* (i.e., a loop that will never terminate).

In most cases, a `for` loop is an abbreviation for a `while` loop, as shown here:

```
initialization
while (condition) {
  body
  increment
}
```

The only exception is when the `continue` statement (see Section 9.6 [The `continue` Statement], page 77) is used inside the loop; changing a `for` statement to a `while` statement in this way can change the effect of the `continue` statement inside the loop.

There is an alternate version of the `for` loop, for iterating over all the indices of an array:

```
for (i in array)
    do something with array[i]
```

See Chapter 10 [Arrays in `awk`], page 81, for more information on this version of the `for` loop.

The `awk` language has a `for` statement in addition to a `while` statement because often a `for` loop is both less work to type and more natural to think of. Counting the number of iterations is very common in loops. It can be easier to think of this counting as part of looping rather than as something to do inside the loop.

The next section has more complicated examples of `for` loops.

9.5 The `break` Statement

The `break` statement jumps out of the innermost `for`, `while`, or `do-while` loop that encloses it. The following example finds the smallest divisor of any integer, and also identifies prime numbers:

```
awk '# find smallest divisor of num
```

```
     { num = $1
       for (div = 2; div*div <= num; div++)
         if (num % div == 0)
           break
       if (num % div == 0)
         printf "Smallest divisor of %d is %d\n", num, div
       else
         printf "%d is prime\n", num  }'
```

When the remainder is zero in the first `if` statement, `awk` immediately *breaks out* of the containing `for` loop. This means that `awk` proceeds immediately to the statement following the loop and continues processing. (This is very different from the `exit` statement which stops the entire `awk` program. See Section 9.8 [The `exit` Statement], page 79.)

Here is another program equivalent to the previous one. It illustrates how the *condition* of a `for` or `while` could just as well be replaced with a `break` inside an `if`:

```
awk '# find smallest divisor of num
     { num = $1
       for (div = 2; ; div++) {
         if (num % div == 0) {
           printf "Smallest divisor of %d is %d\n", num, div
           break
         }
         if (div*div > num) {
           printf "%d is prime\n", num
           break
         }
       }
}'
```

9.6 The `continue` Statement

The `continue` statement, like `break`, is used only inside `for`, `while`, and `do-while` loops. It skips over the rest of the loop body, causing the next cycle around the loop to begin immediately. Contrast this with `break`, which jumps out of the loop altogether. Here is an example:

```
# print names that don't contain the string "ignore"

# first, save the text of each line
{ names[NR] = $0 }

# print what we're interested in
END {
   for (x in names) {
       if (names[x] ~ /ignore/)
           continue
       print names[x]
   }
}
```

If one of the input records contains the string ‘ignore’, this example skips the print statement for that record, and continues back to the first statement in the loop.

This is not a practical example of `continue`, since it would be just as easy to write the loop like this:

```
for (x in names)
  if (names[x] !~ /ignore/)
    print names[x]
```

The `continue` statement in a `for` loop directs `awk` to skip the rest of the body of the loop, and resume execution with the increment-expression of the `for` statement. The following program illustrates this fact:

```
awk 'BEGIN {
     for (x = 0; x <= 20; x++) {
         if (x == 5)
             continue
         printf ("%d ", x)
     }
     print ""
}'
```

This program prints all the numbers from 0 to 20, except for 5, for which the `printf` is skipped. Since the increment `x++` is not skipped, `x` does not remain stuck at 5. Contrast the `for` loop above with the `while` loop:

```
awk 'BEGIN {
     x = 0
     while (x <= 20) {
         if (x == 5)
             continue
         printf ("%d ", x)
         x++
     }
     print ""
}'
```

This program loops forever once `x` gets to 5.

As described above, the `continue` statement has no meaning when used outside the body of a loop.

9.7 The next Statement

The `next` statement forces `awk` to immediately stop processing the current record and go on to the next record. This means that no further rules are executed for the current record. The rest of the current rule’s action is not executed either.

Contrast this with the effect of the getline function (see Section 3.7 [Explicit Input with getline], page 30). That too causes awk to read the next record immediately, but it does not alter the flow of control in any way. So the rest of the current action executes with a new input record.

At the highest level, awk program execution is a loop that reads an input record and then tests each rule's pattern against it. If you think of this loop as a for statement whose body contains the rules, then the next statement is analogous to a continue statement: it skips to the end of the body of this implicit loop, and executes the increment (which reads another record).

For example, if your awk program works only on records with four fields, and you don't want it to fail when given bad input, you might use this rule near the beginning of the program:

```
NF != 4 {
  printf("line %d skipped: doesn't have 4 fields", FNR) > "/dev/stderr"
  next
}
```

so that the following rules will not see the bad record. The error message is redirected to the standard error output stream, as error messages should be. See Section 4.7 [Standard I/O Streams], page 44.

According to the POSIX standard, the behavior is undefined if the next statement is used in a BEGIN or END rule. gawk will treat it as a syntax error.

If the next statement causes the end of the input to be reached, then the code in the END rules, if any, will be executed. See Section 6.7 [BEGIN and END Special Patterns], page 53.

9.8 The exit Statement

The exit statement causes awk to immediately stop executing the current rule and to stop processing input; any remaining input is ignored.

If an exit statement is executed from a BEGIN rule the program stops processing everything immediately. No input records are read. However, if an END rule is present, it is executed (see Section 6.7 [BEGIN and END Special Patterns], page 53).

If exit is used as part of an END rule, it causes the program to stop immediately.

An exit statement that is part of an ordinary rule (that is, not part of a BEGIN or END rule) stops the execution of any further automatic rules, but the END rule is executed if there is one. If you do not want the END rule to do its job in this case, you can set a variable to nonzero before the exit statement, and check that variable in the END rule.

If an argument is supplied to exit, its value is used as the exit status code for the awk process. If no argument is supplied, exit returns status zero (success).

For example, let's say you've discovered an error condition you really don't know how to handle. Conventionally, programs report this by exiting with a nonzero status. Your awk program can do this using an exit statement with a nonzero argument. Here's an example of this:

```
BEGIN {
       if (("date" | getline date_now) < 0) {
         print "Can't get system date" > "/dev/stderr"
         exit 4
       }
}
```

10 Arrays in awk

An *array* is a table of values, called *elements*. The elements of an array are distinguished by their indices. *Indices* may be either numbers or strings. Each array has a name, which looks like a variable name, but must not be in use as a variable name in the same awk program.

10.1 Introduction to Arrays

The awk language has one-dimensional *arrays* for storing groups of related strings or numbers.

Every awk array must have a name. Array names have the same syntax as variable names; any valid variable name would also be a valid array name. But you cannot use one name in both ways (as an array and as a variable) in one awk program.

Arrays in awk superficially resemble arrays in other programming languages; but there are fundamental differences. In awk, you don't need to specify the size of an array before you start to use it. Additionally, any number or string in awk may be used as an array index.

In most other languages, you have to *declare* an array and specify how many elements or components it contains. In such languages, the declaration causes a contiguous block of memory to be allocated for that many elements. An index in the array must be a positive integer; for example, the index 0 specifies the first element in the array, which is actually stored at the beginning of the block of memory. Index 1 specifies the second element, which is stored in memory right after the first element, and so on. It is impossible to add more elements to the array, because it has room for only as many elements as you declared.

A contiguous array of four elements might look like this, conceptually, if the element values are 8, "foo", "" and 30:

```
+---------+---------+--------+---------+
|    8    |  "foo"  |   ""   |    30   |      value
+---------+---------+--------+---------+
     0         1         2         3          index
```

Only the values are stored; the indices are implicit from the order of the values. 8 is the value at index 0, because 8 appears in the position with 0 elements before it.

Arrays in awk are different: they are *associative*. This means that each array is a collection of pairs: an index, and its corresponding array element value:

Element 4	Value 30
Element 2	Value "foo"
Element 1	Value 8
Element 3	Value ""

We have shown the pairs in jumbled order because their order is irrelevant.

One advantage of an associative array is that new pairs can be added at any time. For example, suppose we add to the above array a tenth element whose value is `"number ten"`. The result is this:

```
Element 10     Value "number ten"
Element 4      Value 30
Element 2      Value "foo"
Element 1      Value 8
Element 3      Value ""
```

Now the array is *sparse* (i.e., some indices are missing): it has elements 1–4 and 10, but doesn't have elements 5, 6, 7, 8, or 9.

Another consequence of associative arrays is that the indices don't have to be positive integers. Any number, or even a string, can be an index. For example, here is an array which translates words from English into French:

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

Here we decided to translate the number 1 in both spelled-out and numeric form—thus illustrating that a single array can have both numbers and strings as indices.

When `awk` creates an array for you, e.g., with the `split` built-in function, that array's indices are consecutive integers starting at 1. (See Section 11.3 [Built-in Functions for String Manipulation], page 90.)

10.2 Referring to an Array Element

The principal way of using an array is to refer to one of its elements. An array reference is an expression which looks like this:

```
array[index]
```

Here, *array* is the name of an array. The expression *index* is the index of the element of the array that you want.

The value of the array reference is the current value of that array element. For example, `foo[4.3]` is an expression for the element of array `foo` at index 4.3.

If you refer to an array element that has no recorded value, the value of the reference is `""`, the null string. This includes elements to which you have not assigned any value, and elements that have been deleted (see Section 10.6 [The `delete` Statement], page 85). Such a reference automatically creates that array element, with the null string as its value. (In some cases, this is unfortunate, because it might waste memory inside `awk`).

You can find out if an element exists in an array at a certain index with the expression:

```
index in array
```

This expression tests whether or not the particular index exists, without the side effect of creating that element if it is not present. The expression has the value 1 (true) if *array*[*index*] exists, and 0 (false) if it does not exist.

For example, to test whether the array `frequencies` contains the index `"2"`, you could write this statement:

```
if ("2" in frequencies) print "Subscript \"2\" is present."
```

Note that this is *not* a test of whether or not the array `frequencies` contains an element whose *value* is `"2"`. (There is no way to do that except to scan all the elements.) Also, this *does not* create `frequencies["2"]`, while the following (incorrect) alternative would do so:

```
if (frequencies["2"] != "") print "Subscript \"2\" is present."
```

10.3 Assigning Array Elements

Array elements are lvalues: they can be assigned values just like `awk` variables:

```
array[subscript] = value
```

Here *array* is the name of your array. The expression *subscript* is the index of the element of the array that you want to assign a value. The expression *value* is the value you are assigning to that element of the array.

10.4 Basic Example of an Array

The following program takes a list of lines, each beginning with a line number, and prints them out in order of line number. The line numbers are not in order, however, when they are first read: they are scrambled. This program sorts the lines by making an array using the line numbers as subscripts. It then prints out the lines in sorted order of their numbers. It is a very simple program, and gets confused if it encounters repeated numbers, gaps, or lines that don't begin with a number.

```
{
  if ($1 > max)
    max = $1
  arr[$1] = $0
}

END {
  for (x = 1; x <= max; x++)
    print arr[x]
}
```

The first rule keeps track of the largest line number seen so far; it also stores each line into the array `arr`, at an index that is the line's number.

The second rule runs after all the input has been read, to print out all the lines.

When this program is run with the following input:

```
5  I am the Five man
2  Who are you?  The new number two!
4  . . . And four on the floor
1  Who is number one?
3  I three you.
```

its output is this:

```
1  Who is number one?
2  Who are you?  The new number two!
3  I three you.
4  . . . And four on the floor
5  I am the Five man
```

If a line number is repeated, the last line with a given number overrides the others.

Gaps in the line numbers can be handled with an easy improvement to the program's `END` rule:

```
END {
  for (x = 1; x <= max; x++)
    if (x in arr)
      print arr[x]
}
```

10.5 Scanning all Elements of an Array

In programs that use arrays, often you need a loop that executes once for each element of an array. In other languages, where arrays are contiguous and indices are limited to positive integers, this is easy: the largest index is one less than the length of the array, and you can find all the valid indices by counting from zero up to that value. This technique won't do the job in `awk`, since any number or string may be an array index. So `awk` has a special kind of `for` statement for scanning an array:

```
for (var in array)
  body
```

This loop executes *body* once for each different value that your program has previously used as an index in *array*, with the variable *var* set to that index.

Here is a program that uses this form of the `for` statement. The first rule scans the input records and notes which words appear (at least once) in the input, by storing a 1 into the array `used` with the word as index. The second rule scans the elements of `used` to find all the distinct words that appear in the input. It prints each word that is more than 10 characters long, and also prints the number of such words. See Chapter 11 [Built-in Functions], page 89, for more information on the built-in function `length`.

```
# Record a 1 for each word that is used at least once.
{
  for (i = 1; i <= NF; i++)
    used[$i] = 1
}

# Find number of distinct words more than 10 characters long.
END {
  for (x in used)
    if (length(x) > 10) {
      ++num_long_words
      print x
  }
  print num_long_words, "words longer than 10 characters"
}
```

See Appendix B [Sample Program], page 119, for a more detailed example of this type.

The order in which elements of the array are accessed by this statement is determined by the internal arrangement of the array elements within `awk` and cannot be controlled or changed. This can lead to problems if new elements are added to *array* by statements in *body*; you cannot predict whether or not the `for` loop will reach them. Similarly, changing *var* inside the loop can produce strange results. It is best to avoid such things.

10.6 The `delete` Statement

You can remove an individual element of an array using the `delete` statement:

```
delete array[index]
```

You can not refer to an array element after it has been deleted; it is as if you had never referred to it and had never given it any value. You can no longer obtain any value the element once had.

Here is an example of deleting elements in an array:

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

This example removes all the elements from the array `frequencies`.

If you delete an element, a subsequent `for` statement to scan the array will not report that element, and the `in` operator to check for the presence of that element will return 0:

```
delete foo[4]
```

```
if (4 in foo)
  print "This will never be printed"
```

It is not an error to delete an element which does not exist.

10.7 Using Numbers to Subscript Arrays

An important aspect of arrays to remember is that array subscripts are *always* strings. If you use a numeric value as a subscript, it will be converted to a string value before it is used for subscripting (see Section 8.9 [Conversion of Strings and Numbers], page 67).

This means that the value of the `CONVFMT` can potentially affect how your program accesses elements of an array. For example:

```
a = b = 12.153
data[a] = 1
CONVFMT = "%2.2f"
if (b in data)
    printf "%s is in data", b
else
    printf "%s is not in data", b
```

should print '`12.15 is not in data`'. The first statement gives both `a` and `b` the same numeric value. Assigning to `data[a]` first gives `a` the string value `"12.153"` (using the default conversion value of `CONVFMT`, `"%.6g"`), and then assigns 1 to `data["12.153"]`. The program then changes the value of `CONVFMT`. The test '`(b in data)`' forces `b` to be converted to a string, this time `"12.15"`, since the value of `CONVFMT` only allows two significant digits. This test fails, since `"12.15"` is a different string from `"12.153"`.

According to the rules for conversions (see Section 8.9 [Conversion of Strings and Numbers], page 67), integer values are always converted to strings as integers, no matter what the value of `CONVFMT` may happen to be. So the usual case of

```
for (i = 1; i <= maxsub; i++)
    do something with array[i]
```

will work, no matter what the value of `CONVFMT`.

Like many things in `awk`, the majority of the time things work as you would expect them to work. But it is useful to have a precise knowledge of the actual rules, since sometimes they can have a subtle effect on your programs.

10.8 Multi-dimensional Arrays

A multi-dimensional array is an array in which an element is identified by a sequence of indices, not a single index. For example, a two-dimensional array requires two indices. The usual way (in most languages, including `awk`) to refer to an element of a two-dimensional array named `grid` is with `grid[`*x*`,`*y*`]`.

Multi-dimensional arrays are supported in awk through concatenation of indices into one string. What happens is that awk converts the indices into strings (see Section 8.9 [Conversion of Strings and Numbers], page 67) and concatenates them together, with a separator between them. This creates a single string that describes the values of the separate indices. The combined string is used as a single index into an ordinary, one-dimensional array. The separator used is the value of the built-in variable SUBSEP.

For example, suppose we evaluate the expression foo[5,12]="value" when the value of SUBSEP is "@". The numbers 5 and 12 are converted to strings and concatenated with an '@' between them, yielding "5@12"; thus, the array element foo["5@12"] is set to "value".

Once the element's value is stored, awk has no record of whether it was stored with a single index or a sequence of indices. The two expressions foo[5,12] and foo[5 SUBSEP 12] always have the same value.

The default value of SUBSEP is the string "\034", which contains a nonprinting character that is unlikely to appear in an awk program or in the input data.

The usefulness of choosing an unlikely character comes from the fact that index values that contain a string matching SUBSEP lead to combined strings that are ambiguous. Suppose that SUBSEP were "@"; then foo["a@b", "c"] and foo["a", "b@c"] would be indistinguishable because both would actually be stored as foo["a@b@c"]. Because SUBSEP is "\034", such confusion can arise only when an index contains the character with ASCII code 034, which is a rare event.

You can test whether a particular index-sequence exists in a "multi-dimensional" array with the same operator in used for single dimensional arrays. Instead of a single index as the left-hand operand, write the whole sequence of indices, separated by commas, in parentheses:

(*subscript1*, *subscript2*, ...) in *array*

The following example treats its input as a two-dimensional array of fields; it rotates this array 90 degrees clockwise and prints the result. It assumes that all lines have the same number of elements.

```
awk '{
     if (max_nf < NF)
          max_nf = NF
     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")
     }
}'
```

When given the input:

```
1 2 3 4 5 6
2 3 4 5 6 1
3 4 5 6 1 2
4 5 6 1 2 3
```

it produces:

```
4 3 2 1
5 4 3 2
6 5 4 3
1 6 5 4
2 1 6 5
3 2 1 6
```

10.9 Scanning Multi-dimensional Arrays

There is no special `for` statement for scanning a "multi-dimensional" array; there cannot be one, because in truth there are no multi-dimensional arrays or elements; there is only a multi-dimensional *way of accessing* an array.

However, if your program has an array that is always accessed as multi-dimensional, you can get the effect of scanning it by combining the scanning `for` statement (see Section 10.5 [Scanning all Elements of an Array], page 84) with the `split` built-in function (see Section 11.3 [Built-in Functions for String Manipulation], page 90). It works like this:

```
for (combined in array) {
  split(combined, separate, SUBSEP)
  ...
}
```

This finds each concatenated, combined index in the array, and splits it into the individual indices by breaking it apart where the value of `SUBSEP` appears. The split-out indices become the elements of the array `separate`.

Thus, suppose you have previously stored in *array*`[1, "foo"]`; then an element with index `"1\034foo"` exists in *array*. (Recall that the default value of `SUBSEP` contains the character with code 034.) Sooner or later the `for` statement will find that index and do an iteration with `combined` set to `"1\034foo"`. Then the `split` function is called as follows:

```
split("1\034foo", separate, "\034")
```

The result of this is to set `separate[1]` to 1 and `separate[2]` to `"foo"`. Presto, the original sequence of separate indices has been recovered.

11 Built-in Functions

Built-in functions are functions that are always available for your `awk` program to call. This chapter defines all the built-in functions in `awk`; some of them are mentioned in other sections, but they are summarized here for your convenience. (You can also define new functions yourself. See Chapter 12 [User-defined Functions], page 95.)

11.1 Calling Built-in Functions

To call a built-in function, write the name of the function followed by arguments in parentheses. For example, `atan2(y + z, 1)` is a call to the function `atan2`, with two arguments.

Whitespace is ignored between the built-in function name and the open-parenthesis, but we recommend that you avoid using whitespace there. User-defined functions do not permit whitespace in this way, and you will find it easier to avoid mistakes by following a simple convention which always works: no whitespace after a function name.

Each built-in function accepts a certain number of arguments. In most cases, any extra arguments given to built-in functions are ignored. The defaults for omitted arguments vary from function to function and are described under the individual functions.

When a function is called, expressions that create the function's actual parameters are evaluated completely before the function call is performed. For example, in the code fragment:

```
i = 4
j = sqrt(i++)
```

the variable `i` is set to 5 before `sqrt` is called with a value of 4 for its actual parameter.

11.2 Numeric Built-in Functions

Here is a full list of built-in functions that work with numbers:

`int(`*x*`)`
: This gives you the integer part of *x*, truncated toward 0. This produces the nearest integer to *x*, located between *x* and 0.

  For example, `int(3)` is 3, `int(3.9)` is 3, `int(-3.9)` is −3, and `int(-3)` is −3 as well.

`sqrt(`*x*`)`
: This gives you the positive square root of *x*. It reports an error if *x* is negative. Thus, `sqrt(4)` is 2.

`exp(`*x*`)`
: This gives you the exponential of *x*, or reports an error if *x* is out of range. The range of values *x* can have depends on your machine's floating point representation.

`log(`*x*`)`
: This gives you the natural logarithm of *x*, if *x* is positive; otherwise, it reports an error.

`sin(`*x*`)`
: This gives you the sine of *x*, with *x* in radians.

`cos(`*x*`)`
: This gives you the cosine of *x*, with *x* in radians.

`atan2(`*y*`, `*x*`)`
: This gives you the arctangent of *y* `/` *x* in radians.

`rand()` This gives you a random number. The values of `rand` are uniformly-distributed between 0 and 1. The value is never 0 and never 1.

Often you want random integers instead. Here is a user-defined function you can use to obtain a random nonnegative integer less than *n*:

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

The multiplication produces a random real number greater than 0 and less than *n*. We then make it an integer (using `int`) between 0 and $n - 1$.

Here is an example where a similar function is used to produce random integers between 1 and *n*. Note that this program will print a new random number for each input record.

```
awk '
# Function to roll a simulated die.
function roll(n) { return 1 + int(rand() * n) }

# Roll 3 six-sided dice and print total number of points.
{
      printf("%d points\n", roll(6)+roll(6)+roll(6))
}'
```

Note: `rand` starts generating numbers from the same point, or *seed*, each time you run `awk`. This means that a program will produce the same results each time you run it. The numbers are random within one `awk` run, but predictable from run to run. This is convenient for debugging, but if you want a program to do different things each time it is used, you must change the seed to a value that will be different in each run. To do this, use `srand`.

`srand(`*x*`)` The function `srand` sets the starting point, or *seed*, for generating random numbers to the value *x*.

Each seed value leads to a particular sequence of "random" numbers. Thus, if you set the seed to the same value a second time, you will get the same sequence of "random" numbers again.

If you omit the argument *x*, as in `srand()`, then the current date and time of day are used for a seed. This is the way to get random numbers that are truly unpredictable.

The return value of `srand` is the previous seed. This makes it easy to keep track of the seeds for use in consistently reproducing sequences of random numbers.

11.3 Built-in Functions for String Manipulation

The functions in this section look at or change the text of one or more strings.

`index(`*in*`, `*find*`)`
This searches the string *in* for the first occurrence of the string *find*, and returns the position in characters where that occurrence begins in the string *in*. For example:

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

prints '`3`'. If *find* is not found, `index` returns 0. (Remember that string indices in `awk` start at 1.)

`length(`*string*`)`
This gives you the number of characters in *string*. If *string* is a number, the length of the digit string representing that number is returned. For example, `length("abcde")`

is 5. By contrast, `length(15 * 35)` works out to 3. How? Well, 15 * 35 = 525, and 525 is then converted to the string ‘`"525"`’, which has three characters.

If no argument is supplied, `length` returns the length of `$0`.

In older versions of `awk`, you could call the `length` function without any parentheses. Doing so is marked as “deprecated” in the POSIX standard. This means that while you can do this in your programs, it is a feature that can eventually be removed from a future version of the standard. Therefore, for maximal portability of your `awk` programs you should always supply the parentheses.

`match(`*string*`,` *regexp*`)`

The `match` function searches the string, *string*, for the longest, leftmost substring matched by the regular expression, *regexp*. It returns the character position, or *index*, of where that substring begins (1, if it starts at the beginning of *string*). If no match if found, it returns 0.

The `match` function sets the built-in variable `RSTART` to the index. It also sets the built-in variable `RLENGTH` to the length in characters of the matched substring. If no match is found, `RSTART` is set to 0, and `RLENGTH` to −1.

For example:

```
awk '{
       if ($1 == "FIND")
         regex = $2
       else {
         where = match($0, regex)
         if (where)
           print "Match of", regex, "found at", where, "in", $0
       }
}'
```

This program looks for lines that match the regular expression stored in the variable `regex`. This regular expression can be changed. If the first word on a line is ‘`FIND`’, `regex` is changed to be the second word on that line. Therefore, given:

```
FIND fo*bar
My program was a foobar
But none of it would doobar
FIND Melvin
JF+KM
This line is property of The Reality Engineering Co.
This file created by Melvin.
```

`awk` prints:

```
Match of fo*bar found at 18 in My program was a foobar
Match of Melvin found at 26 in This file created by Melvin.
```

`split(`*string*`,` *array*`,` *fieldsep*`)`

This divides *string* into pieces separated by *fieldsep*, and stores the pieces in *array*. The first piece is stored in *array*`[1]`, the second piece in *array*`[2]`, and so forth. The string value of the third argument, *fieldsep*, is a regexp describing where to split *string* (much as `FS` can be a regexp describing where to split input records). If the *fieldsep* is omitted, the value of `FS` is used. `split` returns the number of elements created.

The `split` function, then, splits strings into pieces in a manner similar to the way input lines are split into fields. For example:

```
split("auto-da-fe", a, "-")
```

splits the string ‘`auto-da-fe`’ into three fields using ‘`-`’ as the separator. It sets the contents of the array `a` as follows:

```
a[1] = "auto"
a[2] = "da"
a[3] = "fe"
```

The value returned by this call to `split` is 3.

As with input field-splitting, when the value of *fieldsep* is `" "`, leading and trailing whitespace is ignored, and the elements are separated by runs of whitespace.

`sprintf(`*format*`,` *expression1*`,`...`)`

This returns (without printing) the string that `printf` would have printed out with the same arguments (see Section 4.5 [Using `printf` Statements for Fancier Printing], page 38). For example:

```
sprintf("pi = %.2f (approx.)", 22/7)
```

returns the string `"pi = 3.14 (approx.)"`.

`sub(`*regexp*`,` *replacement*`,` *target*`)`

The `sub` function alters the value of *target*. It searches this value, which should be a string, for the leftmost substring matched by the regular expression, *regexp*, extending this match as far as possible. Then the entire string is changed by replacing the matched text with *replacement*. The modified string becomes the new value of *target*.

This function is peculiar because *target* is not simply used to compute a value, and not just any expression will do: it must be a variable, field or array reference, so that `sub` can store a modified value there. If this argument is omitted, then the default is to use and alter `$0`.

For example:

```
str = "water, water, everywhere"
sub(/at/, "ith", str)
```

sets `str` to `"wither, water, everywhere"`, by replacing the leftmost, longest occurrence of '`at`' with '`ith`'.

The `sub` function returns the number of substitutions made (either one or zero).

If the special character '`&`' appears in *replacement*, it stands for the precise substring that was matched by *regexp*. (If the regexp can match more than one string, then this precise substring may vary.) For example:

```
awk '{ sub(/candidate/, "& and his wife"); print }'
```

changes the first occurrence of '`candidate`' to '`candidate and his wife`' on each input line.

Here is another example:

```
awk 'BEGIN {
        str = "daabaaa"
        sub(/a*/, "c&c", str)
        print str
}'
```

prints '`dcaacbaaa`'. This show how '`&`' can represent a non-constant string, and also illustrates the "leftmost, longest" rule.

The effect of this special character ('`&`') can be turned off by putting a backslash before it in the string. As usual, to insert one backslash in the string, you must write two backslashes. Therefore, write '`\\&`' in a string constant to include a literal '`&`' in the replacement. For example, here is how to replace the first '`|`' on each line with an '`&`':

```
awk '{ sub(/\|/, "\\&"); print }'
```

Note: as mentioned above, the third argument to `sub` must be an lvalue. Some versions of `awk` allow the third argument to be an expression which is not an lvalue. In such a case, `sub` would still search for the pattern and return 0 or 1, but the result of the substitution (if any) would be thrown away because there is no place to put it. Such versions of `awk` accept expressions like this:

```
sub(/USA/, "United States", "the USA and Canada")
```

But that is considered erroneous in `gawk`.

`gsub(`*regexp*`,` *replacement*`,` *target*`)`
: This is similar to the `sub` function, except `gsub` replaces *all* of the longest, leftmost, *nonoverlapping* matching substrings it can find. The 'g' in `gsub` stands for "global," which means replace everywhere. For example:

```
awk '{ gsub(/Britain/, "United Kingdom"); print }'
```

replaces all occurrences of the string '`Britain`' with '`United Kingdom`' for all input records.

The `gsub` function returns the number of substitutions made. If the variable to be searched and altered, *target*, is omitted, then the entire input record, `$0`, is used.

As in `sub`, the characters '`&`' and '`\`' are special, and the third argument must be an lvalue.

`substr(`*string*`,` *start*`,` *length*`)`
: This returns a *length*-character-long substring of *string*, starting at character number *start*. The first character of a string is character number one. For example, `substr("washington", 5, 3)` returns `"ing"`.

If *length* is not present, this function returns the whole suffix of *string* that begins at character number *start*. For example, `substr("washington", 5)` returns `"ington"`. This is also the case if *length* is greater than the number of characters remaining in the string, counting from character number *start*.

`tolower(`*string*`)`
: This returns a copy of *string*, with each upper-case character in the string replaced with its corresponding lower-case character. Nonalphabetic characters are left unchanged. For example, `tolower("MiXeD cAsE 123")` returns `"mixed case 123"`.

`toupper(`*string*`)`
: This returns a copy of *string*, with each lower-case character in the string replaced with its corresponding upper-case character. Nonalphabetic characters are left unchanged. For example, `toupper("MiXeD cAsE 123")` returns `"MIXED CASE 123"`.

11.4 Built-in Functions for Input/Output

`close(`*filename*`)`
: Close the file *filename*, for input or output. The argument may alternatively be a shell command that was used for redirecting to or from a pipe; then the pipe is closed.

See Section 3.8 [Closing Input Files and Pipes], page 33, regarding closing input files and pipes. See Section 4.6.2 [Closing Output Files and Pipes], page 43, regarding closing output files and pipes.

`system(`*command*`)`
: The system function allows the user to execute operating system commands and then return to the `awk` program. The `system` function executes the command given by the string *command*. It returns, as its value, the status returned by the command that was executed.

For example, if the following fragment of code is put in your `awk` program:

```
END {
     system("mail -s 'awk run done' operator < /dev/null")
}
```

the system operator will be sent mail when the `awk` program finishes processing input and begins its end-of-input processing.

Note that much the same result can be obtained by redirecting `print` or `printf` into a pipe. However, if your `awk` program is interactive, `system` is useful for cranking up large self-contained programs, such as a shell or an editor.

Some operating systems cannot implement the `system` function. `system` causes a fatal error if it is not supported.

Controlling Output Buffering with `system`

Many utility programs will *buffer* their output; they save information to be written to a disk file or terminal in memory, until there is enough to be written in one operation. This is often more efficient than writing every little bit of information as soon as it is ready. However, sometimes it is necessary to force a program to *flush* its buffers; that is, write the information to its destination, even if a buffer is not full. You can do this from your `awk` program by calling `system` with a null string as its argument:

```
system("")   # flush output
```

12 User-defined Functions

Complicated `awk` programs can often be simplified by defining your own functions. User-defined functions can be called just like built-in ones (see Section 8.12 [Function Calls], page 70), but it is up to you to define them—to tell `awk` what they should do.

12.1 Syntax of Function Definitions

Definitions of functions can appear anywhere between the rules of the `awk` program. Thus, the general form of an `awk` program is extended to include sequences of rules *and* user-defined function definitions.

The definition of a function named *name* looks like this:

```
function name (parameter-list) {
     body-of-function
}
```

name is the name of the function to be defined. A valid function name is like a valid variable name: a sequence of letters, digits and underscores, not starting with a digit. Functions share the same pool of names as variables and arrays.

parameter-list is a list of the function's arguments and local variable names, separated by commas. When the function is called, the argument names are used to hold the argument values given in the call. The local variables are initialized to the null string.

The *body-of-function* consists of `awk` statements. It is the most important part of the definition, because it says what the function should actually *do*. The argument names exist to give the body a way to talk about the arguments; local variables, to give the body places to keep temporary values.

Argument names are not distinguished syntactically from local variable names; instead, the number of arguments supplied when the function is called determines how many argument variables there are. Thus, if three argument values are given, the first three names in *parameter-list* are arguments, and the rest are local variables.

It follows that if the number of arguments is not the same in all calls to the function, some of the names in *parameter-list* may be arguments on some occasions and local variables on others. Another way to think of this is that omitted arguments default to the null string.

Usually when you write a function you know how many names you intend to use for arguments and how many you intend to use as locals. By convention, you should write an extra space between the arguments and the locals, so other people can follow how your function is supposed to be used.

During execution of the function body, the arguments and local variable values hide or *shadow* any variables of the same names used in the rest of the program. The shadowed variables are not accessible in the function definition, because there is no way to name them while their names have been taken away for the local variables. All other variables used in the `awk` program can be referenced or set normally in the function definition.

The arguments and local variables last only as long as the function body is executing. Once the body finishes, the shadowed variables come back.

The function body can contain expressions which call functions. They can even call this function, either directly or by way of another function. When this happens, we say the function is *recursive*.

There is no need in `awk` to put the definition of a function before all uses of the function. This is because `awk` reads the entire program before starting to execute any of it.

12.2 Function Definition Example

Here is an example of a user-defined function, called `myprint`, that takes a number and prints it in a specific format.

```
function myprint(num)
{
     printf "%6.3g\n", num
}
```

To illustrate, here is an `awk` rule which uses our `myprint` function:

```
$3 > 0     { myprint($3) }
```

This program prints, in our special format, all the third fields that contain a positive number in our input. Therefore, when given:

```
 1.2   3.4    5.6   7.8
 9.10 11.12 -13.14 15.16
17.18 19.20  21.22 23.24
```

this program, using our function to format the results, prints:

```
   5.6
  21.2
```

Here is a rather contrived example of a recursive function. It prints a string backwards:

```
function rev (str, len) {
    if (len == 0) {
        printf "\n"
        return
    }
    printf "%c", substr(str, len, 1)
    rev(str, len - 1)
}
```

12.3 Calling User-defined Functions

Calling a function means causing the function to run and do its job. A function call is an expression, and its value is the value returned by the function.

A function call consists of the function name followed by the arguments in parentheses. What you write in the call for the arguments are `awk` expressions; each time the call is executed, these expressions are evaluated, and the values are the actual arguments. For example, here is a call to `foo` with three arguments (the first being a string concatenation):

```
foo(x y, "lose", 4 * z)
```

> **Caution:** whitespace characters (spaces and tabs) are not allowed between the function name and the open-parenthesis of the argument list. If you write whitespace by mistake, `awk` might think that you mean to concatenate a variable with an expression in parentheses. However, it notices that you used a function name and not a variable name, and reports an error.

When a function is called, it is given a *copy* of the values of its arguments. This is called *call by value*. The caller may use a variable as the expression for the argument, but the called function does not know this: it only knows what value the argument had. For example, if you write this code:

```
foo = "bar"
z = myfunc(foo)
```

then you should not think of the argument to `myfunc` as being "the variable `foo`." Instead, think of the argument as the string value, `"bar"`.

If the function `myfunc` alters the values of its local variables, this has no effect on any other variables. In particular, if `myfunc` does this:

```
function myfunc (win) {
  print win
  win = "zzz"
  print win
}
```

to change its first argument variable `win`, this *does not* change the value of `foo` in the caller. The role of `foo` in calling `myfunc` ended when its value, `"bar"`, was computed. If `win` also exists outside of `myfunc`, the function body cannot alter this outer value, because it is shadowed during the execution of `myfunc` and cannot be seen or changed from there.

However, when arrays are the parameters to functions, they are *not* copied. Instead, the array itself is made available for direct manipulation by the function. This is usually called *call by reference*. Changes made to an array parameter inside the body of a function *are* visible outside that function. *This can be very dangerous if you do not watch what you are doing.* For example:

```
function changeit (array, ind, nvalue) {
     array[ind] = nvalue
}

BEGIN {
           a[1] = 1 ; a[2] = 2 ; a[3] = 3
           changeit(a, 2, "two")
           printf "a[1] = %s, a[2] = %s, a[3] = %s\n", a[1], a[2], a[3]
      }
```

prints '`a[1] = 1, a[2] = two, a[3] = 3`', because calling `changeit` stores `"two"` in the second element of `a`.

12.4 The `return` Statement

The body of a user-defined function can contain a `return` statement. This statement returns control to the rest of the `awk` program. It can also be used to return a value for use in the rest of the `awk` program. It looks like this:

`return` *expression*

The *expression* part is optional. If it is omitted, then the returned value is undefined and, therefore, unpredictable.

A `return` statement with no value expression is assumed at the end of every function definition. So if control reaches the end of the function body, then the function returns an unpredictable value. `awk` will not warn you if you use the return value of such a function; you will simply get unpredictable or unexpected results.

Here is an example of a user-defined function that returns a value for the largest number among the elements of an array:

```
function maxelt (vec,   i, ret) {
     for (i in vec) {
          if (ret == "" || vec[i] > ret)
               ret = vec[i]
     }
     return ret
}
```

You call `maxelt` with one argument, which is an array name. The local variables `i` and `ret` are not intended to be arguments; while there is nothing to stop you from passing two or three arguments to `maxelt`, the results would be strange. The extra space before `i` in the function parameter list is to indicate that `i` and `ret` are not supposed to be arguments. This is a convention which you should follow when you define functions.

Here is a program that uses our `maxelt` function. It loads an array, calls `maxelt`, and then reports the maximum number in that array:

```
awk '
function maxelt (vec,   i, ret) {
     for (i in vec) {
          if (ret == "" || vec[i] > ret)
               ret = vec[i]
     }
     return ret
}

# Load all fields of each record into nums.
{
          for(i = 1; i <= NF; i++)
               nums[NR, i] = $i
}

END {
     print maxelt(nums)
}'
```

Given the following input:

```
 1 5 23 8 16
44 3 5 2 8 26
256 291 1396 2962 100
-6 467 998 1101
99385 11 0 225
```

our program tells us (predictably) that:

```
99385
```

is the largest number in our array.

13 Built-in Variables

Most `awk` variables are available for you to use for your own purposes; they never change except when your program assigns values to them, and never affect anything except when your program examines them.

A few variables have special built-in meanings. Some of them `awk` examines automatically, so that they enable you to tell `awk` how to do certain things. Others are set automatically by `awk`, so that they carry information from the internal workings of `awk` to your program.

This chapter documents all the built-in variables of `awk`. Most of them are also documented in the chapters where their areas of activity are described.

13.1 Built-in Variables that Control `awk`

This is a list of the variables which you can change to control how `awk` does certain things.

`CONVFMT`
: This string is used by `awk` to control conversion of numbers to strings (see Section 8.9 [Conversion of Strings and Numbers], page 67). It works by being passed, in effect, as the first argument to the `sprintf` function. Its default value is `"%.6g"`. `CONVFMT` was introduced by the POSIX standard.

`FS`
: `FS` is the input field separator (see Section 3.5 [Specifying how Fields are Separated], page 25). The value is a single-character string or a multi-character regular expression that matches the separations between fields in an input record.

  The default value is `" "`, a string consisting of a single space. As a special exception, this value actually means that any sequence of spaces and tabs is a single separator. It also causes spaces and tabs at the beginning or end of a line to be ignored.

  You can set the value of `FS` on the command line using the '`-F`' option:

  ```
  awk -F, 'program' input-files
  ```

`OFMT`
: This string is used by `awk` to control conversion of numbers to strings (see Section 8.9 [Conversion of Strings and Numbers], page 67) for printing with the `print` statement. It works by being passed, in effect, as the first argument to the `sprintf` function. Its default value is `"%.6g"`. Earlier versions of `awk` also used `OFMT` to specify the format for converting numbers to strings in general expressions; this has been taken over by `CONVFMT`.

`OFS`
: This is the output field separator (see Section 4.3 [Output Separators], page 37). It is output between the fields output by a `print` statement. Its default value is `" "`, a string consisting of a single space.

`ORS`
: This is the output record separator. It is output at the end of every `print` statement. Its default value is a string containing a single newline character, which could be written as `"\n"`. (See Section 4.3 [Output Separators], page 37.)

`RS`
: This is `awk`'s input record separator. Its default value is a string containing a single newline character, which means that an input record consists of a single line of text. (See Section 3.1 [How Input is Split into Records], page 21.)

`SUBSEP`
: `SUBSEP` is the subscript separator. It has the default value of `"\034"`, and is used to separate the parts of the name of a multi-dimensional array. Thus, if you access

foo[12,3], it really accesses foo["12\0343"] (see Section 10.8 [Multi-dimensional Arrays], page 86).

13.2 Built-in Variables that Convey Information

This is a list of the variables that are set automatically by awk on certain occasions so as to provide information to your program.

ARGC
ARGV
: The command-line arguments available to awk programs are stored in an array called ARGV. ARGC is the number of command-line arguments present. See Chapter 14 [Invoking awk], page 105. ARGV is indexed from zero to ARGC - 1. For example:

```
awk 'BEGIN {
       for (i = 0; i < ARGC; i++)
           print ARGV[i]
     }' inventory-shipped BBS-list
```

In this example, ARGV[0] contains "awk", ARGV[1] contains "inventory-shipped", and ARGV[2] contains "BBS-list". The value of ARGC is 3, one more than the index of the last element in ARGV since the elements are numbered from zero.

The names ARGC and ARGV, as well the convention of indexing the array from 0 to ARGC - 1, are derived from the C language's method of accessing command line arguments.

Notice that the awk program is not entered in ARGV. The other special command line options, with their arguments, are also not entered. But variable assignments on the command line *are* treated as arguments, and do show up in the ARGV array.

Your program can alter ARGC and the elements of ARGV. Each time awk reaches the end of an input file, it uses the next element of ARGV as the name of the next input file. By storing a different string there, your program can change which files are read. You can use "-" to represent the standard input. By storing additional elements and incrementing ARGC you can cause additional files to be read.

If you decrease the value of ARGC, that eliminates input files from the end of the list. By recording the old value of ARGC elsewhere, your program can treat the eliminated arguments as something other than file names.

To eliminate a file from the middle of the list, store the null string ("") into ARGV in place of the file's name. As a special feature, awk ignores file names that have been replaced with the null string.

ENVIRON
: This is an array that contains the values of the environment. The array indices are the environment variable names; the values are the values of the particular environment variables. For example, ENVIRON["HOME"] might be '/u/close'. Changing this array does not affect the environment passed on to any programs that awk may spawn via redirection or the system function.

Some operating systems may not have environment variables. On such systems, the array ENVIRON is empty.

FILENAME
: This is the name of the file that awk is currently reading. If awk is reading from the standard input (in other words, there are no files listed on the command line), FILENAME is set to "-". FILENAME is changed each time a new file is read (see Chapter 3 [Reading Input Files], page 21).

`FNR` `FNR` is the current record number in the current file. `FNR` is incremented each time a new record is read (see Section 3.7 [Explicit Input with `getline`], page 30). It is reinitialized to 0 each time a new input file is started.

`NF` `NF` is the number of fields in the current input record. `NF` is set each time a new record is read, when a new field is created, or when `$0` changes (see Section 3.2 [Examining Fields], page 22).

`NR` This is the number of input records `awk` has processed since the beginning of the program's execution. (see Section 3.1 [How Input is Split into Records], page 21). `NR` is set each time a new record is read.

`RLENGTH` `RLENGTH` is the length of the substring matched by the `match` function (see Section 11.3 [Built-in Functions for String Manipulation], page 90). `RLENGTH` is set by invoking the `match` function. Its value is the length of the matched string, or −1 if no match was found.

`RSTART` `RSTART` is the start-index in characters of the substring matched by the `match` function (see Section 11.3 [Built-in Functions for String Manipulation], page 90). `RSTART` is set by invoking the `match` function. Its value is the position of the string where the matched substring starts, or 0 if no match was found.

14 Invoking awk

There are two ways to run awk: with an explicit program, or with one or more program files. Here are templates for both of them; items enclosed in '[...]' in these templates are optional.

14.1 Command Line Options

Options begin with a minus sign, and consist of a single character. If the option takes an argument, then the keyword is immediately followed by an equals sign ('=') and the argument's value. For brevity, the discussion below only refers to the traditional short options; however the long and short options are interchangeable in all contexts.

`-F` *fs* Sets the FS variable to *fs* (see Section 3.5 [Specifying how Fields are Separated], page 25).

`-f` *source-file*
Indicates that the awk program is to be found in *source-file* instead of in the first non-option argument.

`-v` *var*=*val*
Sets the variable *var* to the value *val before* execution of the program begins. Such variable values are available inside the BEGIN rule (see below for a fuller explanation).

The '-v' option can only set one variable, but you can use it more than once, setting another variable each time, like this: '-v foo=1 -v bar=2'.

Any other options are flagged as invalid with a warning message, but are otherwise ignored.

If the '-f' option is *not* used, then the first non-option command line argument is expected to be the program text.

The '-f' option may be used more than once on the command line. If it is, awk reads its program source from all of the named files, as if they had been concatenated together into one big file. This is useful for creating libraries of awk functions. Useful functions can be written once, and then retrieved from a standard place, instead of having to be included into each individual program. You can still type in a program at the terminal and use library functions, by specifying '-f /dev/tty'. awk will read a file from the terminal to use as part of the awk program. After typing your program, type *Control-d* (the end-of-file character) to terminate it. (You may also use '-f -' to read program source from the standard input, but then you will not be able to also use the standard input as a source of data.)

14.2 Other Command Line Arguments

Any additional arguments on the command line are normally treated as input files to be processed in the order specified. However, an argument that has the form *var*=*value*, means to assign the value *value* to the variable *var*—it does not specify a file at all.

All these arguments are made available to your awk program in the ARGV array (see Chapter 13 [Built-in Variables], page 101). Command line options and the program text (if present) are omitted from the ARGV array. All other arguments, including variable assignments, are included.

The distinction between file name arguments and variable-assignment arguments is made when `awk` is about to open the next input file. At that point in execution, it checks the "file name" to see whether it is really a variable assignment; if so, `awk` sets the variable instead of reading a file.

Therefore, the variables actually receive the specified values after all previously specified files have been read. In particular, the values of variables assigned in this fashion are *not* available inside a `BEGIN` rule (see Section 6.7 [`BEGIN` and `END` Special Patterns], page 53), since such rules are run before `awk` begins scanning the argument list. The values given on the command line are processed for escape sequences (see Section 8.1 [Constant Expressions], page 57).

In some earlier implementations of `awk`, when a variable assignment occurred before any file names, the assignment would happen *before* the `BEGIN` rule was executed. Some applications came to depend upon this "feature." When `awk` was changed to be more consistent, the '`-v`' option was added to accommodate applications that depended upon this old behavior.

The variable assignment feature is most useful for assigning to variables such as `RS`, `OFS`, and `ORS`, which control input and output formats, before scanning the data files. It is also useful for controlling state if multiple passes are needed over a data file. For example:

```
awk 'pass == 1  { pass 1 stuff }
     pass == 2  { pass 2 stuff }' pass=1 datafile pass=2 datafile
```

Given the variable assignment feature, the '`-F`' option is not strictly necessary. It remains for historical compatibility.

Appendix A awk Summary

This appendix provides a brief summary of the `awk` command line and the `awk` language. It is designed to serve as "quick reference." It is therefore terse, but complete.

A.1 Command Line Options Summary

The command line consists of options to `awk` itself, the `awk` program text (if not supplied via the '`-f`' option), and values to be made available in the `ARGC` and `ARGV` predefined `awk` variables:

```
awk [options] -f source-file [--] file ...
awk [options] [--] 'program' file ...
```

The options that `awk` accepts are:

`-F` *fs*
: Use *fs* for the input field separator (the value of the `FS` predefined variable).

`-f` *program-file*
: Read the `awk` program source from the file *program-file*, instead of from the first command line argument.

`-v` *var*=*val*
: Assign the variable *var* the value *val* before program execution begins.

`--`
: Signal the end of options. This is useful to allow further arguments to the `awk` program itself to start with a '`-`'. This is mainly for consistency with the argument parsing conventions of POSIX.

Any other options are flagged as invalid, but are otherwise ignored. See Chapter 14 [Invoking `awk`], page 105, for more details.

A.2 Language Summary

An `awk` program consists of a sequence of pattern-action statements and optional function definitions.

```
pattern     { action statements }

function name(parameter list)     { action statements }
```

`awk` first reads the program source from the *program-file*(s) if specified, or from the first non-option argument on the command line. The '`-f`' option may be used multiple times on the command line. `awk` reads the program text from all the *program-file* files, effectively concatenating them in the order they are specified. This is useful for building libraries of `awk` functions, without having to include them in each new `awk` program that uses them. To use a library function in a file from a program typed in on the command line, specify '`-f /dev/tty`'; then type your program, and end it with a *`Control-d`*. See Chapter 14 [Invoking `awk`], page 105.

awk compiles the program into an internal form, and then proceeds to read each file named in the ARGV array. If there are no files named on the command line, awk reads the standard input.

If a "file" named on the command line has the form '*var*=*val*', it is treated as a variable assignment: the variable *var* is assigned the value *val*. If any of the files have a value that is the null string, that element in the list is skipped.

For each line in the input, awk tests to see if it matches any *pattern* in the awk program. For each pattern that the line matches, the associated *action* is executed.

A.3 Variables and Fields

awk variables are dynamic; they come into existence when they are first used. Their values are either floating-point numbers or strings. awk also has one-dimension arrays; multiple-dimensional arrays may be simulated. There are several predefined variables that awk sets as a program runs; these are summarized below.

A.3.1 Fields

As each input line is read, awk splits the line into *fields*, using the value of the FS variable as the field separator. If FS is a single character, fields are separated by that character. Otherwise, FS is expected to be a full regular expression. In the special case that FS is a single blank, fields are separated by runs of blanks and/or tabs.

Each field in the input line may be referenced by its position, $1, $2, and so on. $0 is the whole line. The value of a field may be assigned to as well. Field numbers need not be constants:

```
n = 5
print $n
```

prints the fifth field in the input line. The variable NF is set to the total number of fields in the input line.

References to nonexistent fields (i.e., fields after $NF) return the null-string. However, assigning to a nonexistent field (e.g., $(NF+2) = 5) increases the value of NF, creates any intervening fields with the null string as their value, and causes the value of $0 to be recomputed, with the fields being separated by the value of OFS.

See Chapter 3 [Reading Input Files], page 21, for a full description of the way awk defines and uses fields.

A.3.2 Built-in Variables

awk's built-in variables are:

ARGC The number of command line arguments (not including options or the awk program itself).

`ARGV`	The array of command line arguments. The array is indexed from 0 to `ARGC` − 1. Dynamically changing the contents of `ARGV` can control the files used for data.
`CONVFMT`	The conversion format to use when converting numbers to strings.
`ENVIRON`	An array containing the values of the environment variables. The array is indexed by variable name, each element being the value of that variable. Thus, the environment variable `HOME` would be in `ENVIRON["HOME"]`. Its value might be '`/u/close`'. Changing this array does not affect the environment seen by programs which `awk` spawns via redirection or the `system` function. Some operating systems do not have environment variables. The array `ENVIRON` is empty when running on these systems.
`FILENAME`	The name of the current input file. If no files are specified on the command line, the value of `FILENAME` is '`-`'.
`FNR`	The input record number in the current input file.
`FS`	The input field separator, a blank by default.
`NF`	The number of fields in the current input record.
`NR`	The total number of input records seen so far.
`OFMT`	The output format for numbers for the `print` statement, `"%.6g"` by default.
`OFS`	The output field separator, a blank by default.
`ORS`	The output record separator, by default a newline.
`RS`	The input record separator, by default a newline. `RS` is exceptional in that only the first character of its string value is used for separating records. If `RS` is set to the null string, then records are separated by blank lines. When `RS` is set to the null string, then the newline character always acts as a field separator, in addition to whatever value `FS` may have.
`RSTART`	The index of the first character matched by `match`; 0 if no match.
`RLENGTH`	The length of the string matched by `match`; −1 if no match.
`SUBSEP`	The string used to separate multiple subscripts in array elements, by default `"\034"`.

See Chapter 13 [Built-in Variables], page 101, for more information.

A.3.3 Arrays

Arrays are subscripted with an expression between square brackets ('`[`' and '`]`'). Array subscripts are *always* strings; numbers are converted to strings as necessary, following the standard conversion rules (see Section 8.9 [Conversion of Strings and Numbers], page 67).

If you use multiple expressions separated by commas inside the square brackets, then the array subscript is a string consisting of the concatenation of the individual subscript values, converted to strings, separated by the subscript separator (the value of `SUBSEP`).

The special operator `in` may be used in an `if` or `while` statement to see if an array has an index consisting of a particular value.

```
if (val in array)
        print array[val]
```

If the array has multiple subscripts, use `(i, j, ...) in array` to test for existence of an element.

The `in` construct may also be used in a `for` loop to iterate over all the elements of an array. See Section 10.5 [Scanning all Elements of an Array], page 84.

An element may be deleted from an array using the `delete` statement.

See Chapter 10 [Arrays in `awk`], page 81, for more detailed information.

A.3.4 Data Types

The value of an `awk` expression is always either a number or a string.

Certain contexts (such as arithmetic operators) require numeric values. They convert strings to numbers by interpreting the text of the string as a numeral. If the string does not look like a numeral, it converts to 0.

Certain contexts (such as concatenation) require string values. They convert numbers to strings by effectively printing them with `sprintf`. See Section 8.9 [Conversion of Strings and Numbers], page 67, for the details.

To force conversion of a string value to a number, simply add 0 to it. If the value you start with is already a number, this does not change it.

To force conversion of a numeric value to a string, concatenate it with the null string.

The `awk` language defines comparisons as being done numerically if both operands are numeric, or if one is numeric and the other is a numeric string. Otherwise one or both operands are converted to strings and a string comparison is performed.

Uninitialized variables have the string value `""` (the null, or empty, string). In contexts where a number is required, this is equivalent to 0.

See Section 8.2 [Variables], page 59, for more information on variable naming and initialization; see Section 8.9 [Conversion of Strings and Numbers], page 67, for more information on how variable values are interpreted.

A.4 Patterns and Actions

An `awk` program is mostly composed of rules, each consisting of a pattern followed by an action. The action is enclosed in ‘`{`’ and ‘`}`’. Either the pattern may be missing, or the action may be missing, but, of course, not both. If the pattern is missing, the action is executed for every single line of input. A missing action is equivalent to this action,

```
{ print }
```

which prints the entire line.

Comments begin with the '`#`' character, and continue until the end of the line. Blank lines may be used to separate statements. Normally, a statement ends with a newline, however, this is not the case for lines ending in a '`,`', '`{`', '`?`', '`:`', '`&&`', or '`||`'. Lines ending in `do` or `else` also have their statements automatically continued on the following line. In other cases, a line can be continued by ending it with a '\', in which case the newline is ignored.

Multiple statements may be put on one line by separating them with a '`;`'. This applies to both the statements within the action part of a rule (the usual case), and to the rule statements.

See Section 2.5 [Comments in `awk` Programs], page 18, for information on `awk`'s commenting convention; see Section 2.6 [`awk` Statements versus Lines], page 19, for a description of the line continuation mechanism in `awk`.

A.4.1 Patterns

`awk` patterns may be one of the following:

`/`*regular expression*`/`
relational expression
pattern `&&` *pattern*
pattern `||` *pattern*
pattern `?` *pattern* `:` *pattern*
`(`*pattern*`)`
`!` *pattern*
pattern1`,` *pattern2*
`BEGIN`
`END`

`BEGIN` and `END` are two special kinds of patterns that are not tested against the input. The action parts of all `BEGIN` rules are merged as if all the statements had been written in a single `BEGIN` rule. They are executed before any of the input is read. Similarly, all the `END` rules are merged, and executed when all the input is exhausted (or when an `exit` statement is executed). `BEGIN` and `END` patterns cannot be combined with other patterns in pattern expressions. `BEGIN` and `END` rules cannot have missing action parts.

For '`/`*regular-expression*`/`' patterns, the associated statement is executed for each input line that matches the regular expression. Regular expressions are extensions of those in `egrep`, and are summarized below.

A *relational expression* may use any of the operators defined below in the section on actions. These generally test whether certain fields match certain regular expressions.

The '`&&`', '`||`', and '`!`' operators are logical "and," logical "or," and logical "not," respectively, as in C. They do short-circuit evaluation, also as in C, and are used for combining more primitive pattern expressions. As in most languages, parentheses may be used to change the order of evaluation.

The '`?:`' operator is like the same operator in C. If the first pattern matches, then the second pattern is matched against the input record; otherwise, the third is matched. Only one of the second and third patterns is matched.

The '*pattern1*`,` *pattern2*' form of a pattern is called a range pattern. It matches all input lines starting with a line that matches *pattern1*, and continuing until a line that matches *pattern2*, inclusive. A range pattern cannot be used as an operand to any of the pattern operators.

See Chapter 6 [Patterns], page 47, for a full description of the pattern part of `awk` rules.

A.4.2 Regular Expressions

Regular expressions are the extended kind found in `egrep`. They are composed of characters as follows:

c	matches the character *c* (assuming *c* is a character with no special meaning in regexps).
`\`*c*	matches the literal character *c*.
`.`	matches any character except newline.
`^`	matches the beginning of a line or a string.
`$`	matches the end of a line or a string.
`[`*abc*...`]`	matches any of the characters *abc*... (character class).
`[^`*abc*...`]`	matches any character except *abc*... and newline (negated character class).
r1`|`*r2*	matches either *r1* or *r2* (alternation).
r1r2	matches *r1*, and then *r2* (concatenation).
r`+`	matches one or more *r*'s.
r`*`	matches zero or more *r*'s.
r`?`	matches zero or one *r*'s.
`(`*r*`)`	matches *r* (grouping).

See Section 6.2 [Regular Expressions as Patterns], page 47, for a more detailed explanation of regular expressions.

The escape sequences allowed in string constants are also valid in regular expressions (see Section 8.1 [Constant Expressions], page 57).

A.4.3 Actions

Action statements are enclosed in braces, '`{`' and '`}`'. Action statements consist of the usual assignment, conditional, and looping statements found in most languages. The operators, control statements, and input/output statements available are patterned after those in C.

A.4.3.1 Operators

The operators in `awk`, in order of increasing precedence, are:

`= += -= *= /= %= ^=`	Assignment. Both absolute assignment (*var*`=`*value*) and operator assignment (the other forms) are supported.
`?:`	A conditional expression, as in C. This has the form *expr1* `?` *expr2* `:` *expr3*. If *expr1* is true, the value of the expression is *expr2*; otherwise it is *expr3*. Only one of *expr2* and *expr3* is evaluated.
`||`	Logical "or".
`&&`	Logical "and".
`~ !~`	Regular expression match, negated match.
`< <= > >= != ==`	The usual relational operators.
blank	String concatenation.
`+ -`	Addition and subtraction.
`* / %`	Multiplication, division, and modulus.
`+ - !`	Unary plus, unary minus, and logical negation.
`^`	Exponentiation ('`**`' may also be used, and '`**=`' for the assignment operator, but they are not specified in the POSIX standard).
`++ --`	Increment and decrement, both prefix and postfix.
`$`	Field reference.

See Chapter 8 [Expressions as Action Statements], page 57, for a full description of all the operators listed above. See Section 3.2 [Examining Fields], page 22, for a description of the field reference operator.

A.4.3.2 Control Statements

The control statements are as follows:

```
if (condition) statement [ else statement ]
while (condition) statement
do statement while (condition)
for (expr1; expr2; expr3) statement
for (var in array) statement
break
continue
delete array[index]
exit [ expression ]
{ statements }
```

See Chapter 9 [Control Statements in Actions], page 73, for a full description of all the control statements listed above.

A.4.3.3 I/O Statements

The input/output statements are as follows:

`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 `NF`, `FNR`.

`getline` *var* `<`*file*
: Set *var* from next record of *file*.

`next`
: Stop processing the current input record. The next input record is read and processing starts over with the first pattern in the `awk` program. If the end of the input data is reached, the `END` rule(s), if any, are executed.

`print`
: Prints the current record.

`print` *expr-list*
: Prints expressions.

`print` *expr-list* `>` *file*
: Prints expressions on *file*.

`printf` *fmt, expr-list*
: Format and print.

`printf` *fmt, expr-list* `> file`
: Format and print on *file*.

Other input/output redirections are also allowed. For `print` and `printf`, '`>>` *file*' appends output to the *file*, and '`|` *command*' writes on a pipe. In a similar fashion, '*command* `| getline`' pipes input into `getline`. `getline` returns 0 on end of file, and −1 on an error.

See Section 3.7 [Explicit Input with `getline`], page 30, for a full description of the `getline` statement. See Chapter 4 [Printing Output], page 35, for a full description of `print` and `printf`. Finally, see Section 9.7 [The `next` Statement], page 78, for a description of how the `next` statement works.

A.4.3.4 `printf` Summary

The `awk` `printf` statement and `sprintf` function accept the following conversion specification formats:

`%c`
: An ASCII character. If the argument used for '`%c`' is numeric, it is treated as a character and printed. Otherwise, the argument is assumed to be a string, and the only first character of that string is printed.

`%d`
`%i`
: A decimal number (the integer part).

`%e`
: A floating point number of the form '[-]`d.ddddddE`[+-]`dd`'.

`%f`
: A floating point number of the form [-]`ddd.dddddd`.

`%g`	Use ‘`%e`’ or ‘`%f`’ conversion, whichever produces a shorter string, with nonsignificant zeros suppressed.
`%o`	An unsigned octal number (again, an integer).
`%s`	A character string.
`%x`	An unsigned hexadecimal number (an integer).
`%X`	Like ‘`%x`’, except use ‘`A`’ through ‘`F`’ instead of ‘`a`’ through ‘`f`’ for decimal 10 through 15.
`%%`	A single ‘`%`’ character; no argument is converted.

There are optional, additional parameters that may lie between the ‘`%`’ and the control letter:

`-`	The expression should be left-justified within its field.
width	The field should be padded to this width. If *width* has a leading zero, then the field is padded with zeros. Otherwise it is padded with blanks.
`.`*prec*	A number indicating the maximum width of strings or digits to the right of the decimal point.

Either or both of the *width* and *prec* values may be specified as ‘`*`’. In that case, the particular value is taken from the argument list.

See Section 4.5 [Using `printf` Statements for Fancier Printing], page 38, for examples and for a more detailed description.

A.4.3.5 Numeric Functions

`awk` has the following predefined arithmetic functions:

`atan2(`*y*`,` *x*`)`	returns the arctangent of *y/x* in radians.
`cos(`*expr*`)`	returns the cosine in radians.
`exp(`*expr*`)`	the exponential function.
`int(`*expr*`)`	truncates to integer.
`log(`*expr*`)`	the natural logarithm function.
`rand()`	returns a random number between 0 and 1.
`sin(`*expr*`)`	returns the sine in radians.
`sqrt(`*expr*`)`	the square root function.
`srand(`*expr*`)`	use *expr* as a new seed for the random number generator. If no *expr* is provided, the time of day is used. The return value is the previous seed for the random number generator.

A.4.3.6 String Functions

`awk` has the following predefined string functions:

`gsub(`*r*`,` *s*`,` *t*`)`
: for each substring matching the regular expression *r* in the string *t*, substitute the string *s*, and return the number of substitutions. If *t* is not supplied, use `$0`.

`index(`*s*`,` *t*`)`
: returns the index of the string *t* in the string *s*, or 0 if *t* is not present.

`length(`*s*`)`
: returns the length of the string *s*. The length of `$0` is returned if no argument is supplied.

`match(`*s*`,` *r*`)`
: returns the position in *s* where the regular expression *r* occurs, or 0 if *r* is not present, and sets the values of `RSTART` and `RLENGTH`.

`split(`*s*`,` *a*`,` *r*`)`
: splits the string *s* into the array *a* on the regular expression *r*, and returns the number of fields. If *r* is omitted, `FS` is used instead.

`sprintf(`*fmt*`,` *expr-list*`)`
: prints *expr-list* according to *fmt*, and returns the resulting string.

`sub(`*r*`,` *s*`,` *t*`)`
: this is just like `gsub`, but only the first matching substring is replaced.

`substr(`*s*`,` *i*`,` *n*`)`
: returns the *n*-character substring of *s* starting at *i*. If *n* is omitted, the rest of *s* is used.

`tolower(`*str*`)`
: returns a copy of the string *str*, with all the upper-case characters in *str* translated to their corresponding lower-case counterparts. Nonalphabetic characters are left unchanged.

`toupper(`*str*`)`
: returns a copy of the string *str*, with all the lower-case characters in *str* translated to their corresponding upper-case counterparts. Nonalphabetic characters are left unchanged.

`system(`*cmd-line*`)`
: Execute the command *cmd-line*, and return the exit status.

A.4.3.7 String Constants

String constants in `awk` are sequences of characters enclosed between double quotes (`"`). Within strings, certain *escape sequences* are recognized, as in C. These are:

`\\`	A literal backslash.
`\a`	The "alert" character; usually the ASCII BEL character.
`\b`	Backspace.
`\f`	Formfeed.
`\n`	Newline.

`\r`	Carriage return.
`\t`	Horizontal tab.
`\v`	Vertical tab.
`\x`*hex digits*	The character represented by the string of hexadecimal digits following the '`\x`'. As in ANSI C, all following hexadecimal digits are considered part of the escape sequence. (This feature should tell us something about language design by committee.) E.g., `"\x1B"` is a string containing the ASCII ESC (escape) character. (The '`\x`' escape sequence is not in POSIX `awk`.)
`\`*ddd*	The character represented by the 1-, 2-, or 3-digit sequence of octal digits. Thus, `"\033"` is also a string containing the ASCII ESC (escape) character.
`\`*c*	The literal character *c*.

The escape sequences may also be used inside constant regular expressions (e.g., the regexp `/[ \t\f\n\r\v]/` matches whitespace characters).

See Section 8.1 [Constant Expressions], page 57.

A.5 Functions

Functions in `awk` are defined as follows:

```
function name(parameter list) { statements }
```

Actual parameters supplied in the function call are used to instantiate the formal parameters declared in the function. Arrays are passed by reference, other variables are passed by value.

If there are fewer arguments passed than there are names in *parameter-list*, the extra names are given the null string as value. Extra names have the effect of local variables.

The open-parenthesis in a function call of a user-defined function must immediately follow the function name, without any intervening white space. This is to avoid a syntactic ambiguity with the concatenation operator.

The word `func` may be used in place of `function` (but not in POSIX `awk`).

Use the `return` statement to return a value from a function.

See Chapter 12 [User-defined Functions], page 95, for a more complete description.

Appendix B Sample Program

The following example is a complete `awk` program, which prints the number of occurrences of each word in its input. It illustrates the associative nature of `awk` arrays by using strings as subscripts. It also demonstrates the '`for` *x* `in` *array*' construction. Finally, it shows how `awk` can be used in conjunction with other utility programs to do a useful task of some complexity with a minimum of effort. Some explanations follow the program listing.

```
awk '
# Print list of word frequencies
{
    for (i = 1; i <= NF; i++)
        freq[$i]++
}

END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}'
```

The first thing to notice about this program is that it has two rules. The first rule, because it has an empty pattern, is executed on every line of the input. It uses `awk`'s field-accessing mechanism (see Section 3.2 [Examining Fields], page 22) to pick out the individual words from the line, and the built-in variable `NF` (see Chapter 13 [Built-in Variables], page 101) to know how many fields are available.

For each input word, an element of the array `freq` is incremented to reflect that the word has been seen an additional time.

The second rule, because it has the pattern `END`, is not executed until the input has been exhausted. It prints out the contents of the `freq` table that has been built up inside the first action.

Note that this program has several problems that would prevent it from being useful by itself on real text files:

- Words are detected using the `awk` convention that fields are separated by whitespace and that other characters in the input (except newlines) don't have any special meaning to `awk`. This means that punctuation characters count as part of words.
- The `awk` language considers upper and lower case characters to be distinct. Therefore, '`foo`' and '`Foo`' are not treated by this program as the same word. This is undesirable since in normal text, words are capitalized if they begin sentences, and a frequency analyzer should not be sensitive to that.
- The output does not come out in any useful order. You're more likely to be interested in which words occur most frequently, or having an alphabetized table of how frequently each word occurs.

The way to solve these problems is to use some of the more advanced features of the `awk` language. First, we use `tolower` to remove case distinctions. Next, we use `gsub` to remove punctuation characters. Finally, we use the system `sort` utility to process the output of the `awk` script. First, here is the new version of the program:

```
awk '
# Print list of word frequencies
{
    $0 = tolower($0)    # remove case distinctions
    gsub(/[^a-z0-9_ \t]/, "", $0)  # remove punctuation
    for (i = 1; i <= NF; i++)
        freq[$i]++
}

END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}'
```

Assuming we have saved this program in a file named '`frequency.awk`', and that the data is in '`file1`', the following pipeline

```
awk -f frequency.awk file1 | sort +1 -nr
```

produces a table of the words appearing in '`file1`' in order of decreasing frequency.

The `awk` program suitably massages the data and produces a word frequency table, which is not ordered.

The `awk` script's output is then sorted by the `sort` command and printed on the terminal. The options given to `sort` in this example specify to sort using the second field of each input line (skipping one field), that the sort keys should be treated as numeric quantities (otherwise '`15`' would come before '`5`'), and that the sorting should be done in descending (reverse) order.

We could have even done the `sort` from within the program, by changing the `END` action to:

```
END {
    sort = "sort +1 -nr"
    for (word in freq)
        printf "%s\t%d\n", word, freq[word] | sort
    close(sort)
}'
```

See the general operating system documentation for more information on how to use the `sort` command.

Appendix C Glossary

Action	A series of `awk` statements attached to a rule. If the rule's pattern matches an input record, the `awk` language executes the rule's action. Actions are always enclosed in curly braces. See Chapter 7 [Overview of Actions], page 55.
Amazing `awk` Assembler	Henry Spencer at the University of Toronto wrote a retargetable assembler completely as `awk` scripts. It is thousands of lines long, including machine descriptions for several 8-bit microcomputers. It is a good example of a program that would have been better written in another language.
ANSI	The American National Standards Institute. This organization produces many standards, among them the standard for the C programming language.
Assignment	An `awk` expression that changes the value of some `awk` variable or data object. An object that you can assign to is called an *lvalue*. See Section 8.7 [Assignment Expressions], page 64.
`awk` Language	The language in which `awk` programs are written.
`awk` Program	An `awk` program consists of a series of *patterns* and *actions*, collectively known as *rules*. For each input record given to the program, the program's rules are all processed in turn. `awk` programs may also contain function definitions.
`awk` Script	Another name for an `awk` program.
Built-in Function	The `awk` language provides built-in functions that perform various numerical, time stamp related, and string computations. Examples are `sqrt` (for the square root of a number) and `substr` (for a substring of a string). See Chapter 11 [Built-in Functions], page 89.
Built-in Variable	`ARGC`, `ARGV`, `CONVFMT`, `ENVIRON`, `FILENAME`, `FNR`, `FS`, `NF`, `NR`, `OFMT`, `OFS`, `ORS`, `RLENGTH`, `RSTART`, `RS`, and `SUBSEP`, are the variables that have special meaning to `awk`. Changing some of them affects `awk`'s running environment. See Chapter 13 [Built-in Variables], page 101.
Braces	See "Curly Braces."
C	The system programming language that most GNU software is written in. The `awk` programming language has C-like syntax, and this manual points out similarities between `awk` and C when appropriate.
CHEM	A preprocessor for `pic` that reads descriptions of molecules and produces `pic` input for drawing them. It was written by Brian Kernighan, and is available from `netlib@research.att.com`.
Compound Statement	A series of `awk` statements, enclosed in curly braces. Compound statements may be nested. See Chapter 9 [Control Statements in Actions], page 73.
Concatenation	Concatenating two strings means sticking them together, one after another, giving a new string. For example, the string '`foo`' concatenated with the string '`bar`' gives the string '`foobar`'. See Section 8.4 [String Concatenation], page 61.

Conditional Expression
: An expression using the '`?:`' ternary operator, such as *expr1* `?` *expr2* `:` *expr3*. The expression *expr1* is evaluated; if the result is true, the value of the whole expression is the value of *expr2* otherwise the value is *expr3*. In either case, only one of *expr2* and *expr3* is evaluated. See Section 8.11 [Conditional Expressions], page 69.

Constant Regular Expression
: A constant regular expression is a regular expression written within slashes, such as '`/foo/`'. This regular expression is chosen when you write the `awk` program, and cannot be changed doing its execution. See Section 6.2.1 [How to Use Regular Expressions], page 47.

Comparison Expression
: A relation that is either true or false, such as `(a < b)`. Comparison expressions are used in `if`, `while`, and `for` statements, and in patterns to select which input records to process. See Section 8.5 [Comparison Expressions], page 62.

Curly Braces
: The characters '`{`' and '`}`'. Curly braces are used in `awk` for delimiting actions, compound statements, and function bodies.

Data Objects
: These are numbers and strings of characters. Numbers are converted into strings and vice versa, as needed. See Section 8.9 [Conversion of Strings and Numbers], page 67.

Dynamic Regular Expression
: A dynamic regular expression is a regular expression written as an ordinary expression. It could be a string constant, such as `"foo"`, but it may also be an expression whose value may vary. See Section 6.2.1 [How to Use Regular Expressions], page 47.

Escape Sequences
: A special sequence of characters used for describing nonprinting characters, such as '`\n`' for newline, or '`\033`' for the ASCII ESC (escape) character. See Section 8.1 [Constant Expressions], page 57.

Field
: When `awk` reads an input record, it splits the record into pieces separated by whitespace (or by a separator regexp which you can change by setting the built-in variable `FS`). Such pieces are called fields. See Section 3.1 [How Input is Split into Records], page 21.

Format
: Format strings are used to control the appearance of output in the `printf` statement. Also, data conversions from numbers to strings are controlled by the format string contained in the built-in variable `CONVFMT`. See Section 4.5.2 [Format-Control Letters], page 38.

Function
: A specialized group of statements often used to encapsulate general or program-specific tasks. `awk` has a number of built-in functions, and also allows you to define your own. See Chapter 11 [Built-in Functions], page 89. Also, see Chapter 12 [User-defined Functions], page 95.

`gawk`
: The GNU implementation of `awk`.

GNU
: "GNU's not Unix". An on-going project of the Free Software Foundation to create a complete, freely distributable, POSIX-compliant computing environment.

Input Record
: A single chunk of data read in by `awk`. Usually, an `awk` input record consists of one line of text. See Section 3.1 [How Input is Split into Records], page 21.

Keyword
: In the `awk` language, a keyword is a word that has special meaning. Keywords are reserved and may not be used as variable names.

: `awk`'s keywords are: `if`, `else`, `while`, `do…while`, `for`, `for…in`, `break`, `continue`, `delete`, `next`, `function`, `func`, and `exit`.

Lvalue
: An expression that can appear on the left side of an assignment operator. In most languages, lvalues can be variables or array elements. In `awk`, a field designator can also be used as an lvalue.

Number
: A numeric valued data object. The `awk` implementation uses double precision floating point to represent numbers.

Pattern
: Patterns tell `awk` which input records are interesting to which rules.

  A pattern is an arbitrary conditional expression against which input is tested. If the condition is satisfied, the pattern is said to *match* the input record. A typical pattern might compare the input record against a regular expression. See Chapter 6 [Patterns], page 47.

POSIX
: The name for a series of standards being developed by the IEEE that specify a Portable Operating System interface. The "IX" denotes the Unix heritage of these standards. The main standard of interest for `awk` users is P1003.2, the Command Language and Utilities standard.

Range (of input lines)
: A sequence of consecutive lines from the input file. A pattern can specify ranges of input lines for `awk` to process, or it can specify single lines. See Chapter 6 [Patterns], page 47.

Recursion
: When a function calls itself, either directly or indirectly. If this isn't clear, refer to the entry for "recursion."

Redirection
: Redirection means performing input from other than the standard input stream, or output to other than the standard output stream.

  You can redirect the output of the `print` and `printf` statements to a file or a system command, using the '`>`', '`>>`', and '`|`' operators. You can redirect input to the `getline` statement using the '`<`' and '`|`' operators. See Section 4.6 [Redirecting Output of `print` and `printf`], page 42.

Regular Expression
: See "regexp."

Regexp
: Short for *regular expression*. A regexp is a pattern that denotes a set of strings, possibly an infinite set. For example, the regexp '`R.*xp`' matches any string starting with the letter '`R`' and ending with the letters '`xp`'. In `awk`, regexps are used in patterns and in conditional expressions. Regexps may contain escape sequences. See Section 6.2 [Regular Expressions as Patterns], page 47.

Rule
: A segment of an `awk` program, that specifies how to process single input records. A rule consists of a *pattern* and an *action*. `awk` reads an input record; then, for each rule, if the input record satisfies the rule's pattern, `awk` executes the rule's action. Otherwise, the rule does nothing for that input record.

Side Effect
: A side effect occurs when an expression has an effect aside from merely producing a value. Assignment expressions, increment expressions and function calls have side effects. See Section 8.7 [Assignment Expressions], page 64.

Special File
: A file name interpreted internally by `awk`, instead of being handed directly to the underlying operating system. For example, '`/dev/stdin`'. See Section 4.7 [Standard I/O Streams], page 44.

Stream Editor
: A program that reads records from an input stream and processes them one or more at a time. This is in contrast with batch programs, which may expect to read their input

files in entirety before starting to do anything, and with interactive programs, which require input from the user.

String
: A datum consisting of a sequence of characters, such as '`I am a string`'. Constant strings are written with double-quotes in the `awk` language, and may contain escape sequences. See Section 8.1 [Constant Expressions], page 57.

Whitespace
: A sequence of blank or tab characters occurring inside an input record or a string.

Index

Short Contents

Table of Contents