Util2 Package:
Unix-like and Other Utilities
For OS/2, Win32, and DOS
Version 2.9

Brian Yoder
beyoder@us.ibm.com

7 February 1998

Contents

Introduction

This package contains a set of Unix-like and other miscellaneous command-line utilities. Each is available for 16-bit DOS, Win32, and 32-bit OS/2 command lines. The Win32 versions work with Windows 95 and Windows NT command-line sessions and support long filenames. The OS/2 versions work with HPFS long filenames and can run in an OS/2 window.

The package is available inside IBM as the Util2 Package. It is available outside IBM as the UTLOS2 Package from the various Internet sites that mirror the IBM Employee Written Software (EWS) packages.

The following utility programs are included:

ccp	:	Conditional copy: A better XCOPY. Very nice!
cdx	:	Change directory – a much better CD
chmod	:	Change file mode – a better ATTRIB
crc	:	Calculate CRC for file(s)
crcchk	:	Calculate and check CRC for file(s)
du	:	Display file space usage
grep	:	Search text files for regular expression
ls	:	List directory – a much better DIR
strings	:	Show printable character strings
txtcut	:	Text file preprocessor for cut, awk, and Perl

Most of these utilities are based on Unix commands of the same or similar name but are more friendly to DOS, Windows, and OS/2 users and don't require that you have a Unix background or bias. They accept both \ and / as path separators. They show fields (such as system, hidden, and archive attributes) that Windows has and don't show fields (such as the Unix-like user/group/other permissions) that DOS and Windows 95 don't have and that Windows NT has in but a far different flavor.

Filename Matching and Patterns

The utilities in this package match filenames using Bourne shell style filename patterns, a superset of the DOS, Win32, and OS/2 "wildcard" filename matching characters.

A file specification consists of an optional drive, optional path information, and a filename. The filename may consist of Bourne shell style pattern-matching characters. DOS's filename pattern matching is simplistic, Win32's and OS/2's is a lot better, but Bourne shell pattern matching is better still. Also, all versions (DOS, Win32, and OS2) of the commands in this package support the same enhanced level of pattern-matching.

Basic Rules

- **A dot is just another character and has no special meaning.**

  To the commands in this package, a filename is just a string of characters. The dot has no special meaning—it's just another character—so there is no concept of a "file extension". This is consistent with Win32 filesystems, OS/2's HPFS, and with Unix and Linux filesystems.

  Note that "`*.*`" will *not* match a filename that doesn't contain a dot. The pattern "`*`" matches any filename. The pattern "`*.*`" matches any filename that contains a dot. This is important to remember!

  Note, however, that this is consistent across the DOS, Win32, and OS/2 versions of these utilities, unlike any of the native DOS, Win32, and OS/2 commands such as `DIR` and `COPY`. This is also consistent with native Unix and Linux commands.

- **Backslashes and forward slashes are accepted as path separators.**

  Either a `\` (backslash) or `/` (forward slash) character may be used as a filename path separator on the command lines of these utilities. UNC-style names may also be entered with either backslashes or forward slashes.

  When one of these commands prints a path name, it uses the last path separator character that it finds on the last path name in its command line arguments. If you use \ then these commands print path names using \; if you use / then these commands print path names using /.

If there are no path separators at all in any of the path names on the command line, then these commands print path names using / as the default path separator. If you prefer \ to be used as the default output path separator in this case, then set the `USEP` environment variable to \.

Patterns

A filename can contain one or more of the following patterns. Note that letters within patterns are case-*in*sensitive:

`*`	Matches any string, including the null string. *For example,* `*ab*` matches `ab`, `blab`, `babies`, and `BABEL`. `*` matches any string, including `TEMP`, `WINDOWS`, `makefile`, or `myfile.c`.
`?`	Matches any single character. *For example,* `ab?c` matches any filename that is 4 characters long, begins with `ab`, and ends with `c`.
`.`	Remember that "`.`" matches a dot *For example,* `*.*` matches `myfile.c`, `config.sys`, and `zip.hlp`. `*.*` does *not* match `TEMP`, `WINDOWS`, or `makefile`. This is because these names do not contain a dot character.
`[...]`	Matches any one of the enclosed characters. *For example,* `[ABC]*` matches any filename that begins with `A`, `B`, or `C`.
`[.-.]`	Matches any character between the enclosed pair, inclusive (range). *For example,* `[A-Z]*` matches any filename that begins with a letter. `[A-LN-Z]*` matches any filename that doesn't begin with `M`.
`[!...]`	Matches any single character except one of those enclosed. *For example,* `[!XYZ]*` matches any filename that does not begin with `X`, `Y`, or `Z`.

Put a backslash before the following characters if they are part of the filename to be matched. This removes their special meaning:

```
[ ] { } !
```

In addition

Enclosed characters can be combined with ranges. Thus, `[ABCM-Z]*` matches any filename that begins with A, B, C, or M through Z.

Additionally, any pattern may be followed by:

`{m}`	Matches exactly `m` occurrences of the pattern.
`{m,}`	Matches at least `m` occurrences of the pattern.
`{m,n}`	Matches at least `m` but no more than `n` occurrences of the pattern.

where: `m` and `n` must be integers from 0 to 255, inclusive.

Examples

1. The pattern `[A-Z][0-9A-Z.]{0,}` matches any filename that begins with a letter that is followed only by letters or numbers, and may contain one or more dots.

2. The pattern `[!.]{1,}` matches any filename that does not contain a dot. Literally, the pattern means: Match any name that contains one or more occurrences of any character that is not a dot.

3. The pattern `*TXT*` matches any filename that contains the string "TXT", either in the beginning, at the end, or in the middle.

4. Again, the pattern `*` matches any filename. The pattern `*.*` matches any filename that contains a dot. Note that `*.*` will not match a filename that doesn't contain a dot! This is important to remember!

ccp—Conditional Copy

The **ccp** command conditionally copies files to a target directory if they are missing from the target directory or have a different size, time, or date on the target directory. It preserves the name, read/write mode (read-only attribute), system and hidden attributes, and modification date and time of the files that it copies. It can copy entire subdirectory trees as `XCOPY /S` does or just portions of subdirectory trees. It can also exclude files and subdirectory trees that match one or more optional exclusion patterns.

Usage

```
ccp [ -flags ] [ -- ] source ... [ ! Xsource ... ] targetdir
```

The **ccp** command conditionally copies files whose names match the source file specification(s). It ignores hidden and system files. It copies files to the target directory. The target directory must already exist.

If there is only one *source* file specification and no *Xsource* file specifications, then the *targetdir* is optional. In this case, it defaults to `.` (a dot, which means the current directory).

A *source* file specification consists of an optional drive, an optional path, and a filename. The filename may contain pattern-matching characters. If a *source* file specification is the name of a directory, then **ccp** appends "`/*`" to it, matching all files in the directory.

If a **!** is specified, then **ccp** will exclude (not copy) any source file whose source name matches one of the *Xsource* patterns (listed between the `!` and the target directory). Each *Xsource* pattern must match the full source name to exclude the source file. An *Xsource* pattern may, of course, contain drive and path information.

Generally, an *Xsource* pattern should start with a "`*`" character or with the drive and path of a source file (as specified on the command line) or it will not exclude any source files. This is because **ccp** compares each *Xsource* pattern with the path and name of each source file as it was specified on the command line, and including any subdirectory names that **ccp** added due to the `-s` flag. See the examples for more details.

Both `\` (backslash) and `/` (forward slash) characters are interpreted as path separators in source file specifications, the *Xsource* (exclude source) patterns, and the target directory specifications. Therefore, the \ (backslash) character cannot be used as an escape character in an *Xsource* pattern.

This program writes the names of source files (both those copied and, if the `-x` flag is specified, those excluded) to standard output. It writes all other messages, errors, and copy count statistics to standard error.

Flags

By default, **ccp** only copies a source file if it is missing from the target directory, has a different size on the target directory, or has a different modification date and time on the target directory. The behavior of **ccp** can be altered by the following flags:

`-l` (letter el) If a file with the same name as a source file exists in the target directory, copy the source file only if its modification time is later than that of the file in the target directory. In other words, only copy source files that are later than those in the target.

`-d` If the target directory doesn't already exist, create it before attempting to copy any files to it.

Note that if you specify `-s` with or without `-d`, **ccp** always creates any path components *within* the target directory that don't already exist.

`-e` If a file with the same name as a source file doesn't exist in the target directory, then don't copy the source file. In other words, only copy files that already exist in the target.

`-f` Force copy even if target file is read-only.

`-n` Just display the names of source files that would have been copied, but don't actually copy any files. If the `-d` flag is also specified and the target directory doesn't exist, then the target directory is also not actually created.

The `-n` flag is an excellent way to see just what files would be copied and which files would be excluded without actually copying anything.

`-s` Descend subdirectories while searching for files to copy, just as `XCOPY /S` does. The subdirectory structure (relative to the source directory) is preserved within the target directory. When a file is copied, any subdirectories that don't already exist within the target directory are created by **ccp**.

If the specified target directory doesn't exist, it is not created unless the `-d` flag is also specified. The `-s` flag without `-d` only creates subdirectories within the target directory but never creates the target directory.

`-S` Also copy system and hidden files.

Since system files are quite often also read-only files, the `-S` flag should usually be combined with the `-f` flag to force updates. Be very careful when updating system and hidden files!

`-t` For each file copied, show the path and name of each target file in addition to the path and name of each source file, as follows:

```
sourcename -> targetname
```

`-x` Also display the names of source files that are being excluded (not copied). One of the following characters is displayed in front of each filename that **ccp** is excluding from copying:

`!` before files that you want to exclude (via `!` *Xsource*), and

`x` before files that **ccp** decides to exclude (not copy).

Examples

1. `ccp a:* .`

   This command copies all files from the current directory on drive `A` to the current directory if they don't exist in the current directory or have a different size, date, or time in the current directory.

2. `ccp a:*`

   This command is the same as the previous one. Since there's only one file specification, the target directory defaults to the current directory.

3. `ccp * ! *.cod *xyz* a:`

   This command conditionally copies all files in the current directory, except for those whose names end with **.cod** or contain **xyz**, to the current directory on drive `A`.

4. `ccp -sd c:\windows\* ! c:\windows\temp\* d:\archive\windows`

   This command conditionally copies the `c:\windows` directory and all of its subdirectories to the `d:\archive\windows` directory, except for any files that are in the `c:\windows\temp` directory. If any path components within the target `d:\archive\windows` directory don't exist they are created before copying the files.

5. `ccp -sd c:/windows/* ! *temp* d:/archive/windows`

   This command works like the previous command except it excludes any files within **c:/windows** that contain **temp** in their name or in any subdirectory of their name.

Also notice that forward slashes work just as well as backslashes. Indeed, when running **ccp** (or any of these other commands) from within a Perl, Python, bash, gawk, or awk script, or from within another scripting language, a backslash is often used to start an escape sequence and a forward slash is a more convenient and robust path separator.

6. `ccp -sd c:/windows/* ! */temp/* d:/archive/windows`

   This command works like the previous command except it excludes any files within **c:/windows** that also have a path component that includes any **temp** subdirectory.

7. `ccp -sd c:/windows/* ! */temp/* *sample.c d:/archive/windows`

   This command works like the previous command except it excludes any files within **c:/windows** that also have a path component that includes any **temp** subdirectory, and it excludes any file named **sample.c**.

   Note that the exclusion pattern for **sample.c** begins with an asterisk. This ensures that the pattern matches any file ending in **sample.c** even if there is path or other information present (such as **test.sample.c** and **tempdir/sample.c**).

8. `ccp -le * d:/tmp`

   This command copies all files in the current directory to the **d:/tmp** directory that already exist in **d:/tmp** but have a later modification date and time than those in **d:/tmp**.

9. `ccp -n *.exe *.doc a:/`

   This command displays the names of all of the **.exe** and **.doc** files in the current directory that are missing from the root directory on drive `A` or that have a different size, date, or time in the root directory of drive `A`. No files are actually copied.

10. `ccp -nelx * ! *.cod d:/mydir`

    This command displays the names of files that would have been copied, but doesn't actually copy anything.

    It looks at all files in the current directory, excluding any that end in **.cod**. It displays the names of the source files (that it would have copied) that exist in the target directory and that are later than their counterparts in the target directory.

    Since you excluded the ***.cod** files, ccp displays the names of the ***.cod** files with a "`!`" before each of their names. Since **ccp** excludes files that either don't exist in the target directory or have the same or earlier modification time in the source directory, ccp displays these latter files with an "x" before each of their names.

11. `ccp -st c:*.exe d:`

    This command copies all of the **.exe** files within the current directory and all of its subdirectories on drive `C` to the current directory on drive `D`. If a subdirectory in which a copied file doesn't exist in the target, **ccp** creates it. Note that **ccp** creates subdirectories within the target only for files that are copied.

    The `-t` flag tells **ccp** to show the path and name of each target files that is copied as well as its source path and name. For subdirectory copies, this provides feedback that lets you know exactly where each source file is being copied.

12. `ccp a:* .`

    This command copies all files from the current directory on drive `A` to the current directory if they don't exist in the current directory or have a different size, date, or time in the current directory.

13. `ccp a:*`

    This command is the same as the previous one. Since there's only one file specification, the target directory defaults to the current directory.

14. `ccp * ! *.cod *xyz* a:`

    This command conditionally copies all files in the current directory, except for those whose names end with **.cod** or contain **xyz**, to the current directory on drive `A`.

15. `ccp -sd c:/windows/* ! c:/windows/temp/* d:/archive/windows`

    This command conditionally copies the **c:/windows** directory and all of its subdirectories to the **d:/archive/windows** directory, except for any files that are in the **c:/windows/temp** directory. If any path components within the target **d:/archive/windows** directory don't exist then **ccp** creates them before copying the files.

16. `ccp -sd c:/windows/* ! *temp* d:/archive/windows`

    This command works like the previous command except it excludes any files within **c:/windows** that contain **temp** in their name or in any subdirectory of their name.

17. `ccp -sd c:/windows/* ! */temp/* d:/archive/windows`

    This command works like the previous command except it excludes any files within **c:/windows** that also have a path component that includes any **temp** subdirectory.

18. `ccp -sd c:/windows/* ! */temp/* *sample.c d:/archive/windows`

    This command works like the previous command except it excludes any files within **c:/windows** that also have a path component that includes any **temp** subdirectory, and it excludes any file named **sample.c**.

Note that the exlusion pattern for **sample.c** begins with an asterisk. This ensures that the pattern matches any file ending in **sample.c** even if there is path or other information present.

19. `ccp -le * d:/tmp`

    This command copies all files in the current directory to the **d:/tmp** directory that already exist in **d:/tmp** but have a later modification date and time than those in **d:/tmp**.

20. `ccp -n *.exe *.doc a:/`

    This command displays the names of all of the **.exe** and **.doc** files in the current directory that are missing from the root directory on drive `A` or that have a different size, date, or time in the root directory of drive `A`. No files are actually copied.

21. `ccp -nelx * ! *.cod d:/mydir`

    This command displays the names of files that would have been copied, but doesn't actually copy anything.

    It looks at all files in the current directory, excluding any that end in **.cod**. It displays the names of the source files (that it would have copied) that exist in the target directory and that are later than their counterparts in the target directory.

    Since you excluded the ***.cod** files, **ccp** displays the names of the ***.cod** files with a `!` before each of their names. Since **ccp** excludes files that either don't exist in the target directory or have the same or earlier modification time in the source directory, ccp displays these latter files with an `x` before each of their names.

22. `ccp -st c:*.exe d:`

    This command copies all of the **.exe** files within the current directory and all of its subdirectories on drive `C` to the current directory on drive `D`. If a subdirectory in which a copied file doesn't exist in the target, **ccp** creates it. Note that **ccp** creates subdirectories within the target only for files that are copied.

    The `-t` flag tells ccp to show the path and name of each target files that is copied as well as its source path and name. For subdirectory copies, this provides feedback that lets you know exactly where each source file is being copied.

cdx—Enhanced Change Directory

The **cdx** command is an enhanced (eXtended) version of the CD (or CHDIR) command. Like CD, it can go to a specified directory. Unlike CD, it can also change the current drive at the same time. It can go to a default HOME directory. And it can search through a set of paths for the specified directory, eliminating the need for a lot of little command files to change directories.

The **c:\tmp** directory must exist and be writable for the Win32 and OS/2 versions of **cdx**. A temporary batch (command) file is written to **c:\tmp** by the **cdx** command for these platforms.

On DOS, the **cdx** command is implemented as **cdx.bat**. The **cdx.bat** script calls **cdir.exe** and **cdir.exe** changes the current directory and drive itself.

For Win32, the **cdx** command is implemented as the **cdx.bat** script. For OS/2, it is implemented as the **cdx.cmd** script. For these platforms, **cdx** calls **cdir.exe** just to get the path and drive change information and then **cdx** itself performs the change. This is because these platforms, like Unix, don't allow a process to change its parent process's environment or working directory.

For DOS, the cdx.bat file just invokes the cdir.exe program. You can, if you wish, run the cdir.exe program directly to change your current working drive and path. Using **cdx** instead just lets your fingers type the same command on DOS, OS/2, and Win32.

This command cannot be named cd, which is the name of the less-capable built-in DOS, Win32, and OS/2 command.

Usage

```
cdx  [d:][path]
```

If both the drive and path are missing, then **cdx** attempts to change the current drive and path to that specified by the HOME environment variable.

If the drive is present, then **cdx** changes your current drive to that drive. If the path is present, then **cdx** changes the current directory on that drive to the the specified path.

If the drive is missing and the path is relative (doesn't begin with `\` or `/` path separator), then **cdx** looks in the current directory for that path. If it doesn't find it, it then looks for the path in the directories specified by the CDPATH environment variable. The list of paths specified by CDPATH should be formatted just like those for the PATH environment variable.

The following are the settings for HOME and CDPATH that I use:

```
SET HOME=c:\u\brian
SET CDPATH=c:\u\brian;c:\Program Files;e:\watcom
```

Examples

The following examples use the settings for HOME and CDPATH shown above.

1. `cdx`

   Go to the `c:\u\brian` (HOME) directory.

2. `cdx a:`

   Go to the current directory on the `A` drive.

3. `cdx d:/tmp`

   Go to the `\tmp` directory on the `D` drive.

4. `cdx include`

   Go to the **include** directory within the current directory. If it doesn't exist, then use the list of directoies in CDPATH and go to the first of the following directories that exists:

   ```
   c:\u\brian\include
   c:\Program Files\include
   e:\watcom\include
   ```

chmod—Change File Mode

The **chmod** command is an alternative to the ATTRIB command. It allows you to change the attributes (file mode) of one or more files. It can process hidden and system files in addition to ordinary files.

Unlike ATTRIB for DOS and OS/2, **chmod** treats a read-only file as lacking the "write" capability rather than having the "read-only" capability.

The Win32 version of **chmod** will change the specified attributes of matching directories. The DOS and OS/2 versions of **chmod** ignore matching directories.

Usage

`chmod [ -R ] [ +-= ][ wsha ] fpsec ...`

If `-R` is specified, then **chmod** recursively descends subdirectories looking for matching files. The `-R` flag is compatible with the AIX and the original Unix versions of **chmod**.

The first argument must contain exactly one of the "`+-=`" file mode operators followed by one or more of the "`wsha`" file mode letters, with no intervening spaces.

One or more file specifications must follow. A file specification consists of some combination of drive, path, and filename. The filename may contain Bourne shell pattern-matching characters.

The file mode operator can be one of the following:

`+`	Sets the specified file mode(s).
`-`	Resets the specified file mode(s).
`=`	Sets the specified file mode(s) and clears the modes that aren't specified.

The file mode letters have the following meanings:

`w`	Write permission.
`s`	System attribute.
`h`	Hidden attribute.
`a`	Archive attribute.

Examples

1. `chmod -w *`

   Reset the write permission for all of the files in the current directory, making them read-only.

2. `chmod -R -w *`

   Reset the write permission for all of the files in the current directory and in all subdirectories, recursively.

3. `chmod +ha *.exe`

   Make the **.exe** files in the current directory hidden, and set their archive bits.

4. `chmod -a *.c`

   Reset the archive attribute for all of the **.c** files in the current directory.

5. `chmod =sh /*`

   For all of the files in the root directory, set their system and hidden attributes and reset their write permission and archive attribute. When complete, the files will be unwritable (read-only), system, and hidden, and their archive attributes will be zero.

crc and crcchk

The **crc** command is used to record the length and 16-bit CRC value (or, optionally, the 32-bit CRC value) for one or more files.

The **crcchk** command is used to record the length and CRC value (either 16-bit or 32-bit) for a list of files and optionally check their lengths and CRC values against a previously-obtained set of values for those files.

The code used to calculate the 16-bit CRC value for an individual file was developed at IBM Austin, Texas, by Jim Czenkusch. The code to calculate the 32-bit CRC has enhancements to pre-condition and post-condition each value so that it is compatible with the 32-bit CRC value generated by PKWARE's PKZIP and InfoZIP's **zip** commands.

Usage

```
crc [ -s ] [ -l ] fspec ...
crcchk [ -l ] infile [ outfile ]
```

The **crc** program gets the length and calculates the CRC value for each file that matches the given command-line file specification(s). If `-s` is specified, then **crc** recursively descends into subdirectories looking for files that match the file specification(s).

The **crcchk** program reads the input text file to get a list of files. It then calcuates the length and CRC of each file listed. If the input file also contains a length and CRC value for a file, those values are compared against the calculated values for that file. This input file can be created directly from the output of either the **crc** or **crcchk** command.

If an output file name is specified to **crcchk**, then each file listed in the input text file, along with its actual length and CRC, is written to the output file. Later, this output file can be read by the **crcchk** program to check to see if any of the listed files has changed.

File lengths are displayed in decimal format. CRCs are displayed in hexadecimal format with a leading `0x`. Filenames are enclosed within double quotes in case they contain embedded spaces or other puncutation.

If the `-l` (letter el) flat is specified, **crc** and **crcchk** generate 32-bit (long) CRCs that are compatible with those generated by PKZIP and **zip**. The default is to generate 16-bit CRCs.

The **crcchk** program only honors the `-l` flag if an output file is specified. It ignores the `-l` flag if no output file was specified.

For each matching file, **crc** writes one line to standard output as follows:

```
length  CRC  "filename"
```

Flags

`-l` (letter el) **crcchk** (if an output file was specified) and **crc** calculate and show long (32-bit) CRC values. The default is to calculate and show 16-bit CRC values.

`-s` Descend subdirectories searching for matching files (**crc** only).

Input File for crcchk

Blank lines are ignored. Lines that begin with `#` are assumed to contain comments and are ignored. Each non-blank, non-comment line must be formatted as follows:

```
[ length  CRC ]  "filename"
```

The *length* and *CRC* values in the line are optional. Either both must be present or both must be missing. If present, the length must be specified in decimal, while the CRC must be specified in hexadecimal as `0x` followed by one or more hexadecimal digits.

If the CRC is `0xNNNN` or shorter (4 or less hex digits), then it is assumed to be a 16-bit CRC. If it contains 5 or more hex digits, then it is assumed to be a 32-bit CRC.

If the length and CRC value are present, then **crcchk** compares the values to those it calculated and logs any discrepancies to stderr.

The *filename* must be enclosed in double quotes if it contains spaces, semicolons, commas, or other punctuation. Double quotes around the filename are optional otherwise.

Output File from crcchk

If an output file is specified, then **crcchk** writes to this output file the name, length, and CRC value for each file that is listed in the input file, one file per line, as follows:

```
length  CRC  "filename"
```

If the CRC in the input text file is a 32-bit CRC, then the CRC in the output file will also be a 32-bit CRC. If the CRC in the input text file is a 16-bit CRC, then the CRC in the output text file will be a 16-bit CRC.

Note: If an output file is specified but a file listed in the input file has no CRC, then **crcchk** checks the `-l` (letter el) flag. If `-l` is specified in this case, **crcchk** generates a 32-bit (long) CRC for the file. Otherwise, it defaults to generating a 16-bit CRC for the file.

Examples

1. I entered the following command on my system (a very long time ago, I might add):

   ```
   crc *.h \*.sys
   ```

   It produced the following output:

   ```
    3159  0xD65B  BMTBL.H
   10780  0x5495  UTIL.H
   15473  0x064B  REGEXP.H
     178  0x4E87  \CONFIG.SYS
   ```

2. I entered the following command on my system (also a long time ago):

   ```
   crc -l crc.doc
   ```

   It produced the following output:

   ```
   2409  0x070E5D3F  CRC.DOC
   ```

3. The `ls \u\brian\bin\*.exe >files.txt` command was run to produce a list of files that I wanted to check with **crcchk**. You should normally specify the full pathname of the files for **ls** so that you can later run **crcchk** from any directory—**crcchk** will attempt to open each file using the name as listed. I then added the comment and blank line to the beginning of the file, as follows:

   ```
   # files.txt created on 05/20/91

   \u\brian\bin\ati.exe
   \u\brian\bin\b.exe
   \u\brian\bin\cdcl.exe
   \u\brian\bin\config.exe
   \u\brian\bin\ccmt.exe
   \u\brian\bin\cdir.exe
   \u\brian\bin\chmod.exe
   ```

The `crcchk files.txt files.crc` command produced the following file named **files.crc**:

```
 7070  0xF359  "\u\brian\bin\ati.exe"
10000  0x8CFB  "\u\brian\bin\b.exe"
27630  0x13AC  "\u\brian\bin\cdcl.exe"
57954  0x5D88  "\u\brian\bin\config.exe"
14401  0x7352  "\u\brian\bin\ccmt.exe"
17827  0xDE07  "\u\brian\bin\cdir.exe"
24053  0xFE94  "\u\brian\bin\chmod.exe"
```

Now, we can run the `crcchk files.crc` command any time we want to see if any of those executable files has been changed. We might also want to backup the **files.crc** file or make it read-only so we don't lose the information it contains.

Note that since the length and CRC values are missing from the input text file, **crcchk** calculates 16-bit CRCs for the output file by default.

du—Directory Usage

The **du** command calculates the total bytes used by a set of files. For simplicity, only the files' sizes are calculated: the amount of space occupied by the disk blocks is not.

Usage

`du [ -shR ] fspec ...`

The sizes of all files that match each file specification are added. The total size (in bytes) and the total number of matching files are written to stdout.

By default, ordinary files are included. Specify `-s` to include system files and `-h` to include hidden files.

Specify `-R` to recurse subdirectories looking for matching files.

Example

The **du** command writes information to stdout in the following format:

```
533908 bytes in 27 files
```

grep—Search for regular expression

The **grep** (get regular expression and print) command is an subset of the Unix **grep** command and even smaller subset of the superb GNU **grep**. It searches one or more files for a pattern, can descend into subdirectories when searching, and can write its output (matching lines along with optional file name, line number, and column number) in one of a user-defined set of customizable formats.

Usage

`grep [ -flags ] [ -- ] "pattern" [ fspec ... ]`

The **grep** command searches for the specified pattern. It writes each line that contains the pattern to standard output. The pattern is a regular expression as described later in this document.

One or more file specifications may be present on the command line. If no file specifications are present, then **grep** reads lines from standard input.

A file specification consists of an optional drive, optional path information, and a filename. The filename may consist of Bourne shell pattern-matching characters.

Binary files (such as ***.obj** and ***.exe**) can be searched in addition to text files.

Flags

The flags that are supported are a subset of those supported by the standard Unix **grep** and of course an even smaller subset of those supported by GNU **grep**:

`-c` Display only a count of matching lines.

`-i` Ignore case when making comparisons (same as `-y`).

- `-l` (letter el) List just the names of files (once) with matching lines. Each file name is separated by a new-line character.
- `-n` Preceed each line with the file name and line number, in the following default form: `filename(line)`
- `-#` (where `#` is a digit from 0 through 9, inclusive): If the `-n` flag is also specified, then a specifying a digit will select a pattern to use when showing the file name, line number, and starting column. If no digit is specified, then `0` (zero) is assumed.
- `-s` Descend subdirectories, also.
- `-v` Display lines that don't contain the pattern.
- `-y` Ignore case when making comparisons (compatible with the RS/6000).

Customizing the Output

Normally, the Unix platform, GNU tools, and Emacs editor family support the standard Unix-style **grep** output format. However, the DOS, Windows, and OS/2 platforms have widely varying (and often uneven) requirements for the output of **grep** and grep-like tools. Of course, GNU Emacs can be customized for different styles, but most other tools are not nearly as accomodating.

Therefore, the output of **grep** can be customized to include the filename, line number, column number, and other characters in the order required by different programs. If the `-n` flag is specified, then a format pattern number may also be specified. If no pattern number is specified, then `-0` (zero) is used as the default.

The **grep** program looks in the `GREPn` environment variable, where `n` is the pattern number. If there is no `GREPn` variable in the environment (for instance, if `-n5` is specified but there is no `GREP5` environment variable defined), then the "`$n($l) : `" pattern is assumed.

The characters from the formatting pattern defined by the environment variable are copied to the output and then the text from the line is added. If a dollar sign `$` meta character is encountered, then it and the next character are interpreted as follows:

- `$n` specifies the name of the file.
- `$l` (letter el) specifies the line number within the file.
- `$c` specifies the starting column within the line.
- `$$` specifies the dollar sign itself.
- `$"` specifies a double quote character.

With the exception of `$"`, double quotes within or surrounding a format pattern are ignored.

If no format pattern is found, the "`$n($l) : `" pattern is assumed. This causes the following information to be shown:

```
filename(linenumber) : text from line
```

Some useful format patterns that may be specified are:

```
set GREP0="$n($l) : "

set GREP1="$n($l:$c) : "

set GREP2="$n:$l:"

set GREP3="$n/$l/$c "

set GREP4="$n,$l,$c "
```

When specified with the `-n` flag, the format patterns shown above are used as follows:

`-n0` is the default that this package's **grep** has shown from the beginning. It mimics the format of error messages from many PC compilers:

```
filename(linenumber) : text from line
```

The `GREP0` environment variable can be changed to another pattern so that -n by itself uses that pattern instead.

`-n1` mimics the format of error messages from some modern C/C++ compilers that also include the column number:

```
filename(line:column) : text from line
```

`-n2` is the Unix and GNU **grep**'s style:

```
filename:linenumber:text from line
```

`-n3` and `-n4` mimic the format used by the UltraEdit32 editor (for Windows platforms only) allows you to select a filename, open it, and put the cursor at the specified line and column within the file. UltraEdit is a very capable shareware editor available from the *www.idmcomp.com* Web site. It's a spectacular bargain with a low shareware price and with capabilities, ease-of-use, and company responsiveness far beyond that of many other editors at any price.

Notes

Lines are limited to a length of 512 characters. Longer lines are split into pieces of 512 characters or less.

A dot within a filename is treated as just another character and not assumed to be the beginning of a file extension. Therefore, you should specify `*` instead of `*.*` when searching all files in a specific directory.

Errors are listed on standard error.

Regular Expressions

The **grep** command supports regular expressions that are combinations of one or more of the following:

- The following expressions match a single character:

`c`	Any ordinary character, other than one of the special pattern-matching characters, matches itself.
`.`	A `.` (period) matches any single character.
`[string]`	A string enclosed in square brackets matches any one character in the string.
`[.-.]`	A range is two characters separated by a dash and enclosed in square brackets. It matches any character that is within the range.
`[^string]`	A string (or range) enclosed in square brackets and preceeded by a `^` (circumflex) matches any character except for the characters in the string (or range). Strings and ranges may be combined as needed, as in: `[a-m0-9xyz]`, which matches `a` through `m`, `0` through `9`, `x`, `y`, or `z`.
`\c`	The `\` (backslash) followed by any character matches that character. This is useful for matching the following special characters: `. * [ ] { } ^ $ \`

- The single-character expressions can be combined into regular expressions as follows:

`*`	Matches zero or more occurences of the previous character.
`{m}`	Matches exactly `m` occurrences of the previous character.
`{m,}`	Matches at least `m` occurrences of the previous character.
`{m,n}`	Matches at least `m` but no more than `n` occurrences of the previous character. `m` and `n` must be integers from 0 to 255, inclusive.

- A regular expression can be restricted to match a string that begins on the first character of the line, ends on the last character of the line, or both, as follows:

`^pattern`	The pattern matches a string that begins on the first character of a line.
`pattern$`	The pattern matches a string that ends on the last character of a line.
`^pattern$`	The pattern matches an entire line.

Examples

1. `grep "the cat" *.txt /*.bat`

   This command searches the files in the current directory that end in **.txt** and the **.bat** files in the root directory for the string `the cat`. Only exact case matches are listed on standard output: occurrences of `The cat` and `the Cat` are not listed.

2. `grep -i "the cat" *.txt /*.bat`

   This command is similar to the previous one except that it performs a case-insensitive search. Occurrences of `The cat` and `the CAT` would be listed in addition to any occurrences of `the cat`.

3. `grep -sn "the {1,}cat" *.txt`

   This command searches all **.txt** files in the current directory and in all subdirectories (recursively) for the pattern. The pattern consists of the word `the`, followed by one or more spaces, and followed by the word `cat`. Therefore, it would match lines that contain `the cat` or `the    cat`. The "`filename(linenumber) : `" string is prepended to each line of each file in which the pattern is found.

4. `grep -i "[a-z][a-z0-9_]{0,}(" *.c`

   This command searches all **.c** files in the current directory for function prototypes and function declarations. The pattern matches any string that begins with a letter, is followed by zero or more letters, numbers, or underscores, and ends with an open parenthesis.

5. `foo | grep "&cont\."`

   This command searches the output of **foo** for the "`&cont.`" string. Note that the period in the pattern was escaped with a backslash. The **grep** command would interpret the "`&cont.`" pattern as meaning the "`&cont`" string followed by any character.

6. `grep "^Usage$" *.doc`

   This command searches all `.doc` files in the current directory for lines that consist of nothing but the string "`Usage`".

7. `grep "streams\[" *.c`

   This command searches all C files in the current directory for lines that contain the "streams[" string. Note that the "`[`" character has to be escaped in the pattern so that it is interpreted by **grep** as a bracket and not as the beginning of a set or range.

8. `grep -v "^$" *.doc`

   This command searches all .doc files in the current directory and lists all lines that are not blank. Note that the `^$` pattern matches any blank line.

ls—List Directory

The **ls** (list directory) command is a loose adaptation of the GNU **ls** command, and is a better DIR command than DIR. It supports multiple file specifications on the command line. It matches filenames using the Bourne shell pattern matching rules and not the DOS or OS/2 "wildcard" rules.

The **ls** command lists the names of ordinary files, hidden files, system files, and directories as you wish. It also lists each name with any drive and path information that was in the original file specification. It can recursively descend subdirectories looking for matching files and or directories if you wish.

In the "long" format, the **ls** command displays each file's attributes as well as its size, date, and time.

By default, **ls** sorts the output by filename, and displays the names in multi-column format if standard output is a character device. When writing multi-column output, **ls** sorts the files by column, newspaper-style, just as the Unix and GNU **ls** commands do (and *unlike* the simplistic sort by row performed by the DOS, Win32, and OS/2 DIR commands).

Usage

```
ls [ -flags ] [ -- ] [ fspec ... ]
```

The flags are optional. If specified, they must consist of a dash followed by one or more contiguous flag characters. The flags control what names are listed and in what format they are listed.

If no flags are specified, then `-fdn` is assumed: list all matching ordinary files and directories, and sort them by name.

Zero or more file specifications may be entered. Each file specification consists of an optional drive, optional path, and filename. Each filename may contain Bourne shell pattern-matching characters (better than DOS's, Win32's, and OS/2's wildcard characters, *and* supported on all of these platforms). The **ls** command lists all files in the specified directory whose names match the filename, according to the flags.

If a file specification is that of a directory, then **ls** appends `/*` to it and matches everything in that directory. If there are no file specifications, then **ls** assumes `*`: match everything in the current directory.

Flags

The following (case-sensitive) flag characters are supported:

`-f` Lists all ordinary (including read-only) files that match the file specification(s). If the `-f` flag is specified, then directories are not listed unless the `-d` flag is also specified.

`-s` Lists all system files.

`-h` Lists all hidden files.

`-d` Lists all subdirectories that match the file specification(s). If the `-d` flag is specified, then ordinary files are not listed unless the `-f` flag is also specified.

`-a` Lists all ordinary files, system files, hidden files, directories, the `.` and `..` directories, and, for DOS only, volume IDs.

`-l` Lists each matching file or directory name in "long" format. For each name, it also displays the attributes, size, time stamp, and date stamp. If `-l` is not specified, the **ls** command lists only the names.

`-c` Uses time of creation instead of time of last modification. If used with `-l`, it shows this time. If used with `-t`, it sorts using this time. (Win32 and OS/2 versions only.)

`-u` Uses time of last access instead of time of last modification. If used with `-l`, it shows this time. If used with `-t`, it sorts using this time (NTFS and HPFS only).

`-x` The same as `-u`, for compatibility with previous versions of **ls**.

`-R` Recursively descends subdirectories.

`-p` Puts a path separator after each name that is a directory.

`-C` Forces multicolumn output. The **ls** command tries to list more than one filename per line, depending upon the length of the longest name and the width of the display screen. Filenames are sorted vertically.

`-m` The same as `-C`, for compatibility with previous versions of **ls**.

`-1` Forces single (one) column output.

`-n` Sorts the output by name (default), in alphabetic order.

`-t` Sorts the output by date and time, from newest to oldest.

`-z` Sorts the output by filesize, from smallest to largest.

`-r` Reverses the order of the sort.

`-o` Don't sort: display filenames in the order in which they occur in each directory.

`-g` Lists each set of matching files as a group in the order that their fspecs are specified. For example, if `*.h *.c` are entered, then the `-g` flag causes all **.h** files to be displayed first and all **.c** files to be displayed next.

The `-g` flag causes the program to make a separate pass through a directory for each fspec, making '**ls**' run slower. The default behavior is to display all matching files as they are found, making only one pass through each directory.

`-L` Shows filenames in lowercase.

`-U` Shows filenames in uppercase.

You can mix and match the `f`, `s`, `h`, and `d` flags to include as many different file types as you wish. If you specify at least one of the `s`, `h`, or `d` flags, then you must explicitly specify `-f` to include ordinary files in the listing. If you don't specify one of the `f`, `s`, `h`, `d`, or `a` flags, then the default is `-fd`: list only ordinary files and directories.

You can specify the `-l` (letter el) flag to list file and/or directory names in "long" format. In this format, each name appears along with its mode, size, date, and time. The mode (attribute) consists of a set of characters that are interpreted as follows:

- The first character is:

  `d` The name is a directory.
  `-` The name is a file.

- The second character is:

  `r` The file is readable (always!).
  `-` The file is not readable (never!).

- The third character is:

  `w` The file is writable.
  `-` The file is read-only.

- The fourth character is:

  `s` The file is a system file.
  `-` The file is not a system file.

- The fifth character is:

  `h` The file is a hidden file.
  `-` The file is not a hidden file.

- The sixth character is:

  `a` The file has been changed (its archive bit is set).
  `-` The file's archive bit is not set.

Notes

An ordinary file is a directory entry whose directory, system hidden, and volid attribute (mode) bits are all zero.

Filenames are listed on standard output. Filespecs that don't match any files are listed on standard error.

A file specification consists of an optional drive, optional path information, and a filename. The filename may consist of Bourne shell pattern-matching characters. DOS's filename pattern matching is simplistic, Win32's and OS/2's is a lot better, but Bourne shell pattern matching is better still. Also, all versions (DOS, Win32, and OS2) of the commands in this package support the same enhanced level of pattern-matching.

A dot within a filename is treated as just another character and not assumed to be the beginning of a file extension. Indeed, the **ls** command has no concept of a file extension. This is compatible with the use of dots within Win32, NTFS, HPFS, and AIX filenames.

Flag characters are case sensitive. Name comparisons are case-insensitive, as specified by DOS, Win32, and OS/2.

If the `-l` (long format) flag is listed, each file's modification time and date are displayed. Specify the `-c` flag along with `-l` to display each file's time of creation. Or, specify the `-u` flag along with `-l` to display each file's time of last access. Note that the `c` and `u` flags don't yield valid times for a file system (such as DOS or OS/2 FAT) that don't maintain them. Also note that the DOS version simply shows the time of last modification for the `-lu` and `-lc` flags.

You may specify more than one set of flags. For example, you may enter the commands `ls -n -l *.c *.h` and `ls -nl *.c *.h` both list all **.c** and **.h** files in long format, sorted by name.

The order in which you specify certain flags is important. For example, if you specify `-l`, `-C`, and/or `-1`, the last one entered takes precedence. Likewise, if you specify `-o`, `-n`, `-z`, and/or `-t`, the last one entered overrides the others.

If the **ls** command is displaying filenames in multicolumn format and/or sorting the names, then **ls** first stores all of the filenames in memory, sorts them (`n`, `t`, or `z` flag), and determines how many columns to display. The DOS version of the **ls** command may run out of memory for large file listings. If it does, you should re-run **ls** with the `-1o` (number 1 and letter o) flags: force single-column output and don't sort.

The DOS version of **ls** defaults to showing all filenames in lowercase since this is more readable (especially on smaller displays). The OS/2 version shows HPFS, NFS, and TVFS filenames as-is and shows all other filenames (including those on FAT filesystems) in lowercase. The Win32 version shows all filenames as-is. If you want uppercase filenames, then specify the `-U flag`. On OS/2 and Win32 platforms, the `-L` flag will force all names to be shown in lowercase.

The Watcom-built versions (Win32, OS/2, and DOS) do not list the volume label.

Enter `ls -?` to get help.

Examples

1. `ls *`
   `ls`
   `ls -fd *`

   These commands all mean the same thing: List all ordinary files and subdirectories within the current directory. They match the same files that the DOS command `DIR *.*` matches.

2. `ls *.*`

   This command lists all ordinary files in the current directory that have an extension (that is, that have a dot in their file name). This is not the same as the `ls *` command nor is it the same as the `DIR *.*` command, since it won't list any files or directories whose name contains no dot.

3. `ls -Cn`

   List all files in the current directory in multi-column format, sorted by name. This is the default behaviour for the **ls** command when stdout is the display (and when **ls** is entered with no flags or arguments).

4. `ls *.c`

   List all files in the current directory that end in the **.c** suffix (or have the **.c** extension, in DOS's words).

5. `ls *.c *.h`
   `ls *.[ch]`

   List all files in the current directory that end in either **.c** or **.h**. These commands show the same results. In the first command, notice that you can enter more than one file specification. In the second command, notice that the one Bourne shell pattern can replace the two simplistic DOS-sytle patterns.

6. `ls -Rd c:/`
   `ls -Rd c:\`

   List all directories and subdirectories on drive `C`, starting at the root directory. This produces a tree listing of drive `C`.

   Both `\` (backslash) and `/` (forward slash) characters are interpreted as path separators. When showing the matching directory entries, **ls** will use the last path separator (if any) that you entered on the command line.

   If you didn't enter a path separator on the command line, then **ls** uses a default path separator of `\` if the `USEP` environment variable is set to `\` and a default path separator of `/` otherwise.

7. `ls -a a:`

   List all files and subdirectories in the current directory of the `A` drive.

8. `ls -lsh *`

   This command lists only hidden and system files in the current directory, along with their mode (attributes), sizes, and time and date stamps.

9. `ls -d *`

   This command lists all subdirectories within the current directory. The DOS `DIR *.` command sort of does the same thing, provided that directory names have no extension and file names always have an extension.

10. `ls -l`

    List each file in the current directory, along with its mode, size, and time and date of last modification.

11. `ls -lc`

    Like `-l`, except display each file's time and date of creation (Win32 and OS/2 versions only).

12. `ls -lu`

    Like -l, except display each file's time and date of last access (NTFS and HPFS only).

13. `ls -R c:/[a-m]*.exe`

    List all **.exe** files on drive `C` that begin with the letters A through M, inclusive.

14. `ls -R c:/my.bat`

    List all occurances of the **my.bat** file on drive `C`. This command performs a "whereis" type of function.

15. `ls -R *.c`

    List all **.c** files in the current directory and, recursively, in any subdirectories within the current directory.

16. `ls -lRdn /`

    List all directories on the current drive in "long" format, sorted by directory name.

17. `ls -lt`

    List all files in the current directory in "long" format, sorted by date and time of modification.

18. `ls -lut`

    List all files in the current directory in "long" format, sorted by date and time of last access (NTFS and HPFS only).

19. `ls -luo`

    List all files in the current directory in "long" format, in the order in which they occur in the directory (unsorted). Display the time of last access for each one.

20. `ls -lRn`
    `ls -l -Rn`
    `ls -l -R -n`

    You may specify flags in one or more groups. For example, these commands all produce identical results.

21. `ls -lC1`

    The same command as `ls -1`. The `C` flag disables the `l` flag, and the `1` flag in turn disables the `C` flag.

strings—Shows printable strings

The **strings** command extracts and shows the printable strings in one or more files including binary files. It is based on the *ix **strings** command, except that it shows offsets in hexadecimal rather than octal.

Usage

```
strings [ -a ] [ -o ] [ -# ] [ -s ] [ fspec ... ]
```

The **strings** command searches one or more files for printable ASCII strings and writes them to standard output. If no files are specified, then the command searches standard input in binary mode.

The **isprint()** library function is used to test for printable ASCII characters. A sequence of printable characters must be terminated with a null, carriage return, or line feed character to be considered a valid string.

Flags

The flags that are supported are similar to those supported by the standard Unix **strings** command:

- `-a` Search the entire file. This flag is present for command-line compatibility: this **strings** command always searches files in their entirety.
- `-o` Also show the offset of each string found. The offset is shown in hexadecimal and not octal, unlike the Unix **strings** command.
- `-#` Specifiy a number for `#`. Strings must have a minimum length of `#` characters before they are shown. The default minimum number of characters is 4.
- `-s` Descend subdirectories, also.
- `-S` Search system and hidden files, also.

txtcut—Text Preprocessor

Text files that contain blank lines, comments preceded by # (pound signs), a varying amount of whitespace between tokens, and other features to make them well-documented and readable (such as INI-style ASCII text files) are not easily processed by a Korn shell or REXX script and can even require a bit of extra work for Perl scripts. Additional logic can be used to skip over blank lines and remove comments, but this logic adds undesirable complexity and slow performance to the script.

One solution is **txtcut**. This program prepares a text file, stripping out comments and blank lines and handling simple strings. It was originally developed as a fast preprocessor for the AIX **cut** command. However, the output of **txtcut** can be piped into **rxqueue**, giving OS/2's REXX the ability to easily and quickly handle text-style INI-like files. Even **awk** and Perl (of which excellent Win32 and OS/2 ports exist) can benefit from **txtcut**.

Information can be stored in an INI-style text file in an easily- maintained and readable fashion using comments, strings, and blank space. The **txtcut** program preprocesses this text file, removing comments and blank lines, processing simple strings, and delimiting the tokens in a consistent manner. The tokens produced by **txtcut** can be easily extracted using the AIX **cut** command, a REXX script, or even an **awk** or Perl script.

The REXX, awk, Perl, and the Unix shells cover a lot of ground, but I have wished for a long time for the capabilities of **txtcut** to make the basic AIX, DOS, Win32, OS/2 toolsets more complete. I even maintain an AIX port of **txtcut** which I use daily and on which the DOS, Win32, and OS/2 ports are based. Text INI-style files that are highly readable by people *and* easily processed by command scripts: **txtcut** gives you the best of both worlds!

Usage

```
txtcut [ -dchar ] [ -n ] [ -l ] [ -c ] [ textfile ]
```

This program prepares a text file for the AIX **cut** command. If the name of the text file is missing, then **txtcut** reads from standard input.

For each line that contains one or more tokens, **txtcut** writes one line to stdout that contains the tokens. A delimiter character is placed between each pair of tokens.

The default delimiter character is a tab character. You may use the `-d` flag to change it to another character.

Flags

`-d#` Specify a character for `#` to be used as the token delimiter. This character is specified just as it is for the **cut** command. The default is a tab character.

`-n` List the filename as the first token in each line. If the text file is being read from standard input, then the filename is listed as "`(stdin)`".

`-l` List the line number within the file as a token.

`-c` List the number of tokens on the line as a token. This count does not include the additional tokens that may be added by the `-n`, `-l` or `-c` flags.

Input Text File Format

Blank lines are ignored. Lines that begin with a `#` or `;` are assumed to be comments and are ignored.

Each nonblank, noncomment line consists of one or more tokens. A token can be:

- An `=` (equal) sign.
- A string of characters enclosed by either double quotes, single quotes, or square brackets.
- A series of contiguous nonblank characters.

If a non-string token that begins with a `#` or a `;` is encountered on a line, it and the remaining tokens are assumed to be comments and are ignored.

Examples

1. Assume that the **sample.txt** file contains the following:

```
# This is a comment line

        f1 = a b c    # First line of tokens
        x y z         # Second line of tokens
        aaa bbbb  "cccc   ddddd"  # Last line of tokens
```

Run the following command:

```
txtcut -d: sample.txt
```

The **txtcut** command writes the following to stdout:

```
f1:=:a:b:c
x:y:z
aaa:bbbb:cccc   ddddd
```

2. Using the same **sample.txt** file from the previous example, we'll cut the third field from each line of the file by running the following command:

```
txtcut sample.txt | cut -f3
```

The following is written to stdout:

```
a
z
cccc   ddddd
```

3. Again, using the same **sample.txt** file from the previous examples, we'll cut the third field from each line of the file. But this time we'll pipe the file into **txtcut**'s standard input and we'll use a colon for the delimiter:

```
cat sample.txt | txtcut -d: | cut -f3 -d:
```

Again, the following is written to stdout:

```
a
z
cccc   ddddd
```

4. Using the same sample.txt file in example 1, we will add the filename, the line number within the file where each line of tokens was found, and the number of tokens in the line. The following shows a set of commands and the output from each of them:

(a) `txtcut -d: sample.txt`

```
f1:=:a:b:c
x:y:z
aaa:bbbb:cccc   ddddd
```

(b) `txtcut -d: -n sample.txt`

```
sample.txt:f1:=:a:b:c
sample.txt:x:y:z
sample.txt:aaa:bbbb:cccc   ddddd
```

(c) `txtcut -d: -l sample.txt`

```
3:f1:=:a:b:c
4:x:y:z
5:aaa:bbbb:cccc   ddddd
```

(d) `txtcut -d: -n -l sample.txt`

```
sample.txt:3:f1:=:a:b:c
sample.txt:4:x:y:z
sample.txt:5:aaa:bbbb:cccc   ddddd
```

(e) `txtcut -d: -n -l -c sample.txt`

```
sample.txt:3:5:f1:=:a:b:c
sample.txt:4:3:x:y:z
sample.txt:5:3:aaa:bbbb:cccc   ddddd
```