NAME

acd − a compiler driver

SYNOPSIS

acd **−v**[*n*] **−vn**[*n*] **−name** *name* **−descr** *descr* **−T** *dir* [*arg* ...]

DESCRIPTION

Acd is a compiler driver, a program that calls the several passes that are needed to compile a source file. It keeps track of all the temporary files used between the passes. It also defines the interface of the compiler, the options the user gets to see.

This text only describes **acd** itself, it says nothing about the different options the C-compiler accepts. (It has nothing to do with any language, other than being a tool to give a compiler a user interface.)

OPTIONS

Acd itself takes five options:

−v[*n*] Sets the diagnostic level to *n* (by default **2**). The higher *n* is, the more output **acd** generates: **−v0** does not produce any output. **−v1** prints the basenames of the programs called. **−v2** prints names and arguments of the programs called. **−v3** shows the commands executed from the description file too. **−v4** shows the program read from the description file too. Levels 3 and 4 use backspace overstrikes that look good when viewing the output with a smart pager.

−vn[*n*] Like **−v** except that no command is executed. The driver is just play-acting.

−name *name*
Acd is normally linked to the name the compiler is to be called with by the user. The basename of this, say **cc**, is the call name of the driver. It plays a role in selecting the proper description file. With the **−name** option one can change this. **Acd −name cc** has the same effect as calling the program as **cc**.

−descr *descr*
Allows one to choose the pass description file of the driver. By default *descr* is the same as *name*, the call name of the program. If *descr* doesn't start with **/**, **./**, or **../** then the file **/usr/lib/***descr***/descr** will be used for the description, otherwise *descr* itself. Thus **cc −descr newcc** calls the C-compiler with a different description file without changing the call name. Finally, if *descr* is **"−"**, standard input is read. (The default lib directory **/usr/lib**, may be changed to *dir* at compile time by **−DLIB=**\"*dir*\". The default *descr* may be set with **−DDESCR=**\"*descr*\" for simple installations on a system without symlinks.)

−T *dir* Temporary files are made in **/tmp** by default, which may be overridden by the environment variable **TMPDIR**, which may be overridden by the **−T** option.

THE DESCRIPTION FILE

The description file is a program interpreted by the driver. It has variables, lists of files, argument parsing commands, and rules for transforming input files.

Syntax

There are four simple objects:

Words, Substitutions, Letters, and Operators.

And there are two ways to group objects:

Lists, forming sequences of anything but letters,

Strings, forming sequences of anything but Words and Operators.

Each object has the following syntax:

Words They are sequences of characters, like **cc**, **−I/usr/include**, **/lib/cpp**. No whitespace and no special characters. The backslash character (\) may be used to make special characters common, except whitespace. A backslash followed by whitespace is completely removed from the input. The sequence **\n** is changed to a newline.

Substitutions

A substitution (henceforth called 'subst') is formed with a **$**, e.g. **$opt**, **$PATH**, **${lib}**, **$***. The variable name after the **$** is made of letters, digits and underscores, or any sequence of characters between parentheses or braces, or a single other character. A subst indicates that the value of the named variable must be substituted in the list or string when fully evaluated.

Letters Letters are the single characters that would make up a word.

Operators

The characters =, +, −, *, <, and > are the operators. The first four must be surrounded by whitespace if they are to be seen as special (they are often used in arguments). The last two are always special.

Lists One line of objects in the description file forms a list. Put parentheses around it and you have a sublist. The values of variables are lists.

Strings Anything that is not yet a word is a string. All it needs is that the substs in it are evaluated, e.g. **$LIBPATH/lib$key.a**. A single subst doesn't make a string, it expands to a list. You need at least one letter or other subst next to it. Strings (and words) may also be formed by enclosing them in double quotes. Only \ and **$** keep their special meaning within quotes.

Evaluation

One thing has to be carefully understood: Substitutions are delayed until the last possible moment, and description files make heavy use of this. Only if a subst is tainted, either because its variable is declared local, or because a subst in its variable's value is tainted, is it immediately substituted. So if a list is assigned to a variable then this list is only checked for tainted substs. Those substs are replaced by the value of their variable. This is called partial evaluation.

Full evaluation expands all substs, the list is flattened, i.e. all parentheses are removed from sublists.

Implosive evaluation is the last that has to be done to a list before it can be used as a command to execute. The substs within a string have been evaluated to lists after full expansion, but a string must be turned into a single word, not a list. To make this happen, a string is first exploded to all possible combinations of words choosing one member of the lists within the string. These words are tried one by one to see if they exist as a file. The first one that exists is taken, if none exists than the first choice is used. As an example, assume **LIBPATH** equals **(/lib /usr/lib)**, **key** is **(c)** and **key** happens to be local. Then we have:

"$LIBPATH/lib$key.a"

before evaluation,

"$LIBPATH/lib(c).a"

after partial evaluation,

"(/lib/libc.a /usr/lib/libc.a)"

after full evaluation, and finally

/usr/lib/libc.a

after implosion, if the file exists.

Operators

The operators modify the way evaluation is done and perform a special function on a list:

* Forces full evaluation on all the list elements following it. Use it to force substitution of the current value of a variable. This is the only operator that forces immediate evaluation.

+ When a + exists in a list that is fully evaluated, then all the elements before the + are imploded and all elements after the + are imploded and added to the list if they are not already in the list. So this operator can be used either for set addition, or to force implosive expansion within a sublist.

− Like +, except that elements after the − are removed from the list.

The set operators can be used to gather options that exclude each other or for their side effect of implosive expansion. You may want to write:

cpp −I$LIBPATH/include

to call cpp with an extra include directory, but **$LIBPATH** is expanded using a filename starting with **−I** so this won't work. Given that any problem in Computer Science can be solved with an extra level of indirection, use this instead:

cpp −I$INCLUDE
INCLUDE = $LIBPATH/include +

Special Variables

There are three special variables used in a description file: **$***, **$<**, and **$>**. These variables are always local and mostly read-only. They will be explained later.

A Program

The lists in a description file form a program that is executed from the first to the last list. The first word in a list may be recognized as a builtin command (only if the first list element is indeed simply a word.) If it is not a builtin command then the list is imploded and used as a UNIX command with arguments.

Indentation (by tabs or spaces) is not just makeup for a program, but are used to group lines together. Some builtin commands need a body. These bodies are simply lines at a deeper indentation.

Empty lines are not ignored either, they have the same indentation level as the line before it. Comments (starting with a # and ending at end of line) have an indentation of their own and can be used as null commands.

Acd will complain about unexpected indentation shifts and empty bodies. Commands can share the same body by placing them at the same indentation level before the indented body. They are then "guards" to the same body, and are tried one by one until one succeeds, after which the body is executed.

Semicolons may be used to separate commands instead of newlines. The commands are then all at the indentation level of the first.

Execution phases

The driver runs in three phases: Initialization, Argument scanning, and Compilation. Not all commands work in all phases. This is further explained below.

The Commands

The commands accept arguments that are usually generic expressions that implode to a word or a list of words. When *var* is specified, then a single word or subst needs to be given, so an assignment can be either *name* **=** *value*, or **$***name* **=** *value*.

var **=** *expr ...*

The partially evaluated list of expressions is assigned to *var*. During the evaluation is *var* marked as local, and after the assignment set from undefined to defined.

unset *var*

Var is set to null and is marked as undefined.

import *var*

If *var* is defined in the environment of **acd** then it is assigned to *var*. The environment variable is split into words at whitespace and colons. Empty space between two colons (**::**) is changed to a dot.

mktemp *var* **[***suffix***]**

Assigns to *var* the name of a new temporary file, usually something like /tmp/acd12345x. If *suffix* is present then it will be added to the temporary file's name. (Use it because some programs require it, or just because it looks good.) **Acd** remembers this file, and will delete it as soon as you stop referencing it.

temporary *word*

Mark the file named by *word* as a temporary file. You have to make sure that the name is stored in some list in imploded form, and not just temporarily created when *word* is evaluated, because then it will be immediately removed and forgotten.

stop *suffix*

Sets the target suffix for the compilation phase. Something like **stop .o** means that the source files must be compiled to object files. At least one **stop** command must be executed before the compilation phase begins. It may not be changed during the compilation phase. (Note: There is no restriction on *suffix*, it need not start with a dot.)

treat *file suffix*

Marks the file as having the given suffix for the compile phase. Useful for sending a **−l** option directly to the loader by treating it as having the **.a** suffix.

numeric *arg*

Checks if *arg* is a number. If not then **acd** will exit with a nice error message.

error *expr ...*

Makes the driver print the error message *expr ...* and exit.

if *expr = expr*

If tests if the two expressions are equal using set comparison, i.e. each expression should contain all the words in the other expression. If the test succeeds then the if-body is executed.

ifdef *var*

Executes the ifdef-body if *var* is defined.

ifndef *var*

Executes the ifndef-body if *var* is undefined.

iftemp *arg*

Executes the iftemp-body if *arg* is a temporary file. Use it when a command has the same file as input and output and you don't want to clobber the source file:

```
transform .o .o
        iftemp $*
                $> = $*
        else
                cp $* $>
        optimize $>
```

ifhash *arg*

Executes the ifhash-body if *arg* is an existing file with a '#' as the very first character. This usually indicates that the file must be pre-processed:

```
transform .s .o
        ifhash $*
                mktemp ASM .s
                $CPP $* > $ASM
        else
                ASM = $*
        $AS −o $> $ASM
        unset ASM
```

else Executes the else-body if the last executed **if**, **ifdef**, **ifndef**, **iftemp**, or **ifhash** was unsuccessful. Note that **else** need not immediately follow an if, but you are advised not to make use of this. It is a "feature" that may not last.

apply *suffix1 suffix2*

Executed inside a transform rule body to transform the input file according to another transform rule that has the given input and output suffixes. The file under **$*** will be replaced by the new file. So if there is a **.c .i** preprocessor rule then the example of **ifhash** can be

replaced by:

```
transform .s .o
        ifhash $*
                apply .c .i
        $AS -o $> $*
```

include *descr*
Reads another description file and replaces the **include** with it. Execution continues with the first list in the new program. The search for *descr* is the same as used for the **−descr** option. Use **include** to switch in different front ends or back ends, or to call a shared description file with a different initialization. Note that *descr* is only evaluated the first time the **include** is called. After that the **include** has been replaced with the included program, so changing its argument won't get you a different file.

arg *string ...*
Arg may be executed in the initialization and scanning phase to post an argument scanning rule, that's all the command itself does. Like an **if** that fails it allows more guards to share the same body.

transform *suffix1 suffix2*
Transform, like **arg**, only posts a rule to transform a file with the suffix *suffix1* into a file with the suffix *suffix2*.

prefer *suffix1 suffix2*
Tells that the transformation rule from *suffix1* to *suffix2* is to be preferred when looking for a transformation path to the stop suffix. Normally the shortest route to the stop suffix is used. **Prefer** is ignored on a **combine**, because the special nature of combines does not allow ambiguity.

The two suffixes on a **transform** or **prefer** may be the same, giving a rule that is only executed when preferred.

combine *suffix-list suffix*
Combine is like **transform** except that it allows a list of input suffixes to match several types of input files that must be combined into one.

scan The scanning phase may be run early from the initialization phase with the **scan** command. Use it if you need to make choices based on the arguments before posting the transformation rules. After running this, **scan** and **arg** become no-ops.

compile
Move on to the compilation phase early, so that you have a chance to run a few extra commands before exiting. This command implies a **scan**.

Any other command is seen as a UNIX command. This is where the < and > operators come into play. They redirect standard input and standard output to the file mentioned after them, just like the shell. **Acd** will stop with an error if the command is not successful.

The Initialization Phase

The driver starts by executing the program once from top to bottom to initialize variables and post argument scanning and transformation rules.

The Scanning Phase

In this phase the driver makes a pass over the command line arguments to process options. Each **arg** rule is tried one by one in the order they were posted against the front of the argument list. If a match is made then the matched arguments are removed from the argument list and the arg-body is executed. If no match can be made then the first argument is moved to the list of files waiting to be transformed and the scan is restarted.

The match is done as follows: Each of the strings after **arg** must match one argument at the front of the argument list. A character in a string must match a character in an argument word, a subst in a string may match 1 to all remaining characters in the argument, preferring the shortest possible match. The

hyphen in a argument starting with a hyphen cannot be matched by a subst. Therefore:

arg −i

matches only the argument **−i**.

arg −O$n

matches any argument that starts with **−O** and is at least three characters long. Lastly,

arg −o $out

matches **−o** and the argument following it, unless that argument starts with a hyphen.

The variable **$*** is set to all the matched arguments before the arg-body is executed. All the substs in the arg strings are set to the characters they match. The variable **$>** is set to null. All the values of the variables are saved and the variables marked local. All variables except **$>** are marked read-only. After the arg-body is executed is the value of **$>** concatenated to the file list. This allows one to stuff new files into the transformation phase. These added names are not evaluated until the start of the next phase.

The Compilation Phase

The files gathered in the file list in the scanning phase are now transformed one by one using the transformation rules. The shortest, or preferred route is computed for each file all the way to the stop suffix. Each file is transformed until it lands at the stop suffix, or at a combine rule. After a while all files are either fully transformed or at a combine rule.

The driver chooses a combine rule that is not on a path from another combine rule and executes it. The file that results is then transformed until it again lands at a combine rule or the stop suffix. This continues until all files are at the stop suffix and the program exits.

The paths through transform rules may be ambiguous and have cycles, they will be resolved. But paths through combines must be unambiguous, because of the many paths from the different files that meet there. A description file will usually have only one combine rule for the loader. However if you do have a combine conflict then put a no-op transform rule in front of one to resolve the problem.

If a file matches a long and a short suffix then the long suffix is preferred. By putting a null input suffix ("") in a rule one can match any file that no other rule matches. You can send unknown files to the loader this way.

The variable **$*** is set to the file to be transformed or the files to be combined before the transform or combine-body is executed. **$>** is set to the output file name, it may again be modified. **$<** is set to the original name of the first file of **$*** with the leading directories and the suffix removed. **$*** will be made up of temporary files after the first rule. **$>** will be another temporary file or the name of the target file (**$<** plus the stop suffix), if the stop suffix is reached.

$> is passed to the next rule; it is imploded and checked to be a single word. This driver does not store intermediate object files in the current directory like most other compilers, but keeps them in **/tmp** too. (Who knows if the current directory can have files created in?) As an example, here is how you can express the "normal" method:

```
transform .s .o
        if $> = $<.o
                # Stop suffix is .o
        else
                $> = $<.o
                temporary $>
        $AS −o $> $*
```

Note that **temporary** is not called if the target is already the object file, or you would lose the intended result! **$>** is known to be a word, because **$<** is local. (Any string whose substs are all expanded changes to a word.)

Predefined Variables

The driver has three variables predefined: **PROGRAM**, set to the call name of the driver, **VERSION**,

the driver's version number, and **ARCH**, set to the name of the default output architecture. The latter is optional, and only defined if **acd** was compiled with **−DARCH=**\"*arch-name*\".

EXAMPLE

As an example a description file for a C compiler is given. It has a front end (ccom), an intermediate code optimizer (opt), a code generator (cg), an assembler (as), and a loader (ld). The compiler can pre-process, but there is also a separate cpp. If the **−D** and options like it are changed to look like **−o** then this example is even as required by POSIX.

```
# The compiler support search path.
C =     /lib /usr/lib /usr/local/lib

# Compiler passes.
CPP =   $C/cpp $CPP_F
CCOM =          $C/ccom $CPP_F
OPT =   $C/opt
CG =    $C/cg
AS =    $C/as
LD =    $C/ld

# Predefined symbols.
CPP_F =         −D__EXAMPLE_CC__

# Library path.
LIBPATH = $USERLIBPATH $C

# Default transformation target.
stop .out

# Preprocessor directives.
arg −D$name
arg −U$name
arg −I$dir
        CPP_F = $CPP_F $*

# Stop suffix.
arg −c
        stop .o

arg −E
        stop .E

# Optimization.
arg −O
        prefer .m .m
        OPT = $OPT -O1

arg −O$n
        numeric $n
        prefer .m .m
        OPT = $OPT $*

# Add debug info to the executable.
arg −g
        CCOM = $CCOM -g
```

```
# Add directories to the library path.
arg −L$dir
        USERLIBPATH = $USERLIBPATH $dir

# −llib must be searched in $LIBPATH later.
arg −l$lib
        $> = $LIBPATH/lib$lib.a

# Change output file.
arg −o$out
arg −o $out
        OUT = $out

# Complain about a missing argument.
arg −o
        error "argument expected after '$*'"

# Any other option (like −s) are for the loader.
arg −$any
        LD = $LD $*

# Preprocess C-source.
transform .c .i
        $CPP $* > $>

# Preprocess C-source and send it to standard output or $OUT.
transform .c .E
        ifndef OUT
                $CPP $*
        else
                $CPP $* > $OUT

# Compile C-source to intermediate code.
transform .c .m
transform .i .m
        $CCOM $* $>

# Intermediate code optimizer.
transform .m .m
        $OPT $* > $>

# Intermediate to assembly.
transform .m .s
        $CG $* > $>

# Assembler to object code.
transform .s .o
        if $> = $<.o
                ifdef OUT
                        $> = $OUT
        $AS −o $> $*

# Combine object files and libraries to an executable.
combine (.o .a) .out
        ifndef OUT
```

```
OUT = a.out
$LD -o $OUT $C/crtso.o $* $C/libc.a
```

FILES

/usr/lib/*descr*/descr − compiler driver description file.

SEE ALSO

cc(1).

ACKNOWLEDGEMENTS

Even though the end result doesn't look much like it, many ideas were nevertheless derived from the ACK compiler driver by Ed Keizer.

BUGS

POSIX requires that if compiling one source file to an object file fails then the compiler should continue with the next source file. There is no way **acd** can do this, it always stops after error. It doesn't even know what an object file is! (The requirement is stupid anyhow.)

If you don't think that tabs are 8 spaces wide, then don't mix them with spaces for indentation.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

anm − print name list

SYNOPSIS

anm [−**gnoprus**] *file* ...

OPTIONS

−g Global symbols only

−n Sort numerically

−o Prepend the filename to each line

−p No sorting—use symbol table order

−r Sort in reverse order

−u List undefined symbols only

−s Sort in section order

EXAMPLES

anm −gn test.o # Print global symbols in numerical order

DESCRIPTION

Anm prints the name list (symbol table) of each ACK format object *file* in the argument list. If no file name is given, *a.out* is used. Each symbol name is preceded by its value, a section indicator and a type indicator. The section indicators are:

U Undefined symbol
A Absolute symbol
− Other symbol

The type indicators are:

F Filename
M Module name
S Section name
E External (global) symbol
− Local symbol

The output is sorted alphabetically, unless otherwise specified. Notice that *anm* can only be used on ACK format object files (that is: *.o* and *.out* files). If you want to get the name list of an executable program use *nm* instead.

SEE ALSO

asize(1), **nm**(1), **ar**(1), **size**(1).

NAME

ar, aal − archivers

SYNOPSIS

ar [**dmpqrtx**][**abciluv**] [*posname*] *archive* [*file* ...]
aal [**dpqrtx**][**clv**] *archive* [*file* ...]

EXAMPLES

ar r libc.a sort.s # Replace *sort*.s in *libc.a*

ar rb a.s libc.a b.s # Insert *b.s* before *a.s* in *libc.a*

DESCRIPTION

Ar allows groups of files to be put together into a single archive. It is normally used for libraries of compiled procedures. *Aal* is like *ar*, but is to be used with the ACK compiler. The following keys are allowed:

- **d**: Delete. *Ar* will delete the named members.
- **m**: Move named files. *Ar* expects *a*, *b*, or *i* to be specified.
- **p**: Print the named files (list them on *stdout*)
- **q**: Quickly append to the end of the archive file.
- **r**: Replace (append when not in archive).
- **t**: Print the archive's table of contents.
- **x**: Extract

The keys may optionally concatencated with one or more of the following:

- **a**: After *posname*
- **b**: Before *posname*
- **c**: Create (suppresses creation message)
- **i**: Before *posname*
- **l**: Local temporary file for work instead of */tmp/ar.$$$$$*
- **u**: Replace only if dated later than member in archive
- **v**: Verbose

SEE ALSO

anm(1), **asize**(1), **nm**(1), **size**(1).

NAME

ascii − strip all the pure ASCII lines from a file

SYNOPSIS

ascii [**−n**] [*file*]

OPTIONS

−n Extract the lines containing nonASCII characters

EXAMPLES

ascii file >outf # Write all the ASCII lines on *outf*

ascii −n <file >outf # Write all the nonASCII lines on *outf*

DESCRIPTION

Sometimes a file contains some nonASCII characters that are in the way. This program allows the lines containing only ASCII characters to be *grepped* from the file. With the **−n** flag, the nonASCII lines are *grepped*. No matter whether the flag is used or not, the program returns an exit status of true if the file is pure ASCII, and false otherwise.

SEE ALSO

tr(1).

NAME

ash, command, getopts, hash, jobs, local, return, setvar, unset − a shell

SYNOPSIS

ash [**-efIijnsxz**] [**+efIijnsxz**] [**-c** *command*] [*arg*] ...

COPYRIGHT

Copyright 1989 by Kenneth Almquist.

DESCRIPTION

Ash is a version of *sh* with features similar to those of the System V shell. This manual page lists all the features of *ash* but concentrates on the ones not in other shells.

Invocation

If the **-c** options is given, then the shell executes the specified shell command. The **-s** flag cause the shell to read commands from the standard input (after executing any command specified with the **-c** option. If neither the **-s** or **-c** options are set, then the first *arg* is taken as the name of a file to read commands from. If this is impossible because there are no arguments following the options, then *ash* will set the **-s** flag and will read commands from the standard input.

The shell sets the initial value of the positional parameters from the *arg*s remaining after any *arg* used as the name of a file of commands is deleted.

The flags (other than **-c**) are set by preceding them with ‘‘-’’ and cleared by preceding them with ‘‘+’’; see the *set* builtin command for a list of flags. If no value is specified for the **-i** flag, the **-s** flag is set, and the standard input and output of the shell are connected to terminals, then the **-i** flag will be set. If no value is specified for the **-j** flag, then the **-j** flag will be set if the **-i** flag is set.

When the shell is invoked with the **-c** option, it is good practice to include the -*i* flag if the command was entered interactively by a user. For compatibility with the System V shell, the -*i* option should come after the **-c** option.

If the first character of argument zero to the shell is ‘‘-’’, the shell is assumed to be a login shell, and the files **/etc/profile** and **.profile** are read if they exist. If the environment variable SHINIT is set on entry to the shell, the commands in SHINIT are normally parsed and executed. SHINIT is not examined if the shell is a login shell, or if it the shell is running a shell procedure. (A shell is considered to be running a shell procedure if neither the **-s** nor the **-c** options are set.)

Control Structures

A *list* is a sequence of zero or more commands separated by newlines, semicolons, or ampersands, and optionally terminated by one of these three characters. (This differs from the System V shell, which requires a list to contain at least one command in most cases.) The commands in a list are executed in the order they are written. If command is followed by an ampersand, the shell starts the command and immediately proceed onto the next command; otherwise it waits for the command to terminate before proceeding to the next one.

‘‘&&’’ and ‘‘||’’ are binary operators. ‘‘&&’’ executes the first command, and then executes the second command iff the exit status of the first command is zero. ‘‘||’’ is similar, but executes the second command iff the exit status of the first command is nonzero. ‘‘&&’’ and ‘‘||’’ both have the same priority.

The ‘‘|’’ operator is a binary operator which feeds the standard output of the first command into the standard input of the second command. The exit status of the ‘‘|’’ operator is the exit status of the second command. ‘‘|’’ has a higher priority than ‘‘||’’ or ‘‘&&’’.

An *if* command looks like

if list
then list
[**elif** list
then list] ...

[**else** list]
fi

A *while* command looks like

while list
do list
done

The two lists are executed repeatedly while the exit status of the first list is zero. The *until* command is similar, but has the word **until** in place of **while**
repeats until the exit status of the first list is zero.

The *for* command looks like

for variable **in** word...
do list
done

The words are expanded, and then the list is executed repeatedly with the variable set to each word in turn. **do** and **done** may be replaced with ‘‘{’’ and ‘‘}’’.

The *break* and *continue* commands look like

break [num]
continue [num]

Break terminates the *num* innermost *for* or *while* loops. *Continue* continues with the next iteration of the *num’th* innermost loop. These are implemented as builtin commands.

The *case* command looks like

case word **in**
pattern**)** list **;;**
...
esac

The pattern can actually be one or more patterns (see *Patterns* below), separated by ‘‘|’’ characters.

Commands may be grouped by writing either

(list**)**

or

{ list; **}**

The first of these executes the commands in a subshell.

A function definition looks like

name **( )** command

A function definition is an executable statement; when executed it installs a function named **name** and returns an exit status of zero. The command is normally a list enclosed between ‘‘{’’ and ‘‘}’’.

Variables may be declared to be local to a function by using a *local* command. This should appear as the first staement of a function, and looks like

local [variable | -] ...

Local is implemented as a builtin command.

When a variable is made local, it inherits the initial value and exported and readonly flags from the variable with the same name in the surrounding scope, if there is one. Otherwise, the variable is initially unset. *Ash* uses dynamic scoping, so that if you make the variable **x** local to function *f*, which then calls function *g*, references to the variable **x** made inside *g* will refer to the variable **x** declared inside *f*, not to the global variable named **x**.

The only special parameter than can be made local is ''-''. Making ''-'' local any shell options that are changed via the *set* command inside the function to be restored to their original values when the function returns.

The *return* command looks like

return [exitstatus]

It terminates the currently executing function. *Return* is implemented as a builtin command.

Simple Commands

A simple command is a sequence of words. The execution of a simple command proceeds as follows. First, the leading words of the form ''name=value'' are stripped off and assigned to the environment of the command. Second, the words are expanded. Third, the first remaining word is taken as the command name that command is located. Fourth, any redirections are performed. Fifth, the command is executed. We look at these operations in reverse order.

The execution of the command varies with the type of command. There are three types of commands: shell functions, builtin commands, and normal programs.

When a shell function is executed, all of the shell positional parameters (except $0, which remains unchanged) are set to the parameters to the shell function. The variables which are explicitly placed in the environment of the command (by placing assignments to them before the function name) are made local to the function and are set to values given. Then the command given in the function definition is executed. The positional parameters are restored to their original values when the command completes.

Shell builtins are executed internally to the shell, without spawning a new process.

When a normal program is executed, the shell runs the program, passing the parameters and the environment to the program. If the program is a shell procedure, the shell will interpret the program in a subshell. The shell will reinitialize itself in this case, so that the effect will be as if a new shell had been invoked to handle the shell procedure, except that the location of commands located in the parent shell will be remembered by the child. If the program is a file beginning with ''#!'', the remainder of the first line specifies an interpreter for the program. The shell (or the operating system, under Berkeley UNIX) will run the interpreter in this case. The arguments to the interpreter will consist of any arguments given on the first line of the program, followed by the name of the program, followed by the arguments passed to the program.

Redirection

Input/output redirections can be intermixed with the words in a simple command and can be placed following any of the other commands. When redirection occurs, the shell saves the old values of the file descriptors and restores them when the command completes. The ''<'', ''>'', and ''>>'' redirections open a file for input, output, and appending, respectively. The ''<&digit'' and ''>&digit'' makes the input or output a duplicate of the file descriptor numbered by the digit. If a minus sign is used in place of a digit, the standard input or standard output are closed.

The ''<< word'' redirection takes input from a *here* document. As the shell encounters ''<<'' redirections, it collects them. The next time it encounters an unescaped newline, it reads the documents in turn. The word following the ''<<'' specifies the contents of the line that terminates the document. If

none of the quoting methods (’’, "", or \) are used to enter the word, then the document is treated like a word inside double quotes: ‘‘$’’ and backquote are expanded and backslash can be used to escape these and to continue long lines. The word cannot contain any variable or command substitutions, and its length (after quoting) must be in the range of 1 to 79 characters. If ‘‘<<-’’ is used in place of ‘‘<<’’, then leading tabs are deleted from the lines of the document. (This is to allow you do indent shell procedures containing here documents in a natural fashion.)

Any of the preceding redirection operators may be preceded by a single digit specifying the file descriptor to be redirected. There cannot be any white space between the digit and the redirection operator.

Path Search

When locating a command, the shell first looks to see if it has a shell function by that name. Then, if PATH does not contain an entry for "%builtin", it looks for a builtin command by that name. Finally, it searches each entry in PATH in turn for the command.

The value of the PATH variable should be a series of entries separated by colons. Each entry consists of a directory name, or a directory name followed by a flag beginning with a percent sign. The current directory should be indicated by an empty directory name.

If no percent sign is present, then the entry causes the shell to search for the command in the specified directory. If the flag is ‘‘%builtin’’ then the list of shell builtin commands is searched. If the flag is ‘‘%func’’ then the directory is searched for a file which is read as input to the shell. This file should define a function whose name is the name of the command being searched for.

Command names containing a slash are simply executed without performing any of the above searches.

The Environment

The environment of a command is a set of name/value pairs. When the shell is invoked, it reads these names and values, sets the shell variables with these names to the corresponding values, and marks the variables as exported. The *export* command can be used to mark additional variables as exported.

The environment of a command is constructed by constructing name/value pairs from all the exported shell variables, and then modifying this set by the assignments which precede the command, if any.

Expansion

The process of evaluating words when a shell procedure is executed is called *expansion*. Expansion consists of four steps: variable substitution, command substitution, word splitting, and file name generation. If a word is the expression following the word **case** in a case statement, the file name which follows a redirection symbol, or an assignment to the environment of a command, then the word cannot be split into multiple words. In these cases, the last two steps of the expansion process are omitted.

Variable Substitution

To be written.

Command Substitution

Ash accepts two syntaxes for command substitution:

‘*list*‘

and

$(*list*)

Either of these may be included in a word. During the command substitution process, the command (syntactly a *list*) will be executed and anything that the command writes to the standard output will be

captured by the shell. The final newline (if any) of the output will be deleted; the rest of the output will be substituted for the command in the word.

Word Splitting

When the value of a variable or the output of a command is substituted, the resulting text is subject to word splitting, unless the dollar sign introducing the variable or backquotes containing the text were enclosed in double quotes. In addition, ‘‘$@’’ is subject to a special type of splitting, even in the presence of double quotes.

Ash uses two different splitting algorithms. The normal approach, which is intended for splitting text separated by which space, is used if the first character of the shell variable IFS is a space. Otherwise an alternative experimental algorithm, which is useful for splitting (possibly empty) fields separated by a separator character, is used.

When performing splitting, the shell scans the replacement text looking for a character (when IFS does not begin with a space) or a sequence of characters (when IFS does begin with a space), deletes the character or sequence of characters, and spits the word into two strings at that point. When IFS begins with a space, the shell deletes either of the strings if they are null. As a special case, if the word containing the replacement text is the null string, the word is deleted.

The variable ‘‘$@’’ is special in two ways. First, splitting takes place between the positional parameters, even if the text is enclosed in double quotes. Second, if the word containing the replacement text is the null string and there are no positional parameters, then the word is deleted. The result of these rules is that "$@" is equivalent to "$1" "$2" ... "$*n*", where *n* is the number of positional parameters. (Note that this differs from the System V shell. The System V documentation claims that "$@" behaves this way; in fact on the System V shell "$@" is equivalent to "" when there are no positional paramteters.)

File Name Generation

Unless the **-f** flag is set, file name generation is performed after word splitting is complete. Each word is viewed as a series of patterns, separated by slashes. The process of expansion replaces the word with the names of all existing files whose names can be formed by replacing each pattern with a string that matches the specified pattern. There are two restrictions on this: first, a pattern cannot match a string containing a slash, and second, a pattern cannot match a string starting with a period unless the first character of the pattern is a period.

If a word fails to match any files and the **-z** flag is not set, then the word will be left unchanged (except that the meta-characters will be converted to normal characters). If the **-z** flag is set, then the word is only left unchanged if none of the patterns contain a character that can match anything besides itself. Otherwise the **-z** flag forces the word to be replaced with the names of the files that it matches, even if there are zero names.

Patterns

A *pattern* consists of normal characters, which match themselves, and meta-characters. The meta-characters are ‘‘!’’, ‘‘*’’, ‘‘?’’, and ‘‘[’’. These characters lose there special meanings if they are quoted. When command or variable substitution is performed and the dollar sign or back quotes are not double quoted, the value of the variable or the output of the command is scanned for these characters and they are turned into meta-characters.

Two exclamation points at the beginning of a pattern function as a ‘‘not’’ operator, causing the pattern to match any string that the remainder of the pattern does *not* match. Other occurances of exclamation points in a pattern match exclamation points. Two exclamation points are required rather than one to decrease the incompatibility with the System V shell (which does not treat exclamation points specially).

An asterisk (‘‘*’’) matches any string of characters. A question mark matches any single character. A left bracket (‘‘[’’) introduces a character class. The end of the character class is indicated by a ‘‘]’’; if the ‘‘]’’ is missing then the ‘‘[’’ matches a ‘‘[’’ rather than introducing a character class. A character

class matches any of the characters between the square brackets. A range of characters may be specified using a minus sign. The character class may be complemented by making an exclamation point the first character of the character class.

To include a ‘‘]’’ in a character class, make it the first character listed (after the ‘‘!’’, if any). To include a minus sign, make it the first or last character listed.

The /u Directory

By convention, the name ‘‘/u/user’’ refers to the home directory of the specified user. There are good reasons why this feature should be supported by the file system (using a feature such as symbolic links) rather than by the shell, but *ash* is capable of performing this mapping if the file system doesn't. If the mapping is done by *ash*, setting the **-f** flag will turn it off.

Character Set

Ash silently discards nul characters. Any other character will be handled correctly by *ash*, including characters with the high order bit set.

Job Names and Job Control

The term *job* refers to a process created by a shell command, or in the case of a pipeline, to the set of processes in the pipeline. The ways to refer to a job are:

%number
%string
%%
process_id

The first form identifies a job by job number. When a command is run, *ash* assigns it a job number (the lowest unused number is assigned). The second form identifies a job by giving a prefix of the command used to create the job. The prefix must be unique. If there is only one job, then the null prefix will identify the job, so you can refer to the job by writing ‘‘%’’. The third form refers to the *current job*. The current job is the last job to be stopped while it was in the foreground. (See the next paragraph.) The last form identifies a job by giving the process id of the last process in the job.

If the operating system that *ash* is running on supports job control, *ash* will allow you to use it. In this case, typing the suspend character (typically ^Z) while running a command will return you to *ash* and will make the suspended command the current job. You can then continue the job in the background by typing *bg*, or you can continue it in the foreground by typing *fg*.

Atty

If the shell variable ATTY is set, and the shell variable TERM is not set to ‘‘emacs’’, then *ash* generates appropriate escape sequences to talk to *atty*(1).

Exit Statuses

By tradition, an exit status of zero means that a command has succeeded and a nonzero exit status indicates that the command failed. This is better than no convention at all, but in practice it is extremely useful to allow commands that succeed to use the exit status to return information to the caller. A variety of better conventions have been proposed, but none of them has met with universal approval. The convention used by *ash* and all the programs included in the *ash* distribution is as follows:

0	Success.
1	Alternate success.
2	Failure.
129-...	Command terminated by a signal.

The *alternate success* return is used by commands to indicate various conditions which are not errors but which can, with a little imagination, be conceived of as less successful than plain success. For example, *test* returns 1 when the tested condition is false and *getopts* returns 1 when there are no more

options. Because this convention is not used universally, the **-e** option of *ash* causes the shell to exit when a command returns 1 even though that contradicts the convention described here.

When a command is terminated by a signal, the uses 128 plus the signal number as the exit code for the command.

Builtin Commands

This concluding section lists the builtin commands which are builtin because they need to perform some operation that can't be performed by a separate process. In addition to these, there are several other commands (*catf*, *echo*, *expr*, *line*, *nlecho*, *test*, ''*:*'', and *true*) which can optionally be compiled into the shell. The builtin commands described below that accept options use the System V Release 2 *getopt*(3) syntax.

bg [*job*] ...
Continue the specified jobs (or the current job if no jobs are given) in the background. This command is only available on systems with Bekeley job control.

command *command arg*...
Execute the specified builtin command. (This is useful when you have a shell function with the same name as a builtin command.)

cd [*directory*]
Switch to the specified directory (default $HOME). If the an entry for CDPATH appears in the environment of the cd command or the shell variable CDPATH is set and the directory name does not begin with a slash, then the directories listed in CDPATH will be searched for the specified directory. The format of CDPATH is the same as that of PATH. In an interactive shell, the cd command will print out the name of the directory that it actually switched to if this is different from the name that the user gave. These may be different either because the CDPATH mechanism was used or because a symbolic link was crossed.

. *file*
The commands in the specified file are read and executed by the shell. A path search is not done to find the file because the directories in PATH generally contain files that are intended to be executed, not read.

eval *string*...
The strings are parsed as shell commands and executed. (This differs from the System V shell, which concatenates the arguments (separated by spaces) and parses the result as a single command.)

exec [*command arg*...]
Unless *command* is omitted, the shell process is replaced with the specified program (which must be a real program, not a shell builtin or function). Any redirections on the exec command are marked as permanent, so that they are not undone when the exec command finishes. If the command is not found, the exec command causes the shell to exit.

exit [*exitstatus*]
Terminate the shell process. If *exitstatus* is given it is used as the exit status of the shell; otherwise the exit status of the preceding command is used.

export *name*...
The specified names are exported so that they will appear in the environment of subsequent commands. The only way to un-export a variable is to unset it. *Ash* allows the value of a variable to be set at the same time it is exported by writing

export name=value

With no arguments the export command lists the names of all exported variables.

fg [*job*]
Move the specified job or the current job to the foreground. This command is only available on systems with Bekeley job control.

getopts *optstring var*
The System V *getopts* command.

hash -rv *command...*
The shell maintains a hash table which remembers the locations of commands. With no arguments whatsoever, the hash command prints out the contents of this table. Entries which have not been looked at since the last *cd* command are marked with an asterisk; it is possible for these entries to be invalid.

With arguments, the hash command removes the specified commands from the hash table (unless they are functions) and then locates them. With the **-v** option, *hash* prints the locations of the commands as it finds them. The **-r** option causes the *hash* command to delete all the entries in the hash table except for functions.

jobid [*job*]
Print the process id's of the processes in the job. If the job argument is omitted, use the current job.

jobs
This command lists out all the background processes which are children of the current shell process.

pwd
Print the current directory. The builtin command may differ from the program of the same name because the builtin command remembers what the current directory is rather than recomputing it each time. This makes it faster. However, if the current directory is renamed, the builtin version of pwd will continue to print the old name for the directory.

read [**-p** *prompt*] [**-e**] *variable...*
The prompt is printed if the **-p** option is specified and the standard input is a terminal. Then a line is read from the standard input. The trailing newline is deleted from the line and the line is split as described in the section on word splitting above, and the pieces are assigned to the variables in order. If there are more pieces than variables, the remaining pieces (along with the characters in IFS that separated them) are assigned to the last variable. If there are more variables than pieces, the remaining variables are assigned the null string.

The **-e** option causes any backslashes in the input to be treated specially. If a backslash is followed by a newline, the backslash and the newline will be deleted. If a backslash is followed by any other character, the backslash will be deleted and the following character will be treated as though it were not in IFS, even if it is.

readonly *name...*
The specified names are marked as read only, so that they cannot be subsequently modified or unset. *Ash* allows the value of a variable to be set at the same time it is marked read only by writing

readonly name=value

With no arguments the readonly command lists the names of all read only variables.

set [{ *-options* | *+options* | **--** }] *arg...*
The *set* command performs three different functions.

With no arguments, it lists the values of all shell variables.

If options are given, it sets the specified option flags, or clears them if the option flags are introduced with a **+** rather than a **-**. Only the first argument to *set* can contain options. The possible options are:

-e Causes the shell to exit when a command terminates with a nonzero exit status, except when the exit status of the command is explicitly tested. The exit status of a command is considered to be explicitly tested if the command is used to control an *if*, *elif*, *while*, or *until*; or if the command is the left hand operand of an ‘‘&&’’ or ‘‘||’’ operator.

-f Turn off file name generation.

-I Cause the shell to ignore end of file conditions. (This doesn’t apply when the shell a script sourced using the ‘‘.’’ command.) The shell will in fact exit if it gets 50 eof’s in a row.

-i Make the shell interactive. This causes the shell to prompt for input, to trap interrupts, to ignore quit and terminate signals, and to return to the main command loop rather than exiting on error.

-j Turns on Berkeley job control, on systems that support it. When the shell starts up, the **-j** is set by default if the **-i** flag is set.

-n Causes the shell to read commands but not execute them. (This is marginally useful for checking the syntax of scripts.)

-s If this flag is set when the shell starts up, the shell reads commands from its standard input. The shell doesn’t examine the value of this flag any other time.

-x If this flag is set, the shell will print out each command before executing it.

-z If this flag is set, the file name generation process may generate zero files. If it is not set, then a pattern which does not match any files will be replaced by a quoted version of the pattern.

The third use of the set command is to set the values of the shell’s positional parameters to the specified *args*. To change the positional parameters without changing any options, use ‘‘--’’ as the first argument to *set*. If no args are present, the set command will leave the value of the positional parameters unchanged, so to set the positional parameters to set of values that may be empty, execute the command

```
shift $#
```

first to clear out the old values of the positional parameters.

setvar *variable value*
Assigns *value* to *variable*. (In general it is better to write *variable=value* rather than using *setvar*. *Setvar* is intended to be used in functions that assign values to variables whose names are passed as parameters.)

shift [*n*]
Shift the positional parameters *n* times. A shift sets the value of $1 to the value of $2, the value of $2 to the value of $3, and so on, decreasing the value of $# by one. If there are zero positional parameters, shifting doesn’t do anything.

trap [*action*] *signal*...
Cause the shell to parse and execute *action* when any of the specified signals are received. The signals are specified by signal number. *Action* may be null or omitted; the former causes the specified signal to be ignored and the latter causes the default action to be taken. When the shell forks off a subshell, it resets trapped (but not ignored) signals to the default action. The trap command has no effect on signals that were ignored on entry to the shell.

umask [*mask*]
Set the value of umask (see *umask*(2)) to the specified octal value. If the argument is omitted, the

umask value is printed.

unset *name...*
The specified variables and functions are unset and unexported. If a given name corresponds to both a variable and a function, both the variable and the function are unset.

wait [*job*]
Wait for the specified job to complete and return the exit status of the last process in the job. If the argument is omitted, wait for all jobs to complete and the return an exit status of zero.

EXAMPLES

The following function redefines the *cd* command:

```
cd() {
        if command cd "$@"
        thenif test -f .enter
        then. .enter
        elsereturn 0
        fi
        fi
}
```

This function causes the file ‘‘.enter’’ to be read when you enter a directory, if it exists. The *command* command is used to access the real *cd* command. The ‘‘return 0’’ ensures that the function will return an exit status of zero if it successfully changes to a directory that does not contain a ‘‘.enter’’ file. Redefining existing commands is not always a good idea, but this example shows that you can do it if you want to.

The suspend function distributed with *ash* looks like

```
# Copyright (C) 1989 by Kenneth Almquist.  All rights reserved.
# This file is part of ash, which is distributed under the terms
# specified by the Ash General Public License.

suspend() {
        local -
        set +j
        kill -TSTP 0
}
```

This turns off job control and then sends a stop signal to the current process group, which suspends the shell. (When job control is turned on, the shell ignores the TSTP signal.) Job control will be turned back on when the function returns because ‘‘-’’ is local to the function. As an example of what *not* to do, consider an earlier version of *suspend*:

```
suspend() {
        suspend_flag=$-
        set +j
        kill -TSTP 0
        set -$suspend_flag
}
```

There are two problems with this. First, **suspend_flag** is a global variable rather than a local one, which will cause problems in the (unlikely) circumstance that the user is using that variable for some other purpose. Second, consider what happens if shell received an interrupt signal after it executes the first *set* command but before it executes the second one. The interrupt signal will abort the shell function, so that the second *set* command will never be executed and job control will be left off. The first version of *suspend* avoids this problem by turning job control off only in a local copy of the shell

options. The local copy of the shell options is discarded when the function is terminated, no matter how it is terminated.

HINTS

Shell variables can be used to provide abbreviations for things which you type frequently. For example, I set

```
export h=$HOME
```

in my .profile so that I can type the name of my home directory simply by typing ‘‘$h’’.

When writing shell procedures, try not to make assumptions about what is imported from the environment. Explicitly unset or initialize all variables, rather than assuming they will be unset. If you use cd, it is a good idea to unset CDPATH.

People sometimes use ‘‘<&-’’ or ‘‘>&-’’ to provide no input to a command or to discard the output of a command. A better way to do this is to redirect the input or output of the command to **/dev/null**.

Word splitting and file name generation are performed by default, and you have to explicitly use double quotes to suppress it. This is backwards, but you can learn to live with it. Just get in the habit of writing double quotes around variable and command substitutions, and omit them only when you really want word splitting and file name generation. If you want word splitting but not file name generation, use the **-f** option.

AUTHORS

Kenneth Almquist

SEE ALSO

echo(1), expr(1), line(1), pwd(1), true(1).

BUGS

When command substitution occurs inside a here document, the commands inside the here document are run with their standard input closed. For example, the following will not word because the standard input of the *line* command will be closed when the command is run:

```
cat <<-!
Line 1: $(line)
Line 2: $(line)
!
```

Unsetting a function which is currently being executed may cause strange behavior.

The shell syntax allows a here document to be terminated by an end of file as well as by a line containing the terminator word which follows the ‘‘<<’’. What this means is that if you mistype the terminator line, the shell will silently swallow up the rest of your shell script and stick it in the here document.

NAME

asize − report the size of an object file

SYNOPSIS

asize *file* ...

EXAMPLES

asize test.o # Give the size of *test.o*

DESCRIPTION

Asize prints for each argument the (decimal) number of bytes used by the different sections, as well as their sum in decimal and hexadecimal. If no *file* is given *a.out* is used. *Asize* can only be used to obtain the size of a *.o* or *.out* file. To obtain the size of an executable, use *size* instead.

SEE ALSO

anm(1), **nm**(1), **ar**(1), **size**(1).

NAME

at, atrun − execute commands at a later time

SYNOPSIS

at *time* [*month day*] [*file*]

EXAMPLES

at 2315 Jan 31 myfile # Myfile executed Jan 31 at 11:15 pm

at 0900 # Job input read from *stdin*

at 0711 4 29 # Read from *stdin*, exec on April 29

DESCRIPTION

At prepares a file to be executed later at the specified time by creating a special entry in */usr/spool/at*. The program *atrun* should be started periodically, for example, every minute by *cron*. *Atrun* checks to see if any files in */usr/spool/at* should now be run, and if so, it runs them and then puts them in */usr/spool/at/past*. The name of the file created in */usr/spool/at* by *at* is YY.DDD.HHMM.UU (where YY, DDD, HH, and MM give the time to execute and UU is a unique number). Note that when the command runs, it will not be able to use *stdin* or *stdout* unless specifically redirected. In the first example above, it might be necessary to put *>/dev/log* on some lines in the shell script *myfile*. The same holds for the commands typed directly to *at*.

SEE ALSO

cron(8).

NAME

banner − print a banner

SYNOPSIS

banner *arg* ...

EXAMPLES

banner happy birthday

Print a banner saying happy birthday

DESCRIPTION

Banner prints its arguments on *stdout* using a matrix of 6 x 6 pixels per character.

NAME

basename, dirname − strip off file prefixes and suffixes

SYNOPSIS

basename *file* [*suffix*]
dirname *file*

OPTIONS

(none)

EXAMPLES

basename /user/ast/file.c
Strips path to yield *file.c*

basename /user/file.c .c
Strips path and *.c* to yield *file*

dirname /user/file.c # Strips basename to yield */user*

DESCRIPTION

Basename removes the initial directory names (if any) yielding the name of the file itself. If a second argument is present, it is interpreted as a suffix and is also stripped, if present.

Dirname removes the final component of a path, yielding the directory a file is in.

These programs are primarily used in shell scripts.

NAME

bc - An arbitrary precision calculator language

SYNTAX

bc [**-lws**] [*file ...*]

VERSION

This man page documents GNU bc version 1.02.

DESCRIPTION

bc is a language that supports arbitrary precision numbers with interactive execution of statements. There are some similarities in the syntax to the C programming language. A standard math library is available by command line option. If requested, the math library is defined before processing any files. **bc** starts by processing code from all the files listed on the command line in the order listed. After all files have been processed, **bc** reads from the standard input. All code is executed as it is read. (If a file contains a command to halt the processor, **bc** will never read from the standard input.)

This version of **bc** contains several extensions beyond traditional **bc** implementations and the POSIX draft standard. Command line options can cause these extensions to print a warning or to be rejected. This document describes the language accepted by this processor. Extensions will be identified as such.

OPTIONS

-l Define the standard math library.

-w Give warnings for extensions to POSIX **bc**.

-s Process exactly the POSIX **bc** language.

NUMBERS

The most basic element in **bc** is the number. Numbers are arbitrary precision numbers. This precision is both in the integer part and the fractional part. All numbers are represented internally in decimal and all computation is done in decimal. (This version truncates results from divide and multiply operations.) There are two attributes of numbers, the length and the scale. The length is the total number of significant decimal digits in a number and the scale is the total number of decimal digits after the decimal point. For example:

.000001 has a length of 6 and scale of 6.
1935.000 has a length of 7 and a scale of 3.

VARIABLES

Numbers are stored in two types of variables, simple variables and arrays. Both simple variables and array variables are named. Names begin with a letter followed by any number of letters, digits and underscores. All letters must be lower case. (Full alpha-numeric names are an extension. In POSIX **bc** all names are a single lower case letter.) The type of variable is clear by the context because all array variable names will be followed by brackets ([]).

There are four special variables, **scale, ibase, obase,** and **last**. **scale** defines how some operations use digits after the decimal point. The default value of **scale** is 0. **ibase** and **obase** define the conversion base for input and output numbers. The default for both input and output is base 10. **last** (an extension) is a variable that has the value of the last printed number. These will be discussed in further detail where appropriate. All of these variables may have values assigned to them as well as used in expressions.

COMMENTS

Comments in **bc** start with the characters **/*** and end with the characters ***/**. Comments may start anywhere and appear as a single space in the input. (This causes comments to delimit other input items. For example, a comment can not be found in the middle of a variable name.) Comments include any newlines (end of line) between the start and the end of the comment.

EXPRESSIONS

The numbers are manipulated by expressions and statements. Since the language was designed to be interactive, statements and expressions are executed as soon as possible. There is no "main" program. Instead, code is executed as it is encountered. (Functions, discussed in detail later, are defined when

encountered.)

A simple expression is just a constant. **bc** converts constants into internal decimal numbers using the current input base, specified by the variable **ibase**. (There is an exception in functions.) The legal values of **ibase** are 2 through 16 (F). Assigning a value outside this range to **ibase** will result in a value of 2 or 16. Input numbers may contain the characters 0-9 and A-F. (Note: They must be capitals. Lower case letters are variable names.) Single digit numbers always have the value of the digit regardless of the value of **ibase**. (i.e. A = 10.) For multi-digit numbers, **bc** changes all input digits greater or equal to ibase to the value of **ibase**-1. This makes the number **FFF** always be the largest 3 digit number of the input base.

Full expressions are similar to many other high level languages. Since there is only one kind of number, there are no rules for mixing types. Instead, there are rules on the scale of expressions. Every expression has a scale. This is derived from the scale of original numbers, the operation performed and in many cases, the value of the variable **scale**. Legal values of the variable **scale** are 0 to the maximum number representable by a C integer.

In the following descriptions of legal expressions, "expr" refers to a complete expression and "var" refers to a simple or an array variable. A simple variable is just a

name

and an array variable is specified as

name[*expr*]

Unless specifically mentioned the scale of the result is the maximum scale of the expressions involved.

- expr The result is the negation of the expression.

++ var The variable is incremented by one and the new value is the result of the expression.

-- var The variable is decremented by one and the new value is the result of the expression.

var ++ The result of the expression is the value of the variable and then the variable is incremented by one.

var -- The result of the expression is the value of the variable and then the variable is decremented by one.

expr + expr
The result of the expression is the sum of the two expressions.

expr - expr
The result of the expression is the difference of the two expressions.

expr * expr
The result of the expression is the product of the two expressions.

expr / expr
The result of the expression is the quotient of the two expressions. The scale of the result is the value of the variable **scale**.

expr % expr
The result of the expression is the "remainder" and it is computed in the following way. To compute a%b, first a/b is computed to **scale** digits. That result is used to compute a-(a/b)*b to the scale of the maximum of **scale**+scale(b) and scale(a). If **scale** is set to zero and both expressions are integers this expression is the integer remainder function.

expr ^ expr
The result of the expression is the value of the first raised to the second. The second expression must be an integer. (If the second expression is not an integer, a warning is generated and the expression is truncated to get an integer value.) The scale of the result is **scale** if the exponent is negative. If the exponent is positive the scale of the result is the minimum of the scale of the first expression times the value of the exponent and the maximum of **scale** and the scale of the first expression. (e.g. scale(a^b) = min(scale(a)*b, max(**scale,** scale(a))).) It should be noted that expr^0 will always return the value of 1.

(expr) This alters the standard precedence to force the evaluation of the expression.

var = expr
The variable is assigned the value of the expression.

var <op>= expr
This is equivalent to "var = var <op> expr" with the exception that the "var" part is evaluated only once. This can make a difference if "var" is an array.

Relational expressions are a special kind of expression that always evaluate to 0 or 1, 0 if the relation is false and 1 if the relation is true. These may appear in any legal expression. (POSIX bc requires that relational expressions are used only in if, while, and for statements and that only one relational test may be done in them.) The relational operators are

expr1 < expr2
The result is 1 if expr1 is strictly less than expr2.

expr1 <= expr2
The result is 1 if expr1 is less than or equal to expr2.

expr1 > expr2
The result is 1 if expr1 is strictly greater than expr2.

expr1 >= expr2
The result is 1 if expr1 is greater than or equal to expr2.

expr1 == expr2
The result is 1 if expr1 is equal to expr2.

expr1 != expr2
The result is 1 if expr1 is not equal to expr2.

Boolean operations are also legal. (POSIX **bc** does NOT have boolean operations). The result of all boolean operations are 0 and 1 (for false and true) as in relational expressions. The boolean operators are:

!expr The result is 1 if expr is 0.

expr && expr
The result is 1 if both expressions are non-zero.

expr || expr
The result is 1 if either expression is non-zero.

The expression precedence is as follows: (lowest to highest)
|| operator, left associative
&& operator, left associative
! operator, nonassociative
Relational operators, left associative
Assignment operator, right associative
+ and - operators, left associative
*, / and % operators, left associative
^ operator, right associative
unary - operator, nonassociative
++ and -- operators, nonassociative

This precedence was chosen so that POSIX compliant **bc** programs will run correctly. This will cause the use of the relational and logical operators to have some unusual behavior when used with assignment expressions. Consider the expression:
a = 3 < 5

Most C programmers would assume this would assign the result of "3 < 5" (the value 1) to the variable "a". What this does in **bc** is assign the value 3 to the variable "a" and then compare 3 to 5. It is best to use parenthesis when using relational and logical operators with the assignment operators.

There are a few more special expressions that are provided in **bc**. These have to do with user defined functions and standard functions. They all appear as "*name***(***parameters***)**". See the section on functions for user defined functions. The standard functions are:

length (expression)
The value of the length function is the number of significant digits in the expression.

read ( ) The read function (an extension) will read a number from the standard input, regardless of where the function occurs. Beware, this can cause problems with the mixing of data and program in the standard input. The best use for this function is in a previously written program that needs input from the user, but never allows program code to be input from the user. The value of the read function is the number read from the standard input using the current value of the variable **ibase** for the conversion base.

scale (expression)
The value of the scale function is the number of digits after the decimal point in the expression.

sqrt (expression)
The value of the sqrt function is the square root of the expression. If the expression is negative, a run time error is generated.

STATEMENTS

Statements (as in most algebraic languages) provide the sequencing of expression evaluation. In **bc** statements are executed "as soon as possible." Execution happens when a newline in encountered and there is one or more complete statements. Due to this immediate execution, newlines are very important in **bc**. In fact, both a semicolon and a newline are used as statement separators. An improperly placed newline will cause a syntax error. Because newlines are statement separators, it is possible to hide a newline by using the backslash character. The sequence "\<nl>", where <nl> is the newline appears to **bc** as whitespace instead of a newline. A statement list is a series of statements separated by semicolons and newlines. The following is a list of **bc** statements and what they do: (Things enclosed in brackets ([]) are optional parts of the statement.)

expression
This statement does one of two things. If the expression starts with "<variable> <assignment> ...", it is considered to be an assignment statement. If the expression is not an assignment statement, the expression is evaluated and printed to the output. After the number is printed, a newline is printed. For example, "a=1" is an assignment statement and "(a=1)" is an expression that has an embedded assignment. All numbers that are printed are printed in the base specified by the variable **obase**. The legal values for **obase** are 2 through BC_BASE_MAX. (See the section LIMITS.) For bases 2 through 16, the usual method of writing numbers is used. For bases greater than 16, **bc** uses a multi-character digit method of printing the numbers where each higher base digit is printed as a base 10 number. The multi-character digits are separated by spaces. Each digit contains the number of characters required to represent the base ten value of "obase-1". Since numbers are of arbitrary precision, some numbers may not be printable on a single output line. These long numbers will be split across lines using the "\" as the last character on a line. The maximum number of characters printed per line is 70. Due to the interactive nature of **bc** printing a number cause the side effect of assigning the printed value the the special variable **last**. This allows the user to recover the last value printed without having to retype the expression that printed the number. Assigning to **last** is legal and will overwrite the last printed value with the assigned value. The newly assigned value will remain until the next number is printed or another value is assigned to **last**.

string The string is printed to the output. Strings start with a double quote character and contain all characters until the next double quote character. All characters are take literally, including any newline. No newline character is printed after the string.

print list
The print statement (an extension) provides another method of output. The "list" is a list of strings and expressions separated by commas. Each string or expression is printed in the order

of the list. No terminating newline is printed. Expressions are evaluated and their value is printed and assigned the the variable **last**. Strings in the print statement are printed to the output and may contain special characters. Special characters start with the backslash character (\). The special characters recognized by **bc** are "b" (bell), "f" (form feed), "n" (newline), "r" (carriage return), "t" (tab), and "\" (backslash). Any other character following the backslash will be ignored. This still does not allow the double quote character to be part of any string.

{ statement_list }
: This is the compound statement. It allows multiple statements to be grouped together for execution.

if (expression) **then** statement1 [**else** statement2]
: The if statement evaluates the expression and executes statement1 or statement2 depending on the value of the expression. If the expression is non-zero, statement1 is executed. If statement2 is present and the value of the expression is 0, then statement2 is executed. (The else clause is an extension.)

while (expression) statement
: The while statement will execute the statement while the expression is non-zero. It evaluates the expression before each execution of the statement. Termination of the loop is caused by a zero expression value or the execution of a break statement.

for ([expression1] ; [expression2] ; [expression3]) statement
: The for statement controls repeated execution of the statement. Expression1 is evaluated before the loop. Expression2 is evaluated before each execution of the statement. If it is non-zero, the statement is evaluated. If it is zero, the loop is terminated. After each execution of the statement, expression3 is evaluated before the reevaluation of expression2. If expression1 or expression3 are missing, nothing is evaluated at the point they would be evaluated. If expression2 is missing, it is the same as substituting the value 1 for expression2. (The optional expressions are an extension. POSIX **bc** requires all three expressions.) The following is equivalent code for the for statement:

```
expression1;
while (expression2) {
  statement;
  expression3;
}
```

break
: This statement causes a forced exit of the most recent enclosing while statement or for statement.

continue
: The continue statement (an extension) causes the most recent enclosing for statement to start the next iteration.

halt
: The halt statement (an extension) is an executed statement that causes the **bc** processor to quit only when it is executed. For example, "if (0 == 1) halt" will not cause **bc** to terminate because the halt is not executed.

return
: Return the value 0 from a function. (See the section on functions.)

return (expression)
: Return the value of the expression from a function. (See the section on functions.)

PSEUDO STATEMENTS

These statements are not statements in the traditional sense. They are not executed statements. Their function is performed at "compile" time.

limits
: Print the local limits enforced by the local version of **bc**. This is an extension.

quit
: When the quit statement is read, the **bc** processor is terminated, regardless of where the quit statement is found. For example, "if (0 == 1) quit" will cause **bc** to terminate.

warranty
Print a longer warranty notice. This is an extension.

FUNCTIONS

Functions provide a method of defining a computation that can be executed later. Functions in **bc** always compute a value and return it to the caller. Function definitions are "dynamic" in the sense that a function is undefined until a definition is encountered in the input. That definition is then used until another definition function for the same name is encountered. The new definition then replaces the older definition. A function is defined as follows:

define *name* **(** *parameters* **) {** *newline*
auto_list statement_list **}**

A function call is just an expression of the form "*name***(***parameters***)**".

Parameters are numbers or arrays (an extension). In the function definition, zero or more parameters are defined by listing their names separated by commas. Numbers are only call by value parameters. Arrays are only call by variable. Arrays are specified in the parameter definition by the notation "*name***[]**". In the function call, actual parameters are full expressions for number parameters. The same notation is used for passing arrays as for defining array parameters. The named array is passed by variable to the function. Since function definitions are dynamic, parameter numbers and types are checked when a function is called. Any mismatch in number or types of parameters will cause a run-time error. A runtime error will also occur for the call to an undefined function.

The *auto_list* is an optional list of variables that are for "local" use. The syntax of the auto list (if present) is "**auto** *name*, ... ;". (The semicolon is optional.) Each *name* is the name of an auto variable. Arrays may be specified by using the same notation as used in parameters. These variables have their values pushed onto a stack at the start of the function. The variables are then initialized to zero and used throughout the execution of the function. At function exit, these variables are popped so that the original value (at the time of the function call) of these variables are restored. The parameters are really auto variables that are initialized to a value provided in the function call. Auto variables are different than traditional local variables in the fact that if function A calls function B, B may access function A's auto variables by just using the same name, unless function B has called them auto variables. Due to the fact that auto variables and parameters are pushed onto a stack, **bc** supports recursive functions.

The function body is a list of **bc** statements. Again, statements are separated by semicolons or newlines. Return statements cause the termination of a function and the return of a value. There are two versions of the return statement. The first form, "**return**", returns the value 0 to the calling expression. The second form, "**return (** *expression* **)**", computes the value of the expression and returns that value to the calling expression. There is an implied "**return (0)**" at the end of every function. This allows a function to terminate and return 0 without an explicit return statement.

Functions also change the usage of the variable **ibase**. All constants in the function body will be converted using the value of **ibase** at the time of the function call. Changes of **ibase** will be ignored during the execution of the function except for the standard function **read**, which will always use the current value of **ibase** for conversion of numbers.

MATH LIBRARY

If **bc** is invoked with the **-l** option, a math library is preloaded and the default scale is set to 20. The math functions will calculate their results to the scale set at the time of their call. The math library defines the following functions:

s (*x*) The sine of x in radians.

c (*x*) The cosine of x in radians.

a (*x*) The arctangent of x.

l (*x*) The natural logarithm of x.

e (*x*) The exponential function of raising e to the value x.

j (*n*,*x*) The bessel function of integer order n of x.

EXAMPLES

In /bin/sh, the following will assign the value of "pi" to the shell variable **pi**.

```
pi=$(echo "scale=10; 4*a(1)" | bc -l)
```

The following is the definition of the exponential function used in the math library. This function is written in POSIX **bc**.

```
scale = 20

/* Uses the fact that e^x = (e^(x/2))^2
   When x is small enough, we use the series:
     e^x = 1 + x + x^2/2! + x^3/3! + ...
*/

define e(x) {
 auto  a, d, e, f, i, m, v, z

 /* Check the sign of x. */
 if (x<0) {
  m = 1
  x = -x
 }

 /* Precondition x. */
 z = scale;
 scale = 4 + z + .44*x;
 while (x > 1) {
  f += 1;
  x /= 2;
 }

 /* Initialize the variables. */
 v = 1+x
 a = x
 d = 1

 for (i=2; 1; i++) {
  e = (a *= x) / (d *= i)
  if (e == 0) {
   if (f>0) while (f--)  v = v*v;
   scale = z
   if (m) return (1/v);
   return (v/1);
  }
  v += e
 }
}
```

The following is code that uses the extended features of **bc** to implement a simple program for calculating checkbook balances. This program is best kept in a file so that it can be used many times without having to retype it at every use.

```
scale=2
print "\nCheck book program!\n"
print "  Remember, deposits are negative transactions.\n"
print "  Exit by a 0 transaction.\n\n"

print "Initial balance? "; bal = read()
bal /= 1
print "\n"
while (1) {
  "current balance = "; bal
  "transaction? "; trans = read()
  if (trans == 0) break;
  bal -= trans
  bal /= 1
}
quit
```

The following is the definition of the recursive factorial function.

```
define f (x) {
  if (x <= 1) return (1);
  return (f(x-1) * x);
}
```

DIFFERENCES

This version of **bc** was implemented from the POSIX P1003.2/D11 draft and contains several differences and extensions relative to the draft and traditional implementations. It is not implemented in the traditional way using *dc(1)*. This version is a single process which parses and runs a byte code translation of the program. There is an "undocumented" option (-c) that causes the program to output the byte code to the standard output instead of running it. It was mainly used for debugging the parser and preparing the math library.

A major source of differences is extensions, where a feature is extended to add more functionality and additions, where new features are added. The following is the list of differences and extensions.

LANG — This version does not conform to the POSIX standard in the processing of the LANG environment variable and all environment variables starting with LC_.

names — Traditional and POSIX **bc** have single letter names for functions, variables and arrays. They have been extended to be multi-character names that start with a letter and may contain letters, numbers and the underscore character.

Strings — Strings are not allowed to contain NUL characters. POSIX says all characters must be included in strings.

last — POSIX **bc** does not have a **last** variable. Some implementations of **bc** use the period (.) in a similar way.

comparisons — POSIX **bc** allows comparisons only in the if statement, the while statement, and the second expression of the for statement. Also, only one relational operation is allowed in each of those statements.

if statement, else clause — POSIX **bc** does not have an else clause.

for statement — POSIX **bc** requires all expressions to be present in the for statement.

&&, ||, ! — POSIX **bc** does not have the logical operators.

read function
: POSIX **bc** does not have a read function.

print statement
: POSIX **bc** does not have a print statement .

continue statement
: POSIX **bc** does not have a continue statement.

array parameters
: POSIX **bc** does not have array parameters. Other implementations of **bc** may have call by value array parameters.

=+, =-, =*, =/, =%, =^
: POSIX **bc** does not require these "old style" assignment operators to be defined. This version may allow these "old style" assignments. Use the limits statement to see if the installed version supports them. If it does support the "old style" assignment operators, the statement "a =- 1" will decrement **a** by 1 instead of setting **a** to the value -1.

spaces in numbers
: Other implementations of **bc** allow spaces in numbers. For example, "x=1 3" would assign the value 13 to the variable x. The same statement would cause a syntax error in this version of **bc**.

errors and execution
: This implementation varies from other implementations in terms of what code will be executed when syntax and other errors are found in the program. If a syntax error is found in a function definition, error recovery tries to find the beginning of a statement and continue to parse the function. Once a syntax error is found in the function, the function will not be callable and becomes undefined. Syntax errors in the interactive execution code will invalidate the current execution block. The execution block is terminated by an end of line that appears after a complete sequence of statements. For example,

```
a = 1
b = 2
```

has two execution blocks and

```
{ a = 1
  b = 2 }
```

has one execution block. Any runtime error will terminate the execution of the current execution block. A runtime warning will not terminate the current execution block.

Interrupts
: During an interactive session, the SIGINT signal (usually generated by the control-C character from the terminal) will cause execution of the current execution block to be interrupted. It will display a "runtime" error indicating which function was interrupted. After all runtime structures have been cleaned up, a message will be printed to notify the user that **bc** is ready for more input. All previously defined functions remain defined and the value of all non-auto variables are the value at the point of interruption. All auto variables and function parameters are removed during the clean up process. During a non-interactive session, the SIGINT signal will terminate the entire run of **bc**.

LIMITS

The following are the limits currently in place for this **bc** processor. Some of them may have been changed by an installation. Use the limits statement to see the actual values.

BC_BASE_MAX
: The maximum output base is currently set at 999. The maximum input base is 16.

BC_DIM_MAX
: This is currently an arbitrary limit of 65535 as distributed. Your installation may be different.

BC_SCALE_MAX
: The number of digits after the decimal point is limited to INT_MAX digits. Also, the number of digits before the decimal point is limited to INT_MAX digits.

BC_STRING_MAX
: The limit on the number of characters in a string is INT_MAX characters.

exponent
: The value of the exponent in the raise operation (^) is limited to LONG_MAX.

multiply
: The multiply routine may yield incorrect results if a number has more than LONG_MAX / 90 total digits. For 32 bit longs, this number is 23,860,929 digits.

code size
: Each function and the "main" program are limited to 10240 bytes of compiled byte code each. This limit (BC_MAX_SEGS) can be easily changed to have more than 10 segments of 1024 bytes.

variable names
: The current limit on the number of unique names is 32767 for each of simple variables, arrays and functions.

FILES

In most installations, **bc** is completely self-contained. Where executable size is of importance or the C compiler does not deal with very long strings, **bc** will read the standard math library from the file /usr/local/lib/libmath.b. (The actual location may vary. It may be /lib/libmath.b.)

DIAGNOSTICS

If any file on the command line can not be opened, **bc** will report that the file is unavailable and terminate. Also, there are compile and run time diagnostics that should be self-explanatory.

BUGS

Error recovery is not very good yet.

AUTHOR

Philip A. Nelson
phil@cs.wwu.edu

ACKNOWLEDGEMENTS

The author would like to thank Steve Sommars (sesv@iwtsf.att.com) for his extensive help in testing the implementation. Many great suggestions were given. This is a much better product due to his involvement.

NAME

bsfilt, colcrt − a colcrt-like backspace filter

SYNOPSIS

bsfilt [**-**] [**-U**] [file ...]

DESCRIPTION

Bsfilt filters backspace sequences from the input *file*(s) (standard input if none) in an approximation of *colcrt*(1). Both the backspace and the character it returns to are removed, unless they form an underline sequence. Underline sequences are treated according to the settings of the **−** and **−U** options.

OPTIONS

− specifies that no underlining of any kind is to be propagated. Without this option or the **−U** option, *bsfilt* approximates underlining with minus signs ('-') in following lines.

−U specifies that underlining with underscore ('_') and backspace (')character sequences is permitted.

SEE ALSO

cawf(1), colcrt(1) and nroff(1).

DIAGNOSTICS

Diagnostic messages are delivered to the standard error file.

HISTORY

Vic Abell of Purdue University wrote *bsfilt* to have a backspace filter for *cawf*(1) that is independent of licensed source code.

BUGS

The maximum length of a line that can be underlined with minus signs is fixed.

Bsfilt does not examine the characters that are being overprinted via backspace operations. Thus, overprinting that is intended to form a new character from several different ones is ineffective and only the last character of the sequence is propagated − e. g., ''oˆH+'', intended to look like a bullet, is reduced to '+'.

NAME

btoa − binary to ascii conversion

SYNOPSIS

btoa [−**adhor**] [*infile*] [*outfile*]

OPTIONS

−a Decode, rather than encode, the file

−d Extracts repair file from diagnosis file

−h Help menu is displayed giving the options

−o The obsolete algorithm is used for backward compatibility

−r Repair a damaged file

EXAMPLES

btoa <a.out >a.btoa # Convert *a.out* to ASCII

btoa −a <a.btoa >a.out # Reverse the above

DESCRIPTION

Btoa is a filter that converts a binary file to ascii for transmission over a telephone line. If two file names are provided, the first in used for input and the second for output. If only one is provided, it is used as the input file. The program is a functionally similar alternative to *uue/uud*, but the encoding is completely different. Since both of these are widely used, both have been provided with MINIX. The file is expanded about 25 percent in the process.

SEE ALSO

uue(1), **uud**(1).

NAME

cal − print a calendar

SYNOPSIS

cal [*month*] *year*

EXAMPLES

cal 3 1992 # Print March 1992

DESCRIPTION

Cal prints a calendar for a month or year. The year can be between 1 and 9999. Note that the year 91 is not a synonym for 1991, but is itself a valid year about 19 centuries ago. The calendar produced is the one used by England and her colonies. Try Sept. 1752, Feb 1900, and Feb 2000. If you do not understand what is going on, look up *Calendar, Gregorian* in a good encyclopedia.

NAME

calendar − reminder service

SYNOPSIS

calendar [−] [**−r**]

OPTIONS

− Work for every user and send mail to him

−r Restrict multiple execution on the same day

EXAMPLES

calendar # Check *calendar* file in current directory

calendar # Normary used under the control of cron(8)

calendar −r # Normary used in /etc/rc file

DESCRIPTION

Basically *calendar* program consults the file *calendar* in the current directory and display lines which contain today's or tomorrow's date. Month-day formats such as '12/25', 'Dec. 25', 'december 25', '*/25', '12/*', '*/*' are recognized. The asterisk means 'all' days or 'all' months. On weekends 'tomorrow' extends through next Monday without any consideration about holidays. To prevent ambiguity, the formats '25 Dec.' and '25/12' are not recognized.

When an argument **−** is present, *calendar* works for all users with a file *calendar* in their login directories and sends them mail. Normally this is done daily under the control of *cron*.

The **−r** option does its the same job as **−** option, but touches the *calendar* to prevents further access on the same day. Normally this is done in the */etc/rc* file on a machine which may be booted several times in one day.

SEE ALSO

cron(8).

NAME

cat − concatenate files and write them to stdout

SYNOPSIS

cat [**−u**] [*file*] ...

OPTIONS

−u Unbuffered output

EXAMPLES

cat file # Display file on the terminal

cat file1 file2 | lpr # Concatenate 2 files and print result

DESCRIPTION

Cat concatenates its input files and copies the result to *stdout*. If no input file is named, or − is encountered as a file name, standard input is used. Output is buffered in 512 byte blocks unless the **−u** flag is given. If you just want to copy a file, *cp* should be used since it is faster.

SEE ALSO

cp(1).

NAME

cawf, nroff − C version of the nroff-like, Amazingly Workable (text) Formatter

SYNOPSIS

cawf [−**c***config*] [−**d***device*] [−**e**] [−**f***font*] [−**h**] [−**m***acros*] [file ...]

DESCRIPTION

Cawf formats the text from the input *file*(s) (standard input if none) in an approximation of *nroff*. It comes closest to duplicating *nroff*'s **man** or **ms** macro package styles. It has some limited support for *nroff*'s **me** macros.

OPTIONS

Options must precede file names.

−**c***config*

defines an alternate path to the device configuration file. Normally the device configuration file is found in *device.cf* in the *cawf* library (see the **FILES** section).

The device configuration file contains device character strings for selecting fonts and the bold or italic type faces. See the **DEVICES** section for more information.

−**d***device*

specifies the name of the output device. There are three built−in devices − ANSI, NONE and NORMAL − and other devices may be defined in the device configuration file. See the **DEVICES** section for more information.

The NORMAL device is the default.

−**e** directs *cawf* to issue an eject (FF or ^L) after the last page.

−**f***font* specifies the one font for the device, declared with the −**d***device* option, that is to be used for the entire document. *Font* must match a font associated with the device's stanza in the device configuration file. See the **DEVICES** section for more information.

No *font* may be specified for the built−in devices ANSI, NONE or NORMAL.

−**h** requests a help display.

−**m***acro* specifies the macro file to be used. The standard *cawf* distribution supplies macro files to support ''−man'', ''−me'' or ''−ms''. *Cawf* finds a macro file by constructing its name from 'm', *acro* and **.mac** − e. g., −**m***an* is converted to **man.mac**. The default directory for macro files is defined when *cawf* is compiled; it's *C:\SYS\LIB\CAWF* in the MS−DOS environment; */usr/lib/cawf* in the UNIX environment.

file ... are the names of files containing *nroff* source text.

NROFF COMPATIBILITY

Cawf accepts the following raw *nroff* requests:

.\"	.ad	.bp	.br	.ce	.de	.di	.ds
.el	.fi	.fl	.ft	.i0	.ie	.if	.in
.it	.lg	.li	.ll	.ls	.na	.ne	.nf
.nr	.ns	.pl	.po	.ps	.rm	.rn	.rr
.rs	.so	.sp	.ta	.ti	.tm	.tr	

and the following in-text codes:

\$	\%	*	\"	\c	\f	\h	\k
\n	\s	\w					

plus the full list of *nroff*/*troff* special characters in the original V7 *troff* manual.

Many restrictions are present; the behavior in general is a subset of *nroff*'s. Of particular note are the following:

- The fully supported nroff request control character is the period. There is limited support for the non−break, acute accent control character.

- Point sizes do not exist; **.ps** is ignored.
- Special vertical spacing − the **.vs** request included − is ignored.
- Conditionals cover only the numeric comparisons >, =, <, >= and <= on **\n(.$**; string comparisons between a macro parameter and a literal; **n** (always true); and **t** (always false). Only single line input is accepted from conditionals; multi−line input − e.g., \(*anything*\) − is not supported.
- The handling of strings is generally primitive.
- Horizontal motion via **\h** must be supplied with a number register interpolation and must be positive - e. g., **\w\n(NN**, where the value in NN is >= 0.
- The **\k** function is reliable only after TAB characters, so it is useful only for measuring table positions.
- The **.di** request only turns output on and off − any macro name is ignored.
- Expressions - e. g., **.sp** - are reasonably general, but the **|**, **&**, and **:** operators do not exist, there must be white space between the end of the *nroff* function and the beginning of the expression, and **\w** requires that quote (') be used as the delimiters. **\w** counts the characters inside the quotes and scales the result in ens, so that, for example, \w'\(bu' equals 4n, and \w'\(bu'/1n equals 4.
- The only acceptable count for the **.it** request is one, and it is effective only with **man**, **me** or **ms** macros.
- The default scaling factor is 'v' for the **.ne**, **.sp**, and **.pl** raw *nroff* requests; it is 'u' for **.nr**; and 'n' for **.in**, **.ll**, **.ls**, **.po**, **.ta** and **.ti**. (A different scaling factor may be specified with a trailing character.)
- Some obsolete or meaningless requests − **.i0**, **.lg** and **.li** − are silently ignored.

White space at the beginning of lines, and embedded white space within lines is dealt with properly. Sentence terminators at ends of lines are understood to imply extra space afterward in filled lines. Tabs are implemented crudely and not exactly, although usually they work as expected. Hyphenation is done only at explicit hyphens, em-dashes, and *nroff* discretionary hyphens. By default bold and italic characters are emulated with backspacing and overprinting, but the **−d** and **−f** options, combined with the contents of the device configuration file, may be used to generate special codes for bold and italic characters. (See the **DEVICES** section for more information.)

MAN MACROS

The **man** macro set replicates the full V7 manual macros, plus a few semi-random oddballs. The full list is:

.AT	.B	.BI	.BR	.BY	.DE	.DS	.DT	.HP	.I
.IB	.IP	.IR	.IX	.LP	.NB	.P	.PD	.PP	.RB
.RE	.RI	.RS	.SH	.SM	.SS	.TH	.TP	.UC	

.BY and **.NB** each take a single string argument (respectively, an indication of authorship and a note about the status of the manual page) and arrange to place it in the page footer. **.AT** and **.IX** do nothing.

ME MACROS

The **me** macro subset has been derived from the *cawf* **ms** macros by Chet Creider <creider@csd.uwo.ca>. It includes:

.(l	.(q	.)l	.)q	.b	.bu	.i	.ip	.lp	.np
.pp	.r	.sh	.sm	.u	.uh				

The .(l C and .(l L options are supported. In addition, the .AB, .AE, .AI, .AU, .DA, .ND, .TL and .UX macros have been retained from the **ms** set, and the .XP macro has been borrowed from the Berkeley additions to the **ms** macro set.

MS MACROS

The **ms** macro set is a substantial subset of the V7 manuscript macros. The macros are:

.AB	.AE	.AI	.AU	.B	.CD	.DA	.DE	.DS	.I
.ID	.IP	.LD	.LG	.LP	.ND	.NH	.NL	.PP	.QE
.QP	.QS	.R	.RE	.RP	.RS	.SH	.SM	.TL	.TP

.UL .UX

Size changes are recognized but ignored, as are **.RP** and **.ND**. **.UL** just prints its argument in italics. **.DS**/**.DE** does not do a keep, nor do any of the other macros that normally imply keeps.

The **DY** string variable is available. The **PD**, **PI**, and **LL** number registers exist and can be changed.

HEADERS AND FOOTERS

Cawf allows the placement of text into the five line header and footer sections from the **LH**, **CH**, **RF**, **LF**, **CF**, and **RF** string variables, via the control of the **.^b** request:

.^b fh 1 enables header string placement on the first page
.^b fh 0 disables header string placement on the first page
.^b HF 1 enables header/footer string placement
.^b HF 0 disables header/footer string placement

There are appropriate **.^b** requests in the distribution **man**, **me** and **ms** macro files. (The **me** and **ms** macro files use another **.^b** request, **.^b NH**, to enable numbered header processing.)

OUTPUT

The default output format supported by *cawf*, in its distributed form, is that appropriate to a dumb terminal, using overprinting for italics (via underlining) and bold. The *nroff* special characters are printed as some vague approximation (it's sometimes extremely vague) to their correct appearance.

One part of *cawf*'s knowledge of the output device, related to the formation of characters, is established by a device file, which is read before the user's input. The search for it begins in *cawf*'s library directory, under the name *term*.**dev** (where *term* is the value of the TERM environment variable). Failing to find that, *cawf* searches for **dumb.dev**. (See the **FILES** section for a description of the path to *cawf*'s library directory.) The device file uses special internal requests to set up resolution, special characters and more normal *nroff* functions to set up page length, etc.

Cawf has limited support for fonts special forms of bold and italic characters. It is provided through the **−c** *config*, **−d***device* and **−f***font* options. See the **DEVICES** section for more information.

Note the distinction between the device and the output device configuration files. The device file typically defines characters and constant output parameters. The output device configuration file defines font and type face codes. It is usually not necessary to define a separate device file for each device represented in the output device configuration file − the *dumb.dev* device file will suffice for almost all representations.

DEVICES

Cawf supports primitive output device configuration for font and type face control. One font may be selected for the entire document by directing *cawf* to issue a font selection control character string at the beginning of the document, and control character strings may be selected for switching between the bold, italic and Roman type faces.

The **−c** *config*, **−d***device* and **−f***font* options direct the font and type face selections.

The **−d***device* option specifies the name of the device. *Cawf* has three built−in devices − ANSI, NONE and NORMAL. When the ANSI device is selected, *cawf* issues the ANSI shadow mode control codes, ''ESC [7 m'', to represent the bold face; the ANSI underscore control codes, ''ESC [4 m'', to represent the italic face; and the ANSI control codes, ''ESC [0 m'', to represent the ROMAN face. No **−f***font* specification is permitted with the ANSI device.

When the NONE device is selected, *cawf* uses no special output codes to represent the type faces. No **−f***font* specification is permitted with the ANSI device.

The NORMAL output device is the default. When it's selected, *cawf* overprints each bold character two times, using three issuances of each bold character, separated by backspace characters; it issues an underscore and backspace before each italic character. No **−f***font* specification is permitted with the ANSI device. The *bsfilt*(1) filter may be used to further process the backspace codes output for a NORMAL device.

All other devices named in the **−d***device* option must be represented by a stanza in the device configuration file. The device configuration file is usually contained in *device.cf* in *cawf's* library directory (see the **FILES** section for more information). An alternate device configuration file path may be specified with the **−c***config* option.

The **DEVICE CONFIGURATION FILE** section describes the organization of the device configuration file. It is easy to add devices to the *device.cf* supplied in the *cawf* distribution.

The **−f***font* option may be used with the **−d***device* option, when the appropriate stanza in the device configuration file contains an entry for the named *font*. The **DEVICE CONFIGURATION FILE** section describes how fonts are defined in device configuration file stanzas.

DEVICE CONFIGURATION FILE

The device configuration file defines the special character codes necessary to direct output devices to select fonts and to produce bold, italic and Roman type faces.

The configuration file is usually found in *device.cf* in *cawf*'s library directory (see the **FILES** section for more information). It is organized into two main parts − comments and device stanzas. Comments are any lines that begin with the pound sign ('#') character. They are informational only and *cawf* ignores them. *Cawf* also ignores empty lines, so they may be used as vertical white space.

Stanzas name devices and define their font and type face control strings. A stanza begins with the name of the device, starting at the beginning of a line and occupying the entire line. The body of the stanza, defining fonts and type faces, is formed of lines beginning with white space (a TAB or space characters) that directly follow the device name.

Individual lines of the stanza body contain a key character, followed by a equal sign, followed by the font name (if a font key) and the output device control codes. *Cawf* issues the font control codes once, at the beginning of output, so only one font may be selected. The type face control codes are issued at each change of type face.

The key characters are: b for bold f for font definition i for italic r for Roman The 'b', 'i' and 'r' key codes are followed by an equal sign ('=') and their control code definition. The 'f' key code is followed by an equal sign ('='), the font name, another equal sign and the font control code definition.

Control code definitions may contain any printable ASCII characters. Non−printable characters may be encoded in octal notation with the '\nnn' form or in hexadecimal with the '\xnn' form. The special code, '\E' (or '\e') represents the ESC control character (\033 or \x1b).

Here's a sample showing the definition for the HP LaserJet III. The stanza name is ''lj3''. All its non−printable characters are ESCs; the first is coded in octal form; the second with '\E'; the rest, in hexadecimal form. TAB is used as the leading white space character for the stanza body lines. # HP LaserJet III

```
lj3
	b=\033(s7B
	i=\E(s1S
	r=\x1b(s0B\x1b(s0S
	f=c10=&l0O(8U(s0p12h10v0s0b3T
	f=c12ibm=&l0O(10U(s0p10.00h12.0v0s0b3T
	f=lg12=&l0O(8U(s12h12v0s0b6T
```

The distribution *device.cf* file defines the following devices and fonts.

epson dot matrix printer in Epson FX-86e/FX-800 mode
Bold: Double-strike
Fonts: none

ibmppds IBM Personal Printer Data Stream (PPDS) protocol
Bold: Double-strike
Italic: Underline
Fonts: none

kxp1124 Panasonic KX−P1124 dot matrix printer in PGM mode
Bold: Emphasized
Fonts:

c10	10 Characters Per Inch (CPI) Courier
c12	12 CPI Courier
bps10	10 CPI Bold PS
bps12	12 CPI Bold PS
p10	10 CPI Prestige
p12	12 CPI Prestige
s10	10 CPI Script
s12	12 CPI Script
ss10	10 CPI Sans Serif
ss12	12 CPI Sans Serif

kxp1180 Panasonic KX−P1180 dot matrix printer in PGM mode
Bold: Emphasized
Fonts:

c10	10 Characters Per Inch (CPI) Courier
c12	12 CPI Courier
bps10	10 CPI Bold PS
bps12	12 CPI Bold PS
p10	10 CPI Prestige
p12	12 CPI Prestige
ss10	10 CPI Sans Serif
ss12	12 CPI Sans Serif

lj3 HP LaserJet III
Fonts:

c10	10 point, 12 Characters Per Inch (CPI) Courier
c12ibm	12 point, 10 CPI Courier, IBM−PC Symbol Set
lg12	12 point, 12 CPI Letter Gothic

vgamono VGA monochrome monitor for MS−DOS
(ANSI.SYS driver required for MS−DOS)
Italic: Reverse-video
Fonts: none

FILES

Cawf resource files are located in the *cawf* library directory − *C:\SYS\LIB\CAWF*, the MS−DOS environment default; or */usr/lib/cawf*, the UNIX environment default. These defaults can be overridden by the CAWFLIB environment variable, or changed in the cawflib.h header file.

common	common device-independent initialization
device.cf	output device configurations
*.dev	device-specific initialization
m*.mac	macro package files

DIAGNOSTICS

Unlike *nroff*, *cawf* complains whenever it sees unknown requests. All diagnostics appear on the standard error file.

HISTORY

Vic Abell of Purdue University <abe@cc.purdue.edu> derived *cawf* from *awf*, ‘‘the Amazingly

Workable (text) Formatter,'' written by Henry Spencer of the University of Toronto. The Toronto work was a supplement to the C News project. The Purdue effort was aimed at producing a C language version that would run on small systems, particularly MS−DOS ones. The adaptation of the **me** macros was done by Chet Creider <creider@csd.uwo.ca>. Chet also contributed ideas for device, font and type face support.

The MS−DOS version of *cawf* has been compiled with version 2.5 of Microsoft's Quick-C compiler. It runs under the Mortis Kern Systems Toolkit KornShell, *ksh*(1), and COMMAND.COM.

BUGS

Nroff and troff mavens will have many complaints. Some may even represent bugs and not deliberate omissions.

Watch out for scaling factors - especially on requests like **w**.

The overprinting required to create bold and italicized characters is tiresome on a slow printer. The *bsfilt*(1) post−filter from this distribution may be used to alleviate that nuisance by managing the back-spacing codes from *cawf*'s NORMAL device output.

The printing of bold and italic characters is sometimes better handled by special printer codes. Use *cawf*'s **−c** *config*, **−d***device* and **−f***font* options to produce special font and device output control codes.

Cawf has a small amount of built-in code for the **man**, **me** and **ms** macro packages, but none for any others.

The stacking for the **.so** request is limited.

SEE ALSO

bsfilt(1), colcrt(1), man(7), me(7), ms(7) and nroff(1).

NAME

cc − C compiler

SYNOPSIS

cc [**−STOUfcimos**] [**−w[aos]**] [**−v[n]**] [**−D***name*]* [**−I***dir*]* [**−L***dir*]* *file*+ [**−l***name*]*

OPTIONS

−D The flag **−D***x[=y]* defines a macro *x* with (optional) value *y*

−I **−I***dir* searches *dir* for include files

−L **−L***dir* searches *dir* for **−l***name* libraries

−O Optimize the code

−S Produce an assembly code file, then stop

−T The flag **−T***dir* tells *cc* and *as* to use *dir* for temporary files

−U Undefine a macro

−E Preprocess to standard output

−c Compile only. Do not link

−f Link with floating point emulation library

−i Use separate I & D space (64K + 64K) (only)

−l The flag **−l***name* causes the library lib*name*.a to be linked

−m Remove unnecessary prototypes after preprocessing (only)

−o Put output on file named by next arg

−s Strip the symbol-table from executable file

−v Verbose; print pass names

−vn Verbose; print pass names but do not run them

−w Suppress warning messages

−ws Suppress strict messages

−wa Suppress all warning and strict messages

−wo Suppress messages about old-style

−.o Do not link the default run-time start-off

EXAMPLES

cc −c file.c # Compile *file.c*

cc −DFOO file.c # Treat the symbol *FOO* as defined

cc −wo −o out file.c # Compile old-style code; output to *out*

DESCRIPTION

This is the C compiler. It has eight passes, as follows:

Program	**Input**	**Output**	**Operation performed**
lib/ncpp	prog.c	prog.i	C preprocessor: #include, #define, #ifdef
lib/irrel	prog.i	prog.i	Removal of unnecessary prototypes
lib/ncem	prog.i	prog.k	Parsing and semantic analysis
lib/nopt	prog.k	prog.m	Optimization of the intermediate code
lib/ncg	prog.m	prog.s	Code generation
bin/as	prog.s	prog.o	Assembly
lib/ld	prog.o	prog.out	Linking
lib/cv	prog.out	a.out	Conversion to MINIX a.out format

In the 68000 versions of MINIX , the preprocessor is not called since the front-end contains the preprocessor. This increases compilation speed.

The main program, *cc* , forks appropriately to call the passes, transmitting flags and arguments. The **−v** flag causes the passes to be listed as they are called, and the **−vn** flag causes the passes to be listed but not called.

The libraries should be made with *aal* (which is the same as *ar* on the 68000 versions), and consist of .o files. The internal order of files inside the library is unimportant, but the order in which the libraries are specified is.

When **−T** is used, the intermediate files end up in the directory specified. Otherwise, **/tmp** is used. When available memory is very limited (e.g., a 512K machine), it may be necessary to run *chmem* to reduce the sizes of the compiler passes that do not fit, typically *ncem* .

On the other hand, if the compiler (or, in fact, almost any program) begins acting strange, it is almost always due to its running out of space, either stack space or scratch file space. The relevant pass can be given more stack space using *chmem* . More space for scratch files can be obtained by removing other files on the device.

If the compiler runs out of memory, it may be necessary to use the **−m** flag. This causes *irrel* to be run, which removes unnecessary prototypes and thus frees up extra table space within the compiler. Beware, however, that running this pass may cause strictly conforming programs to become non-conforming and vice versa, so you should only run this pass as a last resort.

The compiler is derived from the ACK system (Tanenbaum et al., *Communications of the ACM*, Sept. 1983), not from the AT&T portable C compiler. It has been shoehorned onto the PC with some loss of performance.

SEE ALSO

make(1).

NAME

cdiff − context diff

SYNOPSIS

cdiff [−**c***n*] *oldfile newfile*

OPTIONS

−c Provide *n* lines of context

EXAMPLES

cdiff old new >f # Write context diff on *f*

cdiff −c1 old new >f # Use only 1 line of context

DESCRIPTION

Cdiff produces a context diff by first running *diff* and then adding context. Some update programs, like *patch*, can use context diffs to update files, even in the presence of other, independent changes.

SEE ALSO

cmp(1), **diff**(1), **patch**(1).

NAME

cdplay − play audio compact disks

SYNOPSIS

cdplay [−**a**]|[−**r**]

DESCRIPTION

Cdplay is an audio−cdplayer it can be used interactively or non-interactively. With no flags **cdplay** is used interactively. When used interactively **cdplay** can be either in non-playing or playing mode. In non-playing mode you can select tracks to be played. In playing mode **cdplay** shows the track-time, remaining track-time, total-time and remaining total-time. **Cdplay** assumes the drive to be at */dev/cd0*.

During non-playing mode the following commands are available:

cursor-up/down
: browse through the track-list

spacebar
: add track to the play-list

backspace
: remove a track from the play-list

s : start playing the tracks in the play-list. When the play-list is empty start playing from the current position in the track-list until the end of the cd

p : play the track on the current position in the track-list

r : play all tracks in a random order (shuffle play)

e : exit cdplay

During playing mode the following commands are available:

s : stop playing, return to non-playing mode

spacebar
: pause/resume current track

cursor-up/down
: skip to next/previous track

OPTIONS

When one of these options is used **cdplay** does everything in the background.

−a : Play all tracks, **cdplay** tells the drive to play all tracks and then it immediately returns to the user

−r : Play all tracks in random order (shuffle play). **Cdplay** forks of a new process which starts up a track, during that track it sleeps. When the track is finished it wakes up to start the next track and goes to sleep again. The parent process immediately returns to the user.

BUGS

Cdplay does not sense if a cd has changed, so don't switch cd's. If you want to play another cd you have to insert the new cd and start **cdplay** again.

AUTHOR

Michel R. Prevenier (mrpreve@cs.vu.nl)

NAME

cgrep − grep and display context

SYNOPSIS

cgrep [**−a** *n*] [**−b** *n*] [**−f**] [**−l** *n*] [**−n**] [**−w** *n*] *pattern* [*file*] ...

OPTIONS

−a How many lines to display after the matching line

−b How many lines to display before the matching line

−f Suppress file name in the output

−l Lines are truncated to this length before comparison

−n Suppress line numbers in the output

−w Sets window size (same as **−a** n **−b** n)

EXAMPLES

cgrep −w 3 hello file1 # Print 3 lines of context each way

DESCRIPTION

Cgrep is a program like *grep*, except that it also can print a few lines above and/or below the matching lines. It also prints the line numbers of the output.

SEE ALSO

grep(1), **fgrep**(1).

NAME

chgrp − change group

SYNOPSIS

chgrp [−R] [*owner:*]*group file ...*

OPTIONS

−R Change directory hierarchies

EXAMPLES

chgrp system file1 file2
Make *system* the group of the files

chrgp −R other dir1 # Make *other* the group of all files below dir1

DESCRIPTION

The group field (and optionally owner field) of the named files is changed to *group* and *owner* . Alternatively, a decimal gid (uid) may be specified instead of a group name. If the **−R** flag is used, the changes will be applied recursively to all files in named directories. Only the superuser may execute this command to set arbitrary groups. Normal users can only change the group if they own the file, and the group is their own group (Minix), or one of their supplementary groups (Minix-vmd).

SEE ALSO

chown(1), **chmod**(1), **ls**(1), **chown**(2).

NAME

chmem − change memory allocation

SYNOPSIS

chmem [+] [−] [=] *amount file*

EXAMPLES

chmem =50000 a.out # Give *a.out* 50K of stack space

chmem −4000 a.out # Reduce the stack space by 4000 bytes

chmem +1000 file1 # Increase each stack by 1000 bytes

DESCRIPTION

When a program is loaded into memory, it is allocated enough memory for the text and data+bss segments, plus an area for the stack. Data segment growth using *malloc* , *brk* , or *sbrk* eats up stack space from the low end. The amount of stack space to allocate is derived from a field in the executable program's file header. If the combined stack and data segment growth exceeds the stack space allocated, the program will be terminated.

It is therefore important to set the amount of stack space carefully. If too little is provided, the program may crash. If too much is provided, memory will be wasted, and fewer programs will be able to fit in memory and run simultaneously. MINIX does not swap, so that when memory is full, subsequent attempts to fork will fail. The compiler sets the stack space to the largest possible value (for the Intel CPUs, 64K − text − data). For many programs, this value is far too large. Nonrecursive programs that do not call *brk* , *sbrk* , or *malloc* , and do not have any local arrays usually do not need more than 8K of stack space.

The *chmem* command changes the value of the header field that determines the stack allocation, and thus indirectly the total memory required to run the program. The = option sets the stack size to a specific value; the + and − options increment and decrement the current value by the indicated amount. The old and new stack sizes are printed.

SEE ALSO

install(1), **brk**(2).

NAME

chmod − change access mode for files

SYNOPSIS

chmod [−R] *mode file ...*

OPTIONS

−R Change hierarchies recursively

EXAMPLES

chmod 755 file	# Owner: rwx Group: r−x Others: r−x
chmod +x file1 file2	# Make *file1* and *file2* executable
chmod a−w file	# Make *file* read only
chmod u+s file	# Turn on SETUID for *file*
chmod −R o+w dir	# Allow writing for all files in dir

DESCRIPTION

The given mode is applied to each file in the file list. If the **−R** flag is present, the files in a directory will be changed as well. The mode can be either absolute or symbolic. Absolute modes are given as an octal number that represents the new file mode. The mode bits are defined as follows:

4000	Set effective user id on execution to file's owner id
2000	Set effective group id on execution to file's group id
0400	file is readable by the owner of the file
0200	writeable by owner
0100	executable by owner
0070	same as above, for other users in the same group
0007	same as above, for all other users

Symbolic modes modify the current file mode in a specified way. The form is:

[who] op permissions { op permissions ...} {, [who] op ... }

The possibilities for *who* are *u*, *g*, *o*, and *a*, standing for user, group, other and all, respectively. If *who* is omitted, *a* is assumed, but the current umask is used. The op can be +, -, or =; + turns on the given permissions, − turns them off; = sets the permissions exclusively for the given *who*. For example *g=x* sets the group permissions to *--x*.

The possible permissions are *r*, *w*, *x*; which stand for read, write, and execute; *s* turns on the set effective user/group id bits. *s* only makes sense with *u* and *g*; o+s is harmless.

SEE ALSO

ls(1), **chmod**(2).

NAME

cksum − display file checksum and size

SYNOPSIS

cksum [*file* ...]

EXAMPLES

cksum # Display CRC and size of *stdin*

cksum *.c # Display CRC and size of *.c* files

DESCRIPTION

Cksum calculates and writes to standard output the 32-bits CRC of the input *files* , or of stdin if no *files* were specified. The size in bytes of each *file* will be displayed after a space. The name of each *file* will be displayed after another space.

SEE ALSO

crc(1), **sum**(1).

NAME

clr − clear the screen

SYNOPSIS

clr

EXAMPLES

clr # Clear the screen

DESCRIPTION

All text is removed from the screen, resulting in an empty screen with the cursor positioned in the upper left-hand corner.

NAME

cmp − compare two files

SYNOPSIS

cmp [−**ls**] *file1 file2*

OPTIONS

−l Loud mode. Print bytes that differ (in octal)

−s Silent mode. Print nothing, just return exit status

EXAMPLES

cmp file1 file2 # Tell whether the files are the same

cmp −l file1 file2 # Print all corresponding bytes that differ

DESCRIPTION

Two files are compared. If they are identical, exit status 0 is returned. If they differ, exit status 1 is returned. If the files cannot be opened, exit status 2 is returned. If one of the file arguments is −, then *stdin* is compared to the other file.

SEE ALSO

comm(1), **diff**(1).

NAME

comm − print lines common to two sorted files

SYNOPSIS

comm [**−123**] *file1 file2*

OPTIONS

−1 Suppress column 1 (lines present only in *file1*)

−2 Suppress column 2 (lines present only in *file2*)

−3 Suppress column 3 (lines present in both files)

EXAMPLES

comm file1 file2 # Print all three columns

comm −12 file1 file2 # Print only lines common to both files

DESCRIPTION

Two sorted files are read and compared. A three column listing is produced. Files only in *file1* are in column 1; files only in *file2* are in column 2; files common to both files are in column 3. The file name − means *stdin*.

SEE ALSO

cmp(1), **diff**(1), **sort**(1).

NAME

compress, uncompress, zcat − compress a file using modified Lempel-Ziv coding

SYNOPSIS

compress [**−cdfv**] [*file*] ...

OPTIONS

−c Put output on *stdout* instead of on *file.Z*

−d Decompress instead of compress

−f Force output even if there is no saving

−v Verbose mode

EXAMPLES

compress <infile >outfile
Compress 1 file

compress x y z # Compress 3 files to *x.Z*, *y.Z*, and *z.Z*

compress −d file.Z # Decompress *file.Z* to *file*

DESCRIPTION

The listed files (or *stdin*, if none are given) are compressed using the Ziv-Lempel algorithm. If the output is smaller than the input, the output is put on *file.Z* or *stdout* if no files are listed. If *compress* is linked to *uncompress*, the latter is the same as giving the **−d** flag. Similarly, a link to *zcat* decompresses to *stdout*. The MINIX version of *compress* uses 13-bit compression. This means that when compressing files on other systems for transmission to MINIX, be sure that only 13-bit compression is used. On many systems, the default is 16-bit (too big).

SEE ALSO

tar(1).

NAME

cp, cpdir − file copy

SYNOPSIS

cp [−pifsmrRvx] *file1 file2*
cp [−pifsrRvx] *file ... directory*
cpdir [−ifvx] *file1 file2*

OPTIONS

−p Preserve full mode, uid, gid and times

−i Ask before removing existing file

−f Forced remove existing file

−s Make similar, copy some attributes

−m Merge trees, disable the into-a-directory trick

−r Copy directory trees with link structure, etc. intact

−R Copy directory trees and treat special files as ordinary

−v Display what cp is doing

−x Do not cross device boundaries

EXAMPLES

cp oldfile newfile # Copy *oldfile* to *newfile*

cp -R dir1 dir2 # Copy a directory tree

DESCRIPTION

Cp copies one file to another, or copies one or more files to a directory. Special files are normally opened and read, unless **−r** is used. **−r** also copies the link structure, something **−R** doesn't care about. The **−s** option differs from **−p** that it only copies the times if the target file already exists. A normal copy only copies the mode of the file, with the file creation mask applied. Set-uid bits are cleared if the owner cannot be set. (The **−s** flag does not patronize you by clearing bits. Alas **−s** and **−r** are nonstandard.)

Cpdir is a convenient synonym for **cp −psmr** to make a precise copy of a directory tree.

SEE ALSO

cat(1), **mkdir**(1), **rmdir**(1), **ln**(1), **rm**(1).

NAME

crc − print the checksum of the file data

SYNOPSIS

crc *file* ...

EXAMPLES

crc *.c # Print checksums of all the C programs

DESCRIPTION

The checksum of each argument is computed and printed, along with the file length and its name, one file per line. This program is useful for seeing if a file transmitted to another machine has arrived correctly. It is conceptually similar to *sum*, except that it uses a stronger checksum algorithm and also prints the length.

SEE ALSO

cksum(1), **sum**(1).

NAME

ctags - Generates "tags" and (optionally) "refs" files

SYNOPSIS

ctags [**-stvra**] *filesnames*...

DESCRIPTION

ctags generates the "tags" and "refs" files from a group of C source files. The "tags" file is used by Elvis' ":tag" command, control-] command, and -t option. The "refs" file is sometimes used by the *ref(1)* program.

Each C source file is scanned for #define statements and global function definitions. The name of the macro or function becomes the name of a tag. For each tag, a line is added to the "tags" file which contains:

- the name of the tag
- a tab character
- the name of the file containing the tag
- a tab character
- a way to find the particular line within the file.

The filenames list will typically be the names of all C source files in the current directory, like this:

```
$ ctags -stv *.[ch]
```

OPTIONS

-t Include typedefs. A tag will be generated for each user-defined type. Also tags will be generated for struct and enum names. Types are considered to be global if they are defined in a header file, and static if they are defined in a C source file.

-v Include variable declarations. A tag will be generated for each variable, except for those that are declared inside the body of a function.

-s Include static tags. *Ctags* will normally put global tags in the "tags" file, and silently ignore the static tags. This flag causes both global and static tags to be added. The name of a static tag is generated by prefixing the name of the declared item with the name of the file where it is defined, with a colon in between. For example, "static foo(){}" in "bar.c" results in a tag named "bar.c:foo".

-r This causes *ctags* to generate both "tags" and "refs". Without **-r**, it would only generate "tags".

-a Append to "tags", and maybe "refs". Normally, *ctags* overwrites these files each time it is invoked. This flag is useful when you have to many files in the current directory for you to list them on a single command-line; it allows you to split the arguments among several invocations.

FILES

tags A cross-reference that lists each tag name, the name of the source file that contains it, and a way to locate a particular line in the source file.

refs The "refs" file contains the definitions for each tag in the "tags" file, and very little else. This file can be useful, for example, when licensing restrictions prevent you from making the source code to the standard C library readable by everybody, but you still everybody to know what arguments the library functions need.

BUGS

ctags is sensitive to indenting and line breaks. Consequently, it might not discover all of the tags in a file that is formatted in an unusual way.

SEE ALSO

elvis(1), refs(1)

AUTHOR

Steve Kirkendall
kirkenda@cs.pdx.edu

NAME

cut − select out columns of a file

SYNOPSIS

cut [−b | **−c**] *list* [*file...*]
cut −f *list* [**−d** *delim*] [**−s**]

OPTIONS

−b Cut specified bytes

−c Select out specific characters

−d Change the column delimiter to *delim*

−f Select out specific fields that are separated by the

−i Runs of delimiters count as one

−s Suppres lines with no delimiter characters, when used

EXAMPLES

cut −f 2 file # Extract field 2

cut −c 1−2,5 file # Extract character columns 1, 2, and 5

cut −c 1−5,7− file # Extract all columns except 6

DESCRIPTION

[*file...*]" delimiter character (see *delim*)" with the −f option. Lines with no delimiters are passwd through untouched"

Cut extracts one or more fields or columns from a file and writes them on standard output. If the **−f** flag is used, the fields are separated by a delimiter character, normally a tab, but can be changed using the **−d** flag. If the **−c** flag is used, specific columns can be specified. The list can be comma or BLANK separated. The **−f** and **−c** flags are mutually exclusive. Note: The POSIX1003.2 standard requires the option −b to cut out specific bytes in a file. It is intended for systems with multi byte characters (e.g. kanji), since MINIX uses only one byte characters, this option is equivalent to −c. For the same reason, the option −n has no effect and is not listed in this manual page.

SEE ALSO

sed(1), **awk**(9).

NAME

date − print or set the date and time

SYNOPSIS

date [−qsu] [[*MMDDYY*]*hhmm*[*ss*]] [+*format*]

OPTIONS

−q Read the date from *stdin*

−s Set the time (implicit for **−q** or a date string)

−u Print the date as GMT

−t Use this number of seconds instead of current time

EXAMPLES

date # Print the date and time

date 0221921610 # Set date to Feb 21, 1992 at 4:10 p.m.

DESCRIPTION

With the **−q** flag or a numeric argument, *date* sets the GMT time and date. *MMDDYY* refers to the month, day, and year; *hhmmss* refers to the hour, minute and second. Each of the six fields must be exactly two digits, no more and no less. *date* always display the date and time, with the default format for the system. The **−u** flag request GMT time instead of local time. A format may be specified with a + followed by a printf-like string with the following options:

%% % character
%A Name of the day
%B Name of the month
%D mm/dd/yy
%H Decimal hour on 2 digits
%I Decimal hour modulo 12 on 2 digits
%M Decimal minute on 2 digits
%S Decimal seconds on 2 digits
%T HH:MM:SS
%U Decimal week number, Sunday being first day of week
%W Decimal week number, Monday being first day of week
%X Same as %T
%Y Decimal year on 4 digits
%Z Time Zone (if any)
%a Abbreviated name of the day
%b Abbreviated name of the month
%c Appropriate date & time (default format)
%d Decimal day of the month on 2 digits
%e Same as %d, but a space replaces leading 0
%h Same as %b
%j Decimal dey of the year on 3 digits
%m Decimal month on 2 digits
%n Newline character
%p AM or PM
%r 12-hour clock time with AM/PM
%s Number of seconds since the epoch
%t Tab character
%w Decimal day of the week (0=Sunday)
%x Same as %D
%y Decimal year on 2 digits

SEE ALSO

time(2), **ctime**(3), **readclock**(8).

NAME

dd − disk dumper

SYNOPSIS

dd [*option* = *value*] ...

EXAMPLES

dd if=/dev/fd0 of=/dev/fd1
Copy disk 0 to disk 1

dd if=x of=y bs=1w skip=4
Copy *x* to *y*, skipping 4 words

dd if=x of=y count=3 # Copy three 512−byte blocks

DESCRIPTION

This command is intended for copying partial files. The block size, skip count, and number of blocks to copy can be specified. The options are:

if = file	− Input file (default is *stdin*)
of = file	− Output file (default is standard output)
ibs = n	− Input block size (default 512 bytes)
obs = n	− Output block size (default is 512 bytes)
bs = n	− Block size; sets *ibs* and *obs* (default is 512 bytes)
skip = n	− Skip *n* input blocks before reading
seek = n	− Skip *n* output blocks before writing
count = n	− Copy only *n* input blocks
conv = lcase	− Convert upper case letters to lower case
conv = ucase	− Convert lower case letters to upper case
conv = swab	− Swap every pair of bytes
conv = noerror	− Ignore errors and just keep going
conv = silent	− Suppress statistics (Minix specific flag)

Where sizes are expected, they are in bytes. However, the letters **w**, **b**, or **k** may be appended to the number to indicate words (2 bytes), blocks (512 bytes), or K (1024 bytes), respectively. When *dd* is finished, it reports the number of full and partial blocks read and written.

SEE ALSO

vol(1).

NAME

df − report on free disk space and i-nodes

SYNOPSIS

df *special ...*

EXAMPLES

df /dev/ram	# Report on free RAM disk space
df /dev/fd0 /dev/fd1	# Report on diskette space
df	# Report on all mounted devices

DESCRIPTION

The amount of disk space and number of i-nodes, both free and used is reported. If no argument is given, *df* reports on the root device and all mounted file systems.

SEE ALSO

du(1), **fstab**(5).

NAME

dhrystone − integer benchmark

SYNOPSIS

dhrystone

EXAMPLES

dhrystone # Run the dhrystone benchmark

DESCRIPTION

Many years ago, a floating-point benchmark called *whetstone* was popular for benchmarking FORTRAN programs. Nowadays, an integer benchmark called *dhrystone* is widely used for benchmarking UNIX systems. This is it. Be warned, however, that *dhrystone* is entirely CPU bound, and goes blindingly fast on machines with high-speed caches. Although this is a good measure for programs that spend most of their time in some inner loop, it is a poor benchmark for I/O bound applications.

NAME

diff − print differences between two files

SYNOPSIS

diff [**−c** | **−e** | **−C** *n*] [**−br**]*file1 file2*

OPTIONS

−C *n* Produce output that contains *n* lines of context

−b Ignore white space when comparing

−c Produce output that contains three lines of context

−e Produce an *ed*-script to convert *file1* into *file2*

−r Apply *diff* recursively to files and directories of

EXAMPLES

diff file1 file2 # Print differences between 2 files

diff -C 0 file1 file2 # Same as above

diff -C 3 file1 file2 # Output three lines of context with every

diff -c file1 file2 # Same

diff /etc /dev # Compares recursively the directories */etc* and */dev*

diff passwd /etc # Compares *./passwd* to */etc/passwd*

DESCRIPTION

the same name, when *file1* and *file2* are both directories" difference encountered"

Diff compares two files and generates a list of lines telling how the two files differ. Lines may not be longer than 128 characters. If the two arguments on the command line are both directories, *diff* recursively steps through all subdirectories comparing files of the same name. If a file name is found only in one directory, a diagnostic message is written to *stdout*. A file that is of either block special, character special or FIFO special type, cannot be compared to any other file. On the other hand, if there is one directory and one file given on the command line, *diff* tries to compare the file with the same name as *file* in the directory *directory*.

SEE ALSO

cdiff(1), **cmp**(1), **comm**(1), **patch**(1).

NAME

diskcheck − check a disk for bad sectors

SYNOPSIS

diskcheck *device start count*

EXAMPLES

diskcheck /dev/at0 0 1200

Check 1.2 MB diskette

diskcheck /dev/at0 100 50

Check blocks 100 to 149

DESCRIPTION

Diskcheck checks a disk for bad sectors by reading in each sector, writing a known bit pattern onto it, reading it back in and comparing with what was written. This check is then done a second time. Bad sectors are reported. After each sector is tested, the original sector is restored. Only use this program on unmounted partitions. Killing it part way through may result in lost data.

SEE ALSO

format(1).

NAME

dosdir − list an MS-DOS directory [IBM]

SYNOPSIS

dosdir [**−lr**] *drive*

OPTIONS

−l Long listing

−r Recursively descend and print subdirectories

EXAMPLES

dosdir −l A # List root directory on drive A

dosdir −r C x/y # Recursively list directory *x/y*

dosdir −r fd1 # List device */dev/fd1*

DESCRIPTION

Dosdir reads standard IBM PC diskettes or hard disk partitions in MS-DOS format and lists their contents on standard output. Directory names should contain slashes to separate components, even though MS-DOS uses backslashes. The names *dosdir* , *dosread* , and *doswrite* are all links to the same program. The program sees which function to perform by seeing how it was called. A drive code of *A* causes the program to use */dev/dosA*, for example, a link to */dev/fd0*. Similarly, to have hard disk partition 1 be DOS drive C, */dev/dosC* could be a link to */dev/hd1*, and so on for other drive codes. A normal device name may also be used instead of a drive code.

NAME

dosread − read a file from an MS-DOS diskette [IBM]

SYNOPSIS

dosread [**−a**] *drive file*

OPTIONS

−a ASCII file

EXAMPLES

dosread C g/adv >adv # Read file *g/adv* from hard disk

dosread −a A prog.c >x
Read ASCII file *prog.c* from drive A

DESCRIPTION

Dosread reads one MS-DOS file and writes it on standard output. The file name must use slash, not backslash as a separator. ASCII files have the final CTRL-Z stripped, and carriage return plus line feed are mapped to line feed only, the usual MINIX convention. See **dosdir** on the use of single letter drive codes.

NAME

doswrite − write a file onto an MS-DOS diskette [IBM]

SYNOPSIS

doswrite [**−a**] *drive file*

OPTIONS

−a ASCII file

EXAMPLES

doswrite A x/y <z # Write file *z* to disk as *x/y*

doswrite −a B f # Copy *stdin* to MS-DOS file *f*

DESCRIPTION

Doswrite writes its *stdin* to an MS-DOS file. The diskette or partition must be formatted and have an MS-DOS file system already in place, including all the directories leading up to the file. See **dosdir** on the use of single letter drive codes.

NAME

du − print disk usage

SYNOPSIS

du [**−as**] [**−l** *n*] *dir* ...

OPTIONS

−a Give usage for all files

−l List up to *n* levels of subdirectories

−s Summary only

EXAMPLES

du dir # List disk space used by files in dir

du −s dir1 dir2 # Give summaries only

DESCRIPTION

Du examines one or more directories and prints the amount of space occupied by the files in those directories and their subdirectories.

SEE ALSO

df(1).

NAME

echo − print the arguments

SYNOPSIS

echo [**−n**] *argument* ...

OPTIONS

−n No line feed is output when done

EXAMPLES

echo Start Phase 1 # 'Start Phase 1' is printed

echo −n Hello # 'Hello' is printed without a line feed

DESCRIPTION

Echo writes its arguments to standard output. They are separated by blanks and terminated with a line feed unless **−n** is present. This command is used mostly in shell scripts.

SEE ALSO

sh(1).

NAME

ed − editor

SYNOPSIS

ed *file*

OPTIONS

− Suppress line/byte count messages (for in scripts)

EXAMPLES

ed prog.c # Edit *prog.c*

echo '1,$p' | ed - file # Odd way to write 'cat file'

DESCRIPTION

Ed is functionally equivalent to the standard V7 editor, ed. It supports the following commands:

(.)	a: append
(.,.)	c: change
(.,.)	d: delete
	e: edit new file"
	f: print name of edited file"
(1,$)	g: global command
(.)	i: insert
(.,.+1)	j: join lines together
(.)	k: mark
(.)	l: print with special characters in octal
(.,.)	m: move
(.,.)	p: print
	q: quit editor"
(.)	r: read in new file
(.,.)	s: substitute
(1,$)	v: like g, except select lines that do not match
(1,$)	w: write out edited file

Many of the commands can take one or two addresses, as indicated above. The defaults are shown in parentheses. Thus *a* appends to the current line, and *g* works on the whole file as default. The dot refers to the current line. Below is a sample editing session with comments given following the # symbol.

```
ed prog.c                   # Edit prog.c
3,20p                       # Print lines 3 through 20
/whole/                     # Find next occurence of whole
s/whole/while/              # Replace whole by while
g/Buf/s//BUF/g              # Replace Buf by BUF everywhere
w                           # Write the file back
q                           # Exit the editor
```

Ed is provided for its sentimental value. If you want a line-oriented editor, try *ex*. If you want a good editor, use *elle*, *elvis*, or *mined*.

SEE ALSO

elvis(1), **elle**(9), **mined**(9).

NAME

eject − eject removable media

SYNOPSIS

eject *device*

DESCRIPTION

Eject tells a device to eject removable media, usually a floppy or CD-ROM. **Eject** invokes the **DIOCEJECT** ioctl on the device. The media will then be ejected, or allowed to be removed. The call will fail if the device is still in use.

Tapes can't be unloaded with this command, use **mt offline** instead.

SEE ALSO

mt(1), **hd**(4), **sd**(4).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

elvis, ex, vi − The editor

SYNOPSIS

elvis [*flags*] [**+***cmd*] [*files*...]

DESCRIPTION

Elvis is a text editor which emulates **vi**/**ex**.

On systems which pass the program name as an argument, such as Unix and Minix, you may also install **elvis** under the names "ex", "vi", "view", and "input". These extra names would normally be links to elvis; see the "ln" shell command.

When **elvis** is invoked as "vi", it behaves exactly as though it was invoked as "elvis". However, if you invoke **elvis** as "view", then the readonly option is set as though you had given it the "-R" flag. If you invoke **elvis** as "ex", then **elvis** will start up in the colon command mode instead of the visual command mode, as though you had given it the "-e" flag. If you invoke **elvis** as "input" or "edit", then **elvis** will start up in input mode, as though the "-i" flag was given.

OPTIONS

-r To the real vi, this flag means that a previous edit should be recovered. **Elvis**, though, has a separate program, called *elvrec*(1), for recovering files. When you invoke **elvis** with -r, **elvis** will tell you to run **elvrec**.

-R This sets the "readonly" option, so you won't accidentally overwrite a file.

-t *tag* This causes **elvis** to start editing at the given tag.

-m [*file*]
Elvis will search through *file* for something that looks like an error message from a compiler. It will then begin editing the source file that caused the error, with the cursor sitting on the line where the error was detected. If you don't explicitly name a *file*, then "errlist" is assumed.

-e **Elvis** will start up in colon command mode.

-v **Elvis** will start up in visual command mode.

-i **Elvis** will start up in input mode.

-w *winsize*
Sets the "window" option's value to *winsize*.

+*command* or **-c** *command*
If you use the **+***command* parameter, then after the first file is loaded *command* is executed as an EX command. A typical example would be "elvis +237 foo", which would cause **elvis** to start editing foo and then move directly to line 237. The "-c *command*" variant was added for UNIX SysV compatibility.

FILES

/tmp/elv*
During editing, **elvis** stores text in a temporary file. For UNIX, this file will usually be stored in the /tmp directory, and the first three characters will be "elv". For other systems, the temporary files may be stored someplace else; see the version-specific section of the documentation.

tags This is the database used by the **:tags** command and the **-t** option. It is usually created by the **ctags**(1) program.

.exrc or elvis.rc
On UNIX-like systems, a file called ".exrc" in your home directory is executed as a series of **ex** commands. A file by the same name may be executed in the current directory, too. On non-UNIX systems, ".exrc" is usually an invalid file name; there, the initialization file is called "elvis.rc" instead.

SEE ALSO

ctags(1), **ref**(1), **virec**(1), **elvis**(9).

Elvis - A Clone of Vi/Ex, the complete **elvis** documentation.

BUGS

There is no LISP support. Certain other features are missing, too.

Auto-indent mode is not quite compatible with the real vi. Among other things, 0^D and ^^D don't do what you might expect.

Long lines are displayed differently. The real vi wraps long lines onto multiple rows of the screen, but **elvis** scrolls sideways.

AUTHOR

Steve Kirkendall
kirkenda@cs.pdx.edu

Many other people have worked to port **elvis** to various operating systems. To see who deserves credit, run the **:version** command from within **elvis**, or look in the system-specific section of the complete documentation.

NAME

elvrec - Recover the modified version of a file after a crash

SYNOPSIS

elvrec [*preservedfile* [*newfile*]]

DESCRIPTION

If you're editing a file when *elvis* dies, the system crashes, or power fails, the most recent version of your text will be preserved. The preserved text is stored in a special directory; it does NOT overwrite your text file automatically.

The *elvrec* program locates the preserved version of a given file, and writes it over the top of your text file -- or to a new file, if you prefer. The recovered file will have nearly all of your changes.

To see a list of all recoverable files, run *elvrec* with no arguments.

FILES

/usr/preserve/p*

The text that was preserved when *elvis* died.

/usr/preserve/Index

A text file which lists the names of all preserved files, and the names of the /usr/preserve/p* files which contain their preserved text.

BUGS

elvrec is very picky about filenames. You must tell it to recover the file using exactly the same pathname as when you were editing it. The simplest way to do this is to go into the same directory that you were editing, and invoke *elvrec* with the same filename as *elvis*. If that doesn't work, then try running *elvrec* with no arguments, to see exactly which pathname it is using for the desired file.

Due to the permissions on the /usr/preserve directory, on UNIX systems *elvrec* must be run as superuser. This is accomplished by making the *elvrec* executable be owned by "root" and setting its "set user id" bit.

If you're editing a nameless buffer when *elvis* dies, then *elvrec* will pretend that the file was named "foo".

AUTHOR

Steve Kirkendall
kirkenda@cs.pdx.edu

NAME

expand − convert tabs to spaces

SYNOPSIS

expand [−*t1,t2, ...*] [*file*]

OPTIONS

−*t* Tab stop positions

EXAMPLES

expand −16,32,48,64 # Expand *stdin* with tabs every 16 columns

DESCRIPTION

Expand replaces tabs in the named files with the equivalent numbers of spaces. If no files are listed, *stdin* is given. If only one tab is given, the rest are multiples of it. The default is a tab every 8 spaces.

SEE ALSO

unexpand(1).

NAME

expr − evaluate experession

SYNOPSIS

expr *arg* ...

EXAMPLES

x=`expr + 1` # Add 1 to shell variable x

DESCRIPTION

Expr computes the value of its argument and writes the result on standard output. The valid operators, in order of increasing precedence, are listed below. Operators grouped by {...} have the same precedence. The operators are: |, &, {<, <=, ==, !=, >=, >}, {+, −}, *, /, %, and :. Parentheses are permitted.

SEE ALSO

sh(1), **test**(1).

NAME

factor − factor an integer less than 2**31

SYNOPSIS

factor *number*

EXAMPLES

factor 450180 # Print the prime factors of 450180

DESCRIPTION

Factor prints the prime factors of its argument in increasing order. Each factor is printed as many times as it appears in the number.

NAME

fgrep − fixed grep

SYNOPSIS

fgrep [**−cfhlnsv**] [*string_file*] [*string*] [*file*] ...

OPTIONS

−c Count matching lines and only print count, not the lines

−f Take strings from file named in following argument

−h Omit file headers from printout

−l List file names once only

−n Each line is preceded by its line number

−s Status only, no output

−v Print only lines not matching

EXAMPLES

fgrep % prog.c # Print lines containing % sign

fgrep −f pattern prog.c
Take strings from *pattern*

DESCRIPTION

Fgrep is essentially the same as grep, except that it only searches for lines containing literal strings (no wildcard characters). The pattern may consist of several lines with one string to search on each line.

SEE ALSO

cgrep(1), **grep**(1).

NAME

file − make a guess as to a file's type based on contents

SYNOPSIS

file *name* ...

EXAMPLES

file a.out ar.h # Guess at types

DESCRIPTION

File reads the first block of a file and tries to make an intelligent guess about what kind of file it is. It understands about archives, C source programs, executable binaries, shell scripts, and English text.

NAME

find − find files meeting a given condition

SYNOPSIS

find *directory expression*

EXAMPLES

find / −name a.out −print
Print all *a.out* paths

find /usr/ast ! −newer f −ok rm {} ;
Ask before removing

find /usr −size +20 −exec mv {} /big ;
move files > 20 blks

find / −name a.out −o −name '*.o') −exec rm {};
2 conds

DESCRIPTION

Find descends the file tree starting at the given directory checking each file in that directory and its subdirectories against a predicate. If the predicate is true, an action is taken. The predicates may be connected by **−a** (Boolean and), **−o** (Boolean or) and ! (Boolean negation). Each predicate is true under the conditions specified below. The integer *n* may also be +*n* to mean any value greater than *n*, −*n* to mean any value less than *n*, or just *n* for exactly *n*.

−name s	true if current filename is *s* (include shell wild cards)
−size n	true if file size is *n* blocks
−inum n	true if the current file's i-node number is *n*
−mtime n	true if modification time relative to today (in days) is *n*
−links n	true if the number of links to the file is *n*
−newer f	true if the file is newer than *f*
−perm n	true if the file's permission bits = *n* (*n* is in octal)
−user u	true if the uid = *u* (a numerical value, not a login name)
−group g	true if the gid = *g* (a numerical value, not a group name)
−type x	where *x* is **bcdfug** (block, char, dir, regular file, setuid, setgid)
−xdev	do not cross devices to search mounted file systems

Following the expression can be one of the following, telling what to do when a file is found:

−print	print the file name on standard output
−exec	execute a MINIX command, {} stands for the file name
−ok	prompts before executing the command

SEE ALSO

test(1), **xargs**(1).

NAME

finger − user information lookup program

SYNOPSIS

finger [options] name ...

DESCRIPTION

By default **finger** lists the login name, full name, terminal name and write status (as a '*' before the terminal name if write permission is denied), idle time, login time, and office location and phone number (if they are known) for each current UNIX user. (Idle time is minutes if it is a single integer, hours and minutes if a ':' is present, or days and hours if a 'd' is present.)

A longer format also exists and is used by **finger** whenever a list of people's names is given. (Account names as well as first and last names of users are accepted.) This format is multi-line, and includes all the information described above as well as the user's home directory and login shell, any plan which the person has placed in the file **.plan** in their home directory, and the project on which they are working from the file **.project** also in the home directory.

Finger may be used to lookup users on a remote machine. The format is to specify the user as ''user@host.'' If the user name is left off, the standard format listing is provided on the remote machine.

Finger options include:

−m Match arguments only on user name.

−l Force long output format.

−p Suppress printing of the **.plan** files

−s Force short output format.

FILES

/etc/utmp	who file
/etc/passwd	for users names, offices, ...
/usr/adm/lastlog	last login times
~/.plan	plans
~/.project	projects

SEE ALSO

chfn(1), **w**(1), **who**(1).

AUTHOR

Earl T. Cohen

BUGS

Only the first line of the **.project** file is printed.

There is no way to pass arguments to the remote machine as **finger** uses an internet standard port.

NAME

flex, lex - fast lexical analyzer generator

SYNOPSIS

flex [-bcdfinpstvFILT8 -C[efmF] -Sskeleton] *[filename ...]*

DESCRIPTION

flex is a tool for generating *scanners:* programs which recognized lexical patterns in text. *flex* reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called *rules. flex* generates as output a C source file, **lex.yy.c,** which defines a routine **yylex().** This file is compiled and linked with the **-lfl** library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

For full documentation, see **flexdoc(1).** This manual entry is intended for use as a quick reference.

OPTIONS

flex has the following options:

-b Generate backtracking information to *lex.backtrack.* This is a list of scanner states which require backtracking and the input characters on which they do so. By adding rules one can remove backtracking states. If all backtracking states are eliminated and **-f** or **-F** is used, the generated scanner will run faster.

-c is a do-nothing, deprecated option included for POSIX compliance.

NOTE: in previous releases of *flex* **-c** specified table-compression options. This functionality is now given by the **-C** flag. To ease the the impact of this change, when *flex* encounters **-c,** it currently issues a warning message and assumes that **-C** was desired instead. In the future this "promotion" of **-c** to **-C** will go away in the name of full POSIX compliance (unless the POSIX meaning is removed first).

-d makes the generated scanner run in *debug* mode. Whenever a pattern is recognized and the global **yy_flex_debug** is non-zero (which is the default), the scanner will write to *stderr* a line of the form:

```
--accepting rule at line 53 ("the matched text")
```

The line number refers to the location of the rule in the file defining the scanner (i.e., the file that was fed to flex). Messages are also generated when the scanner backtracks, accepts the default rule, reaches the end of its input buffer (or encounters a NUL; the two look the same as far as the scanner's concerned), or reaches an end-of-file.

-f specifies (take your pick) *full table* or *fast scanner.* No table compression is done. The result is large but fast. This option is equivalent to **-Cf** (see below).

-i instructs *flex* to generate a *case-insensitive* scanner. The case of letters given in the *flex* input patterns will be ignored, and tokens in the input will be matched regardless of case. The matched text given in *yytext* will have the preserved case (i.e., it will not be folded).

-n is another do-nothing, deprecated option included only for POSIX compliance.

-p generates a performance report to stderr. The report consists of comments regarding features of the *flex* input file which will cause a loss of performance in the resulting scanner.

-s causes the *default rule* (that unmatched scanner input is echoed to *stdout)* to be suppressed. If the scanner encounters input that does not match any of its rules, it aborts with an error.

-t instructs *flex* to write the scanner it generates to standard output instead of **lex.yy.c.**

-v specifies that *flex* should write to *stderr* a summary of statistics regarding the scanner it generates.

-F specifies that the *fast* scanner table representation should be used. This representation is about as fast as the full table representation (**−f**), and for some sets of patterns will be considerably smaller (and for others, larger). See **flexdoc(1)** for details.

This option is equivalent to **-CF** (see below).

-I instructs *flex* to generate an *interactive* scanner, that is, a scanner which stops immediately rather than looking ahead if it knows that the currently scanned text cannot be part of a longer rule's match. Again, see **flexdoc(1)** for details.

Note, **-I** cannot be used in conjunction with *full* or *fast tables,* i.e., the **-f, -F, -Cf,** or **-CF** flags.

-L instructs *flex* not to generate **#line** directives in **lex.yy.c.** The default is to generate such directives so error messages in the actions will be correctly located with respect to the original *flex* input file, and not to the fairly meaningless line numbers of **lex.yy.c.**

-T makes *flex* run in *trace* mode. It will generate a lot of messages to *stdout* concerning the form of the input and the resultant non-deterministic and deterministic finite automata. This option is mostly for use in maintaining *flex.*

-8 instructs *flex* to generate an 8-bit scanner. On some sites, this is the default. On others, the default is 7-bit characters. To see which is the case, check the verbose **(-v)** output for "equivalence classes created". If the denominator of the number shown is 128, then by default *flex* is generating 7-bit characters. If it is 256, then the default is 8-bit characters.

-C[efmF]

controls the degree of table compression.

-Ce directs *flex* to construct *equivalence classes,* i.e., sets of characters which have identical lexical properties. Equivalence classes usually give dramatic reductions in the final table/object file sizes (typically a factor of 2-5) and are pretty cheap performance-wise (one array look-up per character scanned).

-Cf specifies that the *full* scanner tables should be generated - *flex* should not compress the tables by taking advantages of similar transition functions for different states.

-CF specifies that the alternate fast scanner representation (described in **flexdoc(1))** should be used.

-Cm directs *flex* to construct *meta-equivalence classes*, which are sets of equivalence classes (or characters, if equivalence classes are not being used) that are commonly used together. Meta-equivalence classes are often a big win when using compressed tables, but they have a moderate performance impact (one or two "if" tests and one array look-up per character scanned).

A lone **-C** specifies that the scanner tables should be compressed but neither equivalence classes nor meta-equivalence classes should be used.

The options **-Cf** or **-CF** and **-Cm** do not make sense together - there is no opportunity for meta-equivalence classes if the table is not being compressed. Otherwise the options may be freely mixed.

The default setting is **-Cem,** which specifies that *flex* should generate equivalence classes and meta-equivalence classes. This setting provides the highest degree of table compression. You can trade off faster-executing scanners at the cost of larger tables with the following generally being true:

slowest & smallest
-Cem
-Cm
-Ce
-C
-C{f,F}e
-C{f,F}

fastest & largest

-C options are not cumulative; whenever the flag is encountered, the previous -C settings are forgotten.

-Sskeleton_file

overrides the default skeleton file from which *flex* constructs its scanners. You'll never need this option unless you are doing *flex* maintenance or development.

SUMMARY OF FLEX REGULAR EXPRESSIONS

The patterns in the input are written using an extended set of regular expressions. These are:

x match the character 'x'
. any character except newline
[xyz] a "character class"; in this case, the pattern matches either an 'x', a 'y', or a 'z'
[abj-oZ] a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'
[^A-Z] a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.
[^A-Z\n] any character EXCEPT an uppercase letter or a newline
r* zero or more r's, where r is any regular expression
r+ one or more r's
r? zero or one r's (that is, "an optional r")
r{2,5} anywhere from two to five r's
r{2,} two or more r's
r{4} exactly 4 r's
{name} the expansion of the "name" definition (see above)
"[xyz]\"foo"
the literal string: [xyz]"foo
\X if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of \x. Otherwise, a literal 'X' (used to escape operators such as '*')
\123 the character with octal value 123
\x2a the character with hexadecimal value 2a
(r) match an r; parentheses are used to override precedence (see below)

rs the regular expression r followed by the regular expression s; called "concatenation"

r|s either an r or an s

r/s an r but only if it is followed by an s. The s is not part of the matched text. This type of pattern is called as "trailing context".
^r an r, but only at the beginning of a line
r$ an r, but only at the end of a line. Equivalent

to "r/\n".

<s>r an r, but only in start condition s (see below for discussion of start conditions)

<s1,s2,s3>r same, but in any of start conditions s1, s2, or s3

<<EOF>> an end-of-file

<s1,s2><<EOF>> an end-of-file when in start condition s1 or s2

The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom. Those grouped together have equal precedence.

Some notes on patterns:

- Negated character classes *match newlines* unless "\n" (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., "[^A-Z\n]").
- A rule can have at most one instance of trailing context (the '/' operator or the '$' operator). The start condition, '^', and "<<EOF>>" patterns can only occur at the beginning of a pattern, and, as well as with '/' and '$', cannot be grouped inside parentheses. The following are all illegal:

```
foo/bar$
foo|(bar$)
foo|^bar
<sc1>foo<sc2>bar
```

SUMMARY OF SPECIAL ACTIONS

In addition to arbitrary C code, the following can appear in actions:

- **ECHO** copies yytext to the scanner's output.
- **BEGIN** followed by the name of a start condition places the scanner in the corresponding start condition.
- **REJECT** directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input). **yytext** and **yyleng** are set up appropriately. Note that **REJECT** is a particularly expensive feature in terms scanner performance; if it is used in *any* of the scanner's actions it will slow down *all* of the scanner's matching. Furthermore, **REJECT** cannot be used with the *-f* or *-F* options.

  Note also that unlike the other special actions, **REJECT** is a *branch;* code immediately following it in the action will *not* be executed.
- **yymore()** tells the scanner that the next time it matches a rule, the corresponding token should be *appended* onto the current value of **yytext** rather than replacing it.
- **yyless(n)** returns all but the first *n* characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. **yytext** and **yyleng** are adjusted appropriately (e.g., **yyleng** will now be equal to *n*).
- **unput(c)** puts the character *c* back onto the input stream. It will be the next character scanned.
- **input()** reads the next character from the input stream (this routine is called **yyinput()** if the scanner is compiled using **C++).**

- **yyterminate()** can be used in lieu of a return statement in an action. It terminates the scanner and returns a 0 to the scanner's caller, indicating "all done".

  By default, **yyterminate()** is also called when an end-of-file is encountered. It is a macro and may be redefined.

- **YY_NEW_FILE** is an action available only in <<EOF>> rules. It means "Okay, I've set up a new input file, continue scanning".

- **yy_create_buffer(file, size)** takes a *FILE* pointer and an integer *size*. It returns a YY_BUFFER_STATE handle to a new input buffer large enough to accomodate *size* characters and associated with the given file. When in doubt, use **YY_BUF_SIZE** for the size.

- **yy_switch_to_buffer(new_buffer)** switches the scanner's processing to scan for tokens from the given buffer, which must be a YY_BUFFER_STATE.

- **yy_delete_buffer(buffer)** deletes the given buffer.

VALUES AVAILABLE TO THE USER

- **char *yytext** holds the text of the current token. It may not be modified.

- **int yyleng** holds the length of the current token. It may not be modified.

- **FILE *yyin** is the file which by default *flex* reads from. It may be redefined but doing so only makes sense before scanning begins. Changing it in the middle of scanning will have unexpected results since *flex* buffers its input. Once scanning terminates because an end-of-file has been seen, **void yyrestart(FILE *new_file)** may be called to point *yyin* at the new input file.

- **FILE *yyout** is the file to which **ECHO** actions are done. It can be reassigned by the user.

- **YY_CURRENT_BUFFER** returns a **YY_BUFFER_STATE** handle to the current buffer.

MACROS THE USER CAN REDEFINE

- **YY_DECL** controls how the scanning routine is declared. By default, it is "int yylex()", or, if prototypes are being used, "int yylex(void)". This definition may be changed by redefining the "YY_DECL" macro. Note that if you give arguments to the scanning routine using a K&R-style/non-prototyped function declaration, you must terminate the definition with a semi-colon (;).

- The nature of how the scanner gets its input can be controlled by redefining the **YY_INPUT** macro. YY_INPUT's calling sequence is "YY_INPUT(buf,result,max_size)". Its action is to place up to *max_size* characters in the character array *buf* and return in the integer variable *result* either the number of characters read or the constant YY_NULL (0 on Unix systems) to indicate EOF. The default YY_INPUT reads from the global file-pointer "yyin". A sample redefinition of YY_INPUT (in the definitions section of the input file):

```
%{
#undef YY_INPUT
#define YY_INPUT(buf,result,max_size) \
  { \
  int c = getchar(); \
  result = (c == EOF) ? YY_NULL : (buf[0] = c, 1); \
  }
%}
```

- When the scanner receives an end-of-file indication from YY_INPUT, it then checks the **yywrap()** function. If **yywrap()** returns false (zero), then it is assumed that the function has gone ahead and set up *yyin* to point to another input file, and scanning continues. If it returns true (non-zero), then the scanner terminates, returning 0 to its caller.

  The default **yywrap()** always returns 1. Presently, to redefine it you must first "#undef yywrap", as it is currently implemented as a macro. It is likely that **yywrap()** will soon be defined to be a function rather than a macro.

- YY_USER_ACTION can be redefined to provide an action which is always executed prior to the matched rule's action.
- The macro **YY_USER_INIT** may be redefined to provide an action which is always executed before the first scan.
- In the generated scanner, the actions are all gathered in one large switch statement and separated using **YY_BREAK,** which may be redefined. By default, it is simply a "break", to separate each rule's action from the following rule's.

FILES

flex.skel skeleton scanner.

lex.yy.c generated scanner (called *lexyy.c* on some systems).

lex.backtrack
backtracking information for **-b** flag (called *lex.bck* on some systems).

-lfl library with which to link the scanners.

SEE ALSO

flexdoc(1), lex(1), yacc(1), sed(1), awk(1).

M. E. Lesk and E. Schmidt, *LEX - Lexical Analyzer Generator*

DIAGNOSTICS

reject_used_but_not_detected undefined or

yymore_used_but_not_detected undefined - These errors can occur at compile time. They indicate that the scanner uses **REJECT** or **yymore()** but that *flex* failed to notice the fact, meaning that *flex* scanned the first two sections looking for occurrences of these actions and failed to find any, but somehow you snuck some in (via a #include file, for example). Make an explicit reference to the action in your *flex* input file. (Note that previously *flex* supported a **%used/%unused** mechanism for dealing with this problem; this feature is still supported but now deprecated, and will go away soon unless the author hears from people who can argue compellingly that they need it.)

flex scanner jammed - a scanner compiled with **-s** has encountered an input string which wasn't matched by any of its rules.

flex input buffer overflowed - a scanner rule matched a string long enough to overflow the scanner's internal input buffer (16K bytes - controlled by **YY_BUF_MAX** in "flex.skel").

scanner requires -8 flag - Your scanner specification includes recognizing 8-bit characters and you did not specify the -8 flag (and your site has not installed flex with -8 as the default).

fatal flex scanner internal error--end of buffer missed - This can occur in an scanner which is reentered after a long-jump has jumped out (or over) the scanner's activation frame. Before reentering the scanner, use:

```
yyrestart( yyin );
```

too many %t classes! - You managed to put every single character into its own %t class. *flex* requires that at least one of the classes share characters.

AUTHOR

Vern Paxson, with the help of many ideas and much inspiration from Van Jacobson. Original version by Jef Poskanzer.

See flexdoc(1) for additional credits and the address to send comments to.

DEFICIENCIES / BUGS

Some trailing context patterns cannot be properly matched and generate warning messages ("Dangerous trailing context"). These are patterns where the ending of the first part of the rule matches the beginning of the second part, such as "zx*/xy*", where the 'x*' matches the 'x' at the beginning of the trailing context. (Note that the POSIX draft states that the text matched by such patterns is

undefined.)

For some trailing context rules, parts which are actually fixed-length are not recognized as such, leading to the abovementioned performance loss. In particular, parts using '|' or {n} (such as "foo{3}") are always considered variable-length.

Combining trailing context with the special '|' action can result in *fixed* trailing context being turned into the more expensive *variable* trailing context. For example, this happens in the following example:

```
%%
abc     |
xyz/def
```

Use of unput() invalidates yytext and yyleng.

Use of unput() to push back more text than was matched can result in the pushed-back text matching a beginning-of-line ('^') rule even though it didn't come at the beginning of the line (though this is rare!).

Pattern-matching of NUL's is substantially slower than matching other characters.

flex does not generate correct #line directives for code internal to the scanner; thus, bugs in *flex.skel* yield bogus line numbers.

Due to both buffering of input and read-ahead, you cannot intermix calls to <stdio.h> routines, such as, for example, **getchar(),** with *flex* rules and expect it to work. Call **input()** instead.

The total table entries listed by the **-v** flag excludes the number of table entries needed to determine what rule has been matched. The number of entries is equal to the number of DFA states if the scanner does not use **REJECT,** and somewhat greater than the number of states if it does.

REJECT cannot be used with the *-f* or *-F* options.

Some of the macros, such as **yywrap(),** may in the future become functions which live in the **-lfl** library. This will doubtless break a lot of code, but may be required for POSIX-compliance.

The *flex* internal algorithms need documentation.

NAME

flexdoc - fast lexical analyzer generator

SYNOPSIS

flex [-bcdfinpstvFILT8 -C[efmF] -Sskeleton] *[filename ...]*

DESCRIPTION

flex is a tool for generating *scanners:* programs which recognized lexical patterns in text. *flex* reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called *rules. flex* generates as output a C source file, **lex.yy.c,** which defines a routine **yylex().** This file is compiled and linked with the **-lfl** library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

SOME SIMPLE EXAMPLES

First some simple examples to get the flavor of how one uses *flex.* The following *flex* input specifies a scanner which whenever it encounters the string "username" will replace it with the user's login name:

```
%%
username    printf( "%s", getlogin() );
```

By default, any text not matched by a *flex* scanner is copied to the output, so the net effect of this scanner is to copy its input file to its output with each occurrence of "username" expanded. In this input, there is just one rule. "username" is the *pattern* and the "printf" is the *action.* The "%%" marks the beginning of the rules.

Here's another simple example:

```
    int num_lines = 0, num_chars = 0;

%%
\n    ++num_lines; ++num_chars;
.     ++num_chars;

%%
main()
    {
    yylex();
    printf( "# of lines = %d, # of chars = %d\n",
            num_lines, num_chars );
    }
```

This scanner counts the number of characters and the number of lines in its input (it produces no output other than the final report on the counts). The first line declares two globals, "num_lines" and "num_chars", which are accessible both inside **yylex()** and in the **main()** routine declared after the second "%%". There are two rules, one which matches a newline ("\n") and increments both the line count and the character count, and one which matches any character other than a newline (indicated by the "." regular expression).

A somewhat more complicated example:

```
/* scanner for a toy Pascal-like language */

%{
/* need this for the call to atof() below */
#include <math.h>
%}
```

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+    {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }

{DIGIT}+"."{DIGIT}*         {
            printf( "A float: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if|then|begin|end|procedure|function        {
            printf( "A keyword: %s\n", yytext );
            }

{ID}        printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "An operator: %s\n", yytext );

"{"[^}\n]*"}"     /* eat up one-line comments */

[ \t\n]+          /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );

%%

main( argc, argv )
int argc;
char **argv;
    {
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;

    yylex();
    }
```

This is the beginnings of a simple scanner for a language like Pascal. It identifies different types of *tokens* and reports on what it has seen.

The details of this example will be explained in the following sections.

FORMAT OF THE INPUT FILE

The *flex* input file consists of three sections, separated by a line with just **%%** in it:

```
definitions
%%
rules
%%
user code
```

The *definitions* section contains declarations of simple *name* definitions to simplify the scanner specification, and declarations of *start conditions*, which are explained in a later section.

Name definitions have the form:

```
name definition
```

The "name" is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non-white-space character following the name and continuing to the end of the line. The definition can subsequently be referred to using "{name}", which will expand to "(definition)". For example,

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

defines "DIGIT" to be a regular expression which matches a single digit, and "ID" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to

```
{DIGIT}+"."{DIGIT}*
```

is identical to

```
([0-9])+"."([0-9])*
```

and matches one-or-more digits followed by a '.' followed by zero-or-more digits.

The *rules* section of the *flex* input contains a series of rules of the form:

```
pattern   action
```

where the pattern must be unindented and the action must begin on the same line.

See below for a further description of patterns and actions.

Finally, the user code section is simply copied to **lex.yy.c** verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second **%%** in the input file may be skipped, too.

In the definitions and rules sections, any *indented* text or text enclosed in **%{** and **%}** is copied verbatim to the output (with the %{}'s removed). The %{}'s must appear unindented on lines by themselves.

In the rules section, any indented or %{} text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or %{} text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors (this feature is present for *POSIX* compliance; see below for other such features).

In the definitions section, an unindented comment (i.e., a line beginning with "/*") is also copied verbatim to the output up to the next "*/". Also, any line in the definitions section beginning with '#' is ignored, though this style of comment is deprecated and may go away in the future.

PATTERNS

The patterns in the input are written using an extended set of regular expressions. These are:

```
x         match the character 'x'
.         any character except newline
[xyz]     a "character class"; in this case, the pattern
          matches either an 'x', a 'y', or a 'z'
```

[abj-oZ] a "character class" with a range in it; matches an ’a’, a ’b’, any letter from ’j’ through ’o’, or a ’Z’

[^A-Z] a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.

[^A-Z\n] any character EXCEPT an uppercase letter or a newline

r* zero or more r’s, where r is any regular expression

r+ one or more r’s

r? zero or one r’s (that is, "an optional r")

r{2,5} anywhere from two to five r’s

r{2,} two or more r’s

r{4} exactly 4 r’s

{name} the expansion of the "name" definition (see above)

"[xyz]\"foo"
the literal string: [xyz]"foo

\X if X is an ’a’, ’b’, ’f’, ’n’, ’r’, ’t’, or ’v’, then the ANSI-C interpretation of \x. Otherwise, a literal ’X’ (used to escape operators such as ’*’)

\123 the character with octal value 123

\x2a the character with hexadecimal value 2a

(r) match an r; parentheses are used to override precedence (see below)

rs the regular expression r followed by the regular expression s; called "concatenation"

r|s either an r or an s

r/s an r but only if it is followed by an s. The s is not part of the matched text. This type of pattern is called as "trailing context".

^r an r, but only at the beginning of a line

r$ an r, but only at the end of a line. Equivalent to "r/\n".

<s>r an r, but only in start condition s (see below for discussion of start conditions)

<s1,s2,s3>r
same, but in any of start conditions s1, s2, or s3

<<EOF>> an end-of-file

<s1,s2><<EOF>>
an end-of-file when in start condition s1 or s2

The regular expressions listed above are grouped according to precedence, from highest precedence at

the top to lowest at the bottom. Those grouped together have equal precedence. For example,

```
foo|bar*
```

is the same as

```
(foo)|(ba(r*))
```

since the '*' operator has higher precedence than concatenation, and concatenation higher than alternation ('|'). This pattern therefore matches *either* the string "foo" *or* the string "ba" followed by zero-or-more r's. To match "foo" or zero-or-more "bar"'s, use:

```
foo|(bar)*
```

and to match zero-or-more "foo"'s-or-"bar"'s:

```
(foo|bar)*
```

Some notes on patterns:

- A negated character class such as the example "[^A-Z]" above *will match a newline* unless "\n" (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., "[^A-Z\n]"). This is unlike how many other regular expression tools treat negated character classes, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like [^"]* can match an entire input (overflowing the scanner's input buffer) unless there's another quote in the input.

- A rule can have at most one instance of trailing context (the '/' operator or the '$' operator). The start condition, '^', and "<<EOF>>" patterns can only occur at the beginning of a pattern, and, as well as with '/' and '$', cannot be grouped inside parentheses. A '^' which does not occur at the beginning of a rule or a '$' which does not occur at the end of a rule loses its special properties and is treated as a normal character.

  The following are illegal:

  ```
  foo/bar$
  <sc1>foo<sc2>bar
  ```

  Note that the first of these, can be written "foo/bar\n".

  The following will result in '$' or '^' being treated as a normal character:

  ```
  foo|(bar$)
  foo|^bar
  ```

  If what's wanted is a "foo" or a bar-followed-by-a-newline, the following could be used (the special '|' action is explained below):

  ```
  foo     |
  bar$    /* action goes here */
  ```

  A similar trick will work for matching a foo or a bar-at-the-beginning-of-a-line.

HOW THE INPUT IS MATCHED

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the *flex* input file is chosen.

Once the match is determined, the text corresponding to the match (called the *token)* is made available in the global character pointer **yytext,** and its length in the global integer **yyleng.** The *action* corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, then the *default rule* is executed: the next character in the input is considered matched and copied to the standard output. Thus, the simplest legal *flex* input is:

```
%%
```

which generates a scanner that simply copies its input (one character at a time) to its output.

ACTIONS

Each pattern in a rule has a corresponding action, which can be any arbitrary C statement. The pattern ends at the first non-escaped whitespace character; the remainder of the line is its action. If the action is empty, then when the pattern is matched the input token is simply discarded. For example, here is the specification for a program which deletes all occurrences of "zap me" from its input:

```
%%
"zap me"
```

(It will copy all other characters in the input to the output since they will be matched by the default rule.)

Here is a program which compresses multiple blanks and tabs down to a single blank, and throws away whitespace found at the end of a line:

```
%%
[ \t]+        putchar( ' ' );
[ \t]+$       /* ignore this token */
```

If the action contains a '{', then the action spans till the balancing '}' is found, and the action may cross multiple lines. *flex* knows about C strings and comments and won't be fooled by braces found within them, but also allows actions to begin with **%{** and will consider the action to be all the text up to the next **%}** (regardless of ordinary braces inside the action).

An action consisting solely of a vertical bar ('|') means "same as the action for the next rule." See below for an illustration.

Actions can include arbitrary C code, including **return** statements to return a value to whatever routine called **yylex().** Each time **yylex()** is called it continues processing tokens from where it last left off until it either reaches the end of the file or executes a return. Once it reaches an end-of-file, however, then any subsequent call to **yylex()** will simply immediately return, unless **yyrestart()** is first called (see below).

Actions are not allowed to modify yytext or yyleng.

There are a number of special directives which can be included within an action:

- **ECHO** copies yytext to the scanner's output.
- **BEGIN** followed by the name of a start condition places the scanner in the corresponding start condition (see below).
- **REJECT** directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input). The rule is chosen as described above in "How the Input is Matched", and **yytext** and **yyleng** set up appropriately. It may either be one which matched as much text as the originally chosen rule but came later in the *flex* input file, or one which matched less text. For example, the following will both count the words in the input and call the routine special() whenever "frob" is seen:

```
    int word_count = 0;
%%

frob       special(); REJECT;
[^ \t\n]+   ++word_count;
```

Without the **REJECT,** any "frob"'s in the input would not be counted as words, since the scanner normally executes only one action per token. Multiple **REJECT's** are allowed, each one finding the next best choice to the currently active rule. For example, when the following scanner scans the token "abcd", it will write "abcdabcaba" to the output:

```
%%
a       |
ab      |
abc     |
abcd    ECHO; REJECT;
.|\n    /* eat up any unmatched character */
```

(The first three rules share the fourth's action since they use the special '|' action.) **REJECT** is a particularly expensive feature in terms scanner performance; if it is used in *any* of the scanner's actions it will slow down *all* of the scanner's matching. Furthermore, **REJECT** cannot be used with the *-f* or *-F* options (see below).

Note also that unlike the other special actions, **REJECT** is a *branch;* code immediately following it in the action will *not* be executed.

- **yymore()** tells the scanner that the next time it matches a rule, the corresponding token should be *appended* onto the current value of **yytext** rather than replacing it. For example, given the input "mega-kludge" the following will write "mega-mega-kludge" to the output:

```
%%
mega-    ECHO; yymore();
kludge   ECHO;
```

First "mega-" is matched and echoed to the output. Then "kludge" is matched, but the previous "mega-" is still hanging around at the beginning of **yytext** so the **ECHO** for the "kludge" rule will actually write "mega-kludge". The presence of **yymore()** in the scanner's action entails a minor performance penalty in the scanner's matching speed.

- **yyless(n)** returns all but the first *n* characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. **yytext** and **yyleng** are adjusted appropriately (e.g., **yyleng** will now be equal to *n*). For example, on the input "foobar" the following will write out "foobarbar":

```
%%
foobar    ECHO; yyless(3);
[a-z]+    ECHO;
```

An argument of 0 to **yyless** will cause the entire current input string to be scanned again. Unless you've changed how the scanner will subsequently process its input (using **BEGIN,** for example), this will result in an endless loop.

- **unput(c)** puts the character *c* back onto the input stream. It will be the next character scanned. The following action will take the current token and cause it to be rescanned enclosed in parentheses.

```
{
int i;
```

```
unput( ')' );
for ( i = yyleng - 1; i >= 0; --i )
  unput( yytext[i] );
unput( '(' );
}
```

Note that since each **unput()** puts the given character back at the *beginning* of the input stream, pushing back strings must be done back-to-front.

- **input()** reads the next character from the input stream. For example, the following is one way to eat up C comments:

```
%%
"/*"      {
          register int c;

          for ( ; ; )
            {
            while ( (c = input()) != '*' &&
                    c != EOF )
              ;    /* eat up text of comment */

            if ( c == '*' )
              {
              while ( (c = input()) == '*' )
                ;
              if ( c == '/' )
                break;    /* found the end */
              }

            if ( c == EOF )
              {
              error( "EOF in comment" );
              break;
              }
            }
          }
```

(Note that if the scanner is compiled using **C++,** then **input()** is instead referred to as **yyinput(),** in order to avoid a name clash with the **C++** stream by the name of *input.)*

- **yyterminate()** can be used in lieu of a return statement in an action. It terminates the scanner and returns a 0 to the scanner's caller, indicating "all done". Subsequent calls to the scanner will immediately return unless preceded by a call to **yyrestart()** (see below). By default, **yyterminate()** is also called when an end-of-file is encountered. It is a macro and may be redefined.

THE GENERATED SCANNER

The output of *flex* is the file **lex.yy.c,** which contains the scanning routine **yylex(),** a number of tables used by it for matching tokens, and a number of auxiliary routines and macros. By default, **yylex()** is declared as follows:

```
int yylex()
  {
  ... various definitions and the actions in here ...
  }
```

(If your environment supports function prototypes, then it will be "int yylex(void)".) This definition may be changed by redefining the "YY_DECL" macro. For example, you could use:

```
#undef YY_DECL
#define YY_DECL float lexscan( a, b ) float a, b;
```

to give the scanning routine the name *lexscan*, returning a float, and taking two floats as arguments. Note that if you give arguments to the scanning routine using a K&R-style/non-prototyped function declaration, you must terminate the definition with a semi-colon (;).

Whenever **yylex()** is called, it scans tokens from the global input file *yyin* (which defaults to stdin). It continues until it either reaches an end-of-file (at which point it returns the value 0) or one of its actions executes a *return* statement. In the former case, when called again the scanner will immediately return unless **yyrestart()** is called to point *yyin* at the new input file. (**yyrestart()** takes one argument, a **FILE *** pointer.) In the latter case (i.e., when an action executes a return), the scanner may then be called again and it will resume scanning where it left off.

By default (and for purposes of efficiency), the scanner uses block-reads rather than simple *getc()* calls to read characters from *yyin*. The nature of how it gets its input can be controlled by redefining the **YY_INPUT** macro. YY_INPUT's calling sequence is "YY_INPUT(buf,result,max_size)". Its action is to place up to *max_size* characters in the character array *buf* and return in the integer variable *result* either the number of characters read or the constant YY_NULL (0 on Unix systems) to indicate EOF. The default YY_INPUT reads from the global file-pointer "yyin".

A sample redefinition of YY_INPUT (in the definitions section of the input file):

```
%{
#undef YY_INPUT
#define YY_INPUT(buf,result,max_size) \
    { \
    int c = getchar(); \
    result = (c == EOF) ? YY_NULL : (buf[0] = c, 1); \
    }
%}
```

This definition will change the input processing to occur one character at a time.

You also can add in things like keeping track of the input line number this way; but don't expect your scanner to go very fast.

When the scanner receives an end-of-file indication from YY_INPUT, it then checks the **yywrap()** function. If **yywrap()** returns false (zero), then it is assumed that the function has gone ahead and set up *yyin* to point to another input file, and scanning continues. If it returns true (non-zero), then the scanner terminates, returning 0 to its caller.

The default **yywrap()** always returns 1. Presently, to redefine it you must first "#undef yywrap", as it is currently implemented as a macro. As indicated by the hedging in the previous sentence, it may be changed to a true function in the near future.

The scanner writes its **ECHO** output to the *yyout* global (default, stdout), which may be redefined by the user simply by assigning it to some other **FILE** pointer.

START CONDITIONS

flex provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with "<sc>" will only be active when the scanner is in the start condition named "sc". For example,

```
<STRING>[^"]*        { /* eat up the string body ... */
        ...
        }
```

will be active only when the scanner is in the "STRING" start condition, and

```
<INITIAL,STRING,QUOTE>\.        { /* handle an escape ... */
        ...
        }
```

will be active only when the current start condition is either "INITIAL", "STRING", or "QUOTE".

Start conditions are declared in the definitions (first) section of the input using unindented lines beginning with either **%s** or **%x** followed by a list of names. The former declares *inclusive* start conditions, the latter *exclusive* start conditions. A start condition is activated using the **BEGIN** action. Until the next **BEGIN** action is executed, rules with the given start condition will be active and rules with other start conditions will be inactive. If the start condition is *inclusive*, then rules with no start conditions at all will also be active. If it is *exclusive*, then *only* rules qualified with the start condition will be active. A set of rules contingent on the same exclusive start condition describe a scanner which is independent of any of the other rules in the *flex* input. Because of this, exclusive start conditions make it easy to specify "mini-scanners" which scan portions of the input that are syntactically different from the rest (e.g., comments).

If the distinction between inclusive and exclusive start conditions is still a little vague, here's a simple example illustrating the connection between the two. The set of rules:

```
%s example
%%
<example>foo          /* do something */
```

is equivalent to

```
%x example
%%
<INITIAL,example>foo    /* do something */
```

The default rule (to **ECHO** any unmatched character) remains active in start conditions.

BEGIN(0) returns to the original state where only the rules with no start conditions are active. This state can also be referred to as the start-condition "INITIAL", so **BEGIN(INITIAL)** is equivalent to **BEGIN(0).** (The parentheses around the start condition name are not required but are considered good style.)

BEGIN actions can also be given as indented code at the beginning of the rules section. For example, the following will cause the scanner to enter the "SPECIAL" start condition whenever *yylex()* is called and the global variable *enter_special* is true:

```
        int enter_special;

%x SPECIAL
%%
        if ( enter_special )
            BEGIN(SPECIAL);

<SPECIAL>blahblahblah
...more rules follow...
```

To illustrate the uses of start conditions, here is a scanner which provides two different interpretations of a string like "123.456". By default it will treat it as as three tokens, the integer "123", a dot ('.'), and the integer "456". But if the string is preceded earlier in the line by the string "expect-floats" it will treat it as a single token, the floating-point number 123.456:

```
%{
#include <math.h>
%}
%s expect

%%
expect-floats        BEGIN(expect);

<expect>[0-9]+"."[0-9]+      {
          printf( "found a float, = %f\n",
                atof( yytext ) );
          }
<expect>\n           {
          /* that's the end of the line, so
           * we need another "expect-number"
           * before we'll recognize any more
           * numbers
           */
          BEGIN(INITIAL);
          }

[0-9]+      {
          printf( "found an integer, = %d\n",
                atoi( yytext ) );
          }

"."         printf( "found a dot\n" );
```

Here is a scanner which recognizes (and discards) C comments while maintaining a count of the current input line.

```
%x comment
%%
       int line_num = 1;

"/*"         BEGIN(comment);

<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

Note that start-conditions names are really integer values and can be stored as such. Thus, the above could be extended in the following fashion:

```
%x comment foo
%%
       int line_num = 1;
       int comment_caller;

"/*"         {
             comment_caller = INITIAL;
             BEGIN(comment);
             }
```

```
...

<foo>"/*"    {
             comment_caller = foo;
             BEGIN(comment);
             }

<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(comment_caller);
```

One can then implement a "stack" of start conditions using an array of integers. (It is likely that such stacks will become a full-fledged *flex* feature in the future.) Note, though, that start conditions do not have their own name-space; %s's and %x's declare names in the same fashion as #define's.

MULTIPLE INPUT BUFFERS

Some scanners (such as those which support "include" files) require reading from several input streams. As *flex* scanners do a large amount of buffering, one cannot control where the next input will be read from by simply writing a **YY_INPUT** which is sensitive to the scanning context. **YY_INPUT** is only called when the scanner reaches the end of its buffer, which may be a long time after scanning a statement such as an "include" which requires switching the input source.

To negotiate these sorts of problems, *flex* provides a mechanism for creating and switching between multiple input buffers. An input buffer is created by using:

```
YY_BUFFER_STATE yy_create_buffer( FILE *file, int size )
```

which takes a *FILE* pointer and a size and creates a buffer associated with the given file and large enough to hold *size* characters (when in doubt, use **YY_BUF_SIZE** for the size). It returns a **YY_BUFFER_STATE** handle, which may then be passed to other routines:

```
void yy_switch_to_buffer( YY_BUFFER_STATE new_buffer )
```

switches the scanner's input buffer so subsequent tokens will come from *new_buffer*. Note that **yy_switch_to_buffer()** may be used by yywrap() to sets things up for continued scanning, instead of opening a new file and pointing *yyin* at it.

```
void yy_delete_buffer( YY_BUFFER_STATE buffer )
```

is used to reclaim the storage associated with a buffer.

yy_new_buffer() is an alias for **yy_create_buffer(),** provided for compatibility with the C++ use of *new* and *delete* for creating and destroying dynamic objects.

Finally, the **YY_CURRENT_BUFFER** macro returns a **YY_BUFFER_STATE** handle to the current buffer.

Here is an example of using these features for writing a scanner which expands include files (the **<<EOF>>** feature is discussed below):

```
/* the "incl" state is used for picking up the name
 * of an include file
 */
%x incl

%{
#define MAX_INCLUDE_DEPTH 10
```

```
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];
int include_stack_ptr = 0;
%}

%%
include           BEGIN(incl);

[a-z]+            ECHO;
[^a-z\n]*\n?      ECHO;

<incl>[ \t]*      /* eat the whitespace */
<incl>[^ \t\n]+   { /* got the include file name */
      if ( include_stack_ptr >= MAX_INCLUDE_DEPTH )
         {
         fprintf( stderr, "Includes nested too deeply" );
         exit( 1 );
         }

      include_stack[include_stack_ptr++] =
         YY_CURRENT_BUFFER;

      yyin = fopen( yytext, "r" );

      if ( ! yyin )
         error( ... );

      yy_switch_to_buffer(
         yy_create_buffer( yyin, YY_BUF_SIZE ) );

      BEGIN(INITIAL);
      }

<<EOF>> {
      if ( --include_stack_ptr < 0 )
         {
         yyterminate();
         }

      else
         yy_switch_to_buffer(
              include_stack[include_stack_ptr] );
      }
```

END-OF-FILE RULES

The special rule "<<EOF>>" indicates actions which are to be taken when an end-of-file is encountered and yywrap() returns non-zero (i.e., indicates no further files to process). The action must finish by doing one of four things:

- the special **YY_NEW_FILE** action, if *yyin* has been pointed at a new file to process;
- a *return* statement;
- the special **yyterminate()** action;
- or, switching to a new buffer using **yy_switch_to_buffer()** as shown in the example above.

<<EOF>> rules may not be used with other patterns; they may only be qualified with a list of start conditions. If an unqualified <<EOF>> rule is given, it applies to *all* start conditions which do not already

have <<EOF>> actions. To specify an <<EOF>> rule for only the initial start condition, use

```
<INITIAL><<EOF>>
```

These rules are useful for catching things like unclosed comments. An example:

```
%x quote
%%

...other rules for dealing with quotes...

<quote><<EOF>>   {
        error( "unterminated quote" );
        yyterminate();
        }
<<EOF>>  {
        if ( *++filelist )
           {
           yyin = fopen( *filelist, "r" );
           YY_NEW_FILE;
           }
        else
          yyterminate();
        }
```

MISCELLANEOUS MACROS

The macro **YY_USER_ACTION** can be redefined to provide an action which is always executed prior to the matched rule's action. For example, it could be #define'd to call a routine to convert yytext to lower-case.

The macro **YY_USER_INIT** may be redefined to provide an action which is always executed before the first scan (and before the scanner's internal initializations are done). For example, it could be used to call a routine to read in a data table or open a logging file.

In the generated scanner, the actions are all gathered in one large switch statement and separated using **YY_BREAK,** which may be redefined. By default, it is simply a "break", to separate each rule's action from the following rule's. Redefining **YY_BREAK** allows, for example, C++ users to #define YY_BREAK to do nothing (while being very careful that every rule ends with a "break" or a "return"!) to avoid suffering from unreachable statement warnings where because a rule's action ends with "return", the **YY_BREAK** is inaccessible.

INTERFACING WITH YACC

One of the main uses of *flex* is as a companion to the *yacc* parser-generator. *yacc* parsers expect to call a routine named **yylex()** to find the next input token. The routine is supposed to return the type of the next token as well as putting any associated value in the global **yylval.** To use *flex* with *yacc*, one specifies the **-d** option to *yacc* to instruct it to generate the file **y.tab.h** containing definitions of all the **%tokens** appearing in the *yacc* input. This file is then included in the *flex* scanner. For example, if one of the tokens is "TOK_NUMBER", part of the scanner might look like:

```
%{
#include "y.tab.h"
%}

%%

[0-9]+       yylval = atoi( yytext ); return TOK_NUMBER;
```

TRANSLATION TABLE

In the name of POSIX compliance, *flex* supports a *translation table* for mapping input characters into groups. The table is specified in the first section, and its format looks like:

```
%t
1       abcd
2       ABCDEFGHIJKLMNOPQRSTUVWXYZ
52       0123456789
6       \t\ \n
%t
```

This example specifies that the characters 'a', 'b', 'c', and 'd' are to all be lumped into group #1, upper-case letters in group #2, digits in group #52, tabs, blanks, and newlines into group #6, and *no other characters will appear in the patterns.* The group numbers are actually disregarded by *flex;* **%t** serves, though, to lump characters together. Given the above table, for example, the pattern "a(AA)*5" is equivalent to "d(ZQ)*0". They both say, "match any character in group #1, followed by zero-or-more pairs of characters from group #2, followed by a character from group #52." Thus **%t** provides a crude way for introducing equivalence classes into the scanner specification.

Note that the **-i** option (see below) coupled with the equivalence classes which *flex* automatically generates take care of virtually all the instances when one might consider using **%t.** But what the hell, it's there if you want it.

OPTIONS

flex has the following options:

-b Generate backtracking information to *lex.backtrack.* This is a list of scanner states which require backtracking and the input characters on which they do so. By adding rules one can remove backtracking states. If all backtracking states are eliminated and **-f** or **-F** is used, the generated scanner will run faster (see the **-p** flag). Only users who wish to squeeze every last cycle out of their scanners need worry about this option. (See the section on PERFORMANCE CONSIDERATIONS below.)

-c is a do-nothing, deprecated option included for POSIX compliance.

NOTE: in previous releases of *flex* **-c** specified table-compression options. This functionality is now given by the **-C** flag. To ease the the impact of this change, when *flex* encounters **-c,** it currently issues a warning message and assumes that **-C** was desired instead. In the future this "promotion" of **-c** to **-C** will go away in the name of full POSIX compliance (unless the POSIX meaning is removed first).

-d makes the generated scanner run in *debug* mode. Whenever a pattern is recognized and the global **yy_flex_debug** is non-zero (which is the default), the scanner will write to *stderr* a line of the form:

```
--accepting rule at line 53 ("the matched text")
```

The line number refers to the location of the rule in the file defining the scanner (i.e., the file that was fed to flex). Messages are also generated when the scanner backtracks, accepts the default rule, reaches the end of its input buffer (or encounters a NUL; at this point, the two look the same as far as the scanner's concerned), or reaches an end-of-file.

-f specifies (take your pick) *full table* or *fast scanner.* No table compression is done. The result is large but fast. This option is equivalent to **-Cf** (see below).

-i instructs *flex* to generate a *case-insensitive* scanner. The case of letters given in the *flex* input patterns will be ignored, and tokens in the input will be matched regardless of case. The matched text given in *yytext* will have the preserved case (i.e., it will not be folded).

-n is another do-nothing, deprecated option included only for POSIX compliance.

-p generates a performance report to stderr. The report consists of comments regarding features of the *flex* input file which will cause a loss of performance in the resulting scanner. Note that the use of *REJECT* and variable trailing context (see the BUGS section in flex(1)) entails a substantial performance penalty; use of *yymore()*, the ^ operator, and the **-I** flag entail minor performance penalties.

-s causes the *default rule* (that unmatched scanner input is echoed to *stdout)* to be suppressed. If the scanner encounters input that does not match any of its rules, it aborts with an error. This option is useful for finding holes in a scanner's rule set.

-t instructs *flex* to write the scanner it generates to standard output instead of **lex.yy.c.**

-v specifies that *flex* should write to *stderr* a summary of statistics regarding the scanner it generates. Most of the statistics are meaningless to the casual *flex* user, but the first line identifies the version of *flex*, which is useful for figuring out where you stand with respect to patches and new releases, and the next two lines give the date when the scanner was created and a summary of the flags which were in effect.

-F specifies that the *fast* scanner table representation should be used. This representation is about as fast as the full table representation (**−f**), and for some sets of patterns will be considerably smaller (and for others, larger). In general, if the pattern set contains both "keywords" and a catch-all, "identifier" rule, such as in the set:

```
"case"    return TOK_CASE;
"switch"  return TOK_SWITCH;
...
"default" return TOK_DEFAULT;
[a-z]+    return TOK_ID;
```

then you're better off using the full table representation. If only the "identifier" rule is present and you then use a hash table or some such to detect the keywords, you're better off using **−F**.

This option is equivalent to **-CF** (see below).

-I instructs *flex* to generate an *interactive* scanner. Normally, scanners generated by *flex* always look ahead one character before deciding that a rule has been matched. At the cost of some scanning overhead, *flex* will generate a scanner which only looks ahead when needed. Such scanners are called *interactive* because if you want to write a scanner for an interactive system such as a command shell, you will probably want the user's input to be terminated with a newline, and without **-I** the user will have to type a character in addition to the newline in order to have the newline recognized. This leads to dreadful interactive performance.

If all this seems to confusing, here's the general rule: if a human will be typing in input to your scanner, use **-I,** otherwise don't; if you don't care about squeezing the utmost performance from your scanner and you don't want to make any assumptions about the input to your scanner, use **-I.**

Note, **-I** cannot be used in conjunction with *full* or *fast tables*, i.e., the **-f, -F, -Cf,** or **-CF** flags.

-L instructs *flex* not to generate **#line** directives. Without this option, *flex* peppers the generated scanner with #line directives so error messages in the actions will be correctly located with respect to the original *flex* input file, and not to the fairly meaningless line numbers of **lex.yy.c.** (Unfortunately *flex* does not presently generate the necessary directives to "retarget" the line numbers for those parts of **lex.yy.c** which it generated. So if there is an error in the generated code, a meaningless line number is reported.)

-T makes *flex* run in *trace* mode. It will generate a lot of messages to *stdout* concerning the form of the input and the resultant non-deterministic and deterministic finite automata. This option is mostly for use in maintaining *flex.*

-8 instructs *flex* to generate an 8-bit scanner, i.e., one which can recognize 8-bit characters. On some sites, *flex* is installed with this option as the default. On others, the default is 7-bit

characters. To see which is the case, check the verbose **(-v)** output for "equivalence classes created". If the denominator of the number shown is 128, then by default *flex* is generating 7-bit characters. If it is 256, then the default is 8-bit characters and the **-8** flag is not required (but may be a good idea to keep the scanner specification portable). Feeding a 7-bit scanner 8-bit characters will result in infinite loops, bus errors, or other such fireworks, so when in doubt, use the flag. Note that if equivalence classes are used, 8-bit scanners take only slightly more table space than 7-bit scanners (128 bytes, to be exact); if equivalence classes are not used, however, then the tables may grow up to twice their 7-bit size.

-C[efmF]

controls the degree of table compression.

-Ce directs *flex* to construct *equivalence classes*, i.e., sets of characters which have identical lexical properties (for example, if the only appearance of digits in the *flex* input is in the character class "[0-9]" then the digits '0', '1', ..., '9' will all be put in the same equivalence class). Equivalence classes usually give dramatic reductions in the final table/object file sizes (typically a factor of 2-5) and are pretty cheap performance-wise (one array look-up per character scanned).

-Cf specifies that the *full* scanner tables should be generated - *flex* should not compress the tables by taking advantages of similar transition functions for different states.

-CF specifies that the alternate fast scanner representation (described above under the **-F** flag) should be used.

-Cm directs *flex* to construct *meta-equivalence classes*, which are sets of equivalence classes (or characters, if equivalence classes are not being used) that are commonly used together. Meta-equivalence classes are often a big win when using compressed tables, but they have a moderate performance impact (one or two "if" tests and one array look-up per character scanned).

A lone **-C** specifies that the scanner tables should be compressed but neither equivalence classes nor meta-equivalence classes should be used.

The options **-Cf** or **-CF** and **-Cm** do not make sense together - there is no opportunity for meta-equivalence classes if the table is not being compressed. Otherwise the options may be freely mixed.

The default setting is **-Cem,** which specifies that *flex* should generate equivalence classes and meta-equivalence classes. This setting provides the highest degree of table compression. You can trade off faster-executing scanners at the cost of larger tables with the following generally being true:

```
slowest & smallest
      -Cem
      -Cm
      -Ce
      -C
      -C{f,F}e
      -C{f,F}
fastest & largest
```

Note that scanners with the smallest tables are usually generated and compiled the quickest, so during development you will usually want to use the default, maximal compression.

-Cfe is often a good compromise between speed and size for production scanners.

-C options are not cumulative; whenever the flag is encountered, the previous -C settings are forgotten.

-Sskeleton_file

overrides the default skeleton file from which *flex* constructs its scanners. You'll never need

this option unless you are doing *flex* maintenance or development.

PERFORMANCE CONSIDERATIONS

The main design goal of *flex* is that it generate high-performance scanners. It has been optimized for dealing well with large sets of rules. Aside from the effects of table compression on scanner speed outlined above, there are a number of options/actions which degrade performance. These are, from most expensive to least:

```
REJECT

pattern sets that require backtracking
arbitrary trailing context

'^' beginning-of-line operator
yymore()
```

with the first three all being quite expensive and the last two being quite cheap.

REJECT should be avoided at all costs when performance is important. It is a particularly expensive option.

Getting rid of backtracking is messy and often may be an enormous amount of work for a complicated scanner. In principal, one begins by using the **-b** flag to generate a *lex.backtrack* file. For example, on the input

```
%%
foo       return TOK_KEYWORD;
foobar    return TOK_KEYWORD;
```

the file looks like:

```
State #6 is non-accepting -
 associated rule line numbers:
       2       3
 out-transitions: [ o ]
 jam-transitions: EOF [ \001-n  p-\177 ]

State #8 is non-accepting -
 associated rule line numbers:
       3
 out-transitions: [ a ]
 jam-transitions: EOF [ \001-`  b-\177 ]

State #9 is non-accepting -
 associated rule line numbers:
       3
 out-transitions: [ r ]
 jam-transitions: EOF [ \001-q  s-\177 ]

Compressed tables always backtrack.
```

The first few lines tell us that there's a scanner state in which it can make a transition on an 'o' but not on any other character, and that in that state the currently scanned text does not match any rule. The state occurs when trying to match the rules found at lines 2 and 3 in the input file. If the scanner is in that state and then reads something other than an 'o', it will have to backtrack to find a rule which is matched. With a bit of headscratching one can see that this must be the state it's in when it has seen "fo". When this has happened, if anything other than another 'o' is seen, the scanner will have to back

up to simply match the 'f' (by the default rule).

The comment regarding State #8 indicates there's a problem when "foob" has been scanned. Indeed, on any character other than a 'b', the scanner will have to back up to accept "foo". Similarly, the comment for State #9 concerns when "fooba" has been scanned.

The final comment reminds us that there's no point going to all the trouble of removing backtracking from the rules unless we're using **-f** or **-F,** since there's no performance gain doing so with compressed scanners.

The way to remove the backtracking is to add "error" rules:

```
%%
foo         return TOK_KEYWORD;
foobar      return TOK_KEYWORD;

fooba       |
foob        |
fo          {
            /* false alarm, not really a keyword */
            return TOK_ID;
            }
```

Eliminating backtracking among a list of keywords can also be done using a "catch-all" rule:

```
%%
foo         return TOK_KEYWORD;
foobar      return TOK_KEYWORD;

[a-z]+      return TOK_ID;
```

This is usually the best solution when appropriate.

Backtracking messages tend to cascade. With a complicated set of rules it's not uncommon to get hundreds of messages. If one can decipher them, though, it often only takes a dozen or so rules to eliminate the backtracking (though it's easy to make a mistake and have an error rule accidentally match a valid token. A possible future *flex* feature will be to automatically add rules to eliminate backtracking).

Variable trailing context (where both the leading and trailing parts do not have a fixed length) entails almost the same performance loss as *REJECT* (i.e., substantial). So when possible a rule like:

```
%%
mouse|rat/(cat|dog)   run();
```

is better written:

```
%%
mouse/cat|dog         run();
rat/cat|dog           run();
```

or as

```
%%
mouse|rat/cat         run();
mouse|rat/dog         run();
```

Note that here the special '|' action does *not* provide any savings, and can even make things worse (see **BUGS** in flex(1)).

Another area where the user can increase a scanner's performance (and one that's easier to implement) arises from the fact that the longer the tokens matched, the faster the scanner will run. This is because with long tokens the processing of most input characters takes place in the (short) inner scanning loop, and does not often have to go through the additional work of setting up the scanning environment (e.g., **yytext)** for the action. Recall the scanner for C comments:

```
%x comment
%%
      int line_num = 1;

"/*"        BEGIN(comment);

<comment>[^*\n]*
<comment>"*"+[^*/\n]*
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

This could be sped up by writing it as:

```
%x comment
%%
      int line_num = 1;

"/*"        BEGIN(comment);

<comment>[^*\n]*
<comment>[^*\n]*\n      ++line_num;
<comment>"*"+[^*/\n]*
<comment>"*"+[^*/\n]*\n ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

Now instead of each newline requiring the processing of another action, recognizing the newlines is "distributed" over the other rules to keep the matched text as long as possible. Note that *adding* rules does *not* slow down the scanner! The speed of the scanner is independent of the number of rules or (modulo the considerations given at the beginning of this section) how complicated the rules are with regard to operators such as '*' and '|'.

A final example in speeding up a scanner: suppose you want to scan through a file containing identifiers and keywords, one per line and with no other extraneous characters, and recognize all the keywords. A natural first approach is:

```
%%
asm      |
auto     |
break    |
... etc ...
volatile |
while    /* it's a keyword */

.|\n     /* it's not a keyword */
```

To eliminate the back-tracking, introduce a catch-all rule:

```
%%
asm      |
auto     |
```

```
break    |
... etc ...
volatile |
while    /* it's a keyword */

[a-z]+   |
.|\n     /* it's not a keyword */
```

Now, if it's guaranteed that there's exactly one word per line, then we can reduce the total number of matches by a half by merging in the recognition of newlines with that of the other tokens:

```
%%
asm\n    |
auto\n   |
break\n  |
... etc ...
volatile\n |
while\n  /* it's a keyword */

[a-z]+\n |
.|\n     /* it's not a keyword */
```

One has to be careful here, as we have now reintroduced backtracking into the scanner. In particular, while *we* know that there will never be any characters in the input stream other than letters or newlines, *flex* can't figure this out, and it will plan for possibly needing backtracking when it has scanned a token like "auto" and then the next character is something other than a newline or a letter. Previously it would then just match the "auto" rule and be done, but now it has no "auto" rule, only a "auto\n" rule. To eliminate the possibility of backtracking, we could either duplicate all rules but without final newlines, or, since we never expect to encounter such an input and therefore don't how it's classified, we can introduce one more catch-all rule, this one which doesn't include a newline:

```
%%
asm\n    |
auto\n   |
break\n  |
... etc ...
volatile\n |
while\n  /* it's a keyword */

[a-z]+\n |
[a-z]+   |
.|\n     /* it's not a keyword */
```

Compiled with **-Cf,** this is about as fast as one can get a *flex* scanner to go for this particular problem.

A final note: *flex* is slow when matching NUL's, particularly when a token contains multiple NUL's. It's best to write rules which match *short* amounts of text if it's anticipated that the text will often include NUL's.

INCOMPATIBILITIES WITH LEX AND POSIX

flex is a rewrite of the Unix *lex* tool (the two implementations do not share any code, though), with some extensions and incompatibilities, both of which are of concern to those who wish to write scanners acceptable to either implementation. At present, the POSIX *lex* draft is very close to the original *lex* implementation, so some of these incompatibilities are also in conflict with the POSIX draft. But the intent is that except as noted below, *flex* as it presently stands will ultimately be POSIX conformant (i.e., that those areas of conflict with the POSIX draft will be resolved in *flex's* favor). Please bear in mind that all the comments which follow are with regard to the POSIX *draft* standard of

Summer 1989, and not the final document (or subsequent drafts); they are included so *flex* users can be aware of the standardization issues and those areas where *flex* may in the near future undergo changes incompatible with its current definition.

flex is fully compatible with *lex* with the following exceptions:

- The undocumented *lex* scanner internal variable **yylineno** is not supported. It is difficult to support this option efficiently, since it requires examining every character scanned and reexamining the characters when the scanner backs up. Things get more complicated when the end of buffer or file is reached or a NUL is scanned (since the scan must then be restarted with the proper line number count), or the user uses the yyless(), unput(), or REJECT actions, or the multiple input buffer functions.

  The fix is to add rules which, upon seeing a newline, increment yylineno. This is usually an easy process, though it can be a drag if some of the patterns can match multiple newlines along with other characters.

  yylineno is not part of the POSIX draft.

- The **input()** routine is not redefinable, though it may be called to read characters following whatever has been matched by a rule. If **input()** encounters an end-of-file the normal **yywrap()** processing is done. A ‘‘real’’ end-of-file is returned by **input()** as *EOF*.

  Input is instead controlled by redefining the **YY_INPUT** macro.

  The *flex* restriction that **input()** cannot be redefined is in accordance with the POSIX draft, but **YY_INPUT** has not yet been accepted into the draft (and probably won’t; it looks like the draft will simply not specify any way of controlling the scanner’s input other than by making an initial assignment to *yyin).*

- *flex* scanners do not use stdio for input. Because of this, when writing an interactive scanner one must explicitly call fflush() on the stream associated with the terminal after writing out a prompt. With *lex* such writes are automatically flushed since *lex* scanners use **getchar()** for their input. Also, when writing interactive scanners with *flex,* the **-I** flag must be used.

- *flex* scanners are not as reentrant as *lex* scanners. In particular, if you have an interactive scanner and an interrupt handler which long-jumps out of the scanner, and the scanner is subsequently called again, you may get the following message:

  ```
  fatal flex scanner internal error--end of buffer missed
  ```

  To reenter the scanner, first use

  ```
  yyrestart( yyin );
  ```

- **output()** is not supported. Output from the **ECHO** macro is done to the file-pointer *yyout* (default *stdout).*

  The POSIX draft mentions that an **output()** routine exists but currently gives no details as to what it does.

- *lex* does not support exclusive start conditions (%x), though they are in the current POSIX draft.

- When definitions are expanded, *flex* encloses them in parentheses. With lex, the following:

  ```
  NAME    [A-Z][A-Z0-9]*
  %%
  foo{NAME}?      printf( "Found it\n" );
  %%
  ```

  will not match the string "foo" because when the macro is expanded the rule is equivalent to

"foo[A-Z][A-Z0-9]*?" and the precedence is such that the '?' is associated with "[A-Z0-9]*". With *flex,* the rule will be expanded to "foo([A-Z][A-Z0-9]*)?" and so the string "foo" will match. Note that because of this, the **^, $, <s>, /,** and **<<EOF>>** operators cannot be used in a *flex* definition.

The POSIX draft interpretation is the same as *flex's.*

- To specify a character class which matches anything but a left bracket (']'), in *lex* one can use "[^]]" but with *flex* one must use "[^\]]". The latter works with *lex,* too.
- The *lex* **%r** (generate a Ratfor scanner) option is not supported. It is not part of the POSIX draft.
- If you are providing your own yywrap() routine, you must include a "#undef yywrap" in the definitions section (section 1). Note that the "#undef" will have to be enclosed in %{}'s.

  The POSIX draft specifies that yywrap() is a function and this is very unlikely to change; so *flex users are warned* that **yywrap()** is likely to be changed to a function in the near future.
- After a call to **unput(),** *yytext* and *yyleng* are undefined until the next token is matched. This is not the case with *lex* or the present POSIX draft.
- The precedence of the **{}** (numeric range) operator is different. *lex* interprets "abc{1,3}" as "match one, two, or three occurrences of 'abc'", whereas *flex* interprets it as "match 'ab' followed by one, two, or three occurrences of 'c'". The latter is in agreement with the current POSIX draft.
- The precedence of the ^ operator is different. *lex* interprets "^foo|bar" as "match either 'foo' at the beginning of a line, or 'bar' anywhere", whereas *flex* interprets it as "match either 'foo' or 'bar' if they come at the beginning of a line". The latter is in agreement with the current POSIX draft.
- To refer to yytext outside of the scanner source file, the correct definition with *flex* is "extern char *yytext" rather than "extern char yytext[]". This is contrary to the current POSIX draft but a point on which *flex* will not be changing, as the array representation entails a serious performance penalty. It is hoped that the POSIX draft will be emended to support the *flex* variety of declaration (as this is a fairly painless change to require of *lex* users).
- *yyin* is *initialized* by *lex* to be *stdin; flex,* on the other hand, initializes *yyin* to NULL and then *assigns* it to *stdin* the first time the scanner is called, providing *yyin* has not already been assigned to a non-NULL value. The difference is subtle, but the net effect is that with *flex* scanners, *yyin* does not have a valid value until the scanner has been called.
- The special table-size declarations such as **%a** supported by *lex* are not required by *flex* scanners; *flex* ignores them.
- The name **FLEX_SCANNER** is #define'd so scanners may be written for use with either *flex* or *lex.*

The following *flex* features are not included in *lex* or the POSIX draft standard:

yyterminate()
<<EOF>>
YY_DECL
#line directives
%{}'s around actions
yyrestart()
comments beginning with '#' (deprecated)
multiple actions on a line

This last feature refers to the fact that with *flex* you can put multiple actions on the same line, separated with semi-colons, while with *lex,* the following

```
foo    handle_foo(); ++num_foos_seen;
```

is (rather surprisingly) truncated to

```
foo    handle_foo();
```

flex does not truncate the action. Actions that are not enclosed in braces are simply terminated at the end of the line.

DIAGNOSTICS

reject_used_but_not_detected undefined or *yymore_used_but_not_detected undefined* - These errors can occur at compile time. They indicate that the scanner uses **REJECT** or **yymore()** but that *flex* failed to notice the fact, meaning that *flex* scanned the first two sections looking for occurrences of these actions and failed to find any, but somehow you snuck some in (via a #include file, for example). Make an explicit reference to the action in your *flex* input file. (Note that previously *flex* supported a **%used/%unused** mechanism for dealing with this problem; this feature is still supported but now deprecated, and will go away soon unless the author hears from people who can argue compellingly that they need it.)

flex scanner jammed - a scanner compiled with **-s** has encountered an input string which wasn't matched by any of its rules.

flex input buffer overflowed - a scanner rule matched a string long enough to overflow the scanner's internal input buffer (16K bytes by default - controlled by **YY_BUF_SIZE** in "flex.skel". Note that to redefine this macro, you must first **#undefine** it).

scanner requires -8 flag - Your scanner specification includes recognizing 8-bit characters and you did not specify the -8 flag (and your site has not installed flex with -8 as the default).

fatal flex scanner internal error--end of buffer missed - This can occur in an scanner which is reentered after a long-jump has jumped out (or over) the scanner's activation frame. Before reentering the scanner, use:

```
yyrestart( yyin );
```

too many %t classes! - You managed to put every single character into its own %t class. *flex* requires that at least one of the classes share characters.

DEFICIENCIES / BUGS

See flex(1).

SEE ALSO

flex(1), lex(1), yacc(1), sed(1), awk(1).

M. E. Lesk and E. Schmidt, *LEX - Lexical Analyzer Generator*

AUTHOR

Vern Paxson, with the help of many ideas and much inspiration from Van Jacobson. Original version by Jef Poskanzer. The fast table representation is a partial implementation of a design done by Van Jacobson. The implementation was done by Kevin Gong and Vern Paxson.

Thanks to the many *flex* beta-testers, feedbackers, and contributors, especially Casey Leedom, benson@odi.com, Keith Bostic, Frederic Brehm, Nick Christopher, Jason Coughlin, Scott David Daniels, Leo Eskin, Chris Faylor, Eric Goldman, Eric Hughes, Jeffrey R. Jones, Kevin B. Kenny, Ronald Lamprecht, Greg Lee, Craig Leres, Mohamed el Lozy, Jim Meyering, Marc Nozell, Esmond Pitt, Jef Poskanzer, Jim Roskind, Dave Tallman, Frank Whaley, Ken Yap, and those whose names have slipped my marginal mail-archiving skills but whose contributions are appreciated all the same.

Thanks to Keith Bostic, John Gilmore, Craig Leres, Bob Mulcahy, Rich Salz, and Richard Stallman for help with various distribution headaches.

Thanks to Esmond Pitt and Earle Horton for 8-bit character support; to Benson Margulies and Fred Burke for C++ support; to Ove Ewerlid for the basics of support for NUL's; and to Eric Hughes for the basics of support for multiple buffers.

Work is being done on extending *flex* to generate scanners in which the state machine is directly represented in C code rather than tables. These scanners may well be substantially faster than those generated using -f or -F. If you are working in this area and are interested in comparing notes and seeing whether redundant work can be avoided, contact Ove Ewerlid (ewerlid@mizar.DoCS.UU.SE).

This work was primarily done when I was at the Real Time Systems Group at the Lawrence Berkeley Laboratory in Berkeley, CA. Many thanks to all there for the support I received.

Send comments to:

Vern Paxson
Computer Science Department
4126 Upson Hall
Cornell University
Ithaca, NY 14853-7501

vern@cs.cornell.edu
decvax!cornell!vern

NAME

fmt - adjust line-length for paragraphs of text

SYNOPSIS

fmt [−*width*] [*files*]...

DESCRIPTION

fmt is a simple text formatter. It inserts or deletes newlines, as necessary, to make all lines in a paragraph be approximately the same width. It preserves indentation and word spacing.

The default line width is 72 characters. You can override this with the −*width* flag. If you don't name any files on the command line, then *fmt* will read from stdin.

It is typically used from within *vi* to adjust the line breaks in a single paragraph. To do this, move the cursor to the top of the paragraph, type "!}fmt", and hit <Return>.

AUTHOR

Steve Kirkendall
kirkenda@cs.pdx.edu

NAME

fold − fold long lines

SYNOPSIS

fold [−*n*] [*file*] ...

OPTIONS

−*n* How long should the output lines be

EXAMPLES

fold −60 # Fold *stdin* to 60 characters

fold file # Fold *file* to 80 characters

DESCRIPTION

Fold takes copies its input from the named file (or *stdin*, if none is specified) to standard output. However, lines longer than the given maximum (default 80) are broken into multiple lines of the maximum length by inserting new line characters.

SEE ALSO

width(1).

NAME

format − format a PC floppy diskette

SYNOPSIS

format [**−v**] *device* [*media-size* [*drive-size*]]

DESCRIPTION

Format allows a user with read-write permission to *device* to format a floppy. Either one of the special floppy devices must be used, see **fd**(4), or an automatic device may be used with the size of the floppy specified on the command line. Two sizes must be given when formatting a low density diskette in a high density drive. For example:

format /dev/at1
format /dev/fd1 1200
format /dev/fd1 360 1200

The first two commands format a 1.2M diskette, the last formats a 360k diskette in a 1.2M drive. A 1.44M drive knows when it's dealing with a low density floppy, so all these commands format a 720k diskette:

format /dev/fd0 720
format /dev/fd0 720 1440
format /dev/ps0

No sizes may be specified when using a special floppy device, a size must be specified when using an automatic device.

OPTIONS

−v Verify the process by reading each track after formatting it. Formatting is normally blind, the controller has no idea whether it succeeds or not. Use **−v** on a new box of cheap diskettes, or on a diskette that may have gone bad. Verifying will increase formatting time by 50%.

SEE ALSO

mkfs(1), **fd**(4).

DIAGNOSTICS

Numbers will be printed on standard output to show that it is busy. The locations of bad sectors are printed on standard error when verifying. The exit code is zero unless there are too many bad spots.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

fortune − print a fortune

SYNOPSIS

fortune

EXAMPLES

fortune # Print a fortune

DESCRIPTION

Fortune prints a fortune at random from the fortunes file, */usr/lib/fortune.dat*. This file consists of pieces of text separated by a line containing only %%.

NAME

fsck, fsck1 − perform file system consistency check

SYNOPSIS

fsck [**−aclmrs**] [*device*] ...

OPTIONS

−a Automatically repair inconsistencies

−c Check and list only the specified i-nodes

−l List the files and directories in the filesytem

−r Prompt user for repairs if inconsistencies are found

−s List the superblock of the file system

EXAMPLES

fsck /dev/hd4 # Check file system on */dev/hd4*

fsck −a /dev/at0 # Automatically fix errors on */dev/at0*

fsck −l /dev/fd0 # List the contents of */dev/fd0*

fsck −c 2 3 /dev/hd3 # Check and list */dev/hd3* i-nodes 2 & 3

DESCRIPTION

Fsck performs consistency checks on the file systems which reside on the specified devices. *Fsck1* is an alternate version for use on obsolete V1 file systems. When either the **−a** or **−r** flags are given, the file system will be repaired if errors are found. Before running *fsck* on a mounted file system, it must first be unmounted. Trying to repair a mounted file system is dangerous and should not be attempted.

To repair the root file system (which cannot be unmounted), first type CTRL-F9 at the console to kill any and all processes. Log back in as **root**, type *sync* to force any buffered changes to disk, run *fsck* on the root file system and immediately reboot the computer by typing *reboot*.

It is necessary to kill all processes before repairing the root file system to prevent them from modifying any disk blocks while *fsck* is running. This is only necessary for the root file system, any other file system can simply be unmounted before it is checked.

SEE ALSO

mkfs(1), **mount**(1).

NAME

gather − gather up the files in a directory for transmission

SYNOPSIS

gather [**−s**] *source_dir [***−d**] dest_dir [**−b**] *bytes* [**−f**] *file*

OPTIONS

−b Desired number of bytes per output file

−d Destination directory

−f Base name of output files

−s Source directory

EXAMPLES

gather # Collect files in current dir into 60K archives

gather −d dir # Put the archives in *dir*

gather −b 90000 # Try to produce 90K archives

gather −s .. −d targ −b 5000
Try to produce 5K archives

DESCRIPTION

It is often useful to collect all the files in a directory into one or more archives for transmission by mail. This program collects all the files in the source directory (default: current directory) and puts them into a shar archive. The shar archive is then compressed and uuencoded. An attempt is made to have the final *.uue* file be about the given size (default: 60K), but since *gather* cannot really predict how much *shar* will add to the file, how much *compress* will reduce the file, and how much *uue* will add again, the sizes can fluctuate. If the **−f** *file* flag is given, the archives will be given the names *file_00.uue*, *file_01.uue* etc. If **−f** is not given, the name of the source directory is used as the base name. Since 7 characters of suffix are appended, the base name should not exceed 7 characters.

NAME

grep − search a file for lines containing a given pattern

SYNOPSIS

grep [−**elnsv**] *pattern* [*file*] ...

OPTIONS

−e −**e** *pattern* is the same as *pattern*

−c Print a count of lines matched

−i Ignore case

−l Print file names, no lines

−n Print line numbers

−s Status only, no printed output

−v Select lines that do not match

EXAMPLES

grep mouse file # Find lines in *file* containing *mouse*

grep [0−9] file # Print lines containing a digit

DESCRIPTION

Grep searches one or more files (by default, *stdin*) and selects out all the lines that match the pattern. All the regular expressions accepted by *ed* and *mined* are allowed. In addition, + can be used instead of ∗ to mean 1 or more occurrences, ? can be used to mean 0 or 1 occurrences, and | can be used between two regular expressions to mean either one of them. Parentheses can be used for grouping. If a match is found, exit status 0 is returned. If no match is found, exit status 1 is returned. If an error is detected, exit status 2 is returned.

SEE ALSO

cgrep(1), **fgrep**(1), **sed**(1), **awk**(9).

NAME

head − print the first few lines of a file

SYNOPSIS

head [−*n*] [*file*] ...

OPTIONS

−*n* How many lines to print

EXAMPLES

head −6 # Print first 6 lines of *stdin*

head −1 file1 file2 # Print first line of two files

DESCRIPTION

The first few lines of one or more files are printed. The default count is 10 lines. The default file is *stdin*.

SEE ALSO

tail(1).

NAME

host − look up host names using domain server

SYNOPSIS

host [-l] [-v] [-w] [-r] [-d] [-t querytype] [-a] host [server]

DESCRIPTION

Host looks for information about Internet hosts. It gets this information from a set of interconnected servers that are spread across the country. By default, it simply converts between host names and Internet addresses. However with the -t or -a options, it can be used to find all of the information about this host that is maintained by the domain server.

The arguments can be either host names or host numbers. The program first attempts to interpret them as host numbers. If this fails, it will treat them as host names. A host number consists of first decimal numbers separated by dots, e.g. 128.6.4.194 A host name consists of names separated by dots, e.g. topaz.rutgers.edu. Unless the name ends in a dot, the local domain is automatically tacked on the end. Thus a Rutgers user can say "host topaz", and it will actually look up "topaz.rutgers.edu". If this fails, the name is tried unchanged (in this case, "topaz"). This same convention is used for mail and other network utilities. The actual suffix to tack on the end is obtained by looking at the results of a "hostname" call, and using everything starting at the first dot. (See below for a description of how to customize the host name lookup.)

The first argument is the host name you want to look up. If this is a number, an "inverse query" is done, i.e. the domain system looks in a separate set of databases used to convert numbers to names.

The second argument is optional. It allows you to specify a particular server to query. If you don’t specify this argument, the default server (normally the local machine) is used.

If a name is specified, you may see output of three different kinds. Here is an example that shows all of them:

```
% host sun4
sun4.rutgers.edu is a nickname for ATHOS.RUTGERS.EDU
ATHOS.RUTGERS.EDU has address 128.6.5.46
ATHOS.RUTGERS.EDU has address 128.6.4.4
ATHOS.RUTGERS.EDU mail is handled by ARAMIS.RUTGERS.EDU
```

The user has typed the command "host sun4". The first line indicates that the name "sun4.rutgers.edu" is actually a nickname. The official host name is "ATHOS.RUTGERS.EDU’. The next two lines show the address. If a system has more than one network interface, there will be a separate address for each. The last line indicates that ATHOS.RUTGERS.EDU does not receive its own mail. Mail for it is taken by ARAMIS.RUTGERS.EDU. There may be more than one such line, since some systems have more than one other system that will handle mail for them. Technically, every system that can receive mail is supposed to have an entry of this kind. If the system receives its own mail, there should be an entry the mentions the system itself, for example "XXX mail is handled by XXX". However many systems that receive their own mail do not bother to mention that fact. If a system has a "mail is handled by" entry, but no address, this indicates that it is not really part of the Internet, but a system that is on the network will forward mail to it. Systems on Usenet, Bitnet, and a number of other networks have entries of this kind.

There are a number of options that can be used before the host name. Most of these options are meaningful only to the staff who have to maintain the domain database.

The option -w causes host to wait forever for a response. Normally it will time out after around a minute.

The option -v causes printout to be in a "verbose" format. This is the official domain master file format, which is documented in the man page for "named". Without this option, output still follows this format in general terms, but some attempt is made to make it more intelligible to normal users. Without -v, "a", "mx", and "cname" records are written out as "has address", "mail is handled by", and "is a nickname for", and TTL and class fields are not shown.

The option -r causes recursion to be turned off in the request. This means that the name server will return only data it has in its own database. It will not ask other servers for more information.

The option -d turns on debugging. Network transactions are shown in detail.

The option -t allows you to specify a particular type of information to be looked up. The arguments are defined in the man page for "named". Currently supported types are a, ns, md, mf, cname, soa, mb, mg, mr, null, wks, ptr, hinfo, minfo, mx, uinfo, uid, gid, unspec, and the wildcard, which may be written as either "any" or "*". Types must be given in lower case. Note that the default is to look first for "a", and then "mx", except that if the verbose option is turned on, the default is only "a".

The option -a (for "all") is equivalent to "-v -t any".

The option -l causes a listing of a complete domain. E.g.

host -l rutgers.edu

will give a listing of all hosts in the rutgers.edu domain. The -t option is used to filter what information is presented, as you would expect. The default is address information, which also include PTR and NS records. The command

host -l -v -t any rutgers.edu

will give a complete download of the zone data for rutgers.edu, in the official master file format. (However the SOA record is listed twice, for arcane reasons.) NOTE: -l is implemented by doing a complete zone transfer and then filtering out the information the you have asked for. This command should be used only if it is absolutely necessary.

CUSTOMIZING HOST NAME LOOKUP

In general, if the name supplied by the user does not have any dots in it, a default domain is appended to the end. This domain can be defined in /etc/resolv.conf, but is normally derived by taking the local hostname after its first dot. The user can override this, and specify a different default domain, using the environment variable *LOCALDOMAIN*. In addition, the user can supply his own abbreviations for host names. They should be in a file consisting of one line per abbreviation. Each line contains an abbreviation, a space, and then the full host name. This file must be pointed to by an environment variable *HOSTALIASES*, which is the name of the file.

See Also

named (8)

BUGS

Unexpected effects can happen when you type a name that is not part of the local domain. Please always keep in mind the fact that the local domain name is tacked onto the end of every name, unless it ends in a dot. Only if this fails is the name used unchanged.

The -l option only tries the first name server listed for the domain that you have requested. If this server is dead, you may need to specify a server manually. E.g. to get a listing of foo.edu, you could try "host -t ns foo.edu" to get a list of all the name servers for foo.edu, and then try "host -l foo.edu xxx" for all xxx on the list of name servers, until you find one that works.

NAME

hostaddr − show ethernet address, IP address or hostname

SYNOPSIS

hostaddr [**−eia**] [**−E** *eth-device*] [**−I** *ip-device*]

DESCRIPTION

Without any of the **−eia** options, **hostaddr** shows the ethernet address, IP address and hostname of the local host on one line in the given order. With options only the wanted fields are shown, still in the same order, not in option order.

OPTIONS

−e Show the ethernet address.

−i Show the IP address. This will not work if no IP address has been found by RARP or by setting it with **ifconfig**(8).

−a Show the fully qualified hostname. The IP address is shown again if it can't be translated to a host name. (This usually indicates that the DNS reverse address translation tables are incomplete.)

SEE ALSO

ifconfig(8), **rarpd**(8), **nonamed**(8), **set_net_default**(8), **boot**(8).

DIAGNOSTICS

"Timeout"

Hostaddr timed out trying to get the IP address. This means that the network connection is out of order or that the setup of the machine is bad.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

ic − integer calculator

SYNOPSIS

ic [*expression*]

EXAMPLES

ic	# Start the calculator
ic 250 300+	# Start calculator with 550 on the stack

DESCRIPTION

Ic is a simple RPN (Reverse Polish Notation) calculator, used for small calculations and base conversions. All calculations are done using 32 bit integers. The standard input is usually a keyboard and the standard output requires a device with a 'termcap' entry. The program starts by interpreting any <args> as commands, where the separation between arguments is considered to be the same as the ENTER key. For example,

"ic 692 784+

After reading the arguments input is from the keyboard.

Stack Operations

The operation of this program is similar to an RPN calculator. A six level stack is used. The ENTER key pushes the stack up one level. For example, '12+5' is entered as '12 ENTER 5 +".

The top two entries on the stack are exchanged by the *x* command, and the stack is rolled down one (popped) by the *p* key. The top of the stack may be cleared by pressing the back-space key. The whole stack and the registers are initialized by a *z*.

Numeric Entry

The input and output bases are initially decimal, but they may be changed using the *i* and *o* commands. The *i* command changes both bases, but the *o* command changes just the output base. These commands take a one character argument of *h*, *d*, *o* or *b* to change to Hexadecimal, Decimal, Octal or Binary. While the input base is hexadecimal the letters *a* through *f* are used to represent the decimal values 10 through 15.

When the input base is decimal: multiply, divide and remainder are signed, otherwise they are performed unsigned.

The output base may also be changed to ASCII (*a*), this causes the least significant 7 bits of a value to be displayed as a character. To input an ASCII value the translate (*t*) command may be used, it accepts one character as its argument.

Calculations

The arithmetic operations supported are: Negate ('.'), Add ('+'), Subtract ('−'), Multiply ('*'), Divide ('/'), and Remainder ('%'). The logical (Boolean) operations available are: NOT ('~'), AND ('&'), OR ('|'), and EXCLUSIVE-OR ('^').

After one of these operations the last top of stack value is saved. It may be restored by pressing *l* (L).

Saving Results

Ten temporary registers are available. The Store (s) command followed by a digit ('0'..'9') will copy the top of the stack to the specified register. The Recall (*r*) command pushes the contents of a register onto the top of the stack.

If the Store command is followed by a '+' preceding the digit, then the top of the stack will be added to the specified 'accumulator' register.

Values may also be written to a file. The *w* command writes the top of the stack, using the current output base, to a file called 'pad' in the current directory. If the user does not have write access to the current directory then the file */tmp/pad_$USER* is used as the scratch pad. The scratch pad file is erased on the first use of the *w* command within each new invocation of 'ic'.

Miscellaneous

The Quit (*q*) key causes an immediate exit. The *m* command temporarily leaves *ic* by invoking the shell as a sub-process. For help while using *ic*, hit the *h* key. If an erroneous key is pressed the bell will sound.

Command Summary

Note that many commands have an alternative key-code available on the extended AT keyboard. This aids entry by including most commands on the right side of the keyboard.

ENTER	Enter (push up)
BS (DEL)	Clear top of stack
h	Help
i	Input base (h, d, o, b)
l (PGDN)	Last top of stack
m	MINIX shell
o	Output base (h, d, o, b, a)
p (DOWN)	Pop stack (roll down)
q (END)	Quit
r (LEFT)	Recall (0-9)
s (RIGHT)	Store [+] (0-9)
t	Translate (char)
w (PGUP)	Write top of stack to scratch pad
x (UP)	Exchange top of stack
z (HOME)	Zero all state
.	Change sign
+ (+)	Add
- (-)	Subtract
*	Multiply
/	Divide
% (sh/5)	Remainder
(tilde)	Not
&	And
|	Or
^	Exclusive-or

Author

Ic was written by Terrence W. Holm.

NAME

id − print the uid and gid

SYNOPSIS

id

EXAMPLES

id # Print the uid and gid

DESCRIPTION

Id prints the current uid and gid, both numerically and symbolically. If the effective uid and gid are different from the real ones, all of them are printed.

Under Minix-vmd the supplementary group IDs are also printed.

SEE ALSO

getuid(2), **getgid**(2), **getgroups**(2).

NAME

ifdef − remove #ifdefs from a file

SYNOPSIS

ifdef [**−t**] [**−d***symbol*] [**−D***symbol*] [**−U***symbol*] [**−I***symbol*] [file]

OPTIONS

−D Define symbol permanently

−I Ignore symbol

−U Undefine symbol permanently

−d Define symbol. It may be #undef'ed later

−t Produce a table of the symbols on *stdout*

EXAMPLES

ifdef −DUNIX file.c >newfile.c
Define *UNIX*

ifdef −D_MINIX −UDOS <x.c >y.c
Define

DESCRIPTION

Ifdef allows conditional code [#ifdef ... #endif] to be selectively removed from C files, but at the same time leaving all other C preprocessor commands intact such as #define, #include etc. Input to *ifdef* is either the file named as the last argument, or *stdin* if no file is named. Output goes to *stdout*.

Symbols may be defined with the **−d** or **−D** flags just like *cpp*, except that the latter option ignores subsequent *#undefs*. It is not permitted to give values to symbols. Similarly, **−U** undefines a symbol and ignores subsequent *#define*s. Symbols defined with **−I** are ignored; any *#ifdef* using an ignored symbol will be left intact.

NAME

install − install files

SYNOPSIS

install [**−lcsz***N*] [**−o** *owner*] [**−g** *group*] [**−m** *mode*] [**−S** *stack*] [*file1*] *file2*
install [**−lcsz***N*] [**−o** *owner*] [**−g** *group*] [**−m** *mode*] [**−S** *stack*] *file ... dir*
install −d [**−o** *owner*] [**−g** *group*] [**−m** *mode*] *directory*

DESCRIPTION

Install puts executables, manual pages, and library files in their proper place in the bin, man, and lib directories. The first two forms of the command are like **cp**(1) copying either one file to another or copying several files to a directory. The "**−d**" form is like **mkdir**(1) with the **−p** flag. *File1* may be omitted if neither **−l** nor **−c** is given to change the attributes of *file2*.

Attributes are always copied from the source file, use the options to change. Note that the source file's attributes are changed with the destination file if they are linked. So copy the file if you change it in a way that makes it read-only. You would otherwise not be able to compile a command again.

OPTIONS

−l Link the destination to the source file instead of copying it. This is done to either save space on a file system with both the source and the bin directories on it, or to install synonyms to a command.

−c Copy the source file to its proper place. This option is the default if **−l** is not given. With **−l**, the file is copied if the link fails.

−s Strip the destination file of its symbol table, *if* it is an executable, and *if* it is actually copied. It has no effect on a link or a non-executable.

−z Compress the executable using **compress**(1) and prepend a header line that calls **zexec**(1) to decompress and execute the binary. This will on average save 40% disk space at the expense of a slower startup time. Like **−s** the file must be actually copied for the flag to have effect.

−N Use **gzip** *−N* to compress the binary. You may see up to 60% space savings, but it will take much longer. *N* is a digit from 1 to 9 telling the compression effort, see **gzip**(1).

−d Make a directory, usually to install files in a separate directory in a library. Intermediate directories in the path are created with the same attributes as the final directory. Only the attributes of the final directory are set if the directory exists.

−o *owner*
Set the owner of the target. This only works if the invoker is the super-user, or if **install** is run setuid root and the invoker is a member of group zero. If **−o** is omitted then the ownership is copied from the source file, or set to the id of the invoker if a directory is made.

−g *group*
Like **−o**, but for the group ownership of the target.

−m *mode*
Mode is an octal number that specifies the mode the target should get. The default is the source file's mode with a **chmod a+rX** applied to it, or 755 for a new directory. Implies **−o 0**, or **−g 0** if a file is to be set-uid or set-gid and the invoker has permission to change ownership. This trick allows a group 0 member to install third party software, even though it expects to be installed by root.

−S *stack*
Sets the maximum amount of heap + stack that an executable may have when running. The argument is a C-style decimal, octal or hexadecimal number, optionally followed by the multipliers **m**, **k**, **w**, and **b** for mega (1024*1024), kilo (1024), "word" (2 or 4), and byte (1). Uppercase **M** is also accepted for those who know what S.I. means. The compilers use **−S 32kw** by default, that translates to 64kb for an 8086, and 128kb for other architectures. This option is ignored on a non-executable.

SEE ALSO

ln(1), **cp**(1), **strip**(1), **compress**(1), **gzip**(1), **zexec**(1), **chown**(8), **chgrp**(8), **chmod**(1), **chmem**(1), **mkdir**(1).

BUGS

Uppercase **K**, **W**, and **B** are also accepted for those who don't know what S.I. means.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

isodir − list ISO9660 or High Sierra directories

SYNOPSIS

isodir −[**lr**] *input_file* [*dir*]

DESCRIPTION

Isodir reads directories on a file system in ISO9660 or High Sierra Group format (usually residing on cdrom) and lists their contents on standard output. Directory names should contain slashes to separate components. The names **isodir**, **isoread**, and **isoinfo** are all links to the same program. The program sees which function to perform by looking how it was called.

−l Lists all info on files and directories (size, date, time)

−r Recursively descend and print subdirectories

BUGS

Only Interchange level-1 is supported. The Red Rock extensions and Interchange level-2 are not implemented.

SEE ALSO

isoread(1), **isoinfo**(1).

AUTHOR

Michel R. Prevenier (mrpreve@cs.vu.nl)

NAME

isoinfo − list an ISO9660 or High Sierra volume descriptor

SYNOPSIS

isoinfo [*input_file*]

DESCRIPTION

Isoinfo reads the volume descriptor from an ISO9660 or High Sierra Group file system (usually residing on cdrom) and lists its contents on standard output. **isodir**, **isoread**, and **isoinfo** are all links to the same program. The program sees which function to perform by looking how it was called.

BUGS

Only Interchange level-1 is supported. The Red Rock extensions and Interchange level-2 are not implemented.

SEE ALSO

isodir(1), **isoread**(1).

AUTHOR

Michel R. Prevenier (mrpreve@cs.vu.nl)

NAME

isoread − read a file in ISO9660 or High Sierra format

SYNOPSIS

isoread [*input_file*] *file*

DESCRIPTION

Isoread reads a file in ISO9660 or High Sierra Group format (usually residing on cdrom) and lists its contents on standard output. The file path should contain slashes to separate components. The names **isodir**, **isoread**, and **isoinfo** are all links to the same program. The program sees which function to perform by looking how it was called.

BUGS

Only Interchange level-1 is supported. The Red Rock extensions and Interchange level-2 are not implemented.

SEE ALSO

isodir(1), **isoinfo**(1).

AUTHOR

Michel R. Prevenier (mrpreve@cs.vu.nl)

NAME

join − relational database operator

SYNOPSIS

join [−**a***n*] [−**e** *s*] [−**o** *list*] [−**t***c*] file1 file2

DESCRIPTION

Join forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is '−', the standard input is used.

File1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

Fields are normally separated by blank, tab or newline. In this case, multiple separators count as one, and leading separators are discarded.

These options are recognized:

−**a***n* In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

−**e** *s* Replace empty output fields by string *s*.

−**o** *list* Each output line comprises the fields specified in *list*, each element of which has the form *n*.*m*, where *n* is a file number and *m* is a field number.

−**t***c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

SEE ALSO

sort(1), **comm**(1), **awk**(1).

BUGS

With default field separation, the collating sequence is that of **sort −b**; with **−t**, the sequence is that of a plain sort.

The conventions of **join**, **sort**, **comm**, **uniq**, **look** and **awk**(1) are wildly incongruous.

NAME

kill − send a signal to a process

SYNOPSIS

kill [−*n*] *process*

OPTIONS

−*n* Signal number to send

−*NAME*
Named signal to send

EXAMPLES

kill 35	# Send signal 15 to process 35
kill −9 40	# Send signal 9 to process 40
kill −2 0	# Send signal 2 to whole terminal process group
kill −HUP -123	# Send a hangup to process group 123

DESCRIPTION

A signal is sent to a given process. By default signal 15 (SIGTERM) is sent. Process 0 means all the processes in the sender's process group. A process group can be signalled by the negative value of the process group ID. Signals may be numerical, or the name of the signal without **SIG**.

SEE ALSO

kill(2), **sigaction**(2).

NAME

last − display recent on-line session records

SYNOPSIS

last [**−f** *file*] [**−r**] [*−n*] [*name*] [*tty*] ...

OPTIONS

−f Use *file* instead of /usr/adm/wtmp

−r Search backwards only to last reboot

−n Print a maximum of *n* lines

EXAMPLES

last reboot # When was the system last rebooted?

last ast # When was the last login for ast?

last −10 tty00 tty01 # Display last 10 logins on tty00 or tty01

DESCRIPTION

Last Searches backward through the login administration file (default is */usr/adm/wtmp*), printing information about previous logins and reboots. During a long search, the SIGQUIT signal (CTRL-\) causes *last* to display how far back it has gone; it then continues.

SEE ALSO

login(1).

NAME

leave − warn when it is time to go home

SYNOPSIS

leave [[+] *hh*[**:**]*mm*]

EXAMPLES

leave 1500	# Issue a warning at 2:55 p.m.
leave 10:00	# Issue a warning at 9:55 a.m.
leave + 30	# Issue a warning in 25 minutes

DESCRIPTION

Leave sets an alarm clock to a specified time and issues a warning 5 minutes before, 1 minute before, and at the time to leave. It then keeps issuing warnings every minute for 10 minutes, then quits. If no time is provided, the program prompts for one.

NAME

ln, clone − create a link to a file

SYNOPSIS

ln [−ifmrRvx] *file* [*name*]
ln [−ifrRvx] *file ... dir*
clone [−ifmvx] *file* [*name*]

OPTIONS

−i Ask if ok to remove a file

−f Remove existing links

−m Merge trees, disable the into-a-directory trick

−rR Recursively link a directory tree

−v Display what ln is doing

−x Do not cross device boundaries

EXAMPLES

ln file newname # Make *newname* a synonym for *file*

ln /usr/games/chess # Create a link called *chess*

DESCRIPTION

A directory entry is created for *name* . The entry points to *file* . Henceforth, *name* and *file* can be used interchangeably. If *name* is not supplied, the last component of *file* is used as the link name. If more than one *file* is supplied or the *name* refers to an existing directory, links will be created in that directory. An existing *name* will not be removed unless the **−i** or **−f** flag is specified.

Clone is a convenient synonym for **ln −fmr** to create a so-called "link farm", a directory full of links to the original tree.

SEE ALSO

cp(1), **link**(2), **unlink**(2).

NAME

loadfont − load a font into the video card

SYNOPSIS

loadfont *fontfile*

EXAMPLES

loadfont iso1.fnt # Loads the ISO 8859-1 (Latin-1) font

DESCRIPTION

Loadfont loads a custom font into the video card (EGA or VGA). The font character size has to be 8x16 pixels and the font file must contain 256 characters for a total size of 4 kilobytes.

Loadfont together with *loadkeys* allow the console and keyboard to be customized to national conventions.

If it exists, the file */etc/font* is loaded as a custom font by */etc/rc* at boot time.

SEE ALSO

console(4).

NAME

loadkeys − load a keyboard map into the keyboard driver

SYNOPSIS

loadkeys *mapfile*

EXAMPLES

loadkeys spanish.map # Load a map for a Spanish keyboard

DESCRIPTION

Loadkeys changes the key number to character mapping. This is necessary for national keyboards that have different symbols on the keys that the standard U.S. English keyboard. The file */etc/keymap* is the first thing loaded by */etc/rc* at boot time if it exists.

SEE ALSO

console(4).

NAME

login − log into the computer

SYNOPSIS

login [*user*]

EXAMPLES

login ast # Login as ast

DESCRIPTION

Login allows a logged in user to login as someone else without first logging out. If a password is needed, *login* will prompt for it.

SEE ALSO

su(1), **init**(8), **getty**(8), **rlogin**(1).

NAME

look − find lines in a sorted list

SYNOPSIS

look [**−df**] *string* [*file*]

DESCRIPTION

Look consults a sorted file and prints all lines that begin with *string*. It uses binary search. The options **−d** and **−f** affect comparisons as in **sort**(1). If no file is specified, **/usr/lib/dict/words** is assumed with collating sequence **−df**.

OPTIONS

−d Dictionary order: compare letters, digits and whitespace.

−f Fold. Upper case letters compare equal to lower case.

FILES

/usr/lib/dict/words Sorted list of English words.

SEE ALSO

sort(1), **spell**(1).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

lp, lpd − copy a file to the line printer

SYNOPSIS

lp [*file* ...]

DESCRIPTION

Each file argument to **lp** is send to the line printer to be printed. Standard input is read and printed if there are no arguments. **Lp** executes **/usr/lib/lpd** with each file as input. **Lpd** puts the file in **/usr/spool/lpd** and starts printing the jobs on **/dev/lp** unless another **lpd** is already running. If **lpd** finds any character in the input that it doesn't know how to handle then it will print the rest of the file without any special treatment. This also means that no formfeed is sent after the file has been printed to force out the page. **Lpd** simply assumes that you know what you are doing. (dumb, eh?)

Note: Don't do anything with a file until it is printed, **lpd** only makes a copy of a file in the spool directory when it is not world readable. If it can be read then it is printed directly.

FILES

/usr/spool/lpd/job*XXXXX*
: Information about a job.

/usr/spool/lpd/tmp*XXXXX*
: Associated file to be printed.

/etc/termcap
: The 'lp' entry describes the printer by the "li#" and "co#" fields. By default 66 lines (li#66), and 80 columns (co#80).

SEE ALSO

lp(4), **termcap**(5), **termcap**(7).

BUGS

Not spooling a world readable file may not be such a smart idea.

A formfeed should be printed and the printer reset after a job full of escape codes, but this may cost paper.

No banner page.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

ls − list the contents of a directory

SYNOPSIS

ls [**−acdfgilqrstu1ACFLMRTX**] [*name*...]

DESCRIPTION

For each file argument, list it. For each directory argument, list its contents. The current working directory is listed when no files are named. Information is printed multicolumn on terminals, single column if the output is redirected. The options control what information is shown and how.

Ls has two sources other then the commands line to draw options from, one is the environment variable **LSOPTS** and is used only when the output of **ls** is displayed on a terminal. The other is the name of **ls** itself. If **ls** is linked to another name, then all the characters after the l are used as flags too, except that f, r, t and x are translated to F, R, T and X. Useful links are **ll**, **lf**, **lm** and **lx**.

Files whose names start with a dot are by default not listed.

Note that standard Minix doesn't have symbolic links or sockets and **−u** and **−c** are no-ops on a V1 file system, since only modified times are stored in V1 inodes.

OPTIONS

−a All entries are listed, even **.** and **..**

−c Use inode changed time for sorting, listing or searching.

−d Do not list contents of directories, but list the directory itself.

−f Do not sort (should also be: treat a file as a directory, but that can't be implemented portably).

−g Suppress the owner name on a long listing (implies **−l**).

−i I-node number printed in first column.

−l Long listing: mode, links, owner, group, size and time. (**ls −lC** uses columns in a wide enough window!)

−n Print numerical user and group id's.

−q Print nongraphic characters as '**?**' (default on terminals).

−r Reverse the sort order.

−s Give size in kilobytes.

−t Sort by time (modified time default), latest first.

−u Use last accessed time for sorting, listing or searching.

−1 Print in one column.

−A List all entries, but not **.** and **..** (This is the default for privileged users.)

−C Print multicolumn (default on terminals).

−F Mark directories with a '/', executables with a '*****', UNIX domain sockets with a '**=**' and symbolic links with a '**@**' behind the name.

−L Print the file referenced by a symbolic link instead of the link.

−M List mode before name (implies **−C**).

−R List directory trees recursively.

−T Group files by type, i.e. regular files together, directories together, etc.

−X Print crunched mode and size before name (implies **−C**). Only the rwx permissions that its caller has on the file, but they are in upper case if the caller owns the file and has given the permission to the callers group or other users. The size is listed in bytes (<= 5K), or rounded up kilo, mega or gigabytes.

SEE ALSO

du(1), **stat**(1), **stat**(2).

BUGS

Having to type **ls −C** when viewing files through **more**(1).

Is only portable to systems with the same st_mode (see **stat**(2)).

The **LSOPTS** variable and the **-M**, **-T** and **-X** flags are not found on other **ls** implementations. (They have there own nonstandard flags.)

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

M, U − conveniently mount and unmount

SYNOPSIS

M *device* [**−r**]
U *device*

OPTIONS

−r Mount read-only

EXAMPLES

M root	# Mount the RAM image on /root
M 0	# Mount /dev/fd0 on /fd0
U fd1	# Unmount /dev/fd1 from /fd1

DESCRIPTION

M and *U* allow easy mounting and unmounting of a device by using only an abbreviated device name or keyword. Special keywords are **root**, **tmp**, and **usr** for the three hard disk partitions Minix runs in. Floppy devices are mounted on **/fd0** or **/fd1**. You can use **0** and **1** instead of **fd0** and **fd1**. A device it doesn't know about is mounted on **/mnt**.

SEE ALSO

mount(1), **umount**(1).

NAME

mail − send and receive electronic mail

SYNOPSIS

mail [**−dpqrv**] [**−f** *file*] [*user*]

OPTIONS

−d Force use of the shell variable *MAILER*

−f Use *file* instead of */usr/spool/mail/user* as mailbox

−p Print all mail and then exit

−q Quit program if SIGINT received

−r Reverse print order, i.e., print oldest first

−v Verbose mode

EXAMPLES

mail ast # Send a message to *ast*

mail # Read your mail

DESCRIPTION

Mail is an extremely simple electronic mail program. It can be used to send or receive email on a single MINIX system, in which case it functions as user agent and local delivery agent. If the flag *MAILER* is defined in *mail.c*, it can also call a transport agent to handle remote mail as well. No such agent is supplied with MINIX.

When called by *user* with no arguments, it examines the mailbox */usr/spool/mail/user*, prints one message (depending on the **−r** flag), and waits for one of the following commands:

<newline>	Go to the next message
−	Print the previous message
!command	Fork off a shell and execute *command*
CTRL-D	Update the mailbox and quit (same as q)
d	Delete the current message and go to the next one
q	Update the mailbox and quit (same as CTRL-D)
p	Print the current message again
s [*file*]	Save message in the named file
x	Exit without updating the mailbox

To send mail, the program is called with the name of the recipient as an argument. The mail is sent, along with a postmark line containing the date. For local delivery, a file named after the recipient in the directory */usr/spool/mail* must be writable.

If the directory */usr/spool/mail* does not exist then the mail is dumped on the console, so that system programs have a way to notify a user on a system that does not have a mail spool.

NAME

make − a program for maintaining large programs

SYNOPSIS

make [**−f** *file*] [**−iknpqrst**] [*option*] ... [*target*]

OPTIONS

−f Use *file* as the makefile

−i Ignore status returned by commands

−k On error, skip to next command

−n Report, but do not execute

−p Print macros and targets

−q Question up-to-dateness of target

−r Rule inhibit; do not use default rules

−s Silent mode

−t Touch files instead of making them

EXAMPLES

make kernel # Make *kernel* up to date

make −n −f mfile # Tell what needs to be done

DESCRIPTION

Make is a program that is normally used for developing large programs consisting of multiple files. It keeps track of which object files depend on which source and header files. When called, it does the minimum amount of recompilation to bring the target file up to date.

The file dependencies are expected in *makefile* or *Makefile* , unless another file is specified with **−f**. *Make* has some default rules built in, for example, it knows how to make *.s* files from *.c* files. Here is a sample *makefile* .

```
d=/user/ast                         # d is a macro
program:  head.s tail.s             # program depends on these
          cc −o program head.s tail.s   # tells how to make program
          echo Program done.        # announce completion
head.s:   $d/def.h head.c           # head.s depends on these
tail.s:   $d/var.h tail.c           # tail.s depends on these
```

A complete description of *make* would require too much space here. Many books on UNIX discuss *make* . Study the numerous *Makefiles* in the MINIX source tree for examples.

SEE ALSO

cc(1).

NAME

makewhatis − build the whatis(5) database

SYNOPSIS

makewhatis *directory*

DESCRIPTION

Makewhatis makes the **whatis**(5) database in the given manual page directory. This database is used by **man**(1) to map titles to manual page names and by **whatis**(1) to give one line descriptions. See **whatis**(5) for a desciption of what a whatis database should look like and the restrictions that are placed on the NAME sections so that **makewhatis** can make whatis lines out of the manual pages.

SEE ALSO

whatis(5).

BUGS

Removing only font and size changes from the NAME section is often not enough.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

man − display online manual pages

SYNOPSIS

man [**−antkf**] [**−M** *path*] [**−s** *section*] *title* ...

DESCRIPTION

Man displays the online manual pages for the specified titles in the specified sections. The sections are as follows:

1 User Commands
Generic commands such as **ls**, **cp**, **grep**.

2 System Calls
Low level routines that directly interface with the kernel.

3 Library Routines
Higher level C language subroutines.

4 Device Files
Describes devices in **/dev**.

5 File Formats
Formats of files handled by various utilities and subroutines.

6 Games
It's not UNIX without an adventure game.

7 Miscellaneous
Macro packages, miscellaneous tidbits.

8 System Utilities
Commands for the System Administrator.

9 Documents
Larger manuals explaining some commands in more detail.

(If you are new to Minix then try **man hier**, it will show you around the file system and give you many pointers to other manual pages.)

By default, **man** will try the following files in a manual page directory for the command **man −s 1 ls**:

cat1/ls.1
cat1/ls.1.Z
man1/ls.1
man1/ls.1.Z

Files in the man[1−8] directories are formatted with **nroff −man**. Those in man9 are formatted with **nroff −mnx**. Files in the cat? directories are preformatted. Files with names ending in **.Z** are decompressed first with **zcat** (see **compress**(1)). The end result is presented to the user using a pager if displaying on the screen.

For each manual page directory in its search path, **man** will first try all the subdirectories of the manual page directory for the files above, and then the directory itself. The directory **/usr/man** contains the standard manual pages, with manual pages for optional packages installed in a subdirectory of /usr/man, with the same structure as /usr/man. The directory **/usr/local/man** contains manual pages for locally added software. By default /usr/local/man is searched first, then /usr/man.

A title is not simply used as a filename, because several titles may refer to the same manual page. Each manual page directory contains a database of titles in the **whatis**(5) file that is created by **makewhatis**(8) from the NAME sections of all the manual pages. A title is searched in this database and the first title on a whatis line is used as a filename.

OPTIONS

The options may be interspersed with the titles to search, and take effect for the titles after them.

−a Show all the manual pages or one line descriptions with the given title in all the specified sections in all the manual directories in the search path. Normally only the first page found is shown.

−n Use **nroff −man** to format manual pages (default).

−t Use **troff −man** to format manual pages.

−f Use **whatis**(1) to show a one line description of the title from the **whatis**(5) file.

−k Use **apropos**(1) to show all the one line descriptions of the title anywhere in the **whatis**(5) files (implies **−a**).

−M *path*
Use *path* as the search path for manual directories.

−s *section*
Section is the section number the page is to be found in, or a comma separated list of sections to use. Normally all sections are searched. The search is always in numerical order no matter what your section list looks like. A single digit is treated as a section number without the **−s** for compatibility with BSD-style **man** commands.

ENVIRONMENT

MANPATH This is a colon separated list of directories to search for manual pages, by default **/usr/local/man:/usr/man**.

PAGER The program to use to display the manual page or one line descriptions on the screen page by page. By default **more**.

FILES

/usr/man/whatis One of the **whatis**(5) databases.

SEE ALSO

nroff(1), **troff**(1), **more**(1), **whatis**(1), **makewhatis**(1), **catman**(1), **whatis**(5), **man**(7).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

mesg − permit or deny messages

SYNOPSIS

mesg [**n**] [**y**]

DESCRIPTION

Mesg with argument **n** forbids messages via **write** and **talk**(1) by revoking non-user write permission on the user's terminal. **Mesg** with argument **y** reinstates permission. All by itself, **mesg** reports the current state without changing it.

FILES

/dev/tty*

SEE ALSO

write(1), **talk**(1).

DIAGNOSTICS

Exit status is 0 if messages are receivable, 1 if not, 2 on error.

NAME

mixer − manipulate mixer settings on a sound card

SYNOPSIS

mixer [−**r**]

DESCRIPTION

Mixer, invoked without arguments, turns the screen into a sound mixer. Levels can be changed with the cursor-left and cursor-right keys. Input and output settings can be toggled with the space bar. For every sound source there are two, or one when mono, sliders. The input controls have only effect when recording with the Dac. These settings can also be used to switch the left and right channels or, when both channels are enabled on both Dac channels, record in mono. To exit the mixer use the 'e' key.

Mixer settings can be stored and restored with the 's' (store) and 'r' keys. When the store function is used **Mixer** will write the settings to a file in the user's home directory called *.mixer*. The restore function reads this file to restore saved settings.

OPTIONS

−r restore settings saved in *.mixer* and exit immediately

AUTHOR

Michel R. Prevenier (mrpreve@cs.vu.nl)

NAME

mkdir − make a directory

SYNOPSIS

mkdir [−p] [**−m** *mode*] *directory* ...

OPTIONS

−m Create directory with mode

−p Create missing intermediate directories

EXAMPLES

mkdir dir # Create *dir* in the current directory

mkdir −p /user/ast/dir # Create the */user/ast* and */user/ast/dir*

DESCRIPTION

The specified directory or directories are created and initialized. If any intermediate directory is missing and **−p** is specified, the missing component will be created and no error displayed if directory already exists. If the **−m** flag is used, this will be equivalent to a chmod on the directory after its creation.

SEE ALSO

chmod(1), **rmdir**(1), **mkdir**(2).

NAME

mkfifo − make a named pipe

SYNOPSIS

mkfifo [−m *mode*] *fifo* ...

OPTIONS

−m Create fifo with specified mode

EXAMPLES

mkfifo pipe # Create *pipe* in the current directory

mkfifo -m a+w systatus
Create the *systatus* writable by all

DESCRIPTION

The specified fifo special files are created. If the **−m** flag is used, this will be equivalent to a chmod on the fifo special file after its creation.

SEE ALSO

chmod(1), **mknod**(2), **mknod**(8).

NAME

mkfs − make a file system

SYNOPSIS

mkfs [**−Ldot**] [**−i** *inodes*] [**−b** *blocks*] *special prototype*

OPTIONS

−L Make a listing on standard output

−d Use mod time of *mkfs* binary for all files

−o Use a drive other than 0 or 1 (safety precaution)

−t Do not test if file system fits on the medium

−1 Make a version 1 file system (for backward compatibility)

−i Number of i-nodes (files)

−b Filesystem size

EXAMPLES

mkfs /dev/fd1 proto # Make a file system on */dev/fd1*

mkfs -b 360 /dev/fd1 # Make empty 360 block file system

mkfs /dev/fd1 360 # Alternate way to specify the size

DESCRIPTION

Mkfs builds a file system and copies specified files to it. The prototype file tells which directories and files to copy to it. If the prototype file cannot be opened, and its name is just a string of digits, an empty file system will be made with the specified number of blocks. A sample prototype file follows. The text following the # sign in the example below is comment. In real prototype files, comments are not allowed.

```
boot                                    # boot block file (ignored)
360 63                                  # blocks and i-nodes
d--755 1 1                              # root directory
  bin    d--755  2 1                    # bin dir: mode (755), uid (2), gid (1)
         sh      ---755 2 1 /user/bin/shell    # shell has mode rwxr-xr-x
         mv      -u-755 2 1 /user/bin/mv       # u = SETUID bit
         login   -ug755 2 1 /user/bin/login    # SETUID and SETGID
  $                                     # end of /bin
  dev    d--755 2 1                     # special files: tty (char), fd0 (block)
         tty     c--777 2 1 4 0         # uid=2, gid=1, major=4, minor=0
         fd0     b--644 2 1 2 0 360     # uid, gid, major, minor, blocks
  $                                     # end of /dev
  user   d--755 12 1                    # user dir: mode (755), uid (12), gid (1)
         ast     d--755 12 1            # /user/ast
         $                              # /user/ast is empty
  $                                     # end of /user
$                                       # end of root directory
```

The first entry on each line (except the first 3 and the $ lines, which terminate directories) is the name the file or directory will get on the new file system. Next comes its mode, with the first character being **−dbc** for regular files, directories, block special files and character special files, respectively. The next two characters are used to specify the SETUID and SETGID bits, as shown above. The last three characters of the mode are the *rwx* protection bits.

Following the mode are the uid and gid. For special files, the major and minor devices are needed. The size in blocks must also be specified for block special files (the MINIX block size is 1K; this can only be changed by changing *BLOCK_SIZE* and then recompiling the operating system).

The maximum size of a file system is 1 Gb for a version 2 file system, and 64 Mb for a version 1 file system. Alas the 8086 *fsck* runs out of memory on a V2 file system larger than 128 Mb, so for the 8086

version of MINIX you have to limit yourself to file systems of that size.

SEE ALSO

mkproto(1), **fsck**(1), **mount**(1).

NAME

mkproto − create a MINIX prototype file

SYNOPSIS

mkproto [**−b** *n*] [**−d** *str*] [**−g** *n*] [**−i** *n*] [**−p** *nnn*] [**−s**] [**−t** *root*] [**−u** *n*] *source_directory* [*prototype_file*]

OPTIONS

−b Number of blocks in the prototype is *n*

−d Indent the prototype file using *str* instead of tab

−g Use *n* as the gid for all files and directories

−i Number of i-nodes in the prototype is *n*

−p Use *nnn* (3 octal digits) as the protection mode

−s Use the same uid, gid and mode as the source files have

−t Use the string *root* as the path prefix for every file

−u Use *n* as the uid for all files and directories

EXAMPLES

mkproto −b360 # Make a 360K prototype of this directory

mkproto −u2 −g1 −p644
Give all files uid 2, gid 1 and mode 644

DESCRIPTION

Mkproto creates an *mkfs* prototype file for the specified source-directory. The prototype file is either written to *stdout* or, if specified, the proto-file.

SEE ALSO

mkfs(1).

NAME

modem − switch the modem and getty state

SYNOPSIS

modem [**−o**] [**−i** *num*] **tty***n*

OPTIONS

−o Turn getty off and set modem to dialout

−i Set line to dialin

EXAMPLES

modem −o tty00 # Set tty00 to dialout

modem −i2 tty00 # Set tty00 to dialin (2 rings)

DESCRIPTION

The *getty* program allows a terminal port to be used for both dialin and dialout. This little program switches the getty state, and also sends some commands to the modem attached to the specified line. If the **−o** flag is presnt, *modem* will put the getty process (if any) connected to the specified line into SUSPEND state, which means that it will not pay attention to that line until it is reset to RESTART state. Also, *modem* will send some (Hayes) commands to the attached modem to disable the auto-nanswer mode. The **−i** flag specifies the number of times the telephone has to ring before the modem may answer the call (to give the operator a chance).

SEE ALSO

term(1), **getty**(8).

NAME

mount − mount a file system

SYNOPSIS

mount *special file* [**−r**]

OPTIONS

−r File system is mounted read-only

EXAMPLES

mount /dev/fd1 /user # Mount diskette 1 on */user*

DESCRIPTION

The file system contained on the special file is mounted on *file*. In the example above, the root directory of the file system in drive 1 can be accessed as **/user** after the mount. When the file system is no longer needed, it must be unmounted before being removed from the drive.

SEE ALSO

df(1), **mkfs**(1), **fsck**(1), **umount**(1), **mount**(2), **fstab**(5).

NAME

mref − make listing and cross reference map of MINIX

SYNOPSIS

mref [**−dlmstx**] [**−p** *n*] [*−n*] *file* ...

OPTIONS

−d Do not produce definition file (global symbol table)

−l Do not produce listing

−m Multiple references on one line are cited only once

−p Set initial page number to *n*

−s Suppress line numbering between procedures

−t Generate output for *troff*

−x Do not produce the cross reference map

−n Number of lines to print per page, default = 50

EXAMPLES

mref *.[*hc***]** # List and cross reference files *.h* and *.c*

mref −60 −t *.c # Produce *troff* input at 60 lines/page

mref −dx −p 100 *.c # Listing only, first page is numbered 100

DESCRIPTION

In default mode, *mref* produces three output files: a numbered listing of the input files (on standard output), a global symbol table (on *symbol.out*), and a cross reference map to the global symbols (on *xref.out*). A global symbol in this context is one present in a #define, PUBLIC, PRIVATE, or SYMBOL statement (the latter being introduced to allow users to explicitly declare certain symbols as global). Any of the three outputs can be suppressed, or alternatively, be made suitable for input to *troff* for typesetting.

NAME

mt − magnetic tape control

SYNOPSIS

mt [**−f** *device*] [*count*]

DESCRIPTION

Mt is a user interface to the magnetic tape commands described in **mtio**(4). It allows one to space a tape forwards or backwards, write end of file markers, etc.

With the **−f** option a tape device can be named, otherwise the environment variable **TAPE** is used if set, otherwise the default device **/dev/nrst4** is assumed. Standard input is used if the tape name is a dash (−). The *count* argument is used to tell how many blocks or files to space or how many file markers to write. It may be a C-style decimal, octal or hexadecimal constant, by default "1".

Command is the action to perform, it may be one of the following, or any unambiguous prefix (like **st** for **status**):

eof, weof Write *count* end-of-file markers.

fsf Forward space *count* file markers.

fsr Forward space *count* records. (The size of a record depends on the tape, and may even be variable, depending on the size of the writes.)

bsf Backwards space *count* files. The count may be zero to backspace to the start of the current file. (A tape device need not support backwards movement, or may be very slow doing it. Rewinding and forward spacing may be better.)

bsr Backwards space *count* records. The tape is positioned after the last block of the previous file if you hit a filemark when spacing backwards. The block count is set to -1 to indicate that the driver has no idea where it is on the previous file.

eom Forward space to the end of media.

rewind Rewind the tape.

offline, rewoffl Rewind and take offline. This may cause some drives to eject the tape.

status Shows the status of the drive, the sense key of the last SCSI error, current file number, current record number, residual count if the last command that encountered end-of-file, and the current block size.

retension Removes tape tension by winding and rewinding the tape completely.

erase Erases the tape completely and rewinds it.

density Sets the density code to read or write the tape to *count*. Density codes supported depend on the drive. This command need not be used if the drive senses the proper density on read and can only write one density.

blksize, blocksize
Sets the block size used to read or write the tape to *count*. This command may be used to select a fixed block size for a variable block size tape. This will speed up I/O for small block sizes. Use a zero *count* to use variable sized blocks again.

ENVIRONMENT

TAPE Tape drive to use if set.

FILES

/dev/nrst4 Default tape device.

SEE ALSO

mtio(4), **st**(4).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

mv − move or rename a file

SYNOPSIS

mv [−ifsmvx] *file1 file2*
mv [−ifsvx] *file ... directory*

OPTIONS

−i Prompt before removing existing files

−f Do not prompt before removing existing files

−s Do not clear any mode bits if owner can't be set

−m Merge directory trees

−v Tell what mv is doing

−x Do not cross device boundaries

EXAMPLES

mv oldname newname # Move *oldname* to *newname*

mv file1 file2 /user/ast # Move two files to */user/ast*

DESCRIPTION

Mv moves one or more files from one place in the file system to another. If the old path and new path are on the same device, it is done by renaming otherwise by copying. If you're not superuser, the copy will destroy the file's owner, group and setuid/setgid bits. *mv* will prompt before removing non-writable files if **stdin** is a terminal or the **−i** flag was specified.

SEE ALSO

cp(1), **rename**(2).

NAME

nm − print name list

SYNOPSIS

nm [**−dgnopru**] [*file*] ...

OPTIONS

−d Print the offsets in decimal instead of in hex

−g Print only external symbols

−n Sort numerically rather than alphabetically

−o Prepend file name to each line rather than only once

−p Do not sort, print in symbol-table order

−r Sort in reverse order

−u Print only undefined symbols

EXAMPLES

nm −n a.out # Print all symbols in numerical order

nm −dg a.out # Print globals alphabetically in decimal

DESCRIPTION

Nm prints the symbol table of executable files when it is available. If no file is given, the symbols in *a.out* are used. The format of the table is somewhat compatible with the one produced by *asld* when used with the **−s** option. The symbol table can be added with *ast*. Assembly language files do not have symbol tables.

SEE ALSO

anm(1), **asize**(1), **ar**(1), **size**(1).

NAME

od − octal dump

SYNOPSIS

od [**−bcdhox**] [*file*] [[**+**] *offset* [**.**][**b**]]

OPTIONS

−b Dump bytes in octal

−c Dump bytes as ASCII characters

−d Dump words in decimal

−h Print addresses in hex (default is octal)

−o Dump words in octal (default)

−v Verbose (list duplicate lines)

−x Dump words in hex

EXAMPLES

od −ox file # Dump *file* in octal and hex

od −d file +1000 # Dump *file* starting at byte 01000

od −c file +10.b # Dump *file* starting at block 10

DESCRIPTION

Od dumps a file in one or more formats. If *file* is missing, *stdin* is dumped. The *offset* argument tells *od* to skip a certain number of bytes or blocks before starting. The offset is in octal bytes, unless it is followed by a '.' for decimal or **b** for blocks or both.

NAME

passwd, chfn, chsh − change a login password, full name or shell

SYNOPSIS

passwd [*user*]
chfn [*user*] *fullname*
chfn [*user*] *shell*

EXAMPLES

passwd	# Change current user's password
passwd ast	# Change ast's password (super−user only)
chsh /usr/bin/mail	# For those who only read mail
chfn 'Jane Doe'	# Current user is Jane Doe

DESCRIPTION

Passwd is used to change your password. It prompts for the old and new passwords. It asks for the new password twice, to reduce the effect of a typing error. *Chfn* changes the full name (GECOS field) in the password file. *Chsh* changes your login shell. Do not forget to copy the modified password file back to the root file system, or the changes will be lost when the system is rebooted.

SEE ALSO

login(1), **su**(1), **crypt**(3), **getpwent**(3).

NAME

paste − paste multiple files together

SYNOPSIS

paste [**−s**] [**−d** *list*] *file...*

OPTIONS

−d Set delimiter used to separate columns to *list*.

−s Print files sequentially, file *k* on line *k*.

EXAMPLES

paste file1 file2 # Print *file1* in col 1, *file2* in col 2

paste −s f1 f2 # Print *f1* on line 1 and *f2* on line 2

paste -d : file1 file2 # Print the lines separated by a colon

DESCRIPTION

Paste concatenates corresponding lines of the given input files and writes them to standard output. The lines of the different files are separated by the delimiters given with the option −s. If no list is given, a tab is substituted for every linefeed, except the last one. If end-of-file is hit on an input file, subsequent lines are empty. Suppose a set of *k* files each has one word per line. Then the *paste* output will have *k* columns, with the contents of file *j* in column *j*. If the **−s** flag is given, then the first file is on line 1, the second file on line 2, etc. In effect, **−s** turns the output sideways.

If a list of delimiters is given, they are used in turn. The C escape sequences \n, \t, \\, and \0 are used for linefeed, tab, backslash, and the null string, respectively.

NAME

patch - apply a diff file to an original

SYNOPSIS

patch [options] [origfile [patchfile]] [+ [options] [origfile]]...

but usually just

patch <patchfile

DESCRIPTION

Patch will take a patch file containing any of the four forms of difference listing produced by the *diff* program and apply those differences to an original file, producing a patched version. By default, the patched version is put in place of the original, with the original file backed up to the same name with the extension ‘‘.orig’’ (‘‘˜’’ on systems that do not support long filenames), or as specified by the **-b**, **-B**, or **-V** switches. The extension used for making backup files may also be specified in the **SIMPLE_BACKUP_SUFFIX** environment variable, which is overridden by above switches.

If the backup file already exists, **patch** creates a new backup file name by changing the first lowercase letter in the last component of the file’s name into uppercase. If there are no more lowercase letters in the name, it removes the first character from the name. It repeats this process until it comes up with a backup file that does not already exist.

You may also specify where you want the output to go with a **-o** switch; if that file already exists, it is backed up first.

If *patchfile* is omitted, or is a hyphen, the patch will be read from standard input.

Upon startup, patch will attempt to determine the type of the diff listing, unless over-ruled by a **-c**, **-e**, **-n**, or **-u** switch. Context diffs (old-style, new-style, and unified) and normal diffs are applied by the *patch* program itself, while ed diffs are simply fed to the *ed* editor via a pipe.

Patch will try to skip any leading garbage, apply the diff, and then skip any trailing garbage. Thus you could feed an article or message containing a diff listing to *patch*, and it should work. If the entire diff is indented by a consistent amount, this will be taken into account.

With context diffs, and to a lesser extent with normal diffs, *patch* can detect when the line numbers mentioned in the patch are incorrect, and will attempt to find the correct place to apply each hunk of the patch. As a first guess, it takes the line number mentioned for the hunk, plus or minus any offset used in applying the previous hunk. If that is not the correct place, *patch* will scan both forwards and backwards for a set of lines matching the context given in the hunk. First *patch* looks for a place where all lines of the context match. If no such place is found, and it’s a context diff, and the maximum fuzz factor is set to 1 or more, then another scan takes place ignoring the first and last line of context. If that fails, and the maximum fuzz factor is set to 2 or more, the first two and last two lines of context are ignored, and another scan is made. (The default maximum fuzz factor is 2.) If *patch* cannot find a place to install that hunk of the patch, it will put the hunk out to a reject file, which normally is the name of the output file plus ‘‘.rej’’ (‘‘#’’ on systems that do not support long filenames). (Note that the rejected hunk will come out in context diff form whether the input patch was a context diff or a normal diff. If the input was a normal diff, many of the contexts will simply be null.) The line numbers on the hunks in the reject file may be different than in the patch file: they reflect the approximate location patch thinks the failed hunks belong in the new file rather than the old one.

As each hunk is completed, you will be told whether the hunk succeeded or failed, and which line (in the new file) *patch* thought the hunk should go on. If this is different from the line number specified in the diff you will be told the offset. A single large offset MAY be an indication that a hunk was installed in the wrong place. You will also be told if a fuzz factor was used to make the match, in which case you should also be slightly suspicious.

If no original file is specified on the command line, *patch* will try to figure out from the leading garbage what the name of the file to edit is. In the header of a context diff, the filename is found from lines beginning with ‘‘***’’ or ‘‘---’’, with the shortest name of an existing file winning. Only context diffs

have lines like that, but if there is an ‘‘Index:’’ line in the leading garbage, *patch* will try to use the filename from that line. The context diff header takes precedence over an Index line. If no filename can be intuited from the leading garbage, you will be asked for the name of the file to patch.

If the original file cannot be found or is read-only, but a suitable SCCS or RCS file is handy, *patch* will attempt to get or check out the file.

Additionally, if the leading garbage contains a ‘‘Prereq: ’’ line, *patch* will take the first word from the prerequisites line (normally a version number) and check the input file to see if that word can be found. If not, *patch* will ask for confirmation before proceeding.

The upshot of all this is that you should be able to say, while in a news interface, the following:

```
| patch -d /usr/src/local/blurfl
```

and patch a file in the blurfl directory directly from the article containing the patch.

If the patch file contains more than one patch, *patch* will try to apply each of them as if they came from separate patch files. This means, among other things, that it is assumed that the name of the file to patch must be determined for each diff listing, and that the garbage before each diff listing will be examined for interesting things such as filenames and revision level, as mentioned previously. You can give switches (and another original file name) for the second and subsequent patches by separating the corresponding argument lists by a ‘+’. (The argument list for a second or subsequent patch may not specify a new patch file, however.)

Patch recognizes the following switches:

−b causes the next argument to be interpreted as the backup extension, to be used in place of ‘‘.orig’’ or ‘‘~’’.

−B causes the next argument to be interpreted as a prefix to the backup file name. If this argument is specified any argument from -b will be ignored.

−c forces *patch* to interpret the patch file as a context diff.

−d causes *patch* to interpret the next argument as a directory, and cd to it before doing anything else.

−D causes *patch* to use the "#ifdef...#endif" construct to mark changes. The argument following will be used as the differentiating symbol. Note that, unlike the C compiler, there must be a space between the **−D** and the argument.

−e forces *patch* to interpret the patch file as an ed script.

−E causes *patch* to remove output files that are empty after the patches have been applied.

−f forces *patch* to assume that the user knows exactly what he or she is doing, and to not ask any questions. It assumes the following: skip patches for which a file to patch can’t be found; patch files even though they have the wrong version for the ‘‘Prereq:’’ line in the patch; and assume that patches are not reversed even if they look like they are. This option does not suppress commentary; use **−s** for that.

−t similar to **−f**, in that it suppresses questions, but makes some different assumptions: skip patches for which a file to patch can’t be found (the same as **−f**); skip patches for which the file has the wrong version for the ‘‘Prereq:’’ line in the patch; and assume that patches are reversed if they look like they are.

−F<number>
sets the maximum fuzz factor. This switch only applies to context diffs, and causes *patch* to ignore up to that many lines in looking for places to install a hunk. Note that a larger fuzz factor increases the odds of a faulty patch. The default fuzz factor is 2, and it may not be set to more than the number of lines of context in the context diff, ordinarily 3.

−l causes the pattern matching to be done loosely, in case the tabs and spaces have been munged in your input file. Any sequence of whitespace in the pattern line will match any sequence in the input file. Normal characters must still match exactly. Each line of the context must still match a line in the input file.

−n forces *patch* to interpret the patch file as a normal diff.

−N causes *patch* to ignore patches that it thinks are reversed or already applied. See also **−R .**

−o causes the next argument to be interpreted as the output file name.

−p<number>
sets the pathname strip count, which controls how pathnames found in the patch file are treated, in case the you keep your files in a different directory than the person who sent out the patch. The strip count specifies how many slashes are to be stripped from the front of the pathname. (Any intervening directory names also go away.) For example, supposing the filename in the patch file was

/u/howard/src/blurfl/blurfl.c

setting **−p** or **−p0** gives the entire pathname unmodified, **−p1** gives

u/howard/src/blurfl/blurfl.c

without the leading slash, **−p4** gives

blurfl/blurfl.c

and not specifying **−p** at all just gives you "blurfl.c", unless all of the directories in the leading path (u/howard/src/blurfl) exist and that path is relative, in which case you get the entire pathname unmodified. Whatever you end up with is looked for either in the current directory, or the directory specified by the **−d** switch.

−r causes the next argument to be interpreted as the reject file name.

−R tells *patch* that this patch was created with the old and new files swapped. (Yes, I'm afraid that does happen occasionally, human nature being what it is.) *Patch* will attempt to swap each hunk around before applying it. Rejects will come out in the swapped format. The **−R** switch will not work with ed diff scripts because there is too little information to reconstruct the reverse operation.

If the first hunk of a patch fails, *patch* will reverse the hunk to see if it can be applied that way. If it can, you will be asked if you want to have the **−R** switch set. If it can't, the patch will continue to be applied normally. (Note: this method cannot detect a reversed patch if it is a normal diff and if the first command is an append (i.e. it should have been a delete) since appends always succeed, due to the fact that a null context will match anywhere. Luckily, most patches add or change lines rather than delete them, so most reversed normal diffs will begin with a delete, which will fail, triggering the heuristic.)

−s makes *patch* do its work silently, unless an error occurs.

−S causes *patch* to ignore this patch from the patch file, but continue on looking for the next patch in the file. Thus

patch -S + -S + <patchfile

will ignore the first and second of three patches.

−u forces *patch* to interpret the patch file as a unified context diff (a unidiff).

−v causes *patch* to print out its revision header and patch level.

−V causes the next argument to be interpreted as a method for creating backup file names. The type of backups made can also be given in the **VERSION_CONTROL** environment variable, which is overridden by this option. The **-B** option overrides this option, causing the prefix to always be used for making backup file names. The value of the **VERSION_CONTROL** environment variable and the argument to the **-V** option are like the GNU Emacs 'version-control' variable; they

also recognize synonyms that are more descriptive. The valid values are (unique abbreviations are accepted):

‘t’ or ‘numbered’
Always make numbered backups.

‘nil’ or ‘existing’
Make numbered backups of files that already have them, simple backups of the others. This is the default.

‘never’ or ‘simple’
Always make simple backups.

−x<number>
sets internal debugging flags, and is of interest only to *patch* patchers.

AUTHOR

Larry Wall <lwall@netlabs.com>
with many other contributors.

ENVIRONMENT

TMPDIR
Directory to put temporary files in; default is /tmp.

SIMPLE_BACKUP_SUFFIX
Extension to use for backup file names instead of ‘‘.orig’’ or ‘‘~’’.

VERSION_CONTROL
Selects when numbered backup files are made.

FILES

$TMPDIR/patch*

SEE ALSO

diff(1)

NOTES FOR PATCH SENDERS

There are several things you should bear in mind if you are going to be sending out patches. First, you can save people a lot of grief by keeping a patchlevel.h file which is patched to increment the patch level as the first diff in the patch file you send out. If you put a Prereq: line in with the patch, it won’t let them apply patches out of order without some warning. Second, make sure you’ve specified the filenames right, either in a context diff header, or with an Index: line. If you are patching something in a subdirectory, be sure to tell the patch user to specify a **−p** switch as needed. Third, you can create a file by sending out a diff that compares a null file to the file you want to create. This will only work if the file you want to create doesn’t exist already in the target directory. Fourth, take care not to send out reversed patches, since it makes people wonder whether they already applied the patch. Fifth, while you may be able to get away with putting 582 diff listings into one file, it is probably wiser to group related patches into separate files in case something goes haywire.

DIAGNOSTICS

Too many to list here, but generally indicative that *patch* couldn’t parse your patch file.

The message ‘‘Hmm...’’ indicates that there is unprocessed text in the patch file and that *patch* is attempting to intuit whether there is a patch in that text and, if so, what kind of patch it is.

Patch will exit with a non-zero status if any reject files were created. When applying a set of patches in a loop it behooves you to check this exit status so you don’t apply a later patch to a partially patched file.

CAVEATS

Patch cannot tell if the line numbers are off in an ed script, and can only detect bad line numbers in a normal diff when it finds a ‘‘change’’ or a ‘‘delete’’ command. A context diff using fuzz factor 3 may have the same problem. Until a suitable interactive interface is added, you should probably do a context diff in these cases to see if the changes made sense. Of course, compiling without errors is a pretty

good indication that the patch worked, but not always.

Patch usually produces the correct results, even when it has to do a lot of guessing. However, the results are guaranteed to be correct only when the patch is applied to exactly the same version of the file that the patch was generated from.

BUGS

Could be smarter about partial matches, excessively deviant offsets and swapped code, but that would take an extra pass.

If code has been duplicated (for instance with #ifdef OLDCODE ... #else ... #endif), *patch* is incapable of patching both versions, and, if it works at all, will likely patch the wrong one, and tell you that it succeeded to boot.

If you apply a patch you've already applied, *patch* will think it is a reversed patch, and offer to unapply the patch. This could be construed as a feature.

NAME

pathchk − check pathname

SYNOPSIS

pathchk [−p] *path ...*

OPTIONS

−p Check against POSIX rules

EXAMPLES

pathchk /usr/src/file.c # Check the accessibility of *file.c*

pathchk −p file # Check posix compliance of *file*

DESCRIPTION

Pathchk verifies path names and display error if paths are too long, contain names that are too long, go thru a non-searchable directory or contain an invalid character in names. If the **−p** flag is specified, the POSIX rules apply instead of the rules of the current file system.

NAME

playwave − play an audio file in MicroSoft PCM wave format

SYNOPSIS

playwave [−**i**] file

DESCRIPTION

Playwave writes the samples in a wave file to */dev/audio*. The wave file must be in Microsoft PCM format.

OPTIONS

−i display information about wave file

BUGS

The highest sample rate that can be used depends on the speed of the system and the size of the DMA buffer used in the driver. (/usr/src/kernel/sb16.h)

AUTHOR

Michel R. Prevenier (mrpreve@cs.vu.nl)

NAME

postmort − perform post-mortem on PC Minix core files

SYNOPSIS

postmort [**−dpt**] **−c** *corefile* **−s** *symbfile*

OPTIONS

−c Use the named corefile

−d Dump all text symbols and segment data

−p Display the kernel process table

−s Use the named symbol file

−t Display a stack backtrace

EXAMPLES

postmort # display the data from the file 'core'

DESCRIPTION

Postmort does a simple static analysis of a PC Minix core file; By default, it looks for the file 'core' in the local directory and loads that for analysis; it also searches for the file 'symbol.out', and if that fails 'a.out', expecting them to contain symbol information for the core file. It is not a fatal error if the symbol files don't exist.

The stack backtrace is slightly tricky, and may go on longer than is really justified, since there's no easy way for it to know when to stop. Treat its results with caution.

NAME

pr − print a file

SYNOPSIS

pr [**−Mfnt**] [**−h** *n*] [**−l** *n*] [**−w** *n*] [−columns] [+*page*] [*file*] ...

OPTIONS

−M Use MINIX style line number

−f Do not fold long lines

−h Take next argument as page header

−l Sets page length in lines

−n Number the output lines

−t Do not print page header or trailer

−w Sets line length in characters

EXAMPLES

pr −w85 −l60 file # Use 85 character line, 60 line page

pr −3 file # List *file* three columns to a page

pr +4 file # Start printing with page 4

DESCRIPTION

Pr formats one or more files for printing. If no files are specified, *stdin* is printed. Options are provided for setting the width and height of the page, the number of columns to use (default 1), and the page to start with, among others.

SEE ALSO

lp(1).

NAME

prep − prepare a text file for statistical analysis

SYNOPSIS

prep [*file*]

OPTIONS

(none)

EXAMPLES

prep infile >outfile # Prepare *infile*

DESCRIPTION

Prep strips off most of the troff commands from a text file and then outputs all the words, one word per line, in the order they occur in the file. This file can then be sorted and compared to a dictionary (as a spelling checker), or used for statistical analyses.

SEE ALSO

nroff(1), **spell**(1).

NAME

pretty − MINIX pretty printer

SYNOPSIS

pretty *file* ...

EXAMPLES

pretty file1 file2 # Convert two files to MINIX layout

DESCRIPTION

Pretty converts one or more C source files to MINIX format by changing their layout. Running this program does not affect the resulting binary programs. Actually, *pretty* is a postprocessor for *indent*, which must be installed in */bin* or */usr/bin*. Although the output is not bad, it is not entirely consistent with the book or even with itself.

NAME

printenv − print out the current environment

SYNOPSIS

printenv

EXAMPLES

printenv # Print the environment

DESCRIPTION

Printenv prints out the current environment strings, one per line.

SEE ALSO

environ(7).

NAME

ps − process status

SYNOPSIS

ps [**−alxU**] [**kernel mm fs**]

OPTIONS

−a Print all processes with controlling terminals

−l Give long listing

−x Include processes without a terminal

EXAMPLES

ps −axl # Print all processes and tasks in long format

DESCRIPTION

Ps prints the status of active processes. Normally only the caller's own processes are listed in short format (the PID, TTY, TIME and CMD fields as explained below). The long listing contains:

F Kernel flags: 001: free slot 002: no memory map 004: sending; 010: receiving 020: inform on pending signals 040: pending signals 100: being traced.

S State: R: runnable W: waiting (on a message) S: sleeping (i.e.,suspended on MM or FS) Z: zombie T: stopped

UID, PID, PPID, PGRP The user, process, parent process and process group ID's.

SZ Size of the process in kilobytes.

RECV Process/task on which a receiving process is waiting or sleeping.

TTY Controlling tty for the process.

TIME Process' cumulative (user + system) execution time.

CMD Command line arguments of the process.

The files */dev/{mem,kmem}* are used to read the system tables and command line arguments from. Terminal names in */dev* are used to generate the mnemonic names in the TTY column, so *ps* is independent of terminal naming conventions.

NAME

pwd − print working directory

SYNOPSIS

pwd

EXAMPLES

pwd # Print the name of the working directory

DESCRIPTION

The full path name of the current working directory is printed.

SEE ALSO

getcwd(3).

NAME

rcp − remote file copy

SYNOPSIS

rcp [−**p**] *file1 file2*
rcp [−**pr**] *file ... directory*

DESCRIPTION

Rcp copies files between machines. Each *file* or *directory* argument is either a remote file name of the form ‘‘rhost:path’’, or a local file name (containing no ‘:’ characters, or a ‘/’ before any ‘:’s).

If the −**r** option is specified and any of the source files are directories, **rcp** copies each subtree rooted at that name; in this case the destination must be a directory.

By default, the mode and owner of *file2* are preserved if it already existed; otherwise the mode of the source file modified by the **umask**(2) on the destination host is used. The −**p** option causes **rcp** to attempt to preserve (duplicate) in its copies the modification times and modes of the source files, ignoring the **umask**.

If *path* is not a full path name, it is interpreted relative to your login directory on *rhost*. A *path* on a remote host may be quoted (using \, ", or ´) so that the metacharacters are interpreted remotely.

Rcp does not prompt for passwords; your current local user name must exist on *rhost* and allow remote command execution via **rsh**(1).

Rcp handles third party copies, where neither source nor target files are on the current machine. Hostnames may also take the form ‘‘rname@rhost’’ to use *rname* rather than the current user name on the remote host. The destination hostname may also take the form ‘‘rhost.rname’’ to support destination machines that are running 4.2BSD versions of **rcp**.

SEE ALSO

cp(1), **ftp**(1), **rsh**(1), **rlogin**(1).

BUGS

Doesn’t detect all cases where the target of a copy might be a file in cases where only a directory should be legal.
Is confused by any output generated by commands in a .profile, or .*shrc file on the remote host.

NAME

readall − read a device quickly to check for bad blocks

SYNOPSIS

readall [**−bt**] *file*

OPTIONS

−b Produce shell script on *stdout* that calls *badblocks*

−t Just print device size

EXAMPLES

readall /dev/hd0 # Read all of */dev/hd0*

readall -b /dev/hd1 >s # Generate shell script on *s*

DESCRIPTION

Readall reads all of the named device in large chunks. It reports about blocks that it cannot read. Unlike *diskcheck*, it does not attempt to write on the disk, making it safer to use when one is worried about a sick system. When the **−b** flag is given, the output is a shell script that calls the *badblocks* program to marked all the bad blocks. Whenever installing MINIX, it is wise to run *readall* with the **−b** flag first on all the hard disks.

SEE ALSO

badblocks(8).

NAME

readfs − read a MINIX file system

SYNOPSIS

readfs [**−il**] *block_special* [*dir*]

OPTIONS

−i Give information about the file, but do not extract files

−l List the files extracted on standard output

EXAMPLES

readfs −l /dev/fd0 # List contents of diskette

DESCRIPTION

Readfs reads a diskette containing a MINIX file system. It can extract all the files from it, give a listing of them, or both. The files extracted can be put in a user-specified directory (default: current directory). If subdirectories are needed, they will be created automatically.

SEE ALSO

mkproto(1).

NAME

recover − recover files that have been removed.

SYNOPSIS

recover *file* ...

EXAMPLES

rm x; recover x # Unremove x

recover a b c # Recover three files

DESCRIPTION

MINIX allows files that have been deleted (e.g., with *rm*) to be restored (in */tmp*). The trick is that when a file is unlinked, its i-node number is kept in the directory entry. As long as the directory entry and disk blocks are not reused, the file can be recovered. This program is actually just a little front end for *de*, which must be installed as a set-uid root executable.

SEE ALSO

de(9).

NAME

recwave − record an audio file in MicroSoft PCM wave format

SYNOPSIS

recwave [−**b** −**s** −**r**] file

DESCRIPTION

Recwav takes samples from */dev/audio* and writes them to *file* in Microsoft PCM wave format.

OPTIONS

−b number of bits to use for one sample. Must be 8 or 16, default is 8

−s enable stereo sampling. 0 = mono (default), 1 = stereo

−r sample rate in samples/sec. 4000 - 44100 (default 22050)

BUGS

The highest sample rate that can be used depends on the speed of the system and the size of the DMA buffer used in the driver. (/usr/src/kernel/sb16.h)

AUTHOR

Michel R. Prevenier (mrpreve@cs.vu.nl)

NAME

ref - Display a C function header

SYNOPSIS

ref [-t] [-c *class*]... [-f *file*]... *tag*

DESCRIPTION

ref quickly locates and displays the header of a function. To do this, *ref* looks in the "tags" file for the line that describes the function, and then scans the source file for the function. When it locates the function, it displays an introductory comment (if there is one), the function's declaration, and the declarations of all arguments.

SEARCH METHOD

ref uses a fairly sophisticated tag look-up algorithm. If you supply a filename via **-f** *file*, then elvis first scans the tags file for a static tag from that file. This search is limited to the tags file in the current directory.

If you supply a classname via **-c** *class*, then elvis searches for a tag from that class. This search is not limited to the current directory; You can supply a list of directories in the environment variable *TAGPATH*, and *ref* will search through the "tags" file in each directory until it finds a tag in the desired class.

If that fails, *ref* will then try to look up an ordinary global tag. This search checks all of the directories listed in *TAGPATH*, too.

If you've given the **-t** flag, then *ref* will simply output the tag line that it found, and then exit. Without **-t**, though, *ref* will search for the tag line. It will try to open the source file, which should be in the same directory as the tags file where the tag was discovered. If the source file doesn't exist, or is unreadable, then *ref* will try to open a file called "*refs*" in that directory. Either way, *ref* will try to locate the tag, and display whatever it finds.

INTERACTION WITH ELVIS

ref is used by *elvis*' shift-K command. If the cursor is located on a word such as "splat", in the file "foo.c", then *elvis* will invoke *ref* with the command "ref -f foo.c splat".

If *elvis* has been compiled with the -DEXTERNAL_TAGS flag, then *elvis* will use *ref* to scan the tags files. This is slower than the built-in tag searching, but it allows *elvis* to access the more sophisticated tag lookup provided by *ref*. Other than that, external tags should act exactly like internal tags.

OPTIONS

-t Output tag info, instead of the function header.

-f *file* The tag might be a static function in *file*. You can use several -f flags to have *ref* consider static tags from more than one file.

-c *class* The tag might be a member of class *class*. You can use several -c flags to have *ref* consider tags from more than one class.

FILES

tags List of function names and their locations, generated by *ctags*.

refs Function headers extracted from source files (optional).

ENVIRONMENT

TAGPATH

List of directories to be searched. The elements in the list are separated by either semicolons (for MS-DOS, Atari TOS, and AmigaDos), or by colons (every other operating system). For each operating system, *ref* has a built-in default which is probably adequate.

NOTES

You might want to generate a "tags" file the directory that contains the source code for standard C library on your system. If licensing restrictions prevent you from making the library source readable by everybody, then you can have *ctags* generate a "refs" file, and make "refs" readable by everybody.

If your system doesn't come with the library source code, then perhaps you can produce something workable from the *lint* libraries.

SEE ALSO

elvis(1), ctags(1)

AUTHOR

Steve Kirkendall
kirkenda@cs.pdx.edu

NAME

remsync - remotely synchronize file trees

SYNOPSIS

remsync −sxv *tree* [*state-file*]
remsync −duxvD *tree* [*state-file* [*diff-file*]]
remsync [**−xv**] *tree* [*diff-file*]

DESCRIPTION

Remsync synchronizes file trees of distant machines, i.e. machines that do not have a fast network between them. It accomplishes this in three steps:

Create a state file containing a description of the machine to be updated.

Compute a file of differences on the source machine using the state file to compare the two file trees.

Update the target machine using the data in the differences file.

This process requires that you move two files, a state file from the target machine to the source machine, and a differences file from the source machine to the target machine. The state file is an ASCII file that may be edited, usually to make **remsync** ignore some files or file trees.

The argument *tree* may be a single file or a directory. A directory is traversed recursively. The *state-file* and *diff-file* arguments may be of any file type. The differences file contains an end marker, so it may be followed by trailing junk. Standard input or output is used if these arguments are omitted or replaced by a minus sign.

State file format

A state file has a line for each file in a tree. A line looks like this formally for a simple file:

name mode owner group length date [*link-number* [**last**]]

The best way to show how each type of file is represented is by example:

```
/               d755            0       0
bin             d755            2       0
    [               644             2       0       233     759160857       1
    cat             755             2       0       3772    768742021
    test            755             2       0       233     759160857       1
dev             d755            0       0
    fd0             b666            0       0       200
    console         c600            10      0       400
    sd2             b600            0       0       a02
    fifo            p700            2       0
opt             ->              usr/opt
usr             ignore (Cross-device link)
```

The root of the tree is always represented by a /, no matter what type of file it may be. Directory entries of the root follow at the same level. Files in subdirectories are indented by two spaces. (Eight spaces are replaced by a TAB.) Normal files have their length and modified time in the state file, devices have their device number in hex, etc. If files are hard linked to each other then they all get an extra "link number" to bind them together. The last link is marked with the word **last**.

One usually only modifies a state file to ignore differences between two files. One does this by replacing the file attributes with the word **ignore**. (**Remsync** generates this keyword too, with the reason why added in parentheses.)

OPTIONS

−s Generate a state file.

−d Generate a differences file. (The default is to apply a differences file.)

−u Only add new files or update files with newer versions.

−x Do not cross device boundaries. This allows one to operate on the root file system for instance ignoring the **/usr** file system.

−D Debug differences file generation. With this flag no file contents are added to the differences file. The result is then human readable.

−v Lists the commands added to the differences file, or the actions done applying a differences file. The output looks like UNIX commands except for the words "add", "restore" and "update" indicating addition of a new file, replacing a file with an older version, or replacement by a newer version.

EXAMPLES

Actions taken by the author to update his notebook "finiah" from his main machine "darask":

finiah# remsync -s /usr /tmp/finiah.state

Edit the state file to ignore .Xauthority files and /usr/var.

finiah# tar cvf /dev/fd0 /tmp/finiah.state

darask# tar xvf /dev/fd0
darask# remsync -dv /usr /tmp/finiah.state | vol 1440 /dev/fd0

finiah# vol 1440 /dev/fd0 | remsync -v /usr

One could add a file compression/decompression program between **remsync** and **vol**, to reduce the number of floppies to move about, but that actually slows things down! (Note that one only needs to shuffle two floppies about if the two machines are adjacent. To update a remote machine it does make sense to use compression to reduce the number of floppies to carry.)

SEE ALSO

synctree(1), **vol**(1), **tar**(1).

NOTES

Nothing stops you from using **remsync** over a fast network of course. **Synctree** can be a bit tedious if you only want to ignore a few files. Editing a state file is then easier.

BUGS

Files are overwritten, not removed, when they are updated. This means that links outside the tree are also updated. The less desirable alternative to this is to break the link before the update.

The verbose option may say that a link is to be created when making a differences file. The link is often already there when the update takes place, so no action is taken, and thus no talk about it. So you may miss a few mutterings about links if you compare the messages.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

rev − reverse the characters on each line of a file

SYNOPSIS

rev [*file*] ...

EXAMPLES

rev file # Reverse each line

DESCRIPTION

Each file is copied to standard output with all the characters of each line reversed, last one first and first one last.

SEE ALSO

rev(1).

NAME

rlogin − remote login

SYNOPSIS

rlogin [**−8EL**] [**−e** *char*] [**−l** *username*] *rhost*
rhost [**−8EL**] [**−e** *char*] [**−l** *username*]

DESCRIPTION

Rlogin connects your terminal on the current local host system *lhost* to the remote host system *rhost.*

Each host has a file **/etc/hosts.equiv** which contains a list of *rhost*'s with which it shares account names. (The host names must be the standard names as described in **rsh**(1).) When you **rlogin** as the same user on an equivalent host, you don't need to give a password. Each user may also have a private equivalence list in a file .rhosts in his login directory. Each line in this file should contain an *rhost* and a *username* separated by a space, giving additional cases where logins without passwords are to be permitted. If the originating user is not equivalent to the remote user, then a login and password will be prompted for on the remote machine as in **login**(1). To avoid some security problems, the .rhosts file must be owned by either the remote user or root.

The remote terminal type is the same as your local terminal type (as given in your environment TERM variable). The terminal or window size is also copied to the remote system if the server supports the option, and changes in size are reflected as well. All echoing takes place at the remote site, so that (except for delays) the rlogin is transparent. Flow control via ^S and ^Q and flushing of input and output on interrupts are handled properly. The optional argument **−8** allows an eight-bit input data path at all times; otherwise parity bits are stripped except when the remote side's stop and start characters are other than ^S/^Q. The argument **−L** allows the rlogin session to be run in litout mode. A line of the form ''~.'' disconnects from the remote host, where ''~'' is the escape character. Similarly, the line ''~^Z'' (where ^Z, control-Z, is the suspend character) will suspend the rlogin session. Substitution of the delayed-suspend character (normally ^Y) for the suspend character suspends the send portion of the rlogin, but allows output from the remote system. A different escape character may be specified by the **−e** option. There is no space separating this option flag and the argument character. With the **−E** option the escape can be turned off.

SEE ALSO

rsh(1), **rhosts**(5).

BUGS

More of the environment should be propagated.

NAME

rm − remove a file

SYNOPSIS

rm [**−ifrRvx**] *name* ...

OPTIONS

−i	Interactive remove: ask before removing
−f	Forced remove: no questions asked
−rR	Remove directories and their contents
−v	Tell what rm is removing
−x	Do not cross device boundaries

EXAMPLES

rm file	# Remove *file*
rm −i *.c	# Remove *.c* files, asking about each

DESCRIPTION

Rm removes one or more files. If a file has no write permission, *rm* asks for permission (type 'y' or 'n') unless **−f** is specified. If the file is a directory, it will be recursively descended and removed if and only if the **−R** flag is present.

SEE ALSO

cp(1), **mv**(1), **rmdir**(1), **unlink**(2), **rmdir**(2).

NAME

rmdir − remove a directory

SYNOPSIS

rmdir *directory ...*

OPTIONS

(none)

EXAMPLES

rmdir /user/ast/foobar # Remove directory *foobar*

rmdir /user/ast/f* # Remove 0 or more directories

DESCRIPTION

The specified directories are removed. Ordinary files are not removed. The directories must be empty.

SEE ALSO

mkdir(1), **rmdir**(2).

NAME

rsh − remote shell

SYNOPSIS

rsh [**−n**] [**−l** *username*] *host* [*command*]
host [**−n**] [**−l** *username*] [*command*]

DESCRIPTION

Rsh connects to the specified *host*, and executes the specified *command*. **Rsh** copies its standard input to the remote command, the standard output of the remote command to its standard output, and the standard error of the remote command to its standard error. Interrupt, quit and terminate signals are propagated to the remote command; **rsh** normally terminates when the remote command does.

The remote username used is the same as your local username, unless you specify a different remote name with the **−l** option. This remote name must be equivalent (in the sense of **rlogin**(1)) to the originating account; no provision is made for specifying a password with a command.

If you omit *command*, then instead of executing a single command, you will be logged in on the remote host using **rlogin**(1).

Shell metacharacters which are not quoted are interpreted on local machine, while quoted metacharacters are interpreted on the remote machine. Thus the command

rsh otherhost cat remotefile >> localfile

appends the remote file *remotefile* to the localfile *localfile*, while

rsh otherhost cat remotefile ">>" otherremotefile

appends *remotefile* to *otherremotefile*.

OPTIONS

−l *username*
Specify the remote user name.

−n Connect standard input of the remote command to /dev/null. Do this if **rsh** should not inadvertently read from standard input.

SEE ALSO

rcp(1), **rlogin**(1), **rhosts**(5).

BUGS

You cannot run an interactive command (like **rogue**(6) or **vi**(1)); use **rlogin**(1).

NAME

rz − receive a file using the zmodem protocol

SYNOPSIS

rz [−**abepqvy**] [−**t** *timeout*]

OPTIONS

−a CP/M to UNIX conventions

−b Binary file

−e Escape for all control characters

−p Protect file if it already exists

−q Quiet; opposite of verbose

−t Set *timeout* in tenths of a second

−v Verbose; opposite of quiet

−y Yes, clobber existing files

EXAMPLES

rz </dev/tty01 >/dev/tty01
Receive a file

DESCRIPTION

The XMODEM, YMODEM, and ZMODEM family of file transfer programs are widely used on personal computers. MINIX supports ZMODEM, the most advanced of the set. The programs *sz* and *rz* are used for sending and receiving, respectively.

Rz and *sz* are programs that uses an error correcting protocol to transfer files over a dial-in serial port from a variety of programs running under various operating systems. *Rz* (Receive ZMODEM) receives files with the ZMODEM batch protocol. Pathnames are supplied by the sending program, and directories are made if necessary (and possible). The meanings of the available options are:

−a
Convert files to UNIX conventions by stripping carriage returns and all characters beginning with the first Control Z (CP/M end of file).

−b
Binary (tell it like it is) file transfer override.

−c
Request 16 bit CRC. XMODEM file transfers default to 8 bit checksum. YMODEM and ZMODEM normally use 16 bit CRC.

−D
Output file data to /dev/null; for testing.

−e
Force sender to escape all control characters; normally XON, XOFF, DLE, CR-@-CR, and Ctrl-X are escaped.

−p
Protect: skip file if destination file exists.

−q
Quiet suppresses verbosity.

−t
Change timeout tenths of seconds (timeout follows flag).

−v
Verbose causes a list of file names to be appended to */tmp/rzlog*. More v's generate more output.

−y
Yes, clobber any existing files with the same name.

SEE ALSO

sz(1), **term**(1).

NAME

sed − stream editor

SYNOPSIS

sed [**−egn**] [**−f** *script_file*] [*edit_script*] [*file*]

OPTIONS

−e Accept multiple commands commands on the commands line

−f The following argument contains the edit script

−g Set the global flag on all substitute commands

−n Only output selected lines

EXAMPLES

sed −f script <file # Run a sed script on *file*

sed ′/pig/s//hog/g′ <file # Replace *pig* by *hog* in the file

DESCRIPTION

Sed is a stream editor. It takes an edit script either from its argument or a file, and performs an edit session on a named file or *stdin*, producing output on *stdout*.

SEE ALSO

cgrep(1), **fgrep**(1), **grep**(1), **awk**(9).

NAME

sh, ., break, case, cd, continue, eval, exec, exit, export, for, if, read, readonly, set, shift, trap, umask, wait, while − shell

SYNOPSIS

sh [−**eiknqstvxu**] [−**c** *str*] **[***file*]

OPTIONS

−c Execute the commands in *str*

−e Quit on error

−i Interactive mode; ignore QUIT, TERMINATE, INTERRUPT

−k Look for name=value everywhere on command line

−n Do not execute commands

−q Change qflag from sig_ign to sig_del

−s Read commands from standard input

−t Exit after reading and executing one command

−v Echo input lines as they are read

−x Trace

−u Unset variables

EXAMPLES

sh script # Run a shell script

DESCRIPTION

Sh is the shell, which forms the user's main interface with the system. On startup, the shell reads /etc/profile and $HOME/.profile, if they exist, and executes any commands they contain. The Minix shell has most of the features of the V7 (Bourne) shell, including redirection of input and output, pipes, magic characters, background processes, and shell scripts. A brief summary follows, but whole books have been written on shell programming alone.

Some of the more common notations are:

```
date                        #  Regular command
sort <file                  #  Redirect stdin (standard input)
sort <file1  >file2         #  Redirect stdin and stdout
cc file.c  2>error          #  Redirect stderr
a.out >f  2>&1              #  Combine standard output and standard error
sort <file1  >>file2        #  Append output to file2
sort <file1  >file2 &       #  Background job
(ls −l; a.out) &            #  Run two background commands sequentially
sort <file | wc             #  Two-process pipeline
sort <f | uniq | wc         #  Three-process pipeline
ls −l *.c                   #  List all files ending in .c
ls −l [a-c]*                #  List all files beginning with a, b, or c
ls −l ?                     #  List all one-character file names
ls \?                       #  List the file whose name is question mark
ls '???'                    #  List the file whose name is three question marks
v=/usr/ast                  #  Set shell variable v
ls −l $v                    #  Use shell variable v
PS1='Hi! '                  #  Change the primary prompt to Hi!
PS2='More: '                #  Change the secondary prompt to More:
ls −l $HOME                 #  List the home directory
echo $PATH                  #  Echo the search path
echo $?                     #  Echo exit status of previous command in decimal
echo $$                     #  Echo shell's pid in decimal
```

```
echo $!          # Echo PID of last background process
echo $#          # Echo number of parameters (shell script)
echo $2          # Echo second parameter (shell script)
echo "$2"        # Echo second parameter without expanding spaces
echo $*          # Echo all parameters (shell script)
echo $@          # Echo all parameters (shell script)
echo "$@"        # Echo all parameters without expanding spaces
```

The shell uses the following variables for specific purposes:

SHELL	the path of the current shell
HOME	the default value for the cd(1) command
PATH	the directories to be searched to find commands
IFS	the internal field separators for command strings
PS1	the primary shell prompt
PS2	the secondary shell prompt

There are various forms of substitution on the shell command line:

‘...‘	Command string between back-quotes is replaced by its output
"..."	Permits variable substitution between quotes
’...’	Inhibits variable substitution between quotes
$VAR	Replaced by contents of variable VAR
${VAR}	Delimits variable VAR from any following string

The expressions below depend on whether or not VAR has ever been set. If VAR has been set, they give:

${VAR-str}	Replace expression by VAR, else by str
${VAR=str}	Replace expression by VAR, else by str and set VAR to str
${VAR?str}	Replace expression by VAR, else print str and exit shell
${VAR+str}	Replace expression by str, else by null string

If a colon is placed after VAR, the expressions depend on whether or not VAR is currently set and non-null.

The shell has a number of built-in commands:

:	return true status
. fn	execute shell script fn on current path
break [n]	break from a for, until or while loop; exit n levels
continue [n]	continue a for, until or while loop; resume nth loop
cd [dir]	change current working directory; move to $HOME
eval cmd	rescan cmd, performing substitutions
eval	rescan the current command line
exec cmd	execute cmd without creating a new process
exec <|>	with no command name, modify shell I/O
exit [n]	exit a shell program, with exit value n
export [var]	export var to shell’s children; list exported variables
pwd	print the name of the current working directory
read var	read a line from stdin and assign to var
readonly [var]	make var readonly; list readonly variables
set -f	set shell flag (+f unsets flag)
set str	set positional parameter to str
set	show the current shell variables
shift	reassign positional parameters (except ${0}) one left
times	print accumulated user and system times for processes
trap arg sigs	trap signals sigs and run arg on receipt
trap	list trapped signals
umask [n]	set the user file creation mask; show the current umask
wait [n]	wait for process pid n; wait for all processes

The shell also contains a programming language, which has the following operators and flow control statements:

#	Comment	The rest of the line is ignored
=	Assignment	Set a shell variable
&&	Logical AND	Execute second command only if first succeeds
||	Logical OR	Execute second command only if first fails
(...)	Group	Execute enclosed commands before continuing

for	For loop (for ... in ... do ... done)
case	Case statement ((case ...) ... ;; ... esac)
esac	Case statement end
while	While loop (while ... do ... done)
do	Do/For/While loop start (do ... until ...)
done	For/While loop end
if	Conditional statement (if ... else ... elif ... fi)
in	For loop selection
then	Conditional statement start
else	Conditional statement alternative
elif	Conditional statement end
until	Do loop end
fi	Conditional statement end

SEE ALSO

echo(1), **expr**(1), **pwd**(1), **true**(1).

NAME

shar − shell archiver

SYNOPSIS

shar *file* ...

EXAMPLES

shar *.c >s	# Collect C programs in shell archive
sh <s	# Extract files from a shell archive

DESCRIPTION

The named files are collected together into a shell archive written onto standard output. The individual files can be extracted by redirecting the shell archive into the shell. The advantage of *shar* over *ar* is that *shar* archives can be read on almost any UNIX system, whereas numerous, incompatible versions of *ar* are in widespread use. Extracting the files from a shell archive requires that *sed* be accessible.

SEE ALSO

sh(1), **unshar**(1).

NAME

size − print text, data, and bss size of a program

SYNOPSIS

size [*file*] ...

EXAMPLES

size file # Print the size of *file*

DESCRIPTION

The text, data, bss, and total sizes for each argument are printed. If no arguments are present, *a.out* is assumed. The amount of memory available for combined stack and data segment growth is printed in the column 'stack.' This is the value manipulated by the *chmem* command. The total amount of memory allocated to the program when it is loaded is listed under 'memory.' This value is just the sum of the other four columns.

SEE ALSO

anm(1), **asize**(1), **ar**(1), **chmem**(1), **install**(1), **nm**(1).

NAME

sleep − suspend execution for a given number of seconds

SYNOPSIS

sleep *seconds*

EXAMPLES

sleep 10 # Suspend execution for 10 sec.

DESCRIPTION

The caller is suspended for the indicated number of seconds. This command is typically used in shell scripts.

SEE ALSO

sleep(3).

NAME

sort − sort a file of ASCII lines

SYNOPSIS

sort [**−bcdfimnru**] [**−t***c*] [**−o** *name*] [*+pos1*] [*−pos2*] *file ...*

OPTIONS

−b Skip leading blanks when making comparisons

−c Check to see if a file is sorted

−d Dictionary order: ignore punctuation

−f Fold upper case onto lower case

−i Ignore nonASCII characters

−m Merge presorted files

−n Numeric sort order

−o Next argument is output file

−r Reverse the sort order

−t Following character is field separator

−u Unique mode (delete duplicate lines)

EXAMPLES

sort −nr file	# Sort keys numerically, reversed
sort +2 −4 file	# Sort using fields 2 and 3 as key
sort +2 −t: −o out	# Field separator is *:*
sort +.3 −.6	# Characters 3 through 5 form the key

DESCRIPTION

Sort sorts one or more files. If no files are specified, *stdin* is sorted. Output is written on standard output, unless **−o** is specified. The options *+pos1 −pos2* use only fields *pos1* up to but not including *pos2* as the sort key, where a field is a string of characters delimited by spaces and tabs, unless a different field delimiter is specified with **−t**. Both *pos1* and *pos2* have the form *m.n* where *m* tells the number of fields and *n* tells the number of characters. Either *m* or *n* may be omitted.

SEE ALSO

comm(1), **grep**(1), **uniq**(1).

NAME

spell − print all words in a file not present in the dictionary

SYNOPSIS

spell *file*

EXAMPLES

spell document # Print the spelling errors on *stdout*

DESCRIPTION

Spell is the MINIX spelling checker. It is actually a short shell script. First, the program *prep* strips off the *roff*, *nroff*, and *troff* control lines, and the punctuation, and lists each word on a separate line. These words are then sorted. The resulting output is then compared to the dictionary. Words present in the file but not present in the dictionary are listed. The dictionary must be located in */usr/lib/dict/words*.

SEE ALSO

nroff(1), **prep**(1).

NAME

split − split a large file into several smaller files

SYNOPSIS

split [−*n*] [*file* [*prefix*]]

OPTIONS

−*n* Number of lines per piece (default: 1000)

EXAMPLES

split −200 file # Split *file* into pieces of 200 lines each

split file z # Split *file* into *zaa*, *zab*, etc.

DESCRIPTION

Split reads *file* and writes it out in *n*-line pieces. By default, the pieces are called *xaa*, *xab*, etc. The optional second argument can be used to provide an alternative prefix for the output file names.

SEE ALSO

cat(1).

NAME

strings − print all the strings in a binary file

SYNOPSIS

strings [−] [−**o**] [−*n*] *file* ...

OPTIONS

−	search whole file, not just data seg
−o	Print octal offset of each string
−*n*	*n* is minimum length string (default = 4)

EXAMPLES

strings −5 a.out	# Print the strings > 4 chars in *a.out*
strings − /bin/sh	# Search entire shell file (text and data)

DESCRIPTION

Strings looks for sequences of ASCII characters followed by a zero byte. These are usually strings. This program is typically used to help identify unknown binary programs

NAME

strip − remove symbol table from executable file

SYNOPSIS

strip [*file*] ...

EXAMPLES

strip a.out # Remove symbols from *a.out*

DESCRIPTION

For each file argument, *strip* removes the symbol table. Strip makes a copy of the file being stripped, so links are lost.

SEE ALSO

install(1).

NAME

stty − set terminal parameters

SYNOPSIS

stty [**−ag]**

stty *encoded-form*

stty *speed* **ispeed** *speed* **ospeed** *speed* **cs5 cs6 cs7 cs8** [−]**parenb** [−]**parodd** [−]**hupcl** [−]**cstopb** [−]**cread** [−]**clocal** [−]**ignbrk** [−]**brkint** [−]**ignpar** [−]**parmrk** [−]**inpck** [−]**istrip** [−]**inlcr** [−]**igncr** [−]**icrnl** [−]**ixon** [−]**ixoff** [−]**ixany** [−]**opost** [−]**onlcr** [−]**xtabs** [−]**onoeot** [−]**isig** [−]**icanon** [−]**iexten** [−]**echo** [−]**echoe** [−]**echok** [−]**echonl** [−]**noflsh** [−]**lflusho eof**=*c* **eol**=*c* **erase**=*c* **erase**=*c* **intr**=*c* **kill**=*c* **quit**=*c* **susp**=*c* **start**=*c* **stop**=*c* **rprnt**=*c* **lnext**=*c* **flush**=*c* **min**=*n* **time**=*n* **rows** *n* **cols** *n* **cooked raw** [−]**evenp** [−]**parity** [−]**oddp** [−]**nl ek sane**

DESCRIPTION

Stty shows or changes the parameters of the terminal connected to standard input. **Stty** takes a myriad of arguments most of which are mapped directly to the flags and special characters described in **tty**(4), so we won't describe them here.

Stty has three forms of operation. First, without any arguments **stty** shows all terminal attributes that are different from the default state. Option **−a** makes **stty** print all terminal attributes, and **−g** lets **stty** print the attributes in a special encoded form, a simple row of colon separated hexadecimal numbers.

In the second form of operation **stty** takes an encoded form as produced by the **−g** option and sets the terminals attributes to its decoded value.

In the third form **stty** interprets a series of flags and parameters settings and modifies the terminal attributes accordingly. Flags can be given as **icanon** or **−icanon** for instance, either setting or clearing the **ICANON** flag. Special character values can by set like **intr=^C** for example, which sets the interrupt character to CTRL-C. You can either use a real CTRL-C, or the two characters '^' and 'C'. In any case it is probably necessary to use quotes to guard it from the shell: **intr='^C'**.

A number alone is interpreted as a baud rate setting for both the input and output rate. The input or the output rate can be set separately with use of the **ispeed** and **ospeed** prefixes to the number. The character size can be set with **cs5**, **cs6**, **cs7** or **cs8**.

The **MIN** and **TIME** value, and the number of rows and columns of the window can also be set using one of the keywords **min**, **time**, **rows** or **cols**, followed by a decimal number that is the value of the setting.

Stty accepts several keywords that are not named by corresponding flags or parameters in **tty**(4). They set several attributes at once:

cooked Same as **icrnl ixon opost onlcr isig icanon iexten echo**, setting all the attributes that are needed for line oriented mode.

raw Same as **−icrnl −ixon −opost −onlcr −isig −icanon −iexten −echo**, setting all the attributes for a raw data channel.

evenp parity
These synonyms are equal to **cs7 parenb −parodd**, setting the line to 7 bits even parity.

oddp Same as **cs7 parenb parodd**, setting the line to 7 bits odd parity.

−parity −evenp −oddp
All synonyms for **cs8 −parenb**, setting the line to 8 bits, no parity.

nl Same as **icrnl**, setting carriage return to line feed input translation.

−nl Same as **−icrnl −inlcr −igncr**, disabling any carriage return or line feed handling.

ek Set the **ERASE** and **KILL** special characters back to the default.

sane Set all attributes to the default except things like the line speed and parity, because their "sane" value is probably what it is right now. The default values are compiled into **stty** from the <termios.h> include file. Use **stty sane; stty -a** to know what they are.

FILES

/etc/ttytab The **init** field of this file may contain an **stty** command to set the attributes to match an attached RS232 terminal or modem.

SEE ALSO

tty(4), **ttytab**(5).

NOTES

The **cooked**, **raw**, **rows** and **cols** keywords are Minix additions beyond the keywords defined by POSIX. **Rows** and **cols** are common UNIX extensions, however. There are more Minix specific flags that match the Minix specific attributes described in **tty**(4).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

su − temporarily log in as superuser or another user

SYNOPSIS

su [*name*]

EXAMPLES

su # Become superuser

su ast # Become *ast*

DESCRIPTION

Su can be used to temporarily login as another user. It prompts for the superuser password. If the correct password is entered, *su* creates a shell with the desired uid. If no name is specified, *root* is assumed. To exit the temporary shell, type CTRL-D. When memory is tight, it is better to become superuser by logging out and then logging in again as *root*, rather than using *su* since the latter creates an extra shell in memory. Any user with a zero group id need not type a password. This makes life easier for the system administrator.

SEE ALSO

login(1).

NAME

sum − compute the checksum and block count of a file

SYNOPSIS

sum *file*

OPTIONS

(none)

EXAMPLES

sum /user/ast/xyz # Checksum */user/ast/xyz*

DESCRIPTION

Sum computes the checksum of one or more files. It is most often used to see if a file copied from another machine has been correctly received. This program works best when both machines use the same checksum algorithm. See also *crc*.

SEE ALSO

cksum(1), **crc**(1).

NAME

svc, ci, co, svclog − shell version control system

SYNOPSIS

ci [**−lu**] *file*
co [**−l**] [**−r** *rev*] *file*
svclog *file*

OPTIONS

−l For *ci*, checkin, checkout again, and lock file

−l For *co*, checkout file and then lock the archive

−u After checking in, do not delete the file

−r Check out revision *rev* instead most recent revision

EXAMPLES

ci −u file # Check in *file*

co −l file # Check out *file* and lock archive

co −r 2 file # Check out version 2

DESCRIPTION

Svc is the Shell Version Control system, patterned on RCS. It maintains a sequence of versions in archive files, so that new versions can be checked in (added to the archive), and old versions can be checked out (made available). To create an archive for *file*, check it in with the **−u** flag. This action will prompt for a log message and then create an archive called *file,S* in the current directory, or in the subdirectory *SVC* if it exists. The file will not be deleted, but will be made unwritable.

To update the file, check it out with the **−l** flag. Then modify it, and check it back in, giving a new message when prompted. After this process has been repeated many times, the archive will contain the entire history. Any version can be checked out using the **−r** flag. To get a printout of the history, use *svclog*.

NAME

synctree − synchronize directory trees.

SYNOPSIS

synctree [**−iuf**] [[*user1@*]*machine1*:]*dir1* [[*user2@*]*machine2*:]*dir2*

DESCRIPTION

Synctree synchronizes the directory tree rooted at *dir2* with *dir1*. It walks recursively through both trees, and deletes and adds files in *dir2* to make it equal to *dir1*. Mode, owner and group are set for each file unless the **−u** flag is given. In its normal mode of operation, synctree will ask if it may delete or add directories assuming that you don't want to. Non-directories are simply deleted or added, but synctree will ask if it needs to update a normal file with a default answer of 'y'. Simply typing return will choose the default answer, typing end-of-file is like typing return to this question and all other questions.

You can specify a hostname and user-id to be used to access *dir1* or *dir2*. Synctree will use **rsh**(1) to run a copy of itself on the remote machine. The call interface mimics that of **rcp**(1), but you can use more than one user@machine prefix if you want to make things really interesting.

Hard links are enforced, an update is done by first deleting the old file so that links to unknown files are broken. Links to files within *dir2* will be restored.

If either directory contains the file **.backup**, then this file will be used as an alternate inode table. This allows one to make a backup copy of a file tree full of special files and differing user-ids on a remote machine under an unpriviledged user-id.

OPTIONS

−i Ask for permission (with default answer 'n') to delete or add any file or directory.

−u Only install newer files, i.e. merge the directory trees.

−f Don't ask, think 'yes' on any question.

SEE ALSO

remsync(1), **cpdir**(1), **rsh**(1), **rcp**(1), **perror**(3).

DIAGNOSTICS

Messages may come from three different processes. One named "Slave" running in *dir1*, one named "Master" running in *dir2*, and synctree itself in a mediator role. The mediator will also perform the task of either the master or the slave if one of them is running locally. You need to know this to interpret the error messages coming from one of these processes. The messages are normally based on **perror**(3). Failure to contact a remote machine will be reported by **rsh**. **Synctree** should have a zero exit status if no errors have been encountered.

BUGS

Directory *dir2* will be created without asking.

The master and slave processes get their error output mixed up sometimes (nice puzzle).

The local and remote machine must use the same file type encoding.

The link replacement strategy may lead to lack of space on a small device. Let **synctree** run to completion and then rerun it to pick up the pieces.

Letting the local process keep its "synctree" name may be a mistake.

It talks too much.

AUTHOR

Kees J. Bot, (kjb@cs.vu.nl)

NAME

sz − send a file using the zmodem protocol

SYNOPSIS

sz [**−LNbdefnopqruvy+**] [**−ci** *command*] [**−Ll** n] [**−t** *timeout*]

OPTIONS

−L Use *n*-byte packets

−N Overwrite if source is newer/longer

−b Binary file

−c Send command for execution

−d Convert dot to slash in names

−e Escape for all control characters

−f Send full path name

−i Send command and return immediately

−l Flow control every *n* packets

−n Overwrite destination if source is newer

−o Use old (16-bit) checksum

−p Protect file if it already exists

−q Quiet; opposite of verbose

−r Resume interrupt file transfer

−t Set *timeout* in tenths of a second

−u Unlink file after successful transmission

−v Verbose; opposite of quiet

−y Yes, clobber existing files

−+ Append to an existing file

EXAMPLES

sz file </dev/tty01 >/dev/tty01

Send *file*

DESCRIPTION

XMODEM, YMODEM, and ZMODEM are a family of protocols that are widely used is the MS-DOS world for transferring information reliably from one computer to another. In all of these protocols, a series of bytes are sent from one computer to the other, and then an acknowledgement is sent back to confirm correct reception. Checksums are used to detect errors so that transmission is reliable even in the face of noisy telephone lines. *Sz* is a program that sends a file sent from another computer using the zmodem protocol. The file can be received using *rz*.

Sz uses the ZMODEM error correcting protocol to send one or more files over a dial-in serial port to a variety of programs running under MINIX, UNIX, MS-DOS, CP/M, VMS, and other operating systems. It is the successor to XMODEM and YMODEM.

ZMODEM greatly simplifies file transfers compared to XMODEM. In addition to a friendly user interface, ZMODEM provides Personal Computer and other users an efficient, accurate, and robust file transfer method.

ZMODEM provides complete end-to-end data integrity between application programs. ZMODEM's 32 bit CRC catches errors that sneak into even the most advanced networks.

Output from another program may be piped to *sz* for transmission by denoting standard input with −:

"ls −l | sz −

The program output is transmitted with the filename *sPID.sz* where PID is the process ID of the *sz* program. If the environment variable *ONAME* is set, that is used instead. In this case, the command:

"ls −l | ONAME=con sz −ay −

will send a 'file' to the PC-DOS console display. The **−y** option instructs the receiver to open the file for writing unconditionally. The **−a** option causes the receiver to convert UNIX newlines to PC-DOS carriage returns and linefeeds. On UNIX systems, additional information about the file is transmitted. If the receiving program uses this information, the transmitted file length controls the exact number of bytes written to the output dataset, and the modify time and file mode are set accordingly.

If *sz* is invoked with $SHELL set and if that variable contains the string *rsh* or *rksh* (restricted shell), *sz* operates in restricted mode. Restricted mode restricts pathnames to the current directory and *PUBDIR* (usually */usr/spool/uucppublic*) and/or subdirectories thereof.

The options and flags available are:

−+
Instruct the receiver to append transmitted data to an existing file.

−a
Convert NL characters in the transmitted file to CR/LF. This is done by the sender for XMODEM and YMODEM, by the receiver for ZMODEM.

−b
Binary override: transfer file without any translation.

−c
Send COMMAND (follows *c*) to the receiver for execution, return with COMMAND's exit status.

−d
Change all instances of '.' to '/' in the transmitted pathname. Thus, C.omenB0000 (which is unacceptable to MS-DOS or CP/M) is transmitted as C/omenB0000. If the resultant filename has more than 8 characters in the stem, a '.' is inserted to allow a total of eleven.

−e
Escape all control characters; normally XON, XOFF, DLE, CR-@-CR, and Ctrl-X are escaped.

−f
Send Full pathname. Normally directory prefixes are stripped from the transmitted filename.

−i
Send COMMAND (follows *i*) to the receiver for execution, return Immediately upon the receiving program's successful reception of the command.

−L
Use ZMODEM sub-packets of length *n* (follows *L*). A larger *n* (32 <= *n* <= 1024) gives slightly higher throughput, a smaller one speeds error recovery. The default is 128 below 300 baud, 256 above 300 baud, or 1024 above 2400 baud.

−l
Wait for the receiver to acknowledge correct data every *n* (32 <= *n* <= 1024) characters. This may be used to avoid network overrun when XOFF flow control is lacking.

−n
Send each file if destination file does not exist. Overwrite destination file if source file is newer than the destination file.

−N
Send each file if destination file does not exist. Overwrite destination file if source file is newer or longer than the destination file.

−o
Disable automatic selection of 32 bit CRC.

−p
Protect existing destination files by skipping transfer if the destination file exists.

−q
Quiet suppresses verbosity.

−r

Resume interrupted file transfer. If the source file is longer than the destination file, the transfer commences at the offset in the source file that equals the length of the destination file.

−t

Change timeout. The timeout, in tenths of seconds, follows, the **−t** flag.

−u

Unlink the file after successful transmission.

−w

Limit the transmit window size to *n* bytes (*n follows* **(enw**).

−v

Verbose causes a list of file names to be appended to */tmp/szlog*.

−y

Instruct a ZMODEM receiving program to overwrite any existing file with the same name.

−Y

Instruct a ZMODEM receiving program to overwrite any existing file with the same name, and to skip any source files that do have a file with the same pathname on the destination system.

Examples

Below are some examples of the use of *sz*.

"sz −a ∗.c

This single command transfers all .c files in the current directory with conversion (**−a**) to end-of-line conventions appropriate to the receiving environment.

"sz −Yan ∗.c ∗.h

Send only the *.c* and *.h* files that exist on both systems, and are newer on the sending system than the corresponding version on the receiving system, converting MINIX to MS-DOS text format.

SEE ALSO

rz(1), **term**(1).

NAME

tail − print the last few lines of a file

SYNOPSIS

tail [**−c** *n*] [**−f] [−n** *n*] [*file*] ...

OPTIONS

−c The count refers to characters

−f On FIFO or special file, keep reading after EOF

−n The count refers to lines

EXAMPLES

tail −n 6 # Print last 6 lines of *stdin*

tail −c 20 file # Print the last 20 characters of *file*

tail −n 1 file1 file2 # Print last line of two files

tail −n +8 file # Print the tail starting with line 8

DESCRIPTION

The last few lines of one or more files are printed. The default count is 10 lines. The default file is *stdin*. If the value of *n* for the **−c** or **−n** flags starts with a + sign, counting starts at the beginning, rather than the end of the file.

SEE ALSO

head(1).

NAME

tar − tape archiver

SYNOPSIS

tar [**Fcotvxp**] [**f**] *tarfile file ...*

OPTIONS

F	Force tar to continue after an error
c	Create a new archive; add named files
o	Set uid/gid to original values on extraction
f	Next argument is name of tarfile
t	Print a table listing the archive's contents
v	Verbose mode-tell what is going on as it happens
x	The named files are extracted from the archive
p	Restore file modes, ignore creation mask
D	Directory only, do not recurse

EXAMPLES

tar c /dev/fd1 . # Back up current directory to */dev/fd1*

tar xv /dev/fd1 file1 file2
Extract two files from the archive

tar cf − | (cd dest; tar xf −)
Copy current directory to *dest*

DESCRIPTION

Tar is a POSIX-compatible archiver, except that it does not use tape. It's primary advantage over *ar* is that the *tar* format is somewhat more standardized than the *ar* format, making it theoretically possible to transport MINIX files to another computer, but do not bet on it. If the target machine runs MS-DOS , try *doswrite* .

SEE ALSO

compress(1), **vol**(1).

NAME

tee − divert stdin to a file

SYNOPSIS

tee [**−ai**] *file* ...

OPTIONS

−a Append to the files, rather than overwriting

−i Ignore interrupts

EXAMPLES

cat file1 file2 | tee x # Save and display two files

pr file | tee x | lpr # Save the output of *pr* on *x*

DESCRIPTION

Tee copies *stdin* to standard output. It also makes copies on all the files listed as arguments.

SEE ALSO

cat(1).

NAME

telnet − user interface to the TELNET protocol

SYNOPSIS

telnet [*host* [*port*]]

DESCRIPTION

Telnet is used to communicate with another host using the **TELNET** protocol. If *telnet* is invoked without arguments, it enters command mode, indicated by its prompt (telnet>). In this mode, it accepts and executes the commands listed below. If it is invoked with arguments, it performs an **open** command (see below) with those arguments.

Once a connection has been opened, *telnet* enters an input mode. The input mode entered will be either character at a time or line by line depending on what the remote system supports.

In character at a time mode, most text typed is immediately sent to the remote host for processing.

In line by line mode, all text is echoed locally, and (normally) only completed lines are sent to the remote host. The local echo character (initially ^E) may be used to turn off and on the local echo (this would mostly be used to enter passwords without the password being echoed).

In either mode, if the *localchars* toggle is TRUE (the default in line mode; see below), the user's *quit*, *intr*, and *flush* characters are trapped locally, and sent as **TELNET** protocol sequences to the remote side. There are options (see **toggle** *autoflush* and **toggle** *autosynch* below) which cause this action to flush subsequent output to the terminal (until the remote host acknowledges the **TELNET** sequence) and flush previous terminal input (in the case of *quit* and *intr*).

While connected to a remote host, *telnet* command mode may be entered by typing the *telnet* escape character (initially ^]). When in command mode, the normal terminal editing conventions are available.

COMMANDS

The following commands are available. Only enough of each command to uniquely identify it need be typed (this is also true for arguments to the **mode**, **set**, **toggle**, and **display** commands).

open *host* [*port*]

Open a connection to the named host. If no port number is specified, *telnet* will attempt to contact a **TELNET** server at the default port. The host specification may be either a host name (see *hosts*(5)) or an Internet address specified in the dot notation (see *inet*(3N)).

close

Close a **TELNET** session and return to command mode.

quit

Close any open **TELNET** session and exit *telnet*. An end of file (in command mode) will also close a session and exit.

z

Suspend *telnet*. This command only works when the user is using the *csh*(1).

mode *type*

Type is either *line* (for line by line mode) or *character* (for character at a time mode). The remote host is asked for permission to go into the requested mode. If the remote host is capable of entering that mode, the requested mode will be entered.

status

Show the current status of *telnet*. This includes the peer one is connected to, as well as the current mode.

display [*argument...*]

Displays all, or some, of the **set** and **toggle** values (see below).

? [*command*]

Get help. With no arguments, *telnet* prints a help summary. If a command is specified, *telnet* will print the help information for just that command.

send *arguments*

Sends one or more special character sequences to the remote host. The following are the arguments which may be specified (more than one argument may be specified at a time):

escape

Sends the current *telnet* escape character (initially ^]).

synch

Sends the **TELNET SYNCH** sequence. This sequence causes the remote system to discard all previously typed (but not yet read) input. This sequence is sent as TCP urgent data (and may not work if the remote system is a 4.2 BSD system -- if it doesn't work, a lower case r may be echoed on the terminal).

brk

Sends the **TELNET BRK** (Break) sequence, which may have significance to the remote system.

ip

Sends the **TELNET IP** (Interrupt Process) sequence, which should cause the remote system to abort the currently running process.

ao

Sends the **TELNET AO** (Abort Output) sequence, which should cause the remote system to flush all output **from** the remote system **to** the user's terminal.

ayt

Sends the **TELNET AYT** (Are You There) sequence, to which the remote system may or may not choose to respond.

ec

Sends the **TELNET EC** (Erase Character) sequence, which should cause the remote system to erase the last character entered.

el

Sends the **TELNET EL** (Erase Line) sequence, which should cause the remote system to erase the line currently being entered.

ga

Sends the **TELNET GA** (Go Ahead) sequence, which likely has no significance to the remote system.

nop

Sends the **TELNET NOP** (No OPeration) sequence.

?

Prints out help information for the **send** command.

set *argument value*

Set any one of a number of *telnet* variables to a specific value. The special value off turns off the function associated with the variable. The values of variables may be interrogated with the **display** command. The variables which may be specified are:

echo

This is the value (initially ^E) which, when in line by line mode, toggles between doing local echoing of entered characters (for normal processing), and suppressing echoing of entered characters (for entering, say, a password).

escape

This is the *telnet* escape character (initially ^[) which causes entry into *telnet* command mode (when connected to a remote system).

interrupt

If *telnet* is in *localchars* mode (see **toggle** *localchars* below) and the *interrupt* character is typed, a **TELNET IP** sequence (see **send** *ip* above) is sent to the remote host.

The initial value for the interrupt character is taken to be the terminal's **intr** character.

quit

If *telnet* is in *localchars* mode (see **toggle** *localchars* below) and the *quit* character is typed, a **TELNET BRK** sequence (see **send** *brk* above) is sent to the remote host. The initial value for the quit character is taken to be the terminal's **quit** character.

flushoutput

If *telnet* is in *localchars* mode (see **toggle** *localchars* below) and the *flushoutput* character is typed, a **TELNET AO** sequence (see **send** *ao* above) is sent to the remote host. The initial value for the flush character is taken to be the terminal's **flush** character.

erase

If *telnet* is in *localchars* mode (see **toggle** *localchars* below), **and** if *telnet* is operating in character at a time mode, then when this character is typed, a **TELNET EC** sequence (see **send** *ec* above) is sent to the remote system. The initial value for the erase character is taken to be the terminal's **erase** character.

kill

If *telnet* is in *localchars* mode (see **toggle** *localchars* below), **and** if *telnet* is operating in character at a time mode, then when this character is typed, a **TELNET EL** sequence (see **send** *el* above) is sent to the remote system. The initial value for the kill character is taken to be the terminal's **kill** character.

eof

If *telnet* is operating in line by line mode, entering this character as the first character on a line will cause this character to be sent to the remote system. The initial value of the eof character is taken to be the terminal's **eof** character.

toggle *arguments...*

Toggle (between TRUE and FALSE) various flags that control how *telnet* responds to events. More than one argument may be specified. The state of these flags may be interrogated with the **display** command. Valid arguments are:

localchars

If this is TRUE, then the *flush*, *interrupt*, *quit*, *erase*, and *kill* characters (see **set** above) are recognized locally, and transformed into (hopefully) appropriate **TELNET** control sequences (respectively *ao*, *ip*, *brk*, *ec*, and *el*; see **send** above). The initial value for this toggle is TRUE in line by line mode, and FALSE in character at a time mode.

autoflush

If *autoflush* and *localchars* are both TRUE, then when the *ao*, *intr*, or *quit* characters are recognized (and transformed into **TELNET** sequences; see **set** above for details), *telnet* refuses to display any data on the user's terminal until the remote system acknowledges (via a **TELNET** *Timing Mark* option) that it has processed those **TELNET** sequences. The initial value for this toggle is TRUE if the terminal user had not done an "stty noflsh", otherwise FALSE (see *stty(1)).*

autosynch

If *autosynch* and *localchars* are both TRUE, then when either the *intr* or *quit* characters is typed (see **set** above for descriptions of the *intr* and *quit* characters), the resulting **TELNET** sequence sent is followed by the **TELNET SYNCH** sequence. This procedure **should** cause the remote system to begin throwing away all previously typed input until both of the **TELNET** sequences have been read and acted upon. The initial value of this toggle is FALSE.

crmod

Toggle carriage return mode. When this mode is enabled, most carriage return

characters received from the remote host will be mapped into a carriage return followed by a line feed. This mode does not affect those characters typed by the user, only those received from the remote host. This mode is not very useful unless the remote host only sends carriage return, but never line feed. The initial value for this toggle is FALSE.

debug

Toggles socket level debugging (useful only to the *super*user*)*. The initial value for this toggle is FALSE.

options

Toggles the display of some internal *telnet* protocol processing (having to do with **TELNET** options). The initial value for this toggle is FALSE.

netdata

Toggles the display of all network data (in hexadecimal format). The initial value for this toggle is FALSE.

?

Displays the legal **toggle** commands.

BUGS

There is no adequate way for dealing with flow control.

On some remote systems, echo has to be turned off manually when in line by line mode.

There is enough settable state to justify a *.telnetrc* file.

No capability for a *.telnetrc* file is provided.

In line by line mode, the terminal's *eof* character is only recognized (and sent to the remote system) when it is the first character on a line.

NAME

term − turn PC into a dumb terminal [IBM]

SYNOPSIS

term [*baudrate*] [*parity*] [*bits_per_character*] [−*dial_string*] [*device*]

EXAMPLES

term 2400 # Talk to modem at 2400 baud

term 1200 7 even # 1200 baud, 7 bits/char, even parity

term 8 9600 /dev/tty01 # 9600 baud, 8 bits/char, no parity, use tty01

term -atdt12345 /dev/tty01
Start with a command to dial out

DESCRIPTION

Term allows MINIX to talk to a terminal or modem over RS232 port 1. The program first sets the baudrate, parity and character length, and then forks. The parent sits in a loop copying from *stdin* (usually the console's keyboard), to the terminal or modem (*/dev/tty00*). The child sits in a loop copying from the terminal or modem (*/dev/tty00*) to standard output. Thus when RS232 port 1 is connected to a modem, every keystroke typed on the keyboard is sent to the modem, and every character arriving from the modem is displayed. Standard input and output may be redirected, to provide a primitive file transfer program, with no checking. Any argument that starts with a minus ('−') is sent out to the modem, usually to dial out. *Term* accepts several commands that are formed by typing the escape character, CTRL-], and a letter. Type CTRL-]? to see a list of commands. The subshell command is very important, it allows you to type in a ZMODEM command to transfer data. Do not quit *term* to do this, or your modem line will be reset! *Term* keeps the modem line open on file descriptor 9 while running the subshell, so you can type

<&9 >&9

at the end of your ZMODEM command to connect it to the modem.

Important note: to use *term*, it is essential that */etc/ttytab* is configured so that there is no shell hanging on */dev/tty01*. If there is, both the shell and term will try to read from */dev/tty01*, and nothing will work.

SEE ALSO

rz(1), **sz**(1).

NAME

termcap − print the current termcap entry

SYNOPSIS

termcap [*type*]

EXAMPLES

termcap # Print the termcap entry

DESCRIPTION

Termcap reads the /etc/termcap entry corresponding to the terminal type supplied as the argument. If none is given, the current $TERM is used. It then prints out all the parameters that apply.

SEE ALSO

termcap(3).

NAME

test, [− test for a condition

SYNOPSIS

test *expr*
[*expr* **]**

OPTIONS

(none)

EXAMPLES

test −r file # See if file is readable

DESCRIPTION

Test checks to see if files exist, are readable, etc. and returns an exit status of zero if true and nonzero if false. The legal operators are

−r file	true if the file is readable
−w file	true if the file is writable
−x file	true if the file is executable
−f file	true if the file is not a directory
−d file	true if the file is a directory
−s file	true if the file exists and has a size > 0
−t fd	true if file descriptor fd (default 1) is a terminal
−z s	true if the string s has zero length
−n s	true if the string s has nonzero length
s1 = s2	true if the strings s1 and s2 are identical
s1 != s2	true if the strings s1 and s2 are different
m −eq m	true if the integers m and n are numerically equal

The operators **−gt**, **−ge**, **−ne**, **−le**, and **−lt** may be used as well. These operands may be combined with **−a** (Boolean and), **−o** (Boolean or), ! (negation). The priority of **−a** is higher than that of **−o**. Parentheses are permitted, but must be escaped to keep the shell from trying to interpret them.

SEE ALSO

expr(1), **sh**(1).

NAME

tget − get termcap values

SYNOPSIS

tget [**−flag** *id*] [**−num** *id*] [**−str** *id*] [**−goto** *col line*] [[**−echo**] *string*]

DESCRIPTION

Tget allows shell scripts access to the **termcap**(3) functions. Flags, numbers and strings can be queried from the termcap database under the entry denoted by the environment variable **$TERM**.

OPTIONS

−flag *id* Set the exit status to zero if the flag *id* is set. All other options except **−echo** set the exit status to *id* being available or not. The last option sets the final exit code.

−num *id*
Print the value of the numeric variable *id*.

−str *id* Print the value of the string variable *id*.

−goto *col line*
Generates instructions to go to the given column and line if the **cm** capability exists.

−echo *string*
Prints *string*. Any other argument that does not start with a dash is also printed.

EXAMPLE

Try this: **tget -str so 'Reverse Video' -str**

ENVIRONMENT

TERM Terminal type.

TERMCAP Path to the termcap database, by default /etc/termcap:/usr/etc/termcap.

SEE ALSO

termcap(3).

DIAGNOSTICS

Tget will fail immediately with a descriptive message if the termcap entry can't be found.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

time − report how long a command takes

SYNOPSIS

time *command*

EXAMPLES

time a.out # Report how long *a.out* takes

time ls −l *.c # Report how long *ls* takes

DESCRIPTION

The command is executed and the real time, user time, and system time (in hours, minutes, and seconds) are printed. Shell scripts cannot be timed.

SEE ALSO

times(2).

NAME

touch − update a file's time of last modification

SYNOPSIS

touch [**−c**] *file* ...

OPTIONS

−c Do not create the file

EXAMPLES

touch *.h # Make the *.h* files look recent

DESCRIPTION

############ NEXT ENTRY HAS NOT BEEN CHECKED ############

The time of last modification is set to the current time. This command is mostly used to trick *make* into thinking that a file is more recent than it really is. If the file being touched does not exist, it is created, unless the **−c** flag is present.

SEE ALSO

utime(2).

NAME

tr − translate character codes

SYNOPSIS

tr [**−cds**] [*string1*] [*string2*]

OPTIONS

−c Complement the set of characters in *string1*

−d Delete all characters specified in *string1*

−s Squeeze all runs of characters in *string1* to one character

EXAMPLES

```
tr '[A-Z]' '[a-z]' <x >y
        # Convert upper case to lower case
tr -d '0123456789' <f1 >f2
        # Delete all digits from f1
```

DESCRIPTION

Tr performs simple character translation. When no flag is specified, each character in *string1* is mapped onto the corresponding character in *string2* .

NAME

traverse − print directory tree under the named directory

SYNOPSIS

traverse *dir*

EXAMPLES

traverse . # Print tree starting at working dir

DESCRIPTION

Traverse prints the tree structure starting at the named directory. All the subdirectories are listed, with the depth shown by indentation.

SEE ALSO

find(1), **ls**(1).

NAME

treecmp − recursively list differences in two directory trees

SYNOPSIS

treecmp [**−cv**] *olddir newdir*

OPTIONS

−c (changes) list the names of changed or new files

−v (verbose) list all directories processed

EXAMPLES

treecmp −v /usr/ast/V1 /usr/ast/V2
Compare two trees

DESCRIPTION

Treecmp recursively descends the directory tree of its second argument and compares all files to those at the corresponding position in the first argument. If the two trees are identical, i.e., all the corresponding directories and files are the same, there is no output. Otherwise, a list of files missing from one of the trees or present in both but whose contents are not identical in both are printed. When the **−c** flag is given, only files that are changed from the old version or are new (i.e., absent in the old version) are listed.

NAME

true, false − exit with the value true or false

SYNOPSIS

true
false

OPTIONS

(none)

EXAMPLES

```
while true              # List the directory until DEL is hit
    do ls −l
    done
```

DESCRIPTION

These commands return the value *true* or *false* . They are used for shell programming.

SEE ALSO

sh(1).

NAME

tsort − topological sort [IBM]

SYNOPSIS

tsort *file*

EXAMPLES

lorder *.s | tsort # Give library ordering

ar cr libc.a `lorder *.s | tsort`
Build library

DESCRIPTION

Tsort accepts a file of lines containing ordered pairs and builds a total ordering from the partial orderings.

NAME

ttt − tic tac toe

SYNOPSIS

ttt

EXAMPLES

ttt # Start the game

DESCRIPTION

This program allows the user to engage in a game of tic tac toe (noughts and crosses) with the computer. The program uses the alpha-beta algorithm, so the user had better be sharp.

NAME

tty − print the device name of this tty

SYNOPSIS

tty [**−s**]

OPTIONS

−s Silent mode, only the exit status is affected.

EXAMPLES

tty # Print the tty name

DESCRIPTION

Print the name of the controlling tty. If the flag **−s** is given, *tty* is equivalent to the call **test −t 0**.

SEE ALSO

ttyname(3).

NAME

umount − unmount a mounted file system

SYNOPSIS

umount *special*

OPTIONS

(none)

EXAMPLES

umount /dev/fd1 # Unmount diskette 1

DESCRIPTION

A mounted file system is unmounted after the cache has been flushed to disk. A diskette should never be removed while it is mounted. If this happens, and is discovered before another diskette is inserted, the original one can be replaced without harm. Attempts to unmount a file system holding working directories or open files will be rejected with a 'device busy' message.

SEE ALSO

mount(1), **umount**(2).

NAME

uname, arch − system info

SYNOPSIS

uname [−**snrvmpa**]
arch [−**snrvmpa**]

OPTIONS

−s System name

−n Node/network name

−r Operating system release

−v Operating system version

−m Machine type

−p Processor family

−a Short for **−snrvm**

EXAMPLES

uname -n # Print the name of the system

arch # Print the name of the system architecture

DESCRIPTION

Uname and *arch* give information about the system. The options indicate which information strings must be printed. These strings are always in the same order. *Uname* and *arch* only differ w.r.t. the default string to print, **−s** and **−p** respectively.

The strings are compiled into the commands except for the node name, it is obtained from the file */etc/hostname.file*. **Uname −m** should return the actual machine type, not the same string as with **−p**.

SEE ALSO

uname(3).

NAME

unexpand − convert spaces to tabs

SYNOPSIS

unexpand [**−a**]

OPTIONS

−a All spaces are unexpanded

EXAMPLES

unexpand oldfile >newfile

Convert leading spaces to tabs

DESCRIPTION

Unexpand replaces spaces in the named files with tabs. If no files are listed, *stdin* is given. The **−a** flag is used to force all sequences of spaces to be expanded, instead of just leading spaces (the default).

SEE ALSO

expand(1).

NAME

uniq − delete consecutive identical lines in a file

SYNOPSIS

uniq [**−cdu**] [−*n*] [+*n*] [*input [output*]]

OPTIONS

−c Give count of identical lines in the input

−d Only duplicate lines are written to output

−u Only unique lines are written to output

−*n* Skip the first *n* columns when matching

+*n* Skip the first *n* fields when matching

EXAMPLES

uniq +2 file # Ignore first 2 fields when comparing

uniq −d inf outf # Write duplicate lines to *outf*

DESCRIPTION

Uniq examines a file for consecutive lines that are identical. All but duplicate entries are deleted, and the file is written to output. The +*n* option skips the first *n* fields, where a field is defined as a run of characters separated by white space. The −*n* option skips the first *n* spaces. Fields are skipped first.

SEE ALSO

sort(1).

NAME

unshar − Remove files from a shell archive

SYNOPSIS

unshar [**−btvx**] *sharfile member ...*

OPTIONS

−b Unshar brutally, overwriting files if need be

−t Tell what is in the archive but do not extract

−v Verbose mode

−x Extract only the members listed

EXAMPLES

unshar arch.sh # Extract all members of the archive

unshar −t arch.sh # List the contents of the archive

unshar −xf1 −xf2 arch.sh
Extract *f1* and *f2* from *arch.sh*

uniq −d inf outf # Write duplicate lines to *outf*

DESCRIPTION

Unshar extracts members of a shell archive, the same as *sh*, except much faster. It expects shell archives created with *shar*. It also has options to list the contents of a shell archive, and to selectively extract some members but not all.

SEE ALSO

sh(1), **shar**(1).

NAME

uud, uudecode − decode a binary file encoded with uue

SYNOPSIS

uud [**−n**] [**−s** *srcdir*] [**−t** *dstdir/*] *file*

OPTIONS

−n Do not verify checksums

−s Name of directory where *.uue* file is

−t Name of directory where output goes

EXAMPLES

uud file.uue # Re-create the original file

uud − <file.uue # The − means use *stdin*

DESCRIPTION

Uud decodes a file encoded with *uue* or UNIX *uuencode*. The decoded file is given the name that the original file had. The name information is part of the encoded file. Mail headers and other junk before the encoded file are skipped.

SEE ALSO

uue(1).

NAME

uue, uuencode − encode a binary file to ASCII (e.g., for mailing)

SYNOPSIS

uue [−*n*] *file* [−]

OPTIONS

−*n* How many lines to put in each file

EXAMPLES

uue file	# Encode *file* to *file.uue*
uue file − >x	# Encode *file* and write on *stdout*
uue −800 file	# Output on *file.uaa*, *file.uab* etc.

DESCRIPTION

Uuencode is a famous program that converts an arbitrary (usually binary) file to an encoding using only 64 ASCII characters. *Uudecode* converts it back to the original file. The *uue* and *uud* programs are the MINIX versions of these programs, and are compatible with the UNIX ones. The files produced can even be sent successfully over BITNET, which is notorious for mangling files. It is possible to have *uue* automatically split the encoded file up into small chunks. The output files then get the suffixes *.uaa*, *.uab*, etc., instead of *.uue*. When *uud* is given *file.uaa* to decode, it automatically includes the subsequent pieces. The encoding takes 3 bytes (24 bits) from the input file and renders it as 4 bytes in the output file.

SEE ALSO

btoa(1), **uud**(1).

NAME

vol − split input on or combine output from several volumes

SYNOPSIS

vol [**−rw1**] [**−b** *blocksize*] [**−m** *multiple*] [*size*] *device*

DESCRIPTION

Vol either reads a large input stream from standard input and distributes it over several volumes or combines volumes and sends them to standard output. The size of the volumes is determined automatically if the device supports this, but may be specified before the argument naming the device if automated detection is not possible or if only part of the physical volume is used. The direction of the data is automatically determined by checking whether the input or output of **vol** is a file or pipe. Use the **−r** or **−w** flag if you want to specify the direction explicitly, in shell scripts for instance.

Vol waits for each new volume to be inserted, typing return makes it continue. If no size is explicitely given then the size of the device is determined each time before it is read or written, so it is possible to mix floppies of different sizes. If the size cannot be determined (probably a tape) then the device is assumed to be infinitely big. **Vol** can be used both for block or character devices. It will buffer the data and use a block size appropriate for fixed or variable block sized tapes.

Vol reads or writes 8192 bytes to block devices, usually floppies. Character devices are read or written using a multiple of 512 bytes. This multiple has an upper limit of 32767 bytes (16-bit machine), 64 kb (32-bit), or even 1 Mb (32-bit VM). The last partial write to a character device is padded with zeros to the block size. If a character device is a tape device that responds to the **mtio**(4) status call then the reported tape block size will be used as the smallest unit. If the tape is a variable block length device then it is read or written like a block device, 8192 bytes at the time, with a minimum unit of one byte.

All sizes may be suffixed by the letters **M**, **k**, **b** or **w** to multiply the number by mega, kilo, block (512), or word (2). The volume size by default in kilobytes if there is no suffix.

OPTIONS

−rw Explicitly specify reading or writing. Almost mandatory in scripts.

−1 Just one volume, start immediately.

−b *blocksize*
Specify the device block size.

−m *multiple*
Specify the maximum read or write size of multiple blocks. The **−b** and **−m** options allow one to modify the block size assumptions that are made above. These assumptions are **−b 1 −m 8192** for block devices or variable length tapes, and **−b 512 −m 65536** for character devices (32 bit machine.) These options will not override the tape block size found out with an **mtio**(4) call. The multiple may be larger then the default if **vol** can allocate the memory required.

EXAMPLES

To back up a tree to floppies as a compressed tarfile:

```
tar cf − . | compress | vol /dev/fd0
```

To restore a tree from 720 kb images from possibly bigger floppies:

```
vol 720 /dev/fd0 | uncompress | tar xfp −
```

Read or write a device with 1024 byte blocks:

```
vol −b 1k /dev/rsd15
```

Read or write a variable block length tape using blocking factor 20 as used by default by many **tar**(1) commands:

```
vol −m 20b /dev/rst5
```

Note that **−m** was used in the last example. It sets the size to use to read or write, **−b** sets the basic block size that may be written in multiples.

SEE ALSO

dd(1), **tar**(1), **mt**(1), **mtio**(4).

NAME

wc − count characters, words, and lines in a file

SYNOPSIS

wc [**−clw**] *file* ...

OPTIONS

−c Print character count

−l Print line count

−w Print word count

EXAMPLES

wc file1 file2 # Print all three counts for both files

wc −l file # Print line count only

DESCRIPTION

Wc reads each argument and computes the number of characters, words and lines it contains. A word is delimited by white space (space, tab, or line feed). If no flags are present, all three counts are printed.

NAME

whatis, apropos − give single line descriptions for manual pages

SYNOPSIS

whatis [**−a**] *title*
apropos *keyword*

DESCRIPTION

Whatis lists the one line description from the **whatis**(5) database describing the title given. It displays all the lines with the title from the first whatis file that has those titles. It uses the same search path as **man**(1).

Apropos searches through all whatis files for the given keywords. It lists any line that has the keyword anywhere on the line.

OPTIONS

−a Search all whatis files.

SEE ALSO

man(1), **grep**(1), **whatis**(5).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

whatsnew − print a newly modified file, marking changes

SYNOPSIS

whatsnew [−*n*] *file.c file.c.cdif*

OPTIONS

−*n* Output line length

EXAMPLES

whatsnew file.c file.c.cdif

Print *file.c* with changes marked

whatsnew −70 file.c file.c.cdif

Same as above, but with 70 col line

DESCRIPTION

It commonly occurs that *cdif*s are posted to USENET. After installing a *cdif* file, it is sometimes desirable to print out the new file, with the changes marked on it. *Whatsnew* does precisely this, with the changes + and ! printed in the right-hand margin.

SEE ALSO

diff(1), **patch**(1).

NAME

whereis − examine system directories for a given file

SYNOPSIS

whereis *file*

EXAMPLES

whereis stat.h # Prints: */usr/include/sys/stat.h*

DESCRIPTION

Whereis searches a fixed set of system directories, */bin*, */lib*, */usr/bin*, and others, and prints all occurrences of the argument name in any of them.

SEE ALSO

man(1), **which**(1).

NAME

which − examine $PATH to see which file will be executed

SYNOPSIS

which *name*

EXAMPLES

which a.out # Tells which *a.out* will be executed

DESCRIPTION

The $PATH shell variable controls the MINIX search rules. If a command *a.out* is given, the shell first tries to find an executable file in the working directory. If that fails, it looks in various system directories, such as */bin* and */usr/bin*. The which command makes the same search and gives the absolute path of the program that will be chosen, followed by other occurrences of the file name along the path.

SEE ALSO

man(1).

NAME

who − print list of currently logged in users

SYNOPSIS

who [*file*]

EXAMPLES

who # Print user names, terminals and times

DESCRIPTION

Who prints a list of currently logged in users. For each one, the user name, terminal, and login time is printed. This program gets its information from the file */etc/utmp*, which is updated by init and login. If the file does not exist, neither of these will create it, and *who* will not work. Note that if you decide to create an empty */usr/adm/wtmp* to enable the login accounting, it will grow forever and eventually fill up your disk unless you manually truncate it from time to time. If an optional file name is provided, the logins in that file will be printed.

SEE ALSO

utmp(5).

NAME

whoami − print current user name

SYNOPSIS

whoami

EXAMPLES

whoami # Print user name

DESCRIPTION

In case you forget who you are logged in as, *whoami* will tell you. If you use *su* to become somebody else, *whoami* will give the current effective user.

SEE ALSO

id(1), **who**(1).

NAME

width − force all the lines of a file to a given width

SYNOPSIS

width [−*n* [*infile* [*outfile*]

OPTIONS

−*n* Outline line size

EXAMPLES

width −60 x y # Copy *x* to *y*, force lines to 60 cols

width x # Copy default (80) column lines to *stdout*

DESCRIPTION

The input file is copied to the output file. All lines are forced to a given size (default: 80 columns) by padding with spaces or truncating. Tabs are expanded to spaces.

SEE ALSO

fold(1).

NAME

write − send a message to a logged-in user

SYNOPSIS

write [**−cv**] *user* [*tty*]

OPTIONS

−c Use cbreak mode

−v Verbose mode

EXAMPLES

write ast # Send a message to ast

write ast tty00 # Send a message to ast on tty00

DESCRIPTION

Write lets a user send messages to another logged-in user. Lines typed by the user appear on the other user's screen a line at a time (a character at a time in the case of cbreak mode). The file */usr/adm/wtmp* is searched to determine which tty to send to. If the user is logged onto more than one terminal, the *tty* argument selects the terminal. Type CTRL- D to terminate the command. Use ! as a shell escape.

SEE ALSO

mail(1).

NAME

yacc − an LALR(1) parser generator

SYNOPSIS

yacc [-dlrtv] [-b *file_prefix* **] [-p** *symbol_prefix* **]** *filename*

DESCRIPTION

Yacc reads the grammar specification in the file *filename* and generates an LR(1) parser for it. The parsers consist of a set of LALR(1) parsing tables and a driver routine written in the C programming language. *Yacc* normally writes the parse tables and the driver routine to the file *y.tab.c*.

The following options are available:

-b *file_prefix*
: The **-b** option changes the prefix prepended to the output file names to the string denoted by *file_prefix*. The default prefix is the character *y*.

-d
: The **-d** option causes the header file *y.tab.h* to be written.

-l
: If the **-l** option is not specified, *yacc* will insert #line directives in the generated code. The #line directives let the C compiler relate errors in the generated code to the user's original code. If the **-l** option is specified, *yacc* will not insert the #line directives. #line directives specified by the user will be retained.

-p *symbol_prefix*
: The **-p** option changes the prefix prepended to yacc-generated symbols to the string denoted by *symbol_prefix*. The default prefix is the string *yy*.

-r
: The **-r** option causes *yacc* to produce separate files for code and tables. The code file is named *y.code.c*, and the tables file is named *y.tab.c*.

-t
: The **-t** option changes the preprocessor directives generated by *yacc* so that debugging statements will be incorporated in the compiled code.

-v
: The **-v** option causes a human-readable description of the generated parser to be written to the file *y.output*.

If the environment variable TMPDIR is set, the string denoted by TMPDIR will be used as the name of the directory where the temporary files are created.

FILES

y.code.c
y.tab.c
y.tab.h
y.output
/tmp/yacc.aXXXXXX
/tmp/yacc.tXXXXXX
/tmp/yacc.uXXXXXX

DIAGNOSTICS

If there are rules that are never reduced, the number of such rules is reported on standard error. If there are any LALR(1) conflicts, the number of conflicts is reported on standard error.

NAME

yap, more − yet another pager

SYNOPSIS

yap [**−cnuq**] [*−n*] [*+command*] [name ...]

DESCRIPTION

Yap is a program allowing the user to examine a continuous text one screenful at a time on a video display terminal. It does so by pausing after each screenful, waiting for the user to type a command. The commands are enumerated later. *Yap*'s main feature is, that it can page both forwards and backwards, even when reading from standard input.

The command line options are:

−n An integer which is the size (in lines) of a page (the initial *page-size*.

−c Normally, *yap* will display each page by beginning at the top of the screen and erasing each line just before it displays on it. If your terminal cannot erase a line, *yap* will clear the screen before it displays a page.
This avoids scrolling the screen, making it easier to read while *yap* is writing. The **-c** option causes *yap* to scroll the screen instead of beginning at the top of the screen. This is also done if your terminal cannot either erase a line or clear the screen.

−u Normally, *yap* handles underlining such as produced by nroff in a manner appropriate to the particular terminal: if the terminal can perform underlining well (t.i., the escape sequences for underlining do not occupy space on the screen), *yap* will underline underlined information in the input. The **-u** option supresses this underlining.

−n Normally, *yap* also recognises escape sequences for stand-out mode or underlining mode in the input, and knows how much space these escape sequences will occupy on the screen, so that *yap* will not fold lines erroneously. The **-n** option supresses this pattern matching.

−q This option will cause *yap* to exit only on the "quit" command.

+command
command is taken to be an initial command to *yap*.

Yap uses the *termcap*(5) database to determine the terminal capabilities and the default *page-size*. It examines the $TERM environment variable to identify the terminal type. If $TERM is not set, it defaults to *dumb*. *Yap* Also examines the $TERMCAP environment variable to locate the *termcap*(5) database. If $TERMCAP is not set, it defaults to */etc/termcap*.

Yap looks in the $YAP environment variable to pre-set flags. For instance, if you prefer the **-c** mode of operation, just set the $YAP environment variable to **-c**.

The commands of *yap* can be bound to sequences of keystrokes. The environment variable $YAP-KEYS may contain the bindings in the form of a list of colon-separated 'name=sequence' pairs. The *name* is a short mnemonic for the command, the *sequence* is the sequence of keystrokes to be typed to invoke the command. This sequence may contain a ^X escape, which means control-X, and a \X escape, which means X. The latter can be used to get the characters '^', '\' and ':' in the sequence. There are two keymaps available, the default one and a user-defined one. You can switch between one and the other with the *change keymap* command.

The *yap* commands are described below. The mnemonics for the commands are given in parentheses. The default key sequences (if any) are given after the mnemonic. Every command takes an optional integer argument, which may be typed before the command. Some commands just ignore it. The integer argument is referred to as *i*. Usually, if *i* is not given, it defaults to 1.

visit previous file (*bf*) **P**
Visit the *i*-th previous file given in the command line.

scroll one line up or go to line (*bl*) **^K** or **k**
If *i* is not given, scroll one line up. Otherwise, *i* will be interpreted as a line number. A page starting with the line indicated will then be displayed.

bottom (*bot*) **l** or **$**
: Go to the last line of the input.

display previous page (*bp*) **-**
: Display the previous page, consisting of *i* lines, (or *page-size* lines if no argument is given).

display previous page and set pagesize (*bps*) **Z**
: Display the previous page, consisting of *i* lines, (or *page-size* lines if no argument is given). If *i* is given, the *page-size* is set to *i*.

scroll up (*bs*) **^B**
: Scroll up *i* lines (or *scroll-size* lines if *i* is not given. Initially, the *scroll-size* is 11).

search backwards for pattern (*bse*) **?**
: Search backwards for the *i*-th occurrence of a regular expression which will be prompted for. If there are less than *i* occurrences of the expression, the position in the file remains unchanged. Otherwise, a page is displayed, starting two lines before the place where the expression was found. The user's erase and kill characters may be used to edit the expression. Erasing back past the first character cancels the search command.
Note: Some systems do not have *regex*(3). On those systems, searches are still supported, but regular expressions are not.

skip lines backwards (*bsl*) **S**
: Skip *i* lines backwards and display a page.

skip pages backwards (*bsp*) **F**
: Skip *i* pages backwards and display a page.

scroll up and set scrollsize (*bss*) **b**
: Scroll up *i* lines (or *scroll-size* lines if *i* is not given. If *i* is given, the *scroll-size* is set to *i*.

change key map (*chm*) **^[**
: Change from the current key map to the other (if there is one).

exchange current page and mark (*exg*) **x**
: Set the mark to the current page, and display the previously marked page.

visit next file (*ff*) **N**
: Visit the *i*-th next file given in the command line.

scroll one line down or go to line (*fl*) **^J** or **^M** or **j**
: If *i* is not given, scroll one line down. Otherwise, *i* will be interpreted as a line number. A page starting with the line indicated will then be displayed.

display next page (*fp*) **<space>**
: Display the next page, consisting of *i* lines, (or *page-size* lines if no argument is given).

display next page and set pagesize (*fps*) **z**
: Display the next page, consisting of *i* lines, (or *page-size* lines if no argument is given). If *i* is given, the *page-size* is set to *i*.

scroll down (*fs*) **^D**
: Scroll down *i* lines (or *scroll-size* lines if no argument is given).

search forwards for pattern (*fse*) **/**
: Search forwards for the *i*-th occurrence of a regular expression which will be prompted for. If there are less than *i* occurrences of the expression, the position in the file remains unchanged. Otherwise, a page is displayed, starting two lines before the place where the expression was found. The user's erase and kill characters may be used to edit the expression. Erasing back past the first character cancels the search command.
Note: Some systems do not have *regex*(3). On those systems, searches are still supported, but regular expressions are not.

skip lines forwards (*fsl*) **s**
: Skip *i* lines and display a page.

skip pages forwards (*fsp*) **f**
Skip *i* pages and display a page.

scroll down and set scrollsize (*fss*) **d**
Scroll down *i* lines (or *scroll-size* lines if *i* is not given. If *i* is given, the *scroll-size* is set to *i*.

help (*hlp*) **h**
Give a short description of all commands that are bound to a key sequence.

set a mark (*mar*) **m**
Set a mark on the current page.

repeat last search (*nse*) **n**
Search for the *i*-th occurrence of the last regular expression entered, in the direction of the last search.

repeat last search in other direction (*nsr*) **r**
Search for the *i*-th occurrence of the last regular expression entered, but in the other direction.

quit (*qui*) **Q** or **q**
Exit from *yap*.

redraw (*red*) **^L**
Redraw the current page.

repeat (*rep*) **.**
Repeat the last command. This does not always make sense, so not all commands can be repeated.

shell escape (*shl*) **!**
Invoke the shell with a command that will be prompted for. In the command, the characters '%' and '!' are replaced with the current file name and the previous shell command respectively. The sequences '\%' and '\!' are replaced by '%' and '!' respectively. The user's erase and kill characters can be used to edit the command. Erasing back past the first character cancels the command.

pipe to shell command (*pip*) **|**
Pipe the current input file into a shell command that will be prompted for. The comments given in the description of the shell escape command apply here too.

go to mark (*tom*) **'**
Display the marked page.

top (*top*) **^^**
Display a page starting with the first line of the input.

visit file (*vis*) **e**
Visit a new file. The filename will be prompted for. If you just type a return, the current file is revisited.

write input to a file (*wrf*) **w**
Write the input to a file, whose name will be prompted for.

The commands take effect immediately, i.e., it is not necessary to type a carriage return. Up to the time when the command sequence itself is given, the user may give an interrupt to cancel the command being formed.

AUTHOR

Ceriel J.H. Jacobs

FILES

/etc/termcap Terminal capabilities data base

SEE ALSO

termcap(5), *regex*(3).

BUGS

Yap will find your terminal very stupid and act like it, if it has no way of placing the cursor on the home position, or cannot either erase a line or insert one.

In lines longer than about 2000 characters, a linefeed is silently inserted.

The percentage, given in the prompt when *yap* reads from a file (and knows it), is not always very accurate.

NAME

yes − an endless stream of the same word

SYNOPSIS

yes [*answer*]

EXAMPLES

yes | script # Answer yes to all questions from the script

DESCRIPTION

Yes sends out an endless stream of y’s, each on one line. One uses it to automatically say "yes" to all questions a command may ask. This is useful for commands that ask too many "Are you sure?" questions. The optional argument makes *yes* use *answer* as the word to print instead of a single y character.