NOTICE: A searchable version of this page is now available in http://www-cgi.cs.cmu.edu/htbin/perl-man

---

PERL -- Practical Extraction and Report Language

This web document is a re-organized version of the "perl.1" man page for PERL version 4.

WWW Notes

For quicker access, this document is now mirrored in many locations. The primary copy is still maintained at Carnegie-Mellon University.

By popular demand, I have also produced a single-page version of this document, suitable for printing. I consider it inferior to the original document, and I do not particularly endorse its use. I do request that people **not** attempt to mirror it.

Hypertext formatting by *Robert Stockton*. Comments and criticisms are welcome.

Introduction to Perl

Perl is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). It combines (in the author's opinion, anyway) some of the best features of C, *sed*, *awk*, and *sh*, so people familiar with those languages should have little difficulty with it. (Language historians will also note some vestiges of *csh*, Pascal, and even BASIC-PLUS.) Expression syntax corresponds quite closely to C expression syntax. Unlike most Unix utilities, *perl* does not arbitrarily limit the size of your data -- if you've got the memory, *perl* can slurp in your whole file as a single string. Recursion is of unlimited depth. And the hash tables used by associative arrays grow as necessary to prevent degraded performance. *Perl* uses sophisticated pattern matching techniques to scan large amounts of data very quickly. Although optimized for scanning text, *perl* can also deal with binary data, and can make dbm files look like associative arrays (where dbm is available). Setuid *perl* scripts are safer than C programs through a dataflow tracing mechanism which prevents many stupid security holes. If you have a problem that would ordinarily use *sed* or *awk* or *sh*, but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then *perl* may be for you. There are also translators to turn your *sed* and *awk* scripts into *perl* scripts. OK, enough hype.

Starting a Perl Script

Upon startup, *perl* looks for your script in one of the following places:

1. Specified line by line via **-e** switches on the command line.
2. Contained in the file specified by the first filename on the command line. (Note that systems supporting the #! notation invoke interpreters this way.)
3. Passed in implicitly via standard input. This only works if there are no filename arguments -- to pass arguments to a *stdin* script you must explicitly specify a - for the script name.

After locating your script, *perl* compiles it to an internal form. If the script is syntactically correct, it is executed.

Options

Note: on first reading this section may not make much sense to you. It's here at the front for easy reference.

A single-character option may be combined with the following option, if any. This is particularly useful when invoking a script using the #! construct which only allows one argument. Example:

```
#!/usr/bin/perl -spi.bak        # same as -s -p -i.bak
...
```

Options include:

-0*digits*
: specifies the record separator ($/) as an octal number. If there are no digits, the null character is the separator. Other switches may precede or follow the digits. For example, if you have a version of *find* which can print filenames terminated by the null character, you can say this:

```
find . -name '*.bak' -print0 | perl -n0e unlink
```

The special value 00 will cause Perl to slurp files in paragraph mode. The value 0777 will cause Perl to slurp files whole since there is no legal character with that value.

-a turns on autosplit mode when used with a **-n** or **-p**. An implicit split command to the @F array is done as the first thing inside the implicit while loop produced by the **-n** or **-p**.

```
perl -ane 'print pop(@F), "\n";'
```

is equivalent to

```
while (<>) {
        @F = split(' ');
        print pop(@F), "\n";
}
```

-c causes *perl* to check the syntax of the script and then exit without executing it.

-d runs the script under the perl debugger. See the section on Debugging.

-D*number*

sets debugging flags. To watch how it executes your script, use **-D14**. (This only works if debugging is compiled into your *perl*.) Another nice value is **-D1024**, which lists your compiled syntax tree. And **-D512** displays compiled regular expressions.

-e *commandline*

may be used to enter one line of script. Multiple **-e** commands may be given to build up a multi-line script. If **-e** is given, *perl* will not look for a script filename in the argument list.

-i*extension*

specifies that files processed by the <> construct are to be edited in-place. It does this by renaming the input file, opening the output file by the same name, and selecting that output file as the default for print statements. The extension, if supplied, is added to the name of the old file to make a backup copy. If no extension is supplied, no backup is made. Saying "perl -p -i.bak -e "s/foo/bar/;" ... " is the same as using the script:

```
#!/usr/bin/perl -pi.bak
s/foo/bar/;
```

which is equivalent to

```
#!/usr/bin/perl
while (<>) {
        if ($ARGV ne $oldargv) {
                rename($ARGV, $ARGV . '.bak');
                open(ARGVOUT, ">$ARGV");
                select(ARGVOUT);
                $oldargv = $ARGV;
        }
        s/foo/bar/;
}
continue {
    print;      # this prints to original filename
}
select(STDOUT);
```

except that the **-i** form doesn't need to compare $ARGV to $oldargv to know when the filename has changed. It does, however, use ARGVOUT for the selected filehandle. Note that *STDOUT* is restored as the default output filehandle after the loop.

You can use eof to locate the end of each input file, in case you want to append to each file, or reset line numbering (see example under eof).

-I*directory*

may be used in conjunction with **-P** to tell the C preprocessor where to look for include files. By default /usr/include and /usr/lib/perl are searched.

-l*octnum*

enables automatic line-ending processing. It has two effects: first, it automatically chops the line terminator when used with **-n** or **-p ,** and second, it assigns $\ to have the value of *octnum* so that any print statements will have that line terminator added back on. If *octnum* is omitted, sets $\ to the current value of $/. For instance, to trim lines to 80 columns:

```
perl -lpe 'substr($_, 80) = ""'
```

Note that the assignment $\ = $/ is done when the switch is processed, so the input record separator can be different than the output record separator if the **-l** switch is followed by a **-0** switch:

```
gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

This sets $\ to newline and then sets $/ to the null character.

-n causes *perl* to assume the following loop around your script, which makes it iterate over filename arguments somewhat like "sed -n" or *awk*:

```
while (<>) {
        ...             # your script goes here
}
```

Note that the lines are not printed by default. See **-p** to have lines printed. Here is an efficient way to delete all files older than a week:

```
gfind . -mtime +7 -print | perl -nle 'unlink;'
```

This is faster than using the -exec switch of find because you don't have to start a process on every filename found.

-p causes *perl* to assume the following loop around your script, which makes it iterate over filename arguments somewhat like *sed*:

```
while (<>) {
        ...             # your script goes here
} continue {
        print;
}
```

Note that the lines are printed automatically. To suppress printing use the **-n** switch. A **-p** overrides a **-n** switch.

-P causes your script to be run through the C preprocessor before compilation by *perl*. (Since both comments and cpp directives begin with the # character, you should avoid starting comments with any words recognized by the C preprocessor such as "if", "else" or "define".)

-s enables some rudimentary switch parsing for switches on the command line after the script name but before any filename arguments (or before a --). Any switch found there is removed from @ARGV and sets the corresponding variable in the *perl* script. The following script prints "true" if and only if the script is invoked with a -xyz switch.

```
#!/usr/bin/perl -s
if ($xyz) { print "true\n"; }
```

-S makes *perl* use the PATH environment variable to search for the script (unless the name of the script starts with a slash). Typically this is used to emulate #! startup on machines that don't support #!, in the following manner:

```
#!/usr/bin/perl
eval "exec /usr/bin/perl -S $0 $*"
        if $running_under_some_shell;
```

The system ignores the first line and feeds the script to /bin/sh, which proceeds to try to execute the *perl* script as a shell script. The shell executes the second line as a normal shell command, and thus starts up the *perl* interpreter. On some systems $0 doesn't always contain the full pathname, so the **-S** tells *perl* to search for the script if necessary. After *perl* locates the script, it parses the lines and ignores them because the variable $running_under_some_shell is never true. A better construct than $* would be ${1+"$@"}, which handles embedded spaces and such in the filenames, but doesn't work if the script is being interpreted by csh. In order to start up sh rather than csh, some systems may have to replace the #! line with a line containing just a colon, which will be politely ignored by perl. Other systems can't control that, and need a totally devious construct that will work under any of csh, sh or perl, such as the following:

```
eval '(exit $?0)' && eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
& eval 'exec /usr/bin/perl -S $0 $argv:q'
        if 0;
```

-u causes *perl* to dump core after compiling your script. You can then take this core dump and turn it into an executable file by using the undump program (not supplied). This speeds startup at the expense of some disk space (which you can minimize by stripping the executable). (Still, a "hello world" executable comes out to about 200K on my machine.) If you are going to run your executable as a set-id program then you should probably compile it using taintperl rather than normal perl. If you want to execute a portion of your script before dumping, use the dump operator instead. Note: availability of undump is platform specific and may not be available for a specific port of perl.

-U allows *perl* to do unsafe operations. Currently the only "unsafe" operations are the unlinking of directories while running as superuser, and running setuid programs with fatal taint checks turned into warnings.

-v prints the version and patchlevel of your *perl* executable.

-w prints warnings about identifiers that are mentioned only once, and scalar variables that are used before being set. Also warns about redefined subroutines, and references to undefined filehandles or filehandles opened readonly that you are attempting to write on. Also warns you if you use == on values that don't look like numbers, and if your subroutines recurse more than 100 deep.

-x*directory*

tells *perl* that the script is embedded in a message. Leading garbage will be discarded until the first line that starts with #! and contains the string "perl". Any meaningful switches on that line will be applied (but only one group of switches, as with normal #! processing). If a directory name is specified, Perl will switch to that directory before running the script. The **-x** switch only controls the the disposal of leading garbage. The script must be terminated with __END__ if there is trailing garbage to be ignored (the script can process any or all of the trailing garbage via the DATA filehandle if desired).

Data Types and Objects

Perl has three data types: scalars, arrays of scalars, and associative arrays of scalars. Normal arrays are indexed by number, and associative arrays by string.

The interpretation of operations and values in perl sometimes depends on the requirements of the context around the operation or value. There are three major contexts: string, numeric and array. Certain operations return array values in contexts wanting an array, and scalar values otherwise. (If this is true of an operation it will be mentioned in the documentation for that operation.) Operations which return scalars don't care whether the context is looking for a string or a number, but scalar variables and values are interpreted as strings or numbers as appropriate to the context. A scalar is interpreted as TRUE in the boolean sense if it is not the null string or 0. Booleans returned by operators are 1 for true and 0 or '' (the null string) for false.

There are actually two varieties of null string: defined and undefined. Undefined null strings are returned when there is no real value for something, such as when there was an error, or at end of file, or when you refer to an uninitialized variable or element of an array. An undefined null string may become defined the first time you access it, but prior to that you can use the defined() operator to determine whether the value is defined or not.

References to scalar variables always begin with '$', even when referring to a scalar that is part of an array. Thus:

```
$days               # a simple scalar variable
$days[28]           # 29th element of array @days
$days{'Feb'}        # one value from an associative array
$#days              # last index of array @days
```

but entire arrays or array slices are denoted by '@':

```
@days               # ($days[0], $days[1],\|... $days[n])
@days[3,4,5]        # same as @days[3.\|.5]
@days{'a','c'}      # same as ($days{'a'},$days{'c'})
```

and entire associative arrays are denoted by '%':

```
%days               # (key1, val1, key2, val2 ...)
```

Any of these eight constructs may serve as an lvalue, that is, may be assigned to. (It also turns out that an assignment is itself an lvalue in certain contexts--see examples under s, tr and chop.) Assignment to a scalar evaluates the righthand side in a scalar context, while assignment to an array or array slice evaluates the righthand side in an array context.

You may find the length of array @days by evaluating "$#days", as in *csh*. (Actually, it's not the length of the array, it's the subscript of the last element, since there is (ordinarily) a 0th element.) Assigning to $#days changes the length of the array. Shortening an array by this method does not actually destroy any values. Lengthening an array that was previously shortened recovers the values that were in those elements. You can also gain some measure of efficiency by preextending an array that is going to get big. (You can also extend an array by assigning to an element that is off the end of the array. This differs from assigning to $#whatever in that intervening values are set to null rather than recovered.) You can truncate an array down to nothing by assigning the null list () to it. The following are exactly equivalent

```
@whatever = ();
$#whatever = $[ - 1;
```

If you evaluate an array in a scalar context, it returns the length of the array. The following is always true:

```
scalar(@whatever) == $#whatever - $[ + 1;
```

If you evaluate an associative array in a scalar context, it returns a value which is true if and only if the array contains any elements. (If there are any elements, the value returned is a string consisting of the number of used buckets and the number of allocated buckets, separated by a slash.)

Multi-dimensional arrays are not directly supported, but see the discussion of the $; variable later for a means of emulating multiple subscripts with an associative array. You could also write a subroutine to turn multiple subscripts into a single subscript.

Every data type has its own namespace. You can, without fear of conflict, use the same name for a scalar variable, an array, an associative array, a filehandle, a subroutine name, and/or a label. Since variable and array references always start with '$', '@', or '%', the "reserved" words aren't in fact reserved with respect to variable names. (They ARE reserved with respect to labels and filehandles, however, which don't have an initial special character. Hint: you could say open(LOG,'logfile') rather than open(log,'logfile'). Using uppercase filehandles also improves readability and protects you from conflict with future reserved words.) Case IS significant--"FOO", "Foo" and "foo" are all different names. Names which start with a letter may also contain digits and underscores. Names which do not start with a letter are limited to one character, e.g. "$%" or "$$". (Most of the one character names have a predefined significance to *perl*. More later.)

Numeric literals are specified in any of the usual floating point or integer formats:

```
12345
12345.67
.23E-10
0xffff      # hex
0377        # octal
4_294_967_296
```

String literals are delimited by either single or double quotes. They work much like shell quotes: double-quoted string literals are subject to backslash and variable substitution; single-quoted strings are not (except for \' and \e). The usual backslash rules apply for making characters such as newline, tab, etc., as well as some more exotic forms:

```
\t              tab
\n              newline
\r              return
\f              form feed
\b              backspace
\a              alarm (bell)
\e              escape
\033            octal char
\x1b            hex char
\c[             control char
```

```
\l              lowercase next char
\u              uppercase next char
\L              lowercase till \E
\U              uppercase till \E
\E              end case modification
```

You can also embed newlines directly in your strings, i.e. they can end on a different line than they begin. This is nice, but if you forget your trailing quote, the error will not be reported until *perl* finds another line containing the quote character, which may be much further on in the script. Variable substitution inside strings is limited to scalar variables, normal array values, and array slices. (In other words, identifiers beginning with $ or @, followed by an optional bracketed expression as a subscript.) The following code segment prints out "The price is $100."

```
$Price = '$100';                    # not interpreted
print "The price is $Price.\n";     # interpreted
```

Note that you can put curly brackets around the identifier to delimit it from following alphanumerics. Also note that a single quoted string must be separated from a preceding word by a space, since single quote is a valid character in an identifier (see Packages).

Two special literals are __LINE__ and __FILE__, which represent the current line number and filename at that point in your program. They may only be used as separate tokens; they will not be interpolated into strings. In addition, the token __END__ may be used to indicate the logical end of the script before the actual end of file. Any following text is ignored, but may be read via the DATA filehandle. (The DATA filehandle may read data only from the main script, but not from any required file or evaluated string.) The two control characters ^D and ^Z are synonyms for __END__.

A word that doesn't have any other interpretation in the grammar will be treated as if it had single quotes around it. For this purpose, a word consists only of alphanumeric characters and underline, and must start with an alphabetic character. As with filehandles and labels, a bare word that consists entirely of lowercase letters risks conflict with future reserved words, and if you use the **-w** switch, Perl will warn you about any such words.

Array values are interpolated into double-quoted strings by joining all the elements of the array with the delimiter specified in the $" variable, space by default. (Since in versions of perl prior to 3.0 the @ character was not a metacharacter in double-quoted strings, the interpolation of @array, $array[EXPR], @array[LIST], $array{EXPR}, or @array{LIST} only happens if array is referenced elsewhere in the program or is predefined.) The following are equivalent:

```
$temp = join($",@ARGV);
system "echo $temp";

system "echo @ARGV";
```

Within search patterns (which also undergo double-quotish substitution) there is a bad ambiguity: Is /$foo[bar]/ to be interpreted as /${foo}[bar]/ (where [bar] is a character class for the regular expression) or as /${foo[bar]}/ (where [bar] is the subscript to array @foo)? If @foo doesn't otherwise exist, then it's obviously a character class. If @foo exists, perl takes a good guess about [bar], and is almost always right. If it does guess wrong, or if you're just plain paranoid, you can force the correct interpretation with curly brackets as above.

A line-oriented form of quoting is based on the shell here-is syntax. Following a << you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item. The terminating string may be either an identifier (a word), or some quoted text. If quoted, the type of quotes you use determines the treatment of the text, just as in regular quoting. An unquoted identifier works like double quotes. There must be no space between the << and the identifier. (If you put a space it will be treated as a null identifier, which is valid, and matches the first blank line--see Merry Christmas example below.) The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

```
        print <<EOF;            # same as above
The price is $Price.
EOF

        print <<"EOF";          # same as above
The price is $Price.
EOF

        print << x 10;          # null identifier is delimiter
Merry Christmas!

        print <<`EOC`;          # execute commands
echo hi there
echo lo there
EOC

        print <<foo, <<bar;     # you can stack them
I said foo.
foo
I said bar.
bar
```

Array literals are denoted by separating individual values by commas, and enclosing the list in parentheses:

```
        (LIST)
```

In a context not requiring an array value, the value of the array literal is the value of the final element, as in the C comma operator. For example,

```
    @foo = ('cc', '-E', $bar);
```

assigns the entire array value to array foo, but

```
    $foo = ('cc', '-E', $bar);
```

assigns the value of variable bar to variable foo. Note that the value of an actual array in a scalar context is the length of the array; the following assigns to $foo the value 3:

```
    @foo = ('cc', '-E', $bar);
    $foo = @foo;                # $foo gets 3
```

You may have an optional comma before the closing parenthesis of an array literal, so that you can say:

```
    @foo = (
        1,
```

```
        2,
        3,
    );
```

When a LIST is evaluated, each element of the list is evaluated in an array context, and the resulting array value is interpolated into LIST just as if each individual element were a member of LIST. Thus arrays lose their identity in a LIST--the list (@foo,@bar,&SomeSub) contains all the elements of @foo followed by all the elements of @bar, followed by all the elements returned by the subroutine named SomeSub.

A list value may also be subscripted like a normal array. Examples:

```
        $time = (stat($file))[8];       # stat returns array value
        $digit = ('a','b','c','d','e','f')[$digit-10];
        return (pop(@foo),pop(@foo))[0];
```

Array lists may be assigned to if and only if each element of the list is an lvalue:

```
    ($a, $b, $c) = (1, 2, 3);

    ($map{'red'}, $map{'blue'}, $map{'green'}) = (0x00f, 0x0f0, 0xf00);
```

The final element may be an array or an associative array:

```
    ($a, $b, @rest) = split;
    local($a, $b, %rest) = @_;
```

You can actually put an array anywhere in the list, but the first array in the list will soak up all the values, and anything after it will get a null value. This may be useful in a local().

An associative array literal contains pairs of values to be interpreted as a key and a value:

```
    # same as map assignment above
    %map = ('red',0x00f,'blue',0x0f0,'green',0xf00);
```

Array assignment in a scalar context returns the number of elements produced by the expression on the right side of the assignment:

```
        $x = (($foo,$bar) = (3,2,1));   # set $x to 3, not 2
```

There are several other pseudo-literals that you should know about. If a string is enclosed by backticks (grave accents), it first undergoes variable substitution just like a double quoted string. It is then interpreted as a command, and the output of that command is the value of the pseudo-literal, like in a shell. In a scalar context, a single string consisting of all the output is returned. In an array context, an array of values is returned, one for each line of output. (You can set $/ to use a different line terminator.) The command is executed each time the pseudo-literal is evaluated. The status value of the command is returned in $? (see Predefined Names for the interpretation of $?). Unlike in *csh*, no translation is done on the return data--newlines remain newlines. Unlike in any of the shells, single quotes do not hide variable names in the command from interpretation. To pass a $ through to the shell you need to hide it with a backslash.

Evaluating a filehandle in angle brackets yields the next line from that file (newline included, so it's never false until EOF, at which time an undefined value is returned). Ordinarily you must assign that value to a variable, but there is one situation where an automatic assignment happens. If (and only if) the input symbol is the only thing inside the conditional of a *while* loop, the value is automatically assigned to the variable "$_". (This may seem like an odd thing to you, but you'll use the construct in almost every *perl* script you write.) Anyway, the following lines are equivalent to each other:

```
while ($_ = <STDIN>) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while $_ = <STDIN>;
print while <STDIN>;
```

The filehandles *STDIN* , *STDOUT* and *STDERR* are predefined. (The filehandles *stdin*, *stdout* and *stderr* will also work except in packages, where they would be interpreted as local identifiers rather than global.) Additional filehandles may be created with the *open* function.

If a <FILEHANDLE> is used in a context that is looking for an array, an array consisting of all the input lines is returned, one line per array element. It's easy to make a LARGE data space this way, so use with care.

The null filehandle <> is special and can be used to emulate the behavior of *sed* and *awk*. Input from <> comes either from standard input, or from each file listed on the command line. Here's how it works: the first time <> is evaluated, the ARGV array is checked, and if it is null, $ARGV[0] is set to '-', which when opened gives you standard input. The ARGV array is then processed as a list of filenames. The loop

```
while (<>) {
        ...                     # code for each line
}
```

is equivalent to the following Perl-like pseudo code:

```
unshift(@ARGV, '-') if $#ARGV < $[;
while ($ARGV = shift) {
        open(ARGV, $ARGV);
        while (<ARGV>) {
                ...             # code for each line
        }
}
```

except that it isn't as cumbersome to say, and will actually work. It really does shift array ARGV and put the current filename into variable ARGV. It also uses filehandle ARGV internally--<> is just a synonym for <ARGV>, which is magical. (The pseudo code above doesn't work because it treats <ARGV> as non-magical.)

You can modify @ARGV before the first <> as long as the array ends up containing the list of filenames you really want. Line numbers ($.) continue as if the input was one big happy file. (But see example under eof for how to reset line numbers on each file.)

If you want to set @ARGV to your own list of files, go right ahead. If you want to pass switches into your script, you can put a loop on the front like this:

```
while ($_ = $ARGV[0], /^-/) {
        shift;
    last if /^--$/;
        /^-D(.*)/ && ($debug = $1);
        /^-v/ && $verbose++;
        ...             # other switches
}
while (<>) {
        ...             # code for each line
}
```

The <> symbol will return FALSE only once. If you call it again after this it will assume you are processing another @ARGV list, and if you haven’t set @ARGV, will input from *STDIN*.

If the string inside the angle brackets is a reference to a scalar variable (e.g. <$foo>), then that variable contains the name of the filehandle to input from.

If the string inside angle brackets is not a filehandle, it is interpreted as a filename pattern to be globbed, and either an array of filenames or the next filename in the list is returned, depending on context. One level of $ interpretation is done first, but you can’t say <$foo> because that’s an indirect filehandle as explained in the previous paragraph. You could insert curly brackets to force interpretation as a filename glob: <${foo}>.

Example:

```
while (<*.c>) {
        chmod 0644, $_;
}
```

is equivalent to

```
open(foo, "echo *.c | tr -s ’ \t\r\f’ ’\\012\\012\\012\\012’");
while (<foo>) {
        chop;
        chmod 0644, $_;
}
```

In fact, it’s currently implemented that way. (Which means it will not work on filenames with spaces in them unless you have /bin/csh on your machine.) Of course, the shortest way to do the above is:

```
chmod 0644, <*.c>;
```

Syntax

A *perl* script consists of a sequence of declarations and commands. The only things that need to be declared in *perl* are report formats and subroutines. See the sections below for more information on those declarations. All uninitialized user-created objects are assumed to start with a null or 0 value until they are defined by some explicit operation such as assignment. The sequence of commands is executed just once, unlike in *sed* and *awk* scripts, where the sequence of commands is executed for each input line. While this means that you must explicitly loop over the lines of your input file (or files), it also means you have much more control over which files and which lines you look at. (Actually, I’m lying--it is possible to do an implicit loop with either the **-n** or **-p** switch.)

A declaration can be put anywhere a command can, but has no effect on the execution of the primary sequence of commands--declarations all take effect at compile time. Typically all the declarations are put at the beginning or the end of the script.

Perl is, for the most part, a free-form language. (The only exception to this is format declarations, for fairly obvious reasons.) Comments are indicated by the # character, and extend to the end of the line. If you attempt to use /* */ C comments, it will be interpreted either as division or pattern matching, depending on the context. So don't do that.

Compound statements

In *perl*, a sequence of commands may be treated as one command by enclosing it in curly brackets. We will call this a BLOCK.

The following compound commands may be used to control flow:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK
LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (ARRAY) BLOCK
LABEL BLOCK continue BLOCK
```

Note that, unlike C and Pascal, these are defined in terms of BLOCKs, not statements. This means that the curly brackets are *required*--no dangling statements allowed. If you want to write conditionals without curly brackets there are several other ways to do it. The following all do the same thing:

```
if (!open(foo)) { die "Can't open $foo: $!"; }
die "Can't open $foo: $!" unless open(foo);
open(foo) || die "Can't open $foo: $!"; # foo or bust!
open(foo) ? 'hi mom' : die "Can't open $foo: $!";
                        # a bit exotic, that last one
```

The *if* statement is straightforward. Since BLOCKs are always bounded by curly brackets, there is never any ambiguity about which *if* an *else* goes with. If you use *unless* in place of *if*, the sense of the test is reversed.

The *while* statement executes the block as long as the expression is true (does not evaluate to the null string or 0). The LABEL is optional, and if present, consists of an identifier followed by a colon. The LABEL identifies the loop for the loop control statements *next*, *last*, and *redo*. If there is a *continue* BLOCK, it is always executed just before the conditional is about to be evaluated again, similarly to the third part of a *for* loop in C. Thus it can be used to increment a loop variable, even when the loop has been continued via the *next* statement (similar to the C "continue" statement).

If the word *while* is replaced by the word *until*, the sense of the test is reversed, but the conditional is still tested before the first iteration.

In either the *if* or the *while* statement, you may replace "(EXPR)" with a BLOCK, and the conditional is true if the value of the last command in that block is true.

The *for* loop works exactly like the corresponding *while* loop:

```
for ($i = 1; $i < 10; $i++) {
        ...
}
```

is the same as

```
$i = 1;
while ($i < 10) {
        ...
} continue {
        $i++;
}
```

The foreach loop iterates over a normal array value and sets the variable VAR to be each element of the array in turn. The variable is implicitly local to the loop, and regains its former value upon exiting the loop. The "foreach" keyword is actually identical to the "for" keyword, so you can use "foreach" for readability or "for" for brevity. If VAR is omitted, $_ is set to each value. If ARRAY is an actual array (as opposed to an expression returning an array value), you can modify each element of the array by modifying VAR inside the loop. Examples:

```
for (@ary) { s/foo/bar/; }

foreach $elem (@elements) {
        $elem *= 2;
}

for ((10,9,8,7,6,5,4,3,2,1,'BOOM')) {
        print $_, "\n"; sleep(1);
}

for (1..15) { print "Merry Christmas\n"; }

foreach $item (split(/:[\\\n:]*/, $ENV{'TERMCAP'})) {
        print "Item: $item\n";
}
```

The BLOCK by itself (labeled or not) is equivalent to a loop that executes once. Thus you can use any of the loop control statements in it to leave or restart the block. The *continue* block is optional. This construct is particularly nice for doing case structures.

```
foo: {
        if (/^abc/) { $abc = 1; last foo; }
        if (/^def/) { $def = 1; last foo; }
        if (/^xyz/) { $xyz = 1; last foo; }
        $nothing = 1;
}
```

There is no official switch statement in perl, because there are already several ways to write the equivalent. In addition to the above, you could write

```
foo: {
        $abc = 1, last foo  if /^abc/;
        $def = 1, last foo  if /^def/;
        $xyz = 1, last foo  if /^xyz/;
        $nothing = 1;
}
```

or

```
foo: {
        /^abc/ && do { $abc = 1; last foo; };
        /^def/ && do { $def = 1; last foo; };
        /^xyz/ && do { $xyz = 1; last foo; };
        $nothing = 1;
}
```

or

```
foo: {
        /^abc/ && ($abc = 1, last foo);
        /^def/ && ($def = 1, last foo);
        /^xyz/ && ($xyz = 1, last foo);
        $nothing = 1;
}
```

or even

```
if (/^abc/)
        { $abc = 1; }
elsif (/^def/)
        { $def = 1; }
elsif (/^xyz/)
        { $xyz = 1; }
else
        {$nothing = 1;}
```

As it happens, these are all optimized internally to a switch structure, so perl jumps directly to the desired statement, and you needn't worry about perl executing a lot of unnecessary statements when you have a string of 50 elsifs, as long as you are testing the same simple scalar variable using ==, eq, or pattern matching as above. (If you're curious as to whether the optimizer has done this for a particular case statement, you can use the -D1024 switch to list the syntax tree before execution.)

Simple statements

The only kind of simple statement is an expression evaluated for its side effects. Every simple statement must be terminated with a semicolon, unless it is the final statement in a block, in which case the semicolon is optional. (Semicolon is still encouraged there if the block takes up more than one line).

Any simple statement may optionally be followed by a single modifier, just before the terminating semicolon. The possible modifiers are:

```
if EXPR
unless EXPR
while EXPR
```

```
until EXPR
```

The *if* and *unless* modifiers have the expected semantics. The *while* and *until* modifiers also have the expected semantics (conditional evaluated first), except when applied to a do-BLOCK or a do-SUBROUTINE command, in which case the block executes once before the conditional is evaluated. This is so that you can write loops like:

```
do {
        $_ = <STDIN>;
        ...
} until $_ eq ".\n";
```

(See the *do* operator below. Note also that the loop control commands described later will NOT work in this construct, since modifiers don't take loop labels. Sorry.)

Expressions

This section has been split up into subsections based upon functionality, loosely based upon the *Perl Reference Guide* by Johan Vromans. This is a departure from the original *Perl* manual, which should simplify on-line browsing. If you have problems with this organization, blame me (rgs@cs.cmu.edu) rather than Larry.

Expression types:

- Flow control operations
- Operators (including File Test operators)
- Arithmetic functions
- Conversion functions
- String functions
- Array and list functions
- File operations
- Directory reading
- I/O operations
- Search and modification operations
- System interaction routines
- IPC and Networking operations
- Miscellaneous operations
- System information

Flow Control Operations

do BLOCK
: Returns the value of the last command in the sequence of commands indicated by BLOCK. When modified by a loop modifier, executes the BLOCK once before testing the loop condition. (On other statements the loop modifiers test the conditional first.)

do SUBROUTINE (LIST)

Executes a SUBROUTINE declared by a *sub* declaration, and returns the value of the last expression evaluated in SUBROUTINE. If there is no subroutine by that name, produces a fatal error. (You may use the "defined" operator to determine if a subroutine exists.) If you pass arrays as part of LIST you may wish to pass the length of the array in front of each array. (See the section on subroutines later on.) The parentheses are required to avoid confusion with the "do EXPR" form.

SUBROUTINE may also be a single scalar variable, in which case the name of the subroutine to execute is taken from the variable.

As an alternate (and preferred) form, you may call a subroutine by prefixing the name with an ampersand: &foo(@args). If you aren't passing any arguments, you don't have to use parentheses. If you omit the parentheses, no @_ array is passed to the subroutine. The & form is also used to specify subroutines to the defined and undef operators:

```
        if (defined &$var) { &$var($parm); undef &$var; }
```

do EXPR

Uses the value of EXPR as a filename and executes the contents of the file as a *perl* script. Its primary use is to include subroutines from a *perl* subroutine library.

```
        do 'stat.pl';

is just like

        eval \`cat stat.pl\`;
```

except that it's more efficient, more concise, keeps track of the current filename for error messages, and searches all the **-I** libraries if the file isn't in the current directory (see also the @INC array in Predefined Names). It's the same, however, in that it does reparse the file every time you call it, so if you are going to use the file inside a loop you might prefer to use -P and #include, at the expense of a little more startup time. (The main problem with #include is that cpp doesn't grok # comments--a workaround is to use ";#" for standalone comments.) Note that the following are NOT equivalent:

```
        do $foo;        # eval a file
        do $foo();      # call a subroutine
```

Note that inclusion of library routines is better done with the "require" operator.

goto LABEL

Finds the statement labeled with LABEL and resumes execution there. Currently you may only go to statements in the main body of the program that are not nested inside a do {} construct. This statement is not implemented very efficiently, and is here only to make the *sed*-to- perl translator easier. I may change its semantics at any time, consistent with support for translated *sed* scripts. Use it at your own risk. Better yet, don't use it at all.

last
last LABEL
last *last* command is like the *break* statement in C (as used in loops); it immediately exits the loop in

question. If the LABEL is omitted, the command refers to the innermost enclosing loop. The *continue* block, if any, is not executed:

```
line: while (<STDIN>) {
        last line if /^$/;      # exit when done with header
        ...
}
```

next LABEL
next The *next* command is like the *continue* statement in C; it starts the next iteration of the loop:

```
line: while (<STDIN>) {
        next line if /^#/;      # discard comments
        ...
}
```

Note that if there were a *continue* block on the above, it would get executed even on discarded lines. If the LABEL is omitted, the command refers to the innermost enclosing loop.

redo
redo LABEL
redo The *redo* command restarts the loop block without evaluating the conditional again. The *continue* block, if any, is not executed. If the LABEL is omitted, the command refers to the innermost enclosing loop. This command is normally used by programs that want to lie to themselves about what was just input:

```
# a simpleminded Pascal comment stripper
# (warning: assumes no { or } in strings)
line: while (<STDIN>) {
        while (s|({.*}.*){.*}|$1 |) {}
        s|{.*}| |;
        if (s|{.*| |) {
                $front = $_;
                while (<STDIN>) {
                        if (/}/) {      # end of comment?
                                s|^|$front{|;
                                redo line;
                        }
                }
        }
        print;
}
```

return LIST
Returns from a subroutine with the value specified. (Note that a subroutine can automatically return the value of the last expression evaluated. That's the preferred method--use of an explicit *return* is a bit slower.)

Operators

Since *perl* expressions work almost exactly like C expressions, only the differences will be mentioned here.

Here’s what *perl* has that C doesn’t:

** The exponentiation operator.

**= The exponentiation assignment operator.

() The null list, used to initialize an array to null.

. Concatenation of two strings.

.= The concatenation assignment operator.

eq String equality (== is numeric equality). For a mnemonic just think of "eq" as a string. (If you are used to the *awk* behavior of using == for either string or numeric equality based on the current form of the comparands, beware! You must be explicit here.)

ne String inequality (!= is numeric inequality).

lt String less than.

gt String greater than.

le String less than or equal.

ge String greater than or equal.

cmp String comparison, returning -1, 0, or 1.

<=> Numeric comparison, returning -1, 0, or 1.

=~ Certain operations search or modify the string "$_" by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or translation. The left argument is what is supposed to be searched, substituted, or translated instead of the default "$_". The return value indicates the success of the operation. (If the right argument is an expression other than a search pattern, substitution, or translation, it is interpreted as a search pattern at run time. This is less efficient than an explicit search, since the pattern must be compiled every time the expression is evaluated.) The precedence of this operator is lower than unary minus and autoincrement/decrement, but higher than everything else.

!~ Just like =~ except the return value is negated.

x The repetition operator. Returns a string consisting of the left operand repeated the number of times specified by the right operand. In an array context, if the left operand is a list in parens, it repeats the list.

```
print '-' x 80;          # print row of dashes
print '-' x80;           # illegal, x80 is identifier
```

```
print "\t" x ($tab/8), ' ' x ($tab%8);  # tab over

@ones = (1) x 80;               # an array of 80 1's
@ones = (5) x @ones;            # set all elements to 5
```

x= The repetition assignment operator. Only works on scalars.

.. The range operator, which is really two different operators depending on the context. In an array context, returns an array of values counting (by ones) from the left value to the right value. This is useful for writing "for (1..10)" loops and for doing slice operations on arrays.

In a scalar context, .. returns a boolean value. The operator is bistable, like a flip-flop, and emulates the line-range (comma) operator of sed, awk, and various editors. Each .. operator maintains its own boolean state. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, AFTER which the range operator becomes false again. (It doesn't become false till the next time the range operator is evaluated. It can test the right operand and become false on the same evaluation it became true (as in awk), but it still returns true once. If you don't want it to test the right operand till the next evaluation (as in sed), use three dots (...) instead of two.) The right operand is not evaluated while the operator is in the "false" state, and the left operand is not evaluated while the operator is in the "true" state. The precedence is a little lower than || and &&. The value returned is either the null string for false, or a sequence number (beginning with 1) for true. The sequence number is reset for each range encountered. The final sequence number in a range has the string 'E0' appended to it, which doesn't affect its numeric value, but gives you something to search for if you want to exclude the endpoint. You can exclude the beginning point by waiting for the sequence number to be greater than 1. If either operand of scalar .. is static, that operand is implicitly compared to the $. variable, the current line number.

Examples:

```
As a scalar operator:
    if (101 .. 200) { print; }  # print 2nd hundred lines

    next line if (1 .. /^$/);   # skip header lines

    s/^/> / if (/^$/ .. eof()); # quote body

As an array operator:
    for (101 .. 200) { print; } # print $_ 100 times

    @foo = @foo[$[ .. $#foo];   # an expensive no-op
    @foo = @foo[$#foo-4 .. $#foo];      # slice last 5 items
```

-x A file test. This unary operator takes one argument, either a filename or a filehandle, and tests the associated file to see if something is true about it. If the argument is omitted, tests $_, except for -t, which tests *STDIN*. It returns 1 for true and '' for false, or the undefined value if the file doesn't exist. Precedence is higher than logical and relational operators, but lower than arithmetic operators. The operator may be any of:

```
-r      File is readable by effective uid/gid.
-w      File is writable by effective uid/gid.
-x      File is executable by effective uid/gid.
-o      File is owned by effective uid.
```

```
-R      File is readable by real uid/gid.
-W      File is writable by real uid/gid.
-X      File is executable by real uid/gid.
-O      File is owned by real uid.
-e      File exists.
-z      File has zero size.
-s      File has non-zero size (returns size).
-f      File is a plain file.
-d      File is a directory.
-l      File is a symbolic link.
-p      File is a named pipe (FIFO).
-S      File is a socket.
-b      File is a block special file.
-c      File is a character special file.
-u      File has setuid bit set.
-g      File has setgid bit set.
-k      File has sticky bit set.
-t      Filehandle is opened to a tty.
-T      File is a text file.
-B      File is a binary file (opposite of -T).
-M      Age of file in days when script started.
-A      Same for access time.
-C      Same for inode change time.
```

The interpretation of the file permission operators -r, -R, -w, -W, -x and -X is based solely on the mode of the file and the uids and gids of the user. There may be other reasons you can't actually read, write or execute the file. Also note that, for the superuser, -r, -R, -w and -W always return 1, and -x and -X return 1 if any execute bit is set in the mode. Scripts run by the superuser may thus need to do a stat() in order to determine the actual mode of the file, or temporarily set the uid to something else.

Example:

```
while (<>) {
        chop;
        next unless -f $_;      # ignore specials
        ...
}
```

Note that -s/a/b/ does not do a negated substitution. Saying -exp($foo) still works as expected, however--only single letters following a minus are interpreted as file tests.

The -T and -B switches work as follows. The first block or so of the file is examined for odd characters such as strange control codes or metacharacters. If too many odd characters (>10%) are found, it's a -B file, otherwise it's a -T file. Also, any file containing null in the first block is considered a binary file. If -T or -B is used on a filehandle, the current stdio buffer is examined rather than the first block. Both -T and -B return TRUE on a null file, or a file at EOF when testing a filehandle.

If any of the file tests (or either stat operator) are given the special filehandle consisting of a solitary underline, then the stat structure of the previous file test (or stat operator) is used, saving a system call. (This doesn't work with -t, and you need to remember that lstat and -l will leave values in the stat structure for the symbolic link, not the real file.)

Example:

```
print "Can do.\n" if -r $a || -w _ || -x _;

stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;
print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;
```

Here is what C has that *perl* doesn't:

unary &
: Address-of operator.

unary *
: Dereference-address operator.

(TYPE)
: Type casting operator.

Like C, *perl* does a certain amount of expression evaluation at compile time, whenever it determines that all of the arguments to an operator are static and have no side effects. In particular, string concatenation happens at compile time between literals that don't do variable substitution. Backslash interpretation also happens at compile time. You can say

```
'Now is the time for all' . "\n" .
'good men to come to.'
```

and this all reduces to one string internally.

The autoincrement operator has a little extra built-in magic to it. If you increment a variable that is numeric, or that has ever been used in a numeric context, you get a normal increment. If, however, the variable has only been used in string contexts since it was set, and has a value that is not null and matches the pattern /^[a-zA-Z]*[0-9]*$/, the increment is done as a string, preserving each character within its range, with carry:

```
print ++($foo = '99');  # prints '100'
print ++($foo = 'a0');  # prints 'a1'
print ++($foo = 'Az');  # prints 'Ba'
print ++($foo = 'zz');  # prints 'aaa'
```

The autodecrement is not magical.

The range operator (in an array context) makes use of the magical autoincrement algorithm if the minimum and maximum are strings. You can say @alphabet = ('A' .. 'Z'); to get all the letters of the alphabet, or $hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15]; to get a hexadecimal digit, or @z2 = ('01' .. '31'); print @z2[$mday]; to get dates with leading zeros. (If the final value specified is not in the sequence that the magical increment would produce, the sequence goes until the next value would be longer than the final value specified.)

The || and && operators differ from C's in that, rather than returning 0 or 1, they return the last value evaluated. Thus, a portable way to find out the home directory might be:

```
$home = $ENV{'HOME'} || $ENV{'LOGDIR'} ||
    (getpwuid($<))[7] || die "You're homeless!\n";
```

Along with the literals and variables mentioned earlier, the operations in the following section can serve as terms in an expression. Some of these operations take a LIST as an argument. Such a list can consist of any combination of scalar arguments or array values; the array values will be included in the list as if each individual element were interpolated at that point in the list, forming a longer single-dimensional array value. Elements of the LIST should be separated by commas. If an operation is listed both with and without parentheses around its arguments, it means you can either use it as a unary operator or as a function call. To use it as a function call, the next token on the same line must be a left parenthesis. (There may be intervening white space.) Such a function then has highest precedence, as you would expect from a function. If any token other than a left parenthesis follows, then it is a unary operator, with a precedence depending only on whether it is a LIST operator or not. LIST operators have lowest precedence. All other unary operators have a precedence greater than relational operators but less than arithmetic operators. See the section on Precedence.

For operators that can be used in either a scalar or array context, failure is generally indicated in a scalar context by returning the undefined value, and in an array context by returning the null list. Remember though that **there is no general rule for converting a list into a scalar**. Each operator decides which sort of scalar it would be most appropriate to return. Some operators return the length of the list that would have been returned in an array context. Some operators return the first value in the list. Some operators return the last value in the list. Some operators return a count of successful operations. In general, they do what you want, unless you want consistency.

Arithmetic Functions

atan2(Y,X)

Returns the arctangent of Y/X in the range -PI to PI.

cos(EXPR)
cos EXPR

Returns the cosine of EXPR (expressed in radians). If EXPR is omitted takes cosine of $_.

exp(EXPR)
exp EXPR

Returns *e* to the power of EXPR. If EXPR is omitted, gives exp($_).

int(EXPR)
int EXPR

Returns the integer portion of EXPR. If EXPR is omitted, uses $_.

log(EXPR)

log EXPR
Returns logarithm (base *e*) of EXPR. If EXPR is omitted, returns log of $_.

rand(EXPR)
rand EXPR
rand Returns a random fractional number between 0 and the value of EXPR. (EXPR should be positive.) If EXPR is omitted, returns a value between 0 and 1. See also srand().

sin(EXPR)
sin EXPR
Returns the sine of EXPR (expressed in radians). If EXPR is omitted, returns sine of $_.

sqrt(EXPR)
sqrt EXPR
Return the square root of EXPR. If EXPR is omitted, returns square root of $_.

srand(EXPR)
srand EXPR
Sets the random number seed for the *rand* operator. If EXPR is omitted, does srand(time).

Conversion Functions

gmtime(EXPR)
gmtime EXPR
Converts a time as returned by the time function to a 9-element array with the time analyzed for the Greenwich timezone. Typically used as follows:

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                              gmtime(time);
```

All array elements are numeric, and come straight out of a struct tm. In particular this means that $mon has the range 0..11 and $wday has the range 0..6. If EXPR is omitted, does gmtime(time).

hex(EXPR)
hex EXPR
Returns the decimal value of EXPR interpreted as an hex string. (To interpret strings that might start with 0 or 0x see oct().) If EXPR is omitted, uses $_.

localtime(EXPR)
localtime EXPR
Converts a time as returned by the time function to a 9-element array with the time analyzed for the local timezone. Typically used as follows:

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                              localtime(time);
```

All array elements are numeric, and come straight out of a struct tm. In particular this means that

$mon has the range 0..11 and $wday has the range 0..6. If EXPR is omitted, does localtime(time).

oct(EXPR)
oct EXPR

Returns the decimal value of EXPR interpreted as an octal string. (If EXPR happens to start off with 0x, interprets it as a hex string instead.) The following will handle decimal, octal and hex in the standard notation:

```
$val = oct($val) if $val =~/^0/;
```

If EXPR is omitted, uses $_.

ord(EXPR)
ord EXPR

Returns the numeric ascii value of the first character of EXPR. If EXPR is omitted, uses $_.

pack(TEMPLATE,LIST)

Takes an array or list of values and packs it into a binary structure, returning the string containing the structure. The TEMPLATE is a sequence of characters that give the order and type of values, as follows:

```
A       An ascii string, will be space padded.
a       An ascii string, will be null padded.
c       A signed char value.
C       An unsigned char value.
s       A signed short value.
S       An unsigned short value.
i       A signed integer value.
I       An unsigned integer value.
l       A signed long value.
L       An unsigned long value.
n       A short in "network" order.
N       A long in "network" order.
f       A single-precision float in the native format.
d       A double-precision float in the native format.
p       A pointer to a string.
v       A short in "VAX" (little-endian) order.
V       A long in "VAX" (little-endian) order.
x       A null byte.
X       Back up a byte.
@       Null fill to absolute position.
u       A uuencoded string.
b       A bit string (ascending bit order, like vec()).
B       A bit string (descending bit order).
h       A hex string (low nybble first).
H       A hex string (high nybble first).
```

Each letter may optionally be followed by a number which gives a repeat count. With all types except "a", "A", "b", "B", "h" and "H", the pack function will gobble up that many values from the LIST. A * for the repeat count means to use however many items are left. The "a" and "A" types gobble just one value, but pack it as a string of length count, padding with nulls or spaces as necessary. (When unpacking, "A" strips trailing spaces and nulls, but "a" does not.) Likewise, the "b" and "B" fields pack a string that many bits long. The "h" and "H" fields pack a string that many nybbles long. Real numbers (floats and doubles) are in the native machine format only; due to the multiplicity of floating formats around, and the lack of a standard "network" representation, no facility for interchange has been made. This means that packed floating point data written on one machine may not be readable on another - even if both use IEEE floating point arithmetic (as the endian-ness of the memory representation is not part of the IEEE spec). Note that perl uses doubles internally for all numeric calculation, and converting from double -> float -> double will lose precision (i.e. unpack("f", pack("f", $foo)) will not in general equal $foo).

Examples:

```
$foo = pack("cccc",65,66,67,68);
# foo eq "ABCD"
$foo = pack("c4",65,66,67,68);
# same thing

$foo = pack("ccxxcc",65,66,67,68);
# foo eq "AB\0\0CD"

$foo = pack("s2",1,2);
# "\1\0\2\0" on little-endian
# "\0\1\0\2" on big-endian

$foo = pack("a4","abcd","x","y","z");
# "abcd"

$foo = pack("aaaa","abcd","x","y","z");
# "axyz"

$foo = pack("a14","abcdefg");
# "abcdefg\0\0\0\0\0\0\0"

$foo = pack("i9pl", gmtime);
# a real struct tm (on my system anyway)

sub bintodec {
    unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}
```

The same template may generally also be used in the unpack function.

unpack(TEMPLATE,EXPR)

Unpack does the reverse of pack: it takes a string representing a structure and expands it out into an array value, returning the array value. (In a scalar context, it merely returns the first value produced.) The TEMPLATE has the same format as in the pack function. Here's a subroutine that does substring:

```
sub substr {
        local($what,$where,$howmuch) = @_;
        unpack("x$where a$howmuch", $what);
}
```

`and then there's`

```
sub ord { unpack("c",$_[0]); }
```

In addition, you may prefix a field with a %<number> to indicate that you want a <number>-bit checksum of the items instead of the items themselves. Default is a 16-bit checksum. For example, the following computes the same number as the System V sum program:

```
while (<>) {
    $checksum += unpack("%16C*", $_);
}
$checksum %= 65536;
```

vec(EXPR,OFFSET,BITS)

Treats a string as a vector of unsigned integers, and returns the value of the bitfield specified. May also be assigned to. BITS must be a power of two from 1 to 32.

Vectors created with vec() can also be manipulated with the logical operators |, & and ^, which will assume a bit vector operation is desired when both operands are strings. This interpretation is not enabled unless there is at least one vec() in your program, to protect older programs.

To transform a bit vector into a string or array of 0's and 1's, use these:

```
$bits = unpack("b*", $vector);
@bits = split(//, unpack("b*", $vector));
```

If you know the exact length in bits, it can be used in place of the *.

String Functions

chop
chop(LIST)
chop(VARIABLE)
chop VARIABLE
chop Chops off the last character of a string and returns the character chopped. It's used primarily to remove the newline from the end of an input record, but is much more efficient than s/\n// because it neither scans nor copies the string. If VARIABLE is omitted, chops $_. Example:

```
while (<>) {
        chop;   # avoid \n on last field
        @array = split(/:/);
        ...
}
```

You can actually chop anything that's an lvalue, including an assignment:

```
chop($cwd = \`pwd\`);
chop($answer = <STDIN>);
```

If you chop a list, each element is chopped. Only the value of the last chop is returned.

crypt(PLAINTEXT,SALT)

Encrypts a string exactly like the crypt() function in the C library. Useful for checking the password file for lousy passwords. Only the guys wearing white hats should do this.

eval(EXPR)

eval EXPR
eval BLOCK

EXPR is parsed and executed as if it were a little *perl* program. It is executed in the context of the current *perl* program, so that any variable settings, subroutine or format definitions remain afterwards. The value returned is the value of the last expression evaluated, just as with subroutines. If there is a syntax error or runtime error, or a die statement is executed, an undefined value is returned by eval, and $@ is set to the error message. If there was no error, $@ is guaranteed to be a null string. If EXPR is omitted, evaluates $_. The final semicolon, if any, may be omitted from the expression.

Note that, since eval traps otherwise-fatal errors, it is useful for determining whether a particular feature (such as dbmopen or symlink) is implemented. It is also Perl's exception trapping mechanism, where the die operator is used to raise exceptions.

If the code to be executed doesn't vary, you may use the eval-BLOCK form to trap run-time errors without incurring the penalty of recompiling each time. The error, if any, is still returned in $@. Evaluating a single-quoted string (as EXPR) has the same effect, except that the eval-EXPR form reports syntax errors at run time via $@, whereas the eval-BLOCK form reports syntax errors at compile time. The eval-EXPR form is optimized to eval-BLOCK the first time it succeeds. (Since the replacement side of a substitution is considered a single-quoted string when you use the e modifier, the same optimization occurs there.) Examples:

```
# make divide-by-zero non-fatal
eval { $answer = $a / $b; }; warn $@ if $@;

# optimized to same thing after first use
eval '$answer = $a / $b'; warn $@ if $@;

# a compile-time error
eval { $answer = };

# a run-time error
eval '$answer =';       # sets $@
```

index(STR,SUBSTR,POSITION)
index(STR,SUBSTR)

Returns the position of the first occurrence of SUBSTR in STR at or after POSITION. If POSITION is omitted, starts searching from the beginning of the string. The return value is based at 0, or whatever you've set the $[variable to. If the substring is not found, returns one less than the base, ordinarily -1.

length(EXPR)
length EXPR

Returns the length in characters of the value of EXPR. If EXPR is omitted, returns length of $_.

q/STRING/
qq/STRING/
qx/STRING/

These are not really functions, but simply syntactic sugar to let you avoid putting too many backslashes into quoted strings. The q operator is a generalized single quote, and the qq operator a generalized double quote. The qx operator is a generalized backquote. Any non-alphanumeric delimiter can be used in place of /, including newline. If the delimiter is an opening bracket or parenthesis, the final delimiter will be the corresponding closing bracket or parenthesis. (Embedded occurrences of the closing bracket need to be backslashed as usual.) Examples:

```
        $foo = q!I said, "You said, 'She said it.'"!;
        $bar = q('This is it.');
        $today = qx{ date };
        $_ .= qq
*** The previous line contains the naughty word "$&".\n
                if /(ibm|apple|awk)/;      # :-)
```

rindex(STR,SUBSTR,POSITION)
rindex(STR,SUBSTR)
Works just like index except that it returns the position of the LAST occurrence of SUBSTR in STR. If POSITION is specified, returns the last occurrence at or before that position.

substr(EXPR,OFFSET,LEN)
substr(EXPR,OFFSET)
Extracts a substring out of EXPR and returns it. First character is at offset 0, or whatever you've set $[to. If OFFSET is negative, starts that far from the end of the string. If LEN is omitted, returns everything to the end of the string. You can use the substr() function as an lvalue, in which case EXPR must be an lvalue. If you assign something shorter than LEN, the string will shrink, and if you assign something longer than LEN, the string will grow to accommodate it. To keep the string the same length you may need to pad or chop your value using sprintf().

Array and List Functions

delete $ASSOC{KEY}
Deletes the specified value from the specified associative array. Returns the deleted value, or the undefined value if nothing was deleted. Deleting from $ENV{} modifies the environment. Deleting from an array bound to a dbm file deletes the entry from the dbm file.

The following deletes all the values of an associative array:

```
foreach $key (keys %ARRAY) {
        delete $ARRAY{$key};
}
```

(But it would be faster to use the *reset* command. Saying undef %ARRAY is faster yet.)

each(ASSOC_ARRAY)
each ASSOC_ARRAY
Returns a 2 element array consisting of the key and value for the next value of an associative array, so that you can iterate over it. Entries are returned in an apparently random order. When the array is entirely read, a null array is returned (which when assigned produces a FALSE (0) value). The next call to each() after that will start iterating again. The iterator can be reset only by reading all the elements from the array. You must not modify the array while iterating over it. There is a single iterator for each associative array, shared by all each(), keys() and values() function calls in

the program. The following prints out your environment like the printenv program, only in a different order:

```
while (($key,$value) = each %ENV) {
        print "$key=$value\n";
}
```

See also keys() and values().

grep(EXPR,LIST)

Evaluates EXPR for each element of LIST (locally setting $_ to each element) and returns the array value consisting of those elements for which the expression evaluated to true. In a scalar context, returns the number of times the expression was true.

```
@foo = grep(!/^#/, @bar);    # weed out comments
```

Note that, since $_ is a reference into the array value, it can be used to modify the elements of the array. While this is useful and supported, it can cause bizarre results if the LIST is not a named array.

join(EXPR,LIST)
join(EXPR,ARRAY)

Joins the separate strings of LIST or ARRAY into a single string with fields separated by the value of EXPR, and returns the string. Example:

```
$_ = join(':',
            $login,$passwd,$uid,$gid,$gcos,$home,$shell);
```

See *split*.

keys(ASSOC_ARRAY)
keys ASSOC_ARRAY

Returns a normal array consisting of all the keys of the named associative array. The keys are returned in an apparently random order, but it is the same order as either the values() or each() function produces (given that the associative array has not been modified). Here is yet another way to print your environment:

```
@keys = keys %ENV;
@values = values %ENV;
while ($#keys >= 0) {
        print pop(@keys), '=', pop(@values), "\n";
}
```

```
or how about sorted by key:
```

```
foreach $key (sort(keys %ENV)) {
        print $key, '=', $ENV{$key}, "\n";
}
```

pop(ARRAY)
pop ARRAY

Pops and returns the last value of the array, shortening the array by 1. Has the same effect as

```
$tmp = $ARRAY[$#ARRAY--];
```

If there are no elements in the array, returns the undefined value.

push(ARRAY,LIST)
Treats ARRAY (@ is optional) as a stack, and pushes the values of LIST onto the end of ARRAY. The length of ARRAY increases by the length of LIST. Has the same effect as

```
for $value (LIST) {
        $ARRAY[++$#ARRAY] = $value;
}
```

but is more efficient.

reverse(LIST)
reverse LIST
In an array context, returns an array value consisting of the elements of LIST in the opposite order. In a scalar context, returns a string value consisting of the bytes of the first element of LIST in the opposite order.

shift(ARRAY)
shift ARRAY
shift Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns the undefined value. If ARRAY is omitted, shifts the @ARGV array in the main program, and the @_ array in subroutines. (This is determined lexically.) See also unshift(), push() and pop(). Shift() and unshift() do the same thing to the left end of an array that push() and pop() do to the right end.

sort(SUBROUTINE LIST)
sort(LIST)
sort SUBROUTINE LIST
sort BLOCK LIST
sort LIST
Sorts the LIST and returns the sorted array value. Nonexistent values of arrays are stripped out. If SUBROUTINE or BLOCK is omitted, sorts in standard string comparison order. If SUBROUTINE is specified, gives the name of a subroutine that returns an integer less than, equal to, or greater than 0, depending on how the elements of the array are to be ordered. (The <=> and cmp operators are extremely useful in such routines.) SUBROUTINE may be a scalar variable name, in which case the value provides the name of the subroutine to use. In place of a SUBROUTINE name, you can provide a BLOCK as an anonymous, in-line sort subroutine.

In the interests of efficiency the normal calling code for subroutines is bypassed, with the following effects: the subroutine may not be a recursive subroutine, and the two elements to be compared are passed into the subroutine not via @_ but as $a and $b (see example below). They are passed by reference so don't modify $a and $b.

Examples:

```
# sort lexically
```

```
@articles = sort @files;

# same thing, but with explicit sort routine
@articles = sort {$a cmp $b} @files;

# same thing in reversed order
@articles = sort {$b cmp $a} @files;

# sort numerically ascending
@articles = sort {$a <=> $b} @files;

# sort numerically descending
@articles = sort {$b <=> $a} @files;

# sort using explicit subroutine name
sub byage {
    $age{$a} <=> $age{$b};      # presuming integers
}
@sortedclass = sort byage @class;

sub reverse { $b cmp $a; }
@harry = ('dog','cat','x','Cain','Abel');
@george = ('gone','chased','yz','Punished','Axed');
print sort @harry;
        # prints AbelCaincatdogx
print sort reverse @harry;
        # prints xdogcatCainAbel
print sort @george, 'to', @harry;
        # prints AbelAxedCainPunishedcatchaseddoggonetoxyz
```

splice(ARRAY,OFFSET,LENGTH,LIST)
splice(ARRAY,OFFSET,LENGTH)
splice(ARRAY,OFFSET)

Removes the elements designated by OFFSET and LENGTH from an array, and replaces them with the elements of LIST, if any. Returns the elements removed from the array. The array grows or shrinks as necessary. If LENGTH is omitted, removes everything from OFFSET onward. The following equivalencies hold (assuming $[== 0):

```
push(@a,$x,$y)\h'|3.5i'splice(@a,$#a+1,0,$x,$y)
pop(@a)\h'|3.5i'splice(@a,-1)
shift(@a)\h'|3.5i'splice(@a,0,1)
unshift(@a,$x,$y)\h'|3.5i'splice(@a,0,0,$x,$y)
$a[$x] = $y\h'|3.5i'splice(@a,$x,1,$y);
```

```
Example, assuming array lengths are passed before arrays:
```

```
sub aeq {       # compare two array values
        local(@a) = splice(@_,0,shift);
        local(@b) = splice(@_,0,shift);
        return 0 unless @a == @b;       # same len?
        while (@a) {
            return 0 if pop(@a) ne pop(@b);
        }
        return 1;
}
if (&aeq($len,@foo[1..$len],0+@bar,@bar)) { ... }
```

split(/PATTERN/,EXPR,LIMIT)
split(/PATTERN/,EXPR)

split(/PATTERN/)

split   Splits a string into an array of strings, and returns it. (If not in an array context, returns the number of fields found and splits into the @_ array. (In an array context, you can force the split into @_ by using ?? as the pattern delimiters, but it still returns the array value.)) If EXPR is omitted, splits the $_ string. If PATTERN is also omitted, splits on whitespace (/[\ \t\n]+/). Anything matching PATTERN is taken to be a delimiter separating the fields. (Note that the delimiter may be longer than one character.) If LIMIT is specified, splits into no more than that many fields (though it may split into fewer). If LIMIT is unspecified, trailing null fields are stripped (which potential users of pop() would do well to remember). A pattern matching the null string (not to be confused with a null pattern //, which is just one member of the set of patterns matching a null string) will split the value of EXPR into separate characters at each point it matches that way. For example:

```
print join(':', split(/ */, 'hi there'));
```

produces the output 'h:i:t:h:e:r:e'.

The LIMIT parameter can be used to partially split a line

```
($login, $passwd, $remainder) = split(/:/, $_, 3);
```

(When assigning to a list, if LIMIT is omitted, perl supplies a LIMIT one larger than the number of variables in the list, to avoid unnecessary work. For the list above LIMIT would have been 4 by default. In time critical applications it behooves you not to split into more fields than you really need.)

If the PATTERN contains parentheses, additional array elements are created from each matching substring in the delimiter.

split(/([,-])/,"1-10,20");

produces the array value

(1,'-',10,',',20)

The pattern /PATTERN/ may be replaced with an expression to specify patterns that vary at runtime. (To do runtime compilation only once, use /$variable/o.) As a special case, specifying a space ('\ ') will split on white space just as split with no arguments does, but leading white space does NOT produce a null first field. Thus, split('\ ') can be used to emulate *awk*'s default behavior, whereas split(/\ /) will give you as many null initial fields as there are leading spaces.

Example:

```
open(passwd, '/etc/passwd');
while (<passwd>) {
        ($login, $passwd, $uid, $gid, $gcos, $home, $shell)
                = split(/:/);
        ...
}
```

(Note that $shell above will still have a newline on it. See chop().) See also *join*.

unshift(ARRAY,LIST)

Does the opposite of a *shift*. Or the opposite of a *push*, depending on how you look at it. Prepends list to the front of the array, and returns the number of elements in the new array.

```
unshift(ARGV, '-e') unless $ARGV[0] =~/^-/;
```

values(ASSOC_ARRAY)
values ASSOC_ARRAY

Returns a normal array consisting of all the values of the named associative array. The values are returned in an apparently random order, but it is the same order as either the keys() or each() function would produce on the same array. See also keys() and each().

File Operations

chmod(LIST)
chmod LIST

Changes the permissions of a list of files. The first element of the list must be the numerical mode. Returns the number of files successfully changed.

```
$cnt = chmod 0755, 'foo', 'bar';
chmod 0755, @executables;
```

chown(LIST)
chown LIST

Changes the owner (and group) of a list of files. The first two elements of the list must be the NUMERICAL uid and gid, in that order. Returns the number of files successfully changed.

```
$cnt = chown $uid, $gid, 'foo', 'bar';
chown $uid, $gid, @filenames;
```

Here's an example that looks up non-numeric uids in the passwd file:

```
print "User: ";
$user = <STDIN>;
chop($user);
print "Files: "
$pattern = <STDIN>;
chop($pattern);
open(pass, '/etc/passwd')
        || die "Can't open passwd: $!\n";
while (<pass>) {
        ($login,$pass,$uid,$gid) = split(/:/);
        $uid{$login} = $uid;
        $gid{$login} = $gid;
}
@ary = <${pattern}>;    # get filenames
if ($uid{$user} eq '') {
        die "$user not in passwd file";
}
else {
        chown $uid{$user}, $gid{$user}, @ary;
}
```

link(OLDFILE,NEWFILE)
Creates a new filename linked to the old filename. Returns 1 for success, 0 otherwise.

lstat(FILEHANDLE)
lstat FILEHANDLE
lstat(EXPR)
lstat SCALARVARIABLE
Does the same thing as the stat() function, but stats a symbolic link instead of the file the symbolic link points to. If symbolic links are unimplemented on your system, a normal stat is done.

mkdir(FILENAME,MODE)
Creates the directory specified by FILENAME, with permissions specified by MODE (as modified by umask). If it succeeds it returns 1, otherwise it returns 0 and sets $! (errno).

readlink(EXPR)
readlink EXPR
Returns the value of a symbolic link, if symbolic links are implemented. If not, gives a fatal error. If there is some system error, returns the undefined value and sets $! (errno). If EXPR is omitted, uses $_.

rename(OLDNAME,NEWNAME)
Changes the name of a file. Returns 1 for success, 0 otherwise. Will not work across filesystem boundaries.

rmdir(FILENAME)
rmdir FILENAME
Deletes the directory specified by FILENAME if it is empty. If it succeeds it returns 1, otherwise it returns 0 and sets $! (errno). If FILENAME is omitted, uses $_.

select(RBITS,WBITS,EBITS,TIMEOUT)
This calls the select system call with the bitmasks specified, which can be constructed using fileno() and vec(), along these lines:

```
$rin = $win = $ein = '';
vec($rin,fileno(STDIN),1) = 1;
vec($win,fileno(STDOUT),1) = 1;
$ein = $rin | $win;
```

If you want to select on many filehandles you might wish to write a subroutine:

```
sub fhbits {
    local(@fhlist) = split(' ',$_[0]);
    local($bits);
    for (@fhlist) {
        vec($bits,fileno($_),1) = 1;
    }
    $bits;
}
$rin = &fhbits('STDIN TTY SOCK');
```

The usual idiom is:

```
($nfound,$timeleft) =
  select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

or to block until something becomes ready:

```
$nfound = select($rout=$rin, $wout=$win,
                        $eout=$ein, undef);
```

Any of the bitmasks can also be undef. The timeout, if specified, is in seconds, which may be fractional. NOTE: not all implementations are capable of returning the $timeleft. If not, they always return $timeleft equal to the supplied $timeout.

stat(FILEHANDLE)
stat FILEHANDLE
stat(EXPR)
stat SCALARVARIABLE

Returns a 13-element array giving the statistics for a file, either the file opened via FILEHANDLE, or named by EXPR. Returns a null list if the stat fails. Typically used as follows:

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
   $atime,$mtime,$ctime,$blksize,$blocks)
       = stat($filename);
```

If stat is passed the special filehandle consisting of an underline, no stat is done, but the current contents of the stat structure from the last stat or filetest are returned. Example:

```
if (-x $file && (($d) = stat(_)) && $d < 0) {
        print "$file is executable NFS file\n";
}
```

(This only works on machines for which the device number is negative under NFS.)

symlink(OLDFILE,NEWFILE)

Creates a new filename symbolically linked to the old filename. Returns 1 for success, 0 otherwise. On systems that don't support symbolic links, produces a fatal error at run time. To check for that, use eval:

```
$symlink_exists = (eval 'symlink("","");', $@ eq '');
```

truncate(FILEHANDLE,LENGTH)
truncate(EXPR,LENGTH)

Truncates the file opened on FILEHANDLE, or named by EXPR, to the specified length. Produces a fatal error if truncate isn't implemented on your system.

unlink(LIST)
unlink LIST

Deletes a list of files. Returns the number of files successfully deleted.

```
$cnt = unlink 'a', 'b', 'c';
unlink @goners;
unlink <*.bak>;
```

Note: unlink will not delete directories unless you are superuser and the **-U** flag is supplied to *perl*. Even if these conditions are met, be warned that unlinking a directory can inflict damage on your filesystem. Use rmdir instead.

utime(LIST)
utime LIST
Changes the access and modification times on each file of a list of files. The first two elements of the list must be the NUMERICAL access and modification times, in that order. Returns the number of files successfully changed. The inode modification time of each file is set to the current time. Example of a "touch" command:

```
#!/usr/bin/perl
$now = time;
utime $now, $now, @ARGV;
```

Directory Reading Routines

closedir(DIRHANDLE)
closedir DIRHANDLE
Closes a directory opened by opendir().

opendir(DIRHANDLE,EXPR)
Opens a directory named EXPR for processing by readdir(), telldir(), seekdir(), rewinddir() and closedir(). Returns true if successful. DIRHANDLEs have their own namespace separate from FILEHANDLEs.

readdir(DIRHANDLE)
readdir DIRHANDLE
Returns the next directory entry for a directory opened by opendir(). If used in an array context, returns all the rest of the entries in the directory. If there are no more entries, returns an undefined value in a scalar context or a null list in an array context.

rewinddir(DIRHANDLE)
rewinddir DIRHANDLE
Sets the current position to the beginning of the directory for the readdir() routine on DIRHANDLE.

seekdir(DIRHANDLE,POS)
Sets the current position for the readdir() routine on DIRHANDLE. POS must be a value returned by telldir(). Has the same caveats about possible directory compaction as the corresponding system library routine.

telldir(DIRHANDLE)
telldir DIRHANDLE
Returns the current position of the readdir() routines on DIRHANDLE. Value may be given to seekdir() to access a particular location in a directory. Has the same caveats about possible directory compaction as the corresponding system library routine.

I/O Operations

binmode(FILEHANDLE)
binmode FILEHANDLE

Arranges for the file to be read in "binary" mode in operating systems that distinguish between binary and text files. Files that are not read in binary mode have CR LF sequences translated to LF on input and LF translated to CR LF on output. Binmode has no effect under Unix. If FILEHANDLE is an expression, the value is taken as the name of the filehandle.

close(FILEHANDLE)
close FILEHANDLE

Closes the file or pipe associated with the file handle. You don't have to close FILEHANDLE if you are immediately going to do another open on it, since open will close it for you. (See *open*.) However, an explicit close on an input file resets the line counter ($.), while the implicit close done by *open* does not. Also, closing a pipe will wait for the process executing on the pipe to complete, in case you want to look at the output of the pipe afterwards. Closing a pipe explicitly also puts the status value of the command into $?. Example:

```
open(OUTPUT, '|sort >foo');     # pipe to sort
...     # print stuff to output
close OUTPUT;           # wait for sort to finish
open(INPUT, 'foo');     # get sort's results
```

FILEHANDLE may be an expression whose value gives the real filehandle name.

dbmclose(ASSOC_ARRAY)
dbmclose ASSOC_ARRAY

Breaks the binding between a dbm file and an associative array. The values remaining in the associative array are meaningless unless you happen to want to know what was in the cache for the dbm file. This function is only useful if you have ndbm.

dbmopen(ASSOC,DBNAME,MODE)

This binds a dbm or ndbm file to an associative array. ASSOC is the name of the associative array. (Unlike normal open, the first argument is NOT a filehandle, even though it looks like one). DBNAME is the name of the database (without the .dir or .pag extension). If the database does not exist, it is created with protection specified by MODE (as modified by the umask). If your system only supports the older dbm functions, you may perform only one dbmopen in your program. If your system has neither dbm nor ndbm, calling dbmopen produces a fatal error.

Values assigned to the associative array prior to the dbmopen are lost. A certain number of values from the dbm file are cached in memory. By default this number is 64, but you can increase it by preallocating that number of garbage entries in the associative array before the dbmopen. You can flush the cache if necessary with the reset command.

If you don't have write access to the dbm file, you can only read associative array variables, not set them. If you want to test whether you can write, either use file tests or try setting a dummy array entry inside an eval, which will trap the error.

Note that functions such as keys() and values() may return huge array values when used on large dbm files. You may prefer to use the each() function to iterate over large dbm files. Example:

```
# print out history file offsets
dbmopen(HIST,'/usr/lib/news/history',0666);
while (($key,$val) = each %HIST) {
        print $key, ' = ', unpack('L',$val), "\n";
}
dbmclose(HIST);
```

eof(FILEHANDLE)
eof()
eof Returns 1 if the next read on FILEHANDLE will return end of file, or if FILEHANDLE is not open. FILEHANDLE may be an expression whose value gives the real filehandle name. (Note that this function actually reads a character and then ungetc's it, so it is not very useful in an interactive context.) An eof without an argument returns the eof status for the last file read. Empty parentheses () may be used to indicate the pseudo file formed of the files listed on the command line, i.e. eof() is reasonable to use inside a while (<>) loop to detect the end of only the last file. Use eof(ARGV) or eof without the parentheses to test EACH file in a while (<>) loop. Examples:

```
# insert dashes just before last line of last file
while (<>) {
        if (eof()) {
                print "--------------\n";
        }
        print;
}

# reset line numbering on each input file
while (<>) {
        print "$.\t$_";
        if (eof) {      # Not eof().
                close(ARGV);
        }
}
```

fcntl(FILEHANDLE,FUNCTION,SCALAR)
Implements the fcntl(2) function. You'll probably have to say

```
require "fcntl.ph";     # probably /usr/local/lib/perl/fcntl.ph
```

first to get the correct function definitions. If fcntl.ph doesn't exist or doesn't have the correct definitions you'll have to roll your own, based on your C header files such as <sys/fcntl.h>. (There is a perl script called h2ph that comes with the perl kit which may help you in this.) Argument processing and value return works just like ioctl below. Note that fcntl will produce a fatal error if used on a machine that doesn't implement fcntl(2).

fileno(FILEHANDLE)
fileno FILEHANDLE
Returns the file descriptor for a filehandle. Useful for constructing bitmaps for select(). If FILEHANDLE is an expression, the value is taken as the name of the filehandle.

flock(FILEHANDLE,OPERATION)

Calls flock(2) on FILEHANDLE. See manual page for flock(2) for definition of OPERATION. Returns true for success, false on failure. Will produce a fatal error if used on a machine that doesn't implement flock(2). Here's a mailbox appender for BSD systems.

```
$LOCK_SH = 1;
$LOCK_EX = 2;
$LOCK_NB = 4;
$LOCK_UN = 8;

sub lock {
    flock(MBOX,$LOCK_EX);
    # and, in case someone appended
    # while we were waiting...
    seek(MBOX, 0, 2);
}

sub unlock {
    flock(MBOX,$LOCK_UN);
}

open(MBOX, ">>/usr/spool/mail/$ENV{'USER'}")
        || die "Can't open mailbox: $!";

do lock();
print MBOX $msg,"\n\n";
do unlock();
```

getc(FILEHANDLE)

getc FILEHANDLE

getc Returns the next character from the input file attached to FILEHANDLE, or a null string at EOF. If FILEHANDLE is omitted, reads from STDIN.

ioctl(FILEHANDLE,FUNCTION,SCALAR)

Implements the ioctl(2) function. You'll probably have to say

```
require "ioctl.ph";     # probably /usr/local/lib/perl/ioctl.ph
```

first to get the correct function definitions. If ioctl.ph doesn't exist or doesn't have the correct definitions you'll have to roll your own, based on your C header files such as <sys/ioctl.h>. (There is a perl script called h2ph that comes with the perl kit which may help you in this.) SCALAR will be read and/or written depending on the FUNCTION--a pointer to the string value of SCALAR will be passed as the third argument of the actual ioctl call. (If SCALAR has no string value but does have a numeric value, that value will be passed rather than a pointer to the string value. To guarantee this to be true, add a 0 to the scalar before using it.) The pack() and unpack() functions are useful for manipulating the values of structures used by ioctl(). The following example sets the erase character to DEL.

```
require 'ioctl.ph';
$sgttyb_t = "ccccs";            # 4 chars and a short
if (ioctl(STDIN,$TIOCGETP,$sgttyb)) {
        @ary = unpack($sgttyb_t,$sgttyb);
        $ary[2] = 127;
        $sgttyb = pack($sgttyb_t,@ary);
        ioctl(STDIN,$TIOCSETP,$sgttyb)
```

```
                    || die "Can’t ioctl: $!";
}
```

The return value of ioctl (and fcntl) is as follows:

```
if OS returns:\h’|3i’perl returns:
  −1\h’|3i’  undefined value
  0\h’|3i’  string "0 but true"
  anything else\h’|3i’  that number
```

Thus perl returns true on success and false on failure, yet you can still easily determine the actual value returned by the operating system:

```
($retval = ioctl(...)) || ($retval = −1);
printf "System returned %d\n", $retval;
```

open(FILEHANDLE,EXPR)
open(FILEHANDLE)
open FILEHANDLE

Opens the file whose filename is given by EXPR, and associates it with FILEHANDLE. If FILEHANDLE is an expression, its value is used as the name of the real filehandle wanted. If EXPR is omitted, the scalar variable of the same name as the FILEHANDLE contains the filename. If the filename begins with "<" or nothing, the file is opened for input. If the filename begins with ">", the file is opened for output. If the filename begins with ">>", the file is opened for appending. (You can put a ’+’ in front of the ’>’ or ’<’ to indicate that you want both read and write access to the file.) If the filename begins with "|", the filename is interpreted as a command to which output is to be piped, and if the filename ends with a "|", the filename is interpreted as command which pipes input to us. (You may not have a command that pipes both in and out.) Opening ’-’ opens *STDIN* and opening ’>-’ opens *STDOUT*. Open returns non-zero upon success, the undefined value otherwise. If the open involved a pipe, the return value happens to be the pid of the subprocess. Examples:

```
$article = 100;
open article || die "Can’t find article $article: $!\n";
while (<article>) {...

open(LOG, ’>>/usr/spool/news/twitlog’);
                                # (log is reserved)

open(article, "caesar <$article |");
                                # decrypt article

open(extract, "|sort >/tmp/Tmp$$");
                                # $$ is our process#

# process argument list of files along with any includes

foreach $file (@ARGV) {
        do process($file, ’fh00’);      # no pun intended
}

sub process {
        local($filename, $input) = @_;
        $input++;               # this is a string increment
        unless (open($input, $filename)) {
```

```
                print STDERR "Can't open $filename: $!\n";
                return;
        }
        while (<$input>) {              # note use of indirection
                if (/^#include "(.*)"/) {
                        do process($1, $input);
                        next;
                }
                ...             # whatever
        }
}
```

You may also, in the Bourne shell tradition, specify an EXPR beginning with ">&", in which case the rest of the string is interpreted as the name of a filehandle (or file descriptor, if numeric) which is to be duped and opened. You may use & after >, >>, <, +>, +>> and +<. The mode you specify should match the mode of the original filehandle. Here is a script that saves, redirects, and restores *STDOUT* and *STDERR*:

```
#!/usr/bin/perl
open(SAVEOUT, ">&STDOUT");
open(SAVEERR, ">&STDERR");

open(STDOUT, ">foo.out") || die "Can't redirect stdout";
open(STDERR, ">&STDOUT") || die "Can't dup stdout";

select(STDERR); $| = 1;         # make unbuffered
select(STDOUT); $| = 1;         # make unbuffered

print STDOUT "stdout 1\n";      # this works for
print STDERR "stderr 1\n";      # subprocesses too

close(STDOUT);
close(STDERR);

open(STDOUT, ">&SAVEOUT");
open(STDERR, ">&SAVEERR");

print STDOUT "stdout 2\n";
print STDERR "stderr 2\n";
```

If you open a pipe on the command "-", i.e. either "|-" or "-|", then there is an implicit fork done, and the return value of open is the pid of the child within the parent process, and 0 within the child process. (Use defined($pid) to determine if the open was successful.) The filehandle behaves normally for the parent, but i/o to that filehandle is piped from/to the *STDOUT*/ STDIN of the child process. In the child process the filehandle isn't opened--i/o happens from/to the new *STDOUT* or *STDIN*. Typically this is used like the normal piped open when you want to exercise more control over just how the pipe command gets executed, such as when you are running setuid, and don't want to have to scan shell commands for metacharacters. The following pairs are more or less equivalent:

```
open(FOO, "|tr '[a-z]' '[A-Z]'");
open(FOO, "|-") || exec 'tr', '[a-z]', '[A-Z]';

open(FOO, "cat -n '$file'|");
open(FOO, "-|") || exec 'cat', '-n', $file;
```

Explicitly closing any piped filehandle causes the parent process to wait for the child to finish, and returns the status value in $?. Note: on any operation which may do a fork, unflushed buffers remain unflushed in both processes, which means you may need to set $| to avoid duplicate output.

The filename that is passed to open will have leading and trailing whitespace deleted. In order to open a file with arbitrary weird characters in it, it's necessary to protect any leading and trailing whitespace thusly:

```
$file =~s#^(\s)#./$1#;
open(FOO, "< $file\0");
```

pipe(READHANDLE,WRITEHANDLE)
Opens a pair of connected pipes like the corresponding system call. Note that if you set up a loop of piped processes, deadlock can occur unless you are very careful. In addition, note that perl's pipes use stdio buffering, so you may need to set $| to flush your WRITEHANDLE after each command, depending on the application. [Requires version 3.0 patchlevel 9.]

print(FILEHANDLE LIST)
print(LIST)
print FILEHANDLE LIST
print LIST
print Prints a string or a comma-separated list of strings. Returns non-zero if successful. FILEHANDLE may be a scalar variable name, in which case the variable contains the name of the filehandle, thus introducing one level of indirection. (NOTE: If FILEHANDLE is a variable and the next token is a term, it may be misinterpreted as an operator unless you interpose a + or put parens around the arguments.) If FILEHANDLE is omitted, prints by default to standard output (or to the last selected output channel--see select()). If LIST is also omitted, prints $_ to *STDOUT*. To set the default output channel to something other than *STDOUT* use the select operation. Note that, because print takes a LIST, anything in the LIST is evaluated in an array context, and any subroutine that you call will have one or more of its expressions evaluated in an array context. Also be careful not to follow the print keyword with a left parenthesis unless you want the corresponding right parenthesis to terminate the arguments to the print--interpose a + or put parens around all the arguments.

printf(FILEHANDLE LIST)
printf(LIST)
printf FILEHANDLE LIST
printf LIST
Equivalent to a "print FILEHANDLE sprintf(LIST)".

read(FILEHANDLE,SCALAR,LENGTH,OFFSET)
read(FILEHANDLE,SCALAR,LENGTH)
Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE. Returns the number of bytes actually read, or undef if there was an error. SCALAR will be grown or shrunk to the length actually read. An OFFSET may be specified to place the read data at some other place than the beginning of the string. This call is actually implemented in terms of stdio's fread call. To get a true read system call, see sysread.

seek(FILEHANDLE,POSITION,WHENCE)
Randomly positions the file pointer for FILEHANDLE, just like the fseek() call of stdio.

FILEHANDLE may be an expression whose value gives the name of the filehandle. Returns 1 upon success, 0 otherwise.

select(FILEHANDLE)
select Returns the currently selected filehandle. Sets the current default filehandle for output, if FILEHANDLE is supplied. This has two effects: first, a *write* or a *print* without a filehandle will default to this FILEHANDLE. Second, references to variables related to output will refer to this output channel. For example, if you have to set the top of form format for more than one output channel, you might do the following:

```
select(REPORT1);
$^ = 'report1_top';
select(REPORT2);
$^ = 'report2_top';
```

FILEHANDLE may be an expression whose value gives the name of the actual filehandle. Thus:

```
$oldfh = select(STDERR); $| = 1; select($oldfh);
```

sprintf(FORMAT,LIST)
Returns a string formatted by the usual printf conventions. The * character is not supported.

sysread(FILEHANDLE,SCALAR,LENGTH,OFFSET)
sysread(FILEHANDLE,SCALAR,LENGTH)
Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE, using the system call read(2). It bypasses stdio, so mixing this with other kinds of reads may cause confusion. Returns the number of bytes actually read, or undef if there was an error. SCALAR will be grown or shrunk to the length actually read. An OFFSET may be specified to place the read data at some other place than the beginning of the string.

syswrite(FILEHANDLE,SCALAR,LENGTH,OFFSET)
syswrite(FILEHANDLE,SCALAR,LENGTH)
Attempts to write LENGTH bytes of data from variable SCALAR to the specified FILEHANDLE, using the system call write(2). It bypasses stdio, so mixing this with prints may cause confusion. Returns the number of bytes actually written, or undef if there was an error. An OFFSET may be specified to place the read data at some other place than the beginning of the string.

tell(FILEHANDLE)
tell FILEHANDLE
tell Returns the current file position for FILEHANDLE. FILEHANDLE may be an expression whose value gives the name of the actual filehandle. If FILEHANDLE is omitted, assumes the file last read.

write(FILEHANDLE)
write(EXPR)
write Writes a formatted record (possibly multi-line) to the specified file, using the format associated with that file. By default the format for a file is the one having the same name is the filehandle, but the format for the current output channel (see *select*) may be set explicitly by assigning the name of the format to the $~variable.

Top of form processing is handled automatically: if there is insufficient room on the current page for the formatted record, the page is advanced by writing a form feed, a special top-of-page format is used to format the new page header, and then the record is written. By default the top-of-page format is the name of the filehandle with "_TOP" appended, but it may be dynamicallly set to the format of your choice by assigning the name to the $^ variable while the filehandle is selected. The number of lines remaining on the current page is in variable $-, which can be set to 0 to force a new page.

If FILEHANDLE is unspecified, output goes to the current default output channel, which starts out as *STDOUT* but may be changed by the *select* operator. If the FILEHANDLE is an EXPR, then the expression is evaluated and the resulting string is used to look up the name of the FILEHANDLE at run time. For more on formats, see the section on formats later on.

Note that write is NOT the opposite of read.

Search and Modification Operations

m/PATTERN/gio
/PATTERN/gio

Searches a string for a pattern match, and returns true (1) or false (’’). If no string is specified via the =~ or !~ operator, the $_ string is searched. (The string specified with =~ need not be an lvalue--it may be the result of an expression evaluation, but remember the =~ binds rather tightly.) See also the section on regular expressions.

If / is the delimiter then the initial ’m’ is optional. With the ’m’ you can use any pair of non-alphanumeric characters as delimiters. This is particularly useful for matching Unix path names that contain ’/’. If the final delimiter is followed by the optional letter ’i’, the matching is done in a case-insensitive manner. PATTERN may contain references to scalar variables, which will be interpolated (and the pattern recompiled) every time the pattern search is evaluated. (Note that $) and $| may not be interpolated because they look like end-of-string tests.) If you want such a pattern to be compiled only once, add an "o" after the trailing delimiter. This avoids expensive run-time recompilations, and is useful when the value you are interpolating won’t change over the life of the script. If the PATTERN evaluates to a null string, the most recent successful regular expression is used instead.

If used in a context that requires an array value, a pattern match returns an array consisting of the subexpressions matched by the parentheses in the pattern, i.e. ($1, $2, $3...). It does NOT actually set $1, $2, etc. in this case, nor does it set $+, $‘, $& or $’. If the match fails, a null array is returned. If the match succeeds, but there were no parentheses, an array value of (1) is returned.

Examples:

```
open(tty, '/dev/tty');
<tty> =~/^y/i && do foo(); # do foo if desired

if (/Version: *([0-9.]*)/) { $version = $1; }
```

```
next if m#^/usr/spool/uucp#;

# poor man's grep
$arg = shift;
while (<>) {
        print if /$arg/o;   # compile only once
}

if (($F1, $F2, $Etc) = ($foo =~/^(\S+)\s+(\S+)\s*(.*)/))
```

This last example splits $foo into the first two words and the remainder of the line, and assigns those three fields to $F1, $F2 and $Etc. The conditional is true if any variables were assigned, i.e. if the pattern matched.

The "g" modifier specifies global pattern matching--that is, matching as many times as possible within the string. How it behaves depends on the context. In an array context, it returns a list of all the substrings matched by all the parentheses in the regular expression. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern. In a scalar context, it iterates through the string, returning TRUE each time it matches, and FALSE when it eventually runs out of matches. (In other words, it remembers where it left off last time and restarts the search at that point.) It presumes that you have not modified the string since the last match. Modifying the string between matches may result in undefined behavior. (You can actually get away with in-place modifications via substr() that do not change the length of the entire string. In general, however, you should be using s///g for such modifications.) Examples:

```
# array context
($one,$five,$fifteen) = (\`uptime\` =~/(\d+\.\d+)/g);

# scalar context
$/ = ""; $* = 1;
while ($paragraph = <>) {
    while ($paragraph =~/[a-z]['")]*[.!?]+['")]*\s/g) {
        $sentences++;
    }
}
print "$sentences\n";
```

?PATTERN?

This is just like the /pattern/ search, except that it matches only once between calls to the *reset* operator. This is a useful optimization when you only want to see the first occurrence of something in each file of a set of files, for instance. Only ?? patterns local to the current package are reset.

s/PATTERN/REPLACEMENT/gieo

Searches a string for a pattern, and if found, replaces that pattern with the replacement text and returns the number of substitutions made. Otherwise it returns false (0). The "g" is optional, and if present, indicates that all occurrences of the pattern are to be replaced. The "i" is also optional, and if present, indicates that matching is to be done in a case-insensitive manner. The "e" is likewise optional, and if present, indicates that the replacement string is to be evaluated as an expression rather than just as a double-quoted string. Any non-alphanumeric delimiter may replace the slashes; if single quotes are used, no interpretation is done on the replacement string (the e modifier overrides this, however); if backquotes are used, the replacement string is a command to execute whose output will be used as the actual replacement text. If the PATTERN is delimited by bracketing quotes, the REPLACEMENT has its own pair of quotes, which may or may not be bracketing quotes, e.g. s(foo)(bar) or s<foo>/bar/. If no string is specified via the =~ or !~ operator, the $_ string is searched and modified. (The string specified with =~ must be a scalar variable, an array element, or an assignment to one of those, i.e. an lvalue.) If the pattern contains a $ that looks like a variable rather than an end-of-string test, the variable will be interpolated into the pattern at run-time. If you only want the pattern compiled once the first time the variable is interpolated, add an "o" at the end. If the PATTERN evaluates to a null string, the most recent successful regular expression is used instead. See also the section on regular expressions. Examples:

```
s/\bgreen\b/mauve/g;                # don't change wintergreen

$path =~s|/usr/bin|/usr/local/bin|;

s/Login: $foo/Login: $bar/; # run-time pattern

($foo = $bar) =~s/bar/foo/;

$_ = 'abc123xyz';
s/\d+/$&*2/e;               # yields 'abc246xyz'
s/\d+/sprintf("%5d",$&)/e;  # yields 'abc  246xyz'
s/\w/$& x 2/eg;             # yields 'aabbcc  224466xxyyzz'

s/([^ ]*) *([^ ]*)/$2 $1/;  # reverse 1st two fields
```

(Note the use of $ instead of \ in the last example. See section on regular expressions.)

study(SCALAR)
study SCALAR
study Takes extra time to study SCALAR ($_ if unspecified) in anticipation of doing many pattern matches on the string before it is next modified. This may or may not save time, depending on the nature and number of patterns you are searching on, and on the distribution of character frequencies in the string to be searched--you probably want to compare runtimes with and without it to see which runs faster. Those loops which scan for many short constant strings (including the constant parts of more complex patterns) will benefit most. You may have only one study active at a time--if you study a different scalar the first is "unstudied". (The way study works is this: a linked list of every character in the string to be searched is made, so we know, for example, where all the 'k' characters are. From each search string, the rarest character is selected, based on some static frequency tables constructed from some C programs and English text. Only those places that contain this "rarest" character are examined.)

For example, here is a loop which inserts index producing entries before any line containing a certain pattern:

```
while (<>) {
        study;
        print ".IX foo\n" if /\bfoo\b/;
        print ".IX bar\n" if /\bbar\b/;
        print ".IX blurfl\n" if /\bblurfl\b/;
        ...
        print;
}
```

In searching for /\bfoo\b/, only those locations in $_ that contain 'f' will be looked at, because 'f' is rarer than 'o'. In general, this is a big win except in pathological cases. The only question is whether it saves you more time than it took to build the linked list in the first place.

Note that if you have to look for strings that you don't know till runtime, you can build an entire loop as a string and eval that to avoid recompiling all your patterns all the time. Together with undefining $/ to input entire files as one record, this can be very fast, often faster than specialized programs like fgrep. The following scans a list of files (@files) for a list of words (@words), and prints out the names of those files that contain a match:

```
$search = 'while (<>) { study;';
foreach $word (@words) {
    $search .= "++\$seen{\$ARGV} if /\\b$word\\b/;\n";
}
$search .= "}";
@ARGV = @files;
undef $/;
eval $search;           # this screams
$/ = "\n";              # put back to normal input delim
foreach $file (sort keys(%seen)) {
    print $file, "\n";
}
```

tr/SEARCHLIST/REPLACEMENTLIST/cds
y/SEARCHLIST/REPLACEMENTLIST/cds

Translates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the =~ or !~ operator, the $_ string is translated. (The string specified with =~ must be a scalar variable, an array element, or an assignment to one of those, i.e. an lvalue.) For *sed* devotees, *y* is provided as a synonym for *tr*. If the SEARCHLIST is delimited by bracketing quotes, the REPLACEMENTLIST has its own pair of quotes, which may or may not be bracketing quotes, e.g. tr[A-Z][a-z] or tr(+-*/)/ABCD/.

If the c modifier is specified, the SEARCHLIST character set is complemented. If the d modifier is specified, any characters specified by SEARCHLIST that are not found in REPLACEMENTLIST are deleted. (Note that this is slightly more flexible than the behavior of some *tr* programs, which delete anything they find in the SEARCHLIST, period.) If the s modifier is specified, sequences of characters that were translated to the same character are squashed down to 1 instance of the character.

If the d modifier was used, the REPLACEMENTLIST is always interpreted exactly as specified. Otherwise, if the REPLACEMENTLIST is shorter than the SEARCHLIST, the final character is replicated till it is long enough. If the REPLACEMENTLIST is null, the SEARCHLIST is replicated. This latter is useful for counting characters in a class, or for squashing character sequences in a class.

Examples:

```
$ARGV[1] =~y/A−Z/a−z/;     \h'|3i'# canonicalize to lower case

$cnt = tr/*/*/;             \h'|3i'# count the stars in $_

$cnt = tr/0−9//;            \h'|3i'# count the digits in $_
```

```
tr/a-zA-Z//s;        \h'|3i'# bookkeeper -> bokeper

($HOST = $host) =~tr/a-z/A-Z/;

y/a-zA-Z/ /cs;       \h'|3i'# change non-alphas to single space

tr/\200-\377/\0-\177/;\h'|3i'# delete 8th bit
```

System Interaction Routines

alarm(SECONDS)
alarm SECONDS

Arranges to have a SIGALRM delivered to this process after the specified number of seconds (minus 1, actually) have elapsed. Thus, alarm(15) will cause a SIGALRM at some point more than 14 seconds in the future. Only one timer may be counting at once. Each call disables the previous timer, and an argument of 0 may be supplied to cancel the previous timer without starting a new one. The returned value is the amount of time remaining on the previous timer.

chdir(EXPR)
chdir EXPR

Changes the working directory to EXPR, if possible. If EXPR is omitted, changes to home directory. Returns 1 upon success, 0 otherwise. See example under *die*.

chroot(FILENAME)
chroot FILENAME

Does the same as the system call of that name. If you don't know what it does, don't worry about it. If FILENAME is omitted, does chroot to $_.

die(LIST)
die LIST

Outside of an eval, prints the value of LIST to *STDERR* and exits with the current value of $! (errno). If $! is 0, exits with the value of ($? >> 8) (\`command\` status). If ($? >> 8) is 0, exits with 255. Inside an eval, the error message is stuffed into $@ and the eval is terminated with the undefined value.

Equivalent examples:

```
die "Can't cd to spool: $!\n"
        unless chdir '/usr/spool/news';

chdir '/usr/spool/news' || die "Can't cd to spool: $!\n"
```

If the value of EXPR does not end in a newline, the current script line number and input line number (if any) are also printed, and a newline is supplied. Hint: sometimes appending ", stopped" to your message will cause it to make better sense when the string "at foo line 123" is appended. Suppose you are running script "canasta".

```
die "/etc/games is no good";
die "/etc/games is no good, stopped";
```

produce, respectively

```
/etc/games is no good at canasta line 123.
/etc/games is no good, stopped at canasta line 123.
```

See also *exit*.

exec(LIST)
exec LIST
If there is more than one argument in LIST, or if LIST is an array with more than one value, calls execvp() with the arguments in LIST. If there is only one scalar argument, the argument is checked for shell metacharacters. If there are any, the entire argument is passed to "/bin/sh -c" for parsing. If there are none, the argument is split into words and passed directly to execvp(), which is more efficient. Note: exec (and system) do not flush your output buffer, so you may need to set $| to avoid lost output. Examples:

```
exec '/bin/echo', 'Your arguments are: ', @ARGV;
exec "sort $outfile | uniq";
```

If you don't really want to execute the first argument, but want to lie to the program you are executing about its own name, you can specify the program you actually want to run by assigning that to a variable and putting the name of the variable in front of the LIST without a comma. (This always forces interpretation of the LIST as a multi-valued list, even if there is only a single scalar in the list.) Example:

```
$shell = '/bin/csh';
exec $shell '-sh';              # pretend it's a login shell
```

exit(EXPR)
exit EXPR
Evaluates EXPR and exits immediately with that value. Example:

```
$ans = <STDIN>;
exit 0 if $ans =~/^[Xx]/;
```

See also *die*. If EXPR is omitted, exits with 0 status.

fork Does a fork() call. Returns the child pid to the parent process and 0 to the child process. Note: unflushed buffers remain unflushed in both processes, which means you may need to set $| to avoid duplicate output.

getlogin
Returns the current login from /etc/utmp, if any. If null, use getpwuid. $login = getlogin || (getpwuid($<))[0] || "Somebody";

getpgrp(PID)
getpgrp PID
Returns the current process group for the specified PID, 0 for the current process. Will produce a fatal error if used on a machine that doesn't implement getpgrp(2). If EXPR is omitted, returns process group of current process.

getppid
: Returns the process id of the parent process.

getpriority(WHICH,WHO)
: Returns the current priority for a process, a process group, or a user. (See getpriority(2).) Will produce a fatal error if used on a machine that doesn't implement getpriority(2).

kill(LIST)
kill LIST
: Sends a signal to a list of processes. The first element of the list must be the signal to send. Returns the number of processes successfully signaled.

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
```

If the signal is negative, kills process groups instead of processes. (On System V, a negative *process* number will also kill process groups, but that's not portable.) You may use a signal name in quotes.

setpgrp(PID,PGRP)
: Sets the current process group for the specified PID, 0 for the current process. Will produce a fatal error if used on a machine that doesn't implement setpgrp(2).

setpriority(WHICH,WHO,PRIORITY)
: Sets the current priority for a process, a process group, or a user. (See setpriority(2).) Will produce a fatal error if used on a machine that doesn't implement setpriority(2).

sleep(EXPR)
sleep EXPR
sleep Causes the script to sleep for EXPR seconds, or forever if no EXPR. May be interrupted by sending the process a SIGALRM. Returns the number of seconds actually slept. You probably cannot mix alarm() and sleep() calls, since sleep() is often implemented using alarm().

syscall(LIST)
syscall LIST
: Calls the system call specified as the first element of the list, passing the remaining elements as arguments to the system call. If unimplemented, produces a fatal error. The arguments are interpreted as follows: if a given argument is numeric, the argument is passed as an int. If not, the pointer to the string value is passed. You are responsible to make sure a string is pre-extended long enough to receive any result that might be written into a string. If your integer arguments are not literals and have never been interpreted in a numeric context, you may need to add 0 to them to force them to look like numbers.

```
require 'syscall.ph';           # may need to run h2ph
syscall(&SYS_write, fileno(STDOUT), "hi there\n", 9);
```

system(LIST)
system LIST
: Does exactly the same thing as "exec LIST" except that a fork is done first, and the parent process

waits for the child process to complete. Note that argument processing varies depending on the number of arguments. The return value is the exit status of the program as returned by the wait() call. To get the actual exit value divide by 256. See also *exec*.

time
: Returns the number of non-leap seconds since 00:00:00 UTC, January 1, 1970. Suitable for feeding to gmtime() and localtime().

times
: Returns a four-element array giving the user and system times, in seconds, for this process and the children of this process.

  ($user,$system,$cuser,$csystem) = times;

umask(EXPR)
umask EXPR
umask
: Sets the umask for the process and returns the old one. If EXPR is omitted, merely returns current umask.

wait
: Waits for a child process to terminate and returns the pid of the deceased process, or -1 if there are no child processes. The status is returned in $?.

waitpid(PID,FLAGS)
: Waits for a particular child process to terminate and returns the pid of the deceased process, or -1 if there is no such child process. The status is returned in $?. If you say

```
require "sys/wait.h";
...
waitpid(-1,&WNOHANG);
```

  then you can do a non-blocking wait for any process. Non-blocking wait is only available on machines supporting either the *waitpid (2)* or *wait4 (2)* system calls. However, waiting for a particular pid with FLAGS of 0 is implemented everywhere. (Perl emulates the system call by remembering the status values of processes that have exited but have not been harvested by the Perl script yet.)

warn(LIST)
warn LIST
: Produces a message on STDERR just like "die", but doesn't exit.

IPC and Networking Operations

accept(NEWSOCKET,GENERICSOCKET)
: Does the same thing that the accept system call does. Returns true if it succeeded, false otherwise. See example in section on Interprocess Communication.

bind(SOCKET,NAME)
: Does the same thing that the bind system call does. Returns true if it succeeded, false otherwise.

NAME should be a packed address of the proper type for the socket. See example in section on Interprocess Communication.

connect(SOCKET,NAME)
: Does the same thing that the connect system call does. Returns true if it succeeded, false otherwise. NAME should be a package address of the proper type for the socket. See example in section on Interprocess Communication.

getpeername(SOCKET)
: Returns the packed sockaddr address of other end of the SOCKET connection.

```
# An internet sockaddr
$sockaddr = 'S n a4 x8';
$hersockaddr = getpeername(S);
($family, $port, $heraddr) =
                unpack($sockaddr,$hersockaddr);
```

getsockname(SOCKET)
: Returns the packed sockaddr address of this end of the SOCKET connection.

```
# An internet sockaddr
$sockaddr = 'S n a4 x8';
$mysockaddr = getsockname(S);
($family, $port, $myaddr) =
                unpack($sockaddr,$mysockaddr);
```

getsockopt(SOCKET,LEVEL,OPTNAME)
: Returns the socket option requested, or undefined if there is an error.

listen(SOCKET,QUEUESIZE)
: Does the same thing that the listen system call does. Returns true if it succeeded, false otherwise. See example in section on Interprocess Communication.

msgctl(ID,CMD,ARG)
: Calls the System V IPC function msgctl. If CMD is &IPC_STAT, then ARG must be a variable which will hold the returned msqid_ds structure. Returns like ioctl: the undefined value for error, "0 but true" for zero, or the actual return value otherwise.

msgget(KEY,FLAGS)
: Calls the System V IPC function msgget. Returns the message queue id, or the undefined value if there is an error.

msgsnd(ID,MSG,FLAGS)
: Calls the System V IPC function msgsnd to send the message MSG to the message queue ID. MSG must begin with the long integer message type, which may be created with pack("L", $type). Returns true if successful, or false if there is an error.

msgrcv(ID,VAR,SIZE,TYPE,FLAGS)
: Calls the System V IPC function msgrcv to receive a message from message queue ID into variable VAR with a maximum message size of SIZE. Note that if a message is received, the message type will be the first thing in VAR, and the maximum length of VAR is SIZE plus the

size of the message type. Returns true if successful, or false if there is an error.

recv(SOCKET,SCALAR,LEN,FLAGS)
: Receives a message on a socket. Attempts to receive LENGTH bytes of data into variable SCALAR from the specified SOCKET filehandle. Returns the address of the sender, or the undefined value if there's an error. SCALAR will be grown or shrunk to the length actually read. Takes the same flags as the system call of the same name.

semctl(ID,SEMNUM,CMD,ARG)
: Calls the System V IPC function semctl. If CMD is &IPC_STAT or &GETALL, then ARG must be a variable which will hold the returned semid_ds structure or semaphore value array. Returns like ioctl: the undefined value for error, "0 but true" for zero, or the actual return value otherwise.

semget(KEY,NSEMS,SIZE,FLAGS)
: Calls the System V IPC function semget. Returns the semaphore id, or the undefined value if there is an error.

semop(KEY,OPSTRING)
: Calls the System V IPC function semop to perform semaphore operations such as signaling and waiting. OPSTRING must be a packed array of semop structures. Each semop structure can be generated with 'pack("sss", $semnum, $semop, $semflag)'. The number of semaphore operations is implied by the length of OPSTRING. Returns true if successful, or false if there is an error. As an example, the following code waits on semaphore $semnum of semaphore id $semid:

```
$semop = pack("sss", $semnum, -1, 0);
die "Semaphore trouble: $!\n" unless semop($semid, $semop);
```

To signal the semaphore, replace "-1" with "1".

send(SOCKET,MSG,FLAGS,TO)
send(SOCKET,MSG,FLAGS)
: Sends a message on a socket. Takes the same flags as the system call of the same name. On unconnected sockets you must specify a destination to send TO. Returns the number of characters sent, or the undefined value if there is an error.

setsockopt(SOCKET,LEVEL,OPTNAME,OPTVAL)
: Sets the socket option requested. Returns undefined if there is an error. OPTVAL may be specified as undef if you don't want to pass an argument.

shmctl(ID,CMD,ARG)
: Calls the System V IPC function shmctl. If CMD is &IPC_STAT, then ARG must be a variable which will hold the returned shmid_ds structure. Returns like ioctl: the undefined value for error, "0 but true" for zero, or the actual return value otherwise.

shmget(KEY,SIZE,FLAGS)
: Calls the System V IPC function shmget. Returns the shared memory segment id, or the undefined value if there is an error.

shmread(ID,VAR,POS,SIZE)

shmwrite(ID,STRING,POS,SIZE)

Reads or writes the System V shared memory segment ID starting at position POS for size SIZE by attaching to it, copying in/out, and detaching from it. When reading, VAR must be a variable which will hold the data read. When writing, if STRING is too long, only SIZE bytes are used; if STRING is too short, nulls are written to fill out SIZE bytes. Return true if successful, or false if there is an error.

shutdown(SOCKET,HOW)

Shuts down a socket connection in the manner indicated by HOW, which has the same interpretation as in the system call of the same name.

socket(SOCKET,DOMAIN,TYPE,PROTOCOL)

Opens a socket of the specified kind and attaches it to filehandle SOCKET. DOMAIN, TYPE and PROTOCOL are specified the same as for the system call of the same name. You may need to run h2ph on sys/socket.h to get the proper values handy in a perl library file. Return true if successful. See the example in the section on Interprocess Communication.

socketpair(SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL)

Creates an unnamed pair of sockets in the specified domain, of the specified type. DOMAIN, TYPE and PROTOCOL are specified the same as for the system call of the same name. If unimplemented, yields a fatal error. Return true if successful.

Miscellaneous Operations

caller(EXPR)
caller Returns the context of the current subroutine call:

```
($package,$filename,$line) = caller;
```

With EXPR, returns some extra information that the debugger uses to print a stack trace. The value of EXPR indicates how many call frames to go back before the current one.

defined(EXPR)
defined EXPR

Returns a boolean value saying whether the lvalue EXPR has a real value or not. Many operations return the undefined value under exceptional conditions, such as end of file, uninitialized variable, system error and such. This function allows you to distinguish between an undefined null string and a defined null string with operations that might return a real null string, in particular referencing elements of an array. You may also check to see if arrays or subroutines exist. Use on predefined variables is not guaranteed to produce intuitive results. Examples:

```
print if defined $switch{'D'};
print "$val\n" while defined($val = pop(@ary));
die "Can't readlink $sym: $!"
        unless defined($value = readlink $sym);
eval '@foo = ()' if defined(@foo);
die "No XYZ package defined" unless defined %_XYZ;
sub foo { defined &$bar ? &$bar(@_) : die "No bar"; }
```

See also undef.

dump LABEL

This causes an immediate core dump. Primarily this is so that you can use the undump program to turn your core dump into an executable binary after having initialized all your variables at the beginning of the program. When the new binary is executed it will begin by executing a "goto LABEL" (with all the restrictions that goto suffers). Think of it as a goto with an intervening core dump and reincarnation. If LABEL is omitted, restarts the program from the top. WARNING: any files opened at the time of the dump will NOT be open any more when the program is reincarnated, with possible resulting confusion on the part of perl. See also -u.

Example:

```
        #!/usr/bin/perl
        require 'getopt.pl';
        require 'stat.pl';
        %days = (
            'Sun',1,
            'Mon',2,
            'Tue',3,
            'Wed',4,
            'Thu',5,
            'Fri',6,
            'Sat',7);

        dump QUICKSTART if $ARGV[0] eq '-d';

    QUICKSTART:
        do Getopt('f');
```

local(LIST)

Declares the listed variables to be local to the enclosing block, subroutine, eval or "do". All the listed elements must be legal lvalues. This operator works by saving the current values of those variables in LIST on a hidden stack and restoring them upon exiting the block, subroutine or eval. This means that called subroutines can also reference the local variable, but not the global one. The LIST may be assigned to if desired, which allows you to initialize your local variables. (If no initializer is given for a particular variable, it is created with an undefined value.) Commonly this is used to name the parameters to a subroutine. Examples:

```
        sub RANGEVAL {
                local($min, $max, $thunk) = @_;
                local($result) = '';
                local($i);

                # Presumably $thunk makes reference to $i

                for ($i = $min; $i < $max; $i++) {
                        $result .= eval $thunk;
                }

                $result;
        }

        if ($sw eq '-v') {
            # init local array with global array
```

```
        local(@ARGV) = @ARGV;
        unshift(@ARGV,'echo');
        system @ARGV;
    }
    # @ARGV restored

    # temporarily add to digits associative array
    if ($base12) {
            # (NOTE: not claiming this is efficient!)
            local(%digits) = (%digits,'t',10,'e',11);
            do parse_num();
    }
```

Note that local() is a run-time command, and so gets executed every time through a loop, using up more stack storage each time until it's all released at once when the loop is exited.

require(EXPR)
require EXPR
require
Includes the library file specified by EXPR, or by $_ if EXPR is not supplied. Has semantics similar to the following subroutine:

```
    sub require {
        local($filename) = @_;
        return 1 if $INC{$filename};
        local($realfilename,$result);
        ITER: {
            foreach $prefix (@INC) {
                $realfilename = "$prefix/$filename";
                if (-f $realfilename) {
                    $result = do $realfilename;
                    last ITER;
                }
            }
            die "Can't find $filename in \@INC";
        }
        die $@ if $@;
        die "$filename did not return true value" unless $result;
        $INC{$filename} = $realfilename;
        $result;
    }
```

Note that the file will not be included twice under the same specified name. The file must return true as the last statement to indicate successful execution of any initialization code, so it's customary to end such a file with "1;" unless you're sure it'll return true otherwise.

reset(EXPR)
reset EXPR
reset Generally used in a *continue* block at the end of a loop to clear variables and reset ?? searches so that they work again. The expression is interpreted as a list of single characters (hyphens allowed for ranges). All variables and arrays beginning with one of those letters are reset to their pristine state. If the expression is omitted, one-match searches (?pattern?) are reset to match again. Only resets variables or searches in the current package. Always returns 1. Examples:

```
    reset 'X';          # reset all X variables
    reset 'a-z';        # reset lower case variables
```

```
reset;              # just reset ?? searches
```

Note: resetting "A-Z" is not recommended since you'll wipe out your ARGV and ENV arrays.

The use of reset on dbm associative arrays does not change the dbm file. (It does, however, flush any entries cached by perl, which may be useful if you are sharing the dbm file. Then again, maybe not.)

scalar(EXPR)
Forces EXPR to be interpreted in a scalar context and returns the value of EXPR.

undef(EXPR)
undef EXPR
undef Undefines the value of EXPR, which must be an lvalue. Use only on a scalar value, an entire array, or a subroutine name (using &). (Undef will probably not do what you expect on most predefined variables or dbm array values.) Always returns the undefined value. You can omit the EXPR, in which case nothing is undefined, but you still get an undefined value that you could, for instance, return from a subroutine. Examples:

```
undef $foo;
undef $bar{'blurfl'};
undef @ary;
undef %assoc;
undef &mysub;
return (wantarray ? () : undef) if $they_blew_it;
```

wantarray
Returns true if the context of the currently executing subroutine is looking for an array value. Returns false if the context is looking for a scalar.

```
return wantarray ? () : undef;
```

System information

getpwnam(NAME)
getgrnam(NAME)
gethostbyname(NAME)
getnetbyname(NAME)
getprotobyname(NAME)
getpwuid(UID)
getgrgid(GID)
getservbyname(NAME,PROTO)
gethostbyaddr(ADDR,ADDRTYPE)
getnetbyaddr(ADDR,ADDRTYPE)
getprotobynumber(NUMBER)
getservbyport(PORT,PROTO)
getpwent
getgrent

gethostent
getnetent
getprotoent
getservent
setpwent
setgrent
sethostent(STAYOPEN)
setnetent(STAYOPEN)
setprotoent(STAYOPEN)
setservent(STAYOPEN)
endpwent
endgrent
endhostent
endnetent
endprotoent
endservent

These routines perform the same functions as their counterparts in the system library. Within an array context, the return values from the various get routines are as follows:

```
($name,$passwd,$uid,$gid,
   $quota,$comment,$gcos,$dir,$shell) = getpw...
($name,$passwd,$gid,$members) = getgr...
($name,$aliases,$addrtype,$length,@addrs) = gethost...
($name,$aliases,$addrtype,$net) = getnet...
($name,$aliases,$proto) = getproto...
($name,$aliases,$port,$proto) = getserv...
```

(If the entry doesn’t exist you get a null list.)

Within a scalar context, you get the name, unless the function was a lookup by name, in which case you get the other thing, whatever it is. (If the entry doesn’t exist you get the undefined value.) For example:

```
$uid = getpwnam
$name = getpwuid
$name = getpwent
$gid = getgrnam
$name = getgrgid
$name = getgrent
etc.
```

The $members value returned by getgr... is a space separated list of the login names of the members of the group.

For the gethost... functions, if the h_errno variable is supported in C, it will be returned to you via $? if the function call fails. The @addrs value returned by a successful call is a list of the raw addresses returned by the corresponding system library call. In the Internet domain, each address is four bytes long and you can unpack it by saying something like:

```
($a,$b,$c,$d) = unpack('C4',$addr[0]);
```

Precedence

Perl operators have the following associativity and precedence:

```
nonassoc        print printf exec system sort reverse
                chmod chown kill unlink utime die return
left            ,
right           = += -= *= etc.
right           ?:
nonassoc        ..
left            ||
left            &&
left            | ^
left            &
nonassoc        == != <=> eq ne cmp
nonassoc        &lt; &gt; &lt;= &gt;= lt gt le ge
nonassoc        chdir exit eval reset sleep rand umask
nonassoc        -r -w -x etc.
left            << >>
left            + - .
left            * / % x
left            =~!~
right           ! ~ and unary minus
right           **
nonassoc        ++ --
left
```

As mentioned earlier, if any list operator (print, etc.) or any unary operator (chdir, etc.) is followed by a left parenthesis as the next token on the same line, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call. Examples:

```
chdir $foo || die;              # (chdir $foo) || die
chdir($foo) || die;             # (chdir $foo) || die
chdir ($foo) || die;            # (chdir $foo) || die
chdir +($foo) || die;           # (chdir $foo) || die
```

but, because * is higher precedence than ||:

```
chdir $foo * 20;                # chdir ($foo * 20)
chdir($foo) * 20;               # (chdir $foo) * 20
chdir ($foo) * 20;              # (chdir $foo) * 20
chdir +($foo) * 20;             # chdir ($foo * 20)

rand 10 * 20;           # rand (10 * 20)
rand(10) * 20;          # (rand 10) * 20
rand (10) * 20;         # (rand 10) * 20
rand +(10) * 20;                # rand (10 * 20)
```

In the absence of parentheses, the precedence of list operators such as print, sort or chmod is either very high or very low depending on whether you look at the left side of operator or the right side of it. For example, in

```
@ary = (1, 3, sort 4, 2);
print @ary;             # prints 1324
```

the commas on the right of the sort are evaluated before the sort, but the commas on the left are evaluated after. In other words, list operators tend to gobble up all the arguments that follow them, and then act like a simple term with regard to the preceding expression. Note that you have to be careful with parens:

```
        # These evaluate exit before doing the print:
        print($foo, exit);      # Obviously not what you want.
        print $foo, exit;       # Nor is this.

        # These do the print before evaluating exit:
        (print $foo), exit;     # This is what you want.
        print($foo), exit;      # Or this.
        print ($foo), exit;     # Or even this.
```

Also note that

```
        print ($foo & 255) + 1, "\n";
```

probably doesn't do what you expect at first glance.

Subroutines

A subroutine may be declared as follows:

```
    sub NAME BLOCK
```

Any arguments passed to the routine come in as array @_, that is ($_[0], $_[1], ...). The array @_ is a local array, but its values are references to the actual scalar parameters. The return value of the subroutine is the value of the last expression evaluated, and can be either an array value or a scalar value. Alternately, a return statement may be used to specify the returned value and exit the subroutine. To create local variables see the *local* operator.

A subroutine is called using the *do* operator or the & operator.

Example:

```
        sub MAX {
                local($max) = pop(@_);
                foreach $foo (@_) {
                        $max = $foo if $max < $foo;
                }
                $max;
        }

        ...
        $bestday = &MAX($mon,$tue,$wed,$thu,$fri);
```

Example:

```
        # get a line, combining continuation lines
        #  that start with whitespace
        sub get_line {
                $thisline = $lookahead;
                line: while ($lookahead = <STDIN>) {
                        if ($lookahead =~/^[ \^\t]/) {
                                $thisline .= $lookahead;
```

```
                }
                else {
                        last line;
                }
        }
        $thisline;
}

$lookahead = <STDIN>;   # get first line
while ($_ = do get_line()) {
        ...
}
```

Use array assignment to a local list to name your formal arguments:

```
sub maybeset {
        local($key, $value) = @_;
        $foo{$key} = $value unless $foo{$key};
}
```

This also has the effect of turning call-by-reference into call-by-value, since the assignment copies the values.

Subroutines may be called recursively. If a subroutine is called using the & form, the argument list is optional. If omitted, no @_ array is set up for the subroutine; the @_ array at the time of the call is visible to subroutine instead.

```
do foo(1,2,3);          # pass three arguments
&foo(1,2,3);            # the same

do foo();               # pass a null list
&foo();                 # the same
&foo;                   # pass no arguments--more efficient
```

Passing By Reference

Sometimes you don't want to pass the value of an array to a subroutine but rather the name of it, so that the subroutine can modify the global copy of it rather than working with a local copy. In perl you can refer to all the objects of a particular name by prefixing the name with a star: *foo. When evaluated, it produces a scalar value that represents all the objects of that name, including any filehandle, format or subroutine. When assigned to within a local() operation, it causes the name mentioned to refer to whatever * value was assigned to it.

Example:

```
sub doubleary {
    local(*someary) = @_;
    foreach $elem (@someary) {
        $elem *= 2;
    }
}
do doubleary(*foo);
do doubleary(*bar);
```

Assignment to *name is currently recommended only inside a local(). You can actually assign to *name anywhere, but the previous referent of *name may be stranded forever. This may or may not bother you.

Note that scalars are already passed by reference, so you can modify scalar arguments without using this mechanism by referring explicitly to the $_[nnn] in question. You can modify all the elements of an array by passing all the elements as scalars, but you have to use the * mechanism to push, pop or change the size of an array. The * mechanism will probably be more efficient in any case.

Since a *name value contains unprintable binary data, if it is used as an argument in a print, or as a %s argument in a printf or sprintf, it then has the value '*name', just so it prints out pretty.

Even if you don't want to modify an array, this mechanism is useful for passing multiple arrays in a single LIST, since normally the LIST mechanism will merge all the array values so that you can't extract out the individual arrays.

Regular Expressions

The patterns used in pattern matching are regular expressions such as those supplied in the Version 8 regexp routines. (In fact, the routines are derived from Henry Spencer's freely redistributable reimplementation of the V8 routines.) In addition, \w matches an alphanumeric character (including "_") and \W a nonalphanumeric. Word boundaries may be matched by \b, and non-boundaries by \B. A whitespace character is matched by \s, non-whitespace by \S. A numeric character is matched by \d, non-numeric by \D. You may use \w, \s and \d within character classes. Also, \n, \r, \f, \t and \NNN have their normal interpretations. Within character classes \b represents backspace rather than a word boundary. Alternatives may be separated by |. The bracketing construct (\ ...\ ) may also be used, in which case \<digit> matches the digit'th substring. (Outside of the pattern, always use $ instead of \ in front of the digit. The scope of $<digit> (and $\`, $& and $') extends to the end of the enclosing BLOCK or eval string, or to the next pattern match with subexpressions. The \<digit> notation sometimes works outside the current pattern, but should not be relied upon.) You may have as many parentheses as you wish. If you have more than 9 substrings, the variables $10, $11, ... refer to the corresponding substring. Within the pattern, \10, \11, etc. refer back to substrings if there have been at least that many left parens before the backreference. Otherwise (for backward compatibilty) \10 is the same as \010, a backspace, and \11 the same as \011, a tab. And so on. (\1 through \9 are always backreferences.)

$+ returns whatever the last bracket match matched. $& returns the entire matched string. ($0 used to return the same thing, but not any more.) $` returns everything before the matched string. $' returns everything after the matched string. Examples:

```
s/^([^ ]*) *([^ ]*)/$2 $1/;     # swap first two words

if (/Time: (..):(..):(..)/) {
        $hours = $1;
        $minutes = $2;
        $seconds = $3;
}
```

By default, the ^ character is only guaranteed to match at the beginning of the string, the $ character only at the end (or before the newline at the end) and *perl* does certain optimizations with the assumption that the string contains only one line. The behavior of ^ and $ on embedded newlines will be inconsistent. You may, however, wish to treat a string as a multi-line buffer, such that the ^ will match after any newline within the string, and $ will match before any newline. At the cost of a little more overhead, you can do this by setting the variable $* to 1. Setting it back to 0 makes *perl* revert to its old behavior.

To facilitate multi-line substitutions, the . character never matches a newline (even when $* is 0). In particular, the following leaves a newline on the $_ string:

```
$_ = <STDIN>;
s/.*(some_string).*/$1/;
```

`If the newline is unwanted, try one of`

```
s/.*(some_string).*\n/$1/;
s/.*(some_string)[^\000]*/$1/;
s/.*(some_string)(.|\n)*/$1/;
chop; s/.*(some_string).*/$1/;
/(some_string)/ && ($_ = $1);
```

Any item of a regular expression may be followed with digits in curly brackets of the form {n,m}, where n gives the minimum number of times to match the item and m gives the maximum. The form {n} is equivalent to {n,n} and matches exactly n times. The form {n,} matches n or more times. (If a curly bracket occurs in any other context, it is treated as a regular character.) The * modifier is equivalent to {0,}, the + modifier to {1,} and the ? modifier to {0,1}. There is no limit to the size of n or m, but large numbers will chew up more memory.

You will note that all backslashed metacharacters in *perl* are alphanumeric, such as \b, \w, \n. Unlike some other regular expression languages, there are no backslashed symbols that aren't alphanumeric. So anything that looks like \\, \(, \), \<, \>, \{, or \} is always interpreted as a literal character, not a metacharacter. This makes it simple to quote a string that you want to use for a pattern but that you are afraid might contain metacharacters. Simply quote all the non-alphanumeric characters:

```
$pattern =~s/(\W)/\\$1/g;
```

Formats

Output record formats for use with the *write* operator may declared as follows:

```
format NAME =
FORMLIST
.
```

If name is omitted, format "STDOUT" is defined. FORMLIST consists of a sequence of lines, each of which may be of one of three types:

1. A comment.
2. A "picture" line giving the format for one output line.
3. An argument line supplying values to plug into a picture line.

Picture lines are printed exactly as they look, except for certain fields that substitute values into the line. Each picture field starts with either @ or ^. The @ field (not to be confused with the array marker @) is the normal case; ^ fields are used to do rudimentary multi-line text block filling. The length of the field is supplied by padding out the field with multiple <, >, or | characters to specify, respectively, left justification, right justification, or centering. As an alternate form of right justification, you may also use # characters (with an optional .) to specify a numeric field. (Use of ^ instead of @ causes the field to be blanked if undefined.) If any of the values supplied for these fields contains a newline, only the text up to the newline is printed. The special field @* can be used for printing multi-line values. It should appear by itself on a line.

The values are specified on the following line, in the same order as the picture fields. The values should be separated by commas.

Picture fields that begin with ^ rather than @ are treated specially. The value supplied must be a scalar variable name which contains a text string. *Perl* puts as much text as it can into the field, and then chops off the front of the string so that the next time the variable is referenced, more of the text can be printed. Normally you would use a sequence of fields in a vertical stack to print out a block of text. If you like, you can end the final field with ..., which will appear in the output if the text was too long to appear in its entirety. You can change which characters are legal to break on by changing the variable $: to a list of the desired characters.

Since use of ^ fields can produce variable length records if the text to be formatted is short, you can suppress blank lines by putting the tilde (~) character anywhere in the line. (Normally you should put it in the front if possible, for visibility.) The tilde will be translated to a space upon output. If you put a second tilde contiguous to the first, the line will be repeated until all the fields on the line are exhausted. (If you use a field of the @ variety, the expression you supply had better not give the same value every time forever!)

Examples:

```
# a report on the /etc/passwd file
format STDOUT_TOP =
                        Passwd File
Name                Login    Office   Uid   Gid Home
------------------------------------------------------------------
.
format STDOUT =
@<<<<<<<<<<<<<<<<<< @||||||| @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<
$name,              $login,  $office,$uid,$gid, $home
.

# a report from a bug report form
format STDOUT_TOP =
                        Bug Reports
@<<<<<<<<<<<<<<<<<<<<<<<     @|||         @>>>>>>>>>>>>>>>>>>>>>>>
$system,                      $%,         $date
------------------------------------------------------------------
.
format STDOUT =
Subject: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
         $subject
Index: @<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
       $index,                       $description
```

```
Priority: @<<<<<<<<<< Date: @<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
          $priority,        $date,   $description
From: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $from,                         $description
Assigned to: @<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
             $programmer,            $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<...
                                     $description
.
```

It is possible to intermix prints with writes on the same output channel, but you'll have to handle $- (lines left on the page) yourself.

If you are printing lots of fields that are usually blank, you should consider using the reset operator between records. Not only is it more efficient, but it can prevent the bug of adding another field and forgetting to zero it.

Interprocess Communication

The IPC facilities of perl are built on the Berkeley socket mechanism. If you don't have sockets, you can ignore this section. The calls have the same names as the corresponding system calls, but the arguments tend to differ, for two reasons. First, perl file handles work differently than C file descriptors. Second, perl already knows the length of its strings, so you don't need to pass that information. Here is a sample client (untested):

```
($them,$port) = @ARGV;
$port = 2345 unless $port;
$them = 'localhost' unless $them;

$SIG{'INT'} = 'dokill';
sub dokill { kill 9,$child if $child; }

require 'sys/socket.ph';

$sockaddr = 'S n a4 x8';
chop($hostname = `hostname`);

($name, $aliases, $proto) = getprotobyname('tcp');
($name, $aliases, $port) = getservbyname($port, 'tcp')
        unless $port =~/^\d+$/;
($name, $aliases, $type, $len, $thisaddr) =
                                gethostbyname($hostname);
($name, $aliases, $type, $len, $thataddr) = gethostbyname($them);

$this = pack($sockaddr, &AF_INET, 0, $thisaddr);
$that = pack($sockaddr, &AF_INET, $port, $thataddr);
```

```
socket(S, &PF_INET, &SOCK_STREAM, $proto) || die "socket: $!";
bind(S, $this) || die "bind: $!";
connect(S, $that) || die "connect: $!";

select(S); $| = 1; select(stdout);

if ($child = fork) {
        while (<>) {
                print S;
        }
        sleep 3;
        do dokill();
}
else {
        while (<S>) {
                print;
        }
}
```

And here's a server:

```
($port) = @ARGV;
$port = 2345 unless $port;

require 'sys/socket.ph';

$sockaddr = 'S n a4 x8';

($name, $aliases, $proto) = getprotobyname('tcp');
($name, $aliases, $port) = getservbyname($port, 'tcp')
        unless $port =~/^\d+$/;

$this = pack($sockaddr, &AF_INET, $port, "\0\0\0\0");

select(NS); $| = 1; select(stdout);

socket(S, &PF_INET, &SOCK_STREAM, $proto) || die "socket: $!";
bind(S, $this) || die "bind: $!";
listen(S, 5) || die "connect: $!";

select(S); $| = 1; select(stdout);

for (;;) {
        print "Listening again\n";
        ($addr = accept(NS,S)) || die $!;
        print "accept ok\n";

        ($af,$port,$inetaddr) = unpack($sockaddr,$addr);
        @inetaddr = unpack('C4',$inetaddr);
        print "$af $port @inetaddr\n";

        while (<NS>) {
                print;
                print NS;
        }
}
```

Predefined Names

The following names have special meaning to *perl*. I could have used alphabetic symbols for some of these, but I didn’t want to take the chance that someone would say reset "a-zA-Z" and wipe them all out. You’ll just have to suffer along with these silly symbols. Most of them have reasonable mnemonics, or analogues in one of the shells.

$_ The default input and pattern-searching space. The following pairs are equivalent:

```
while (<>) {... # only equivalent in while!
while ($_ = <>) {...

/^Subject:/
$_ =~/^Subject:/

y/a-z/A-Z/
$_ =~y/a-z/A-Z/

chop
chop($_)
```

(Mnemonic: underline is understood in certain operations.)

$. The current input line number of the last filehandle that was read. Readonly. Remember that only an explicit close on the filehandle resets the line number. Since <> never does an explicit close, line numbers increase across ARGV files (but see examples under eof). (Mnemonic: many programs use . to mean the current line number.)

$/ The input record separator, newline by default. Works like *awk*’s RS variable, including treating blank lines as delimiters if set to the null string. You may set it to a multicharacter string to match a multi-character delimiter. Note that setting it to "\n\n" means something slightly different than setting it to "", if the file contains consecutive blank lines. Setting it to "" will treat two or more consecutive blank lines as a single blank line. Setting it to "\n\n" will blindly assume that the next input character belongs to the next paragraph, even if it’s a newline. (Mnemonic: / is used to delimit line boundaries when quoting poetry.)

$, The output field separator for the print operator. Ordinarily the print operator simply prints out the comma separated fields you specify. In order to get behavior more like *awk*, set this variable as you would set *awk*’s OFS variable to specify what is printed between fields. (Mnemonic: what is printed when there is a , in your print statement.)

$"" This is like $, except that it applies to array values interpolated into a double-quoted string (or similar interpreted string). Default is a space. (Mnemonic: obvious, I think.)

$\ The output record separator for the print operator. Ordinarily the print operator simply prints out the comma separated fields you specify, with no trailing newline or record separator assumed. In order to get behavior more like *awk*, set this variable as you would set *awk*’s ORS variable to specify what is printed at the end of the print. (Mnemonic: you set $\ instead of adding \n at the end of the print. Also, it’s just like /, but it’s what you get "back" from *perl*.)

$# The output format for printed numbers. This variable is a half-hearted attempt to emulate *awk*’s OFMT variable. There are times, however, when *awk* and *perl* have differing notions of what is in fact numeric. Also, the initial value is %.20g rather than %.6g, so you need to set $# explicitly to get *awk*'s value. (Mnemonic: # is the number sign.)

$% The current page number of the currently selected output channel. (Mnemonic: % is page number in nroff.)

$= The current page length (printable lines) of the currently selected output channel. Default is 60. (Mnemonic: = has horizontal lines.)

$- The number of lines left on the page of the currently selected output channel. (Mnemonic: lines_on_page - lines_printed.)

$~ The name of the current report format for the currently selected output channel. Default is name of the filehandle. (Mnemonic: brother to $^.)

$^ The name of the current top-of-page format for the currently selected output channel. Default is name of the filehandle with "_TOP" appended. (Mnemonic: points to top of page.)

$| If set to nonzero, forces a flush after every write or print on the currently selected output channel. Default is 0. Note that *STDOUT* will typically be line buffered if output is to the terminal and block buffered otherwise. Setting this variable is useful primarily when you are outputting to a pipe, such as when you are running a *perl* script under rsh and want to see the output as it's happening. (Mnemonic: when you want your pipes to be piping hot.)

$$ The process number of the *perl* running this script. (Mnemonic: same as shells.)

$? The status returned by the last pipe close, backtick (\`\`) command or *system* operator. Note that this is the status word returned by the wait() system call, so the exit value of the subprocess is actually ($? >> 8). $? & 255 gives which signal, if any, the process died from, and whether there was a core dump. (Mnemonic: similar to sh and ksh.)

$& The string matched by the last successful pattern match (not counting any matches hidden within a BLOCK or eval enclosed by the current BLOCK). (Mnemonic: like & in some editors.)

$\` The string preceding whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or eval enclosed by the current BLOCK). (Mnemonic: \` often precedes a quoted string.)

$' The string following whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or eval enclosed by the current BLOCK). (Mnemonic: ' often follows a quoted string.) Example:

```
$_ = 'abcdefghi';
/def/;
print "$\`:$&:$'\n";     # prints abc:def:ghi
```

$+ The last bracket matched by the last search pattern. This is useful if you don't know which of a set of alternative patterns matched. For example:

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

(Mnemonic: be positive and forward looking.)

$* Set to 1 to do multiline matching within a string, 0 to tell *perl* that it can assume that strings contain a single line, for the purpose of optimizing pattern matches. Pattern matches on strings containing multiple newlines can produce confusing results when $* is 0. Default is 0. (Mnemonic: * matches multiple things.) Note that this variable only influences the interpretation of ^ and $. A literal newline can be searched for even when $* == 0.

$0 Contains the name of the file containing the *perl* script being executed. Assigning to $0 modifies the argument area that the ps(1) program sees. (Mnemonic: same as sh and ksh.)

$<digit>
Contains the subpattern from the corresponding set of parentheses in the last pattern matched, not counting patterns matched in nested blocks that have been exited already. (Mnemonic: like \digit.)

$[The index of the first element in an array, and of the first character in a substring. Default is 0, but you could set it to 1 to make *perl* behave more like *awk* (or Fortran) when subscripting and when evaluating the index() and substr() functions. (Mnemonic: [begins subscripts.)

$] The string printed out when you say "perl -v". It can be used to determine at the beginning of a script whether the perl interpreter executing the script is in the right range of versions. If used in a numeric context, returns the version + patchlevel / 1000. Example:

```
# see if getc is available
($version,$patchlevel) =
         $] =~/(\d+\.\d+).*\nPatch level: (\d+)/;
print STDERR "(No filename completion available.)\n"
         if $version * 1000 + $patchlevel < 2016;
```

or, used numerically,

```
warn "No checksumming!\n" if $] < 3.019;
```

(Mnemonic: Is this version of perl in the right bracket?)

$; The subscript separator for multi-dimensional array emulation. If you refer to an associative array element as

```
$foo{$a,$b,$c}
```

it really means

```
$foo{join($;, $a, $b, $c)}
```

But don't put

```
@foo{$a,$b,$c}          # a slice--note the @
```

which means

```
($foo{$a},$foo{$b},$foo{$c})
```

Default is "\034", the same as SUBSEP in *awk*. Note that if your keys contain binary data there might not be any safe value for $;. (Mnemonic: comma (the syntactic subscript separator) is a semi-semicolon. Yeah, I know, it's pretty lame, but $, is already taken for something more important.)

$! If used in a numeric context, yields the current value of errno, with all the usual caveats. (This means that you shouldn't depend on the value of $! to be anything in particular unless you've gotten a specific error return indicating a system error.) If used in a string context, yields the corresponding system error string. You can assign to $! in order to set errno if, for instance, you want $! to return the string for error n, or you want to set the exit value for the die operator. (Mnemonic: What just went bang?)

$@ The perl syntax error message from the last eval command. If null, the last eval parsed and executed correctly (although the operations you invoked may have failed in the normal fashion). (Mnemonic: Where was the syntax error "at"?)

$< The real uid of this process. (Mnemonic: it's the uid you came FROM, if you're running setuid.)

$> The effective uid of this process. Example:

```
$< = $>;          # set real uid to the effective uid
($<,$>) = ($>,$<);       # swap real and effective uid
```

(Mnemonic: it's the uid you went TO, if you're running setuid.) Note: $< and $> can only be swapped on machines supporting setreuid().

$(The real gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by getgid(), and the subsequent ones by getgroups(), one of which may be the same as the first number. (Mnemonic: parentheses are used to GROUP things. The real gid is the group you LEFT, if you're running setgid.)

$) The effective gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by getegid(), and the subsequent ones by getgroups(), one of which may be the same as the first number. (Mnemonic: parentheses are used to GROUP things. The effective gid is the group that's RIGHT for you, if you're running setgid.)

Note: $<, $>, $(and $) can only be set on machines that support the corresponding set[re][ug]id() routine. $(and $) can only be swapped on machines supporting setregid().

$: The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. Default is "\ \n-", to break on whitespace or hyphens. (Mnemonic: a "colon" in poetry is a part of a line.)

$^D The current value of the debugging flags. (Mnemonic: value of **-D** switch.)

$^F The maximum system file descriptor, ordinarily 2. System file descriptors are passed to subprocesses, while higher file descriptors are not. During an open, system file descriptors are preserved even if the open fails. Ordinary file descriptors are closed before the open is attempted.

$^I The current value of the inplace-edit extension. Use undef to disable inplace editing. (Mnemonic: value of **-i** switch.)

$^L What formats output to perform a formfeed. Default is \f.

$^P The internal flag that the debugger clears so that it doesn't debug itself. You could conceivable disable debugging yourself by clearing it.

$^T The time at which the script began running, in seconds since the epoch. The values returned by the **-M , -A** and **-C** filetests are based on this value.

$^W The current value of the warning switch. (Mnemonic: related to the **-w** switch.)

$^X The name that Perl itself was executed as, from argv[0].

$ARGV
contains the name of the current file when reading from <>.

@ARGV
The array ARGV contains the command line arguments intended for the script. Note that $#ARGV is the generally number of arguments minus one, since $ARGV[0] is the first argument, NOT the command name. See $0 for the command name.

@INC
The array INC contains the list of places to look for *perl* scripts to be evaluated by the "do EXPR" command or the "require" command. It initially consists of the arguments to any **-I** command line switches, followed by the default *perl* library, probably "/usr/local/lib/perl", followed by ".", to represent the current directory.

%INC
The associative array INC contains entries for each filename that has been included via "do" or "require". The key is the filename you specified, and the value is the location of the file actually found. The "require" command uses this array to determine whether a given file has already been included.

$ENV{expr}
The associative array ENV contains your current environment. Setting a value in ENV changes the environment for child processes.

$SIG{expr}
The associative array SIG is used to set signal handlers for various signals. Example:

```
sub handler {   # 1st argument is signal name
        local($sig) = @_;
        print "Caught a SIG$sig--shutting down\n";
```

```
        close(LOG);
        exit(0);
}

$SIG{'INT'} = 'handler';
$SIG{'QUIT'} = 'handler';
...
$SIG{'INT'} = 'DEFAULT';        # restore default action
$SIG{'QUIT'} = 'IGNORE';        # ignore SIGQUIT
```

The SIG array only contains values for the signals actually set within the perl script.

Packages

Perl provides a mechanism for alternate namespaces to protect packages from stomping on each others variables. By default, a perl script starts compiling into the package known as "main". By use of the *package* declaration, you can switch namespaces. The scope of the package declaration is from the declaration itself to the end of the enclosing block (the same scope as the local() operator). Typically it would be the first declaration in a file to be included by the "require" operator. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a single quote. If the package name is null, the "main" package as assumed.

Only identifiers starting with letters are stored in the packages symbol table. All other symbols are kept in package "main". In addition, the identifiers STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, INC and SIG are forced to be in package "main", even when used for other purposes than their built-in one. Note also that, if you have a package called "m", "s" or "y", the you can't use the qualified form of an identifier since it will be interpreted instead as a pattern match, a substitution or a translation.

Eval'ed strings are compiled in the package in which the eval was compiled in. (Assignments to $SIG{}, however, assume the signal handler specified is in the main package. Qualify the signal handler name if you wish to have a signal handler in a package.) For an example, examine perldb.pl in the perl library. It initially switches to the DB package so that the debugger doesn't interfere with variables in the script you are trying to debug. At various points, however, it temporarily switches back to the main package to evaluate various expressions in the context of the main package.

The symbol table for a package happens to be stored in the associative array of that name prepended with an underscore. The value in each entry of the associative array is what you are referring to when you use the *name notation. In fact, the following have the same effect (in package main, anyway), though the first is more efficient because it does the symbol table lookups at compile time:

```
local(*foo) = *bar;
local($_main{'foo'}) = $_main{'bar'};
```

You can use this to print out all the variables in a package, for instance. Here is dumpvar.pl from the perl library:

```
package dumpvar;
```

```
sub main'dumpvar {
    ($package) = @_;
    local(*stab) = eval("*_$package");
    while (($key,$val) = each(%stab)) {
        {
            local(*entry) = $val;
            if (defined $entry) {
                print "\$$key = '$entry'\n";
            }
            if (defined @entry) {
                print "\@$key = (\n";
                foreach $num ($[ .. $#entry) {
                    print "  $num\t'",$entry[$num],"'\n";
                }
                print ")\n";
            }
            if ($key ne "_$package" && defined %entry) {
                print "\%$key = (\n";
                foreach $key (sort keys(%entry)) {
                    print "  $key\t'",$entry{$key},"'\n";
                }
                print ")\n";
            }
        }
    }
}
```

Note that, even though the subroutine is compiled in package dumpvar, the name of the subroutine is qualified so that its name is inserted into package "main".

Style

Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read.

1. Just because you CAN do something a particular way doesn't mean that you SHOULD do it that way. *Perl* is designed to give you several ways to do anything, so consider picking the most readable one. For instance open(FOO,$foo) || die "Can't open $foo: $!"; is better than die "Can't open $foo: $!" unless open(FOO,$foo); because the second way hides the main point of the statement in a modifier. On the other hand print "Starting analysis\n" if $verbose; is better than $verbose && print "Starting analysis\n"; since the main point isn't whether the user typed -v or not.

   Similarly, just because an operator lets you assume default arguments doesn't mean that you have to make use of the defaults. The defaults are there for lazy systems programmers writing one-shot programs. If you want your program to be readable, consider supplying the argument.

   Along the same lines, just because you *can* omit parentheses in many places doesn't mean that you ought to:

   ```
   return print reverse sort num values array;
   return print(reverse(sort num (values(%array))));
   ```

When in doubt, parenthesize. At the very least it will let some poor schmuck bounce on the % key in vi.

Even if you aren't in doubt, consider the mental welfare of the person who has to maintain the code after you, and who will probably put parens in the wrong place.

2. Don't go through silly contortions to exit a loop at the top or the bottom, when *perl* provides the "last" operator so you can exit in the middle. Just outdent it a little to make it more visible:

```
line:
    for (;;) {
        statements;
    last line if $foo;
        next line if /^#/;
        statements;
    }
```

3. Don't be afraid to use loop labels--they're there to enhance readability as well as to allow multi-level loop breaks. See last example.
4. For portability, when using features that may not be implemented on every machine, test the construct in an eval to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test $] to see if it will be there.
5. Choose mnemonic identifiers.
6. Be consistent.

Debugging

If you invoke *perl* with a **-d** switch, your script will be run under a debugging monitor. It will halt before the first executable statement and ask you for a command, such as:

h	Prints out a help message.

T	Stack trace.

s	Single step. Executes until it reaches the beginning of another statement.

n	Next. Executes over subroutine calls, until it reaches the beginning of the next statement.

f	Finish. Executes statements until it has finished the current subroutine.

c	Continue. Executes until the next breakpoint is reached.

c line	Continue to the specified line. Inserts a one-time-only breakpoint at the specified line.

<CR>
	Repeat last n or s.

l min+incr
	List incr+1 lines starting at min. If min is omitted, starts where last listing left off. If incr is

omitted, previous value of incr is used.

l min-max
List lines in the indicated range.

l line List just the indicated line.

l List next window.

\- List previous window.

w line
List window around line.

l subname
List subroutine. If it's a long subroutine it just lists the beginning. Use "l" to list more.

/pattern/
Regular expression search forward for pattern; the final / is optional.

?pattern?
Regular expression search backward for pattern; the final ? is optional.

L List lines that have breakpoints or actions.

S Lists the names of all subroutines.

t Toggle trace mode on or off.

b line condition
Set a breakpoint. If line is omitted, sets a breakpoint on the line that is about to be executed. If a condition is specified, it is evaluated each time the statement is reached and a breakpoint is taken only if the condition is true. Breakpoints may only be set on lines that begin an executable statement.

b subname condition
Set breakpoint at first executable line of subroutine.

<dt>d line
Delete breakpoint. If line is omitted, deletes the breakpoint on the line that is about to be executed.

D Delete all breakpoints.

a line command
Set an action for line. A multi-line command may be entered by backslashing the newlines.

A Delete all line actions.

< command
: Set an action to happen before every debugger prompt. A multi-line command may be entered by backslashing the newlines.

> command
: Set an action to happen after the prompt when you’ve just given a command to return to executing the script. A multi-line command may be entered by backslashing the newlines.

V package
: List all variables in package. Default is main package.

! number
: Redo a debugging command. If number is omitted, redoes the previous command.

! -number
: Redo the command that was that many commands ago.

H -number
: Display last n commands. Only commands longer than one character are listed. If number is omitted, lists them all.

q or ^D
: Quit.

command
: Execute command as a perl statement. A missing semicolon will be supplied.

p expr
: Same as "print DB’OUT expr". The DB’OUT filehandle is opened to /dev/tty, regardless of where STDOUT may be redirected to.

If you want to modify the debugger, copy perldb.pl from the perl library to your current directory and modify it as necessary. (You’ll also have to put -I. on your command line.) You can do some customization by setting up a .perldb file which contains initialization code. For instance, you could make aliases like these:

```
$DB’alias{’len’} = ’s/^len(.*)/p length($1)/’;
$DB’alias{’stop’} = ’s/^stop (at|in)/b/’;
$DB’alias{’.’} =
  ’s/^\./p "\$DB\’sub(\$DB\’line):\t",\$DB\’line[\$DB\’line]/’;
```

Setuid Scripts

Perl is designed to make it easy to write secure setuid and setgid scripts. Unlike shells, which are based on multiple substitution passes on each line of the script, *perl* uses a more conventional evaluation scheme with fewer hidden "gotchas". Additionally, since the language has more built-in functionality, it has to rely less upon external (and possibly untrustworthy) programs to accomplish its purposes.

In an unpatched 4.2 or 4.3bsd kernel, setuid scripts are intrinsically insecure, but this kernel feature can be disabled. If it is, *perl* can emulate the setuid and setgid mechanism when it notices the otherwise useless setuid/gid bits on perl scripts. If the kernel feature isn't disabled, *perl* will complain loudly that your setuid script is insecure. You'll need to either disable the kernel setuid script feature, or put a C wrapper around the script.

When perl is executing a setuid script, it takes special precautions to prevent you from falling into any obvious traps. (In some ways, a perl script is more secure than the corresponding C program.) Any command line argument, environment variable, or input is marked as "tainted", and may not be used, directly or indirectly, in any command that invokes a subshell, or in any command that modifies files, directories or processes. Any variable that is set within an expression that has previously referenced a tainted value also becomes tainted (even if it is logically impossible for the tainted value to influence the variable). For example:

```
$foo = shift;                   # $foo is tainted
$bar = $foo,'bar';              # $bar is also tainted
$xxx = <>;              # Tainted
$path = $ENV{'PATH'};           # Tainted, but see below
$abc = 'abc';                   # Not tainted

system "echo $foo";             # Insecure
system "/bin/echo", $foo;       # Secure (doesn't use sh)
system "echo $bar";             # Insecure
system "echo $abc";             # Insecure until PATH set

$ENV{'PATH'} = '/bin:/usr/bin';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';

$path = $ENV{'PATH'};           # Not tainted
system "echo $abc";             # Is secure now!

open(FOO,"$foo");               # OK
open(FOO,">$foo");              # Not OK

open(FOO,"echo $foo|");         # Not OK, but...
open(FOO,"-|") || exec 'echo', $foo;    # OK

$zzz = `echo $foo`;             # Insecure, zzz tainted

unlink $abc,$foo;               # Insecure
umask $foo;                     # Insecure

exec "echo $foo";               # Insecure
exec "echo", $foo;              # Secure (doesn't use sh)
exec "sh", '-c', $foo;          # Considered secure, alas
```

The taintedness is associated with each scalar value, so some elements of an array can be tainted, and others not.

If you try to do something insecure, you will get a fatal error saying something like "Insecure dependency" or "Insecure PATH". Note that you can still write an insecure system call or exec, but only by explicitly doing something like the last example above. You can also bypass the tainting mechanism by referencing subpatterns--\c *perl* presumes that if you reference a substring using $1, $2, etc, you knew what you were doing when you wrote the pattern:

```
$ARGV[0] =~/^-P(\w+)$/;
$printer = $1;          # Not tainted
```

This is fairly secure since \w+ doesn't match shell metacharacters. Use of .+ would have been insecure, but *perl* doesn't check for that, so you must be careful with your patterns. This is the ONLY mechanism for untainting user supplied filenames if you want to do file operations on them (unless you make $> equal to $<).

It's also possible to get into trouble with other operations that don't care whether they use tainted values. Make judicious use of the file tests in dealing with any user-supplied filenames. When possible, do opens and such after setting $> = $<. *Perl* doesn't prevent you from opening tainted filenames for reading, so be careful what you print out. The tainting mechanism is intended to prevent stupid mistakes, not to remove the need for thought.

ENVIRONMENT

HOME
: Used if chdir has no argument.

LOGDIR
: Used if chdir has no argument and HOME is not set.

PATH
: Used in executing subprocesses, and in finding the script if -S is used.

PERLLIB
: A colon-separated list of directories in which to look for Perl library files before looking in the standard library and the current directory.

PERLDB
: The command used to get the debugger code. If unset, uses

```
require 'perldb.pl'
```

Apart from these, *perl* uses no other environment variables, except to make them available to the script being executed, and to child processes. However, scripts running setuid would do well to execute the following lines before doing anything else, just to keep people honest:

```
$ENV{'PATH'} = '/bin:/usr/bin';    # or whatever you need
$ENV{'SHELL'} = '/bin/sh' if $ENV{'SHELL'} ne '';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';
```

DIAGNOSTICS

Compilation errors will tell you the line number of the error, with an indication of the next token or token type that was to be examined. (In the case of a script passed to *perl* via **-e** switches, each **-e** is counted as one line.)

Setuid scripts have additional constraints that can produce error messages such as "Insecure dependency". See the section on setuid scripts.

TRAPS

Accustomed *awk* users should take special note of the following:

- Semicolons are required after all simple statements in *perl* (except at the end of a block). Newline is not a statement delimiter.
- Curly brackets are required on ifs and whiles.
- Variables begin with $ or @ in *perl*.
- Arrays index from 0 unless you set $[. Likewise string positions in substr() and index().
- You have to decide whether your array has numeric or string indices.
- Associative array values do not spring into existence upon mere reference.
- You have to decide whether you want to use string or numeric comparisons.
- Reading an input line does not split it for you. You get to split it yourself to an array. And the *split* operator has different arguments.
- The current input line is normally in $_, not $0. It generally does not have the newline stripped. ($0 is the name of the program executed.)
- $<digit> does not refer to fields--it refers to substrings matched by the last match pattern.
- The *print* statement does not add field and record separators unless you set $, and $\.
- You must open your files before you print to them.
- The range operator is "..", not comma. (The comma operator works as in C.)
- The match operator is "=~", not "~". ("~" is the one's complement operator, as in C.)
- The exponentiation operator is "**", not "^". ("^" is the XOR operator, as in C.)
- The concatenation operator is ".", not the null string. (Using the null string would render "/pat/ /pat/" unparsable, since the third slash would be interpreted as a division operator--the tokener is in fact slightly context sensitive for operators like /, ?, and <. And in fact, . itself can be the beginning of a number.)
- *Next*, *exit* and *continue* work differently.
- The following variables work differently

```
Awk                     Perl
ARGC                    $#ARGV
ARGV[0]                 $0
FILENAME                $ARGV
FNR                     $. - something
FS                      (whatever you like)
NF                      $#Fld, or some such
NR                      $.
OFMT                    $#
OFS                     $,
ORS                     $\
RLENGTH                 length($&)
RS                      $/
RSTART                  length($\`)
SUBSEP                  $;
```

- When in doubt, run the *awk* construct through a2p and see what it gives you.

Cerebral C programmers should take note of the following:

- Curly brackets are required on ifs and whiles.
- You should use "elsif" rather than "else if"
- *Break* and *continue* become *last* and *next*, respectively.
- There's no switch statement.
- Variables begin with $ or @ in *perl*.
- Printf does not implement *.
- Comments begin with #, not /*.
- You can't take the address of anything.
- ARGV must be capitalized.
- The "system" calls link, unlink, rename, etc. return nonzero for success, not 0.
- Signal handlers deal with signal names, not numbers.

Seasoned *sed* programmers should take note of the following:

- Backreferences in substitutions use $ rather than \.
- The pattern matching metacharacters (,), and | do not have backslashes in front.
- The range operator is .. rather than comma.

Sharp shell programmers should take note of the following:

- The backtick operator does variable interpretation without regard to the presence of single quotes in the command.
- The backtick operator does no translation of the return value, unlike csh.
- Shells (especially csh) do several levels of substitution on each command line. *Perl* does substitution only in certain constructs such as double quotes, backticks, angle brackets and search patterns.
- Shells interpret scripts a little bit at a time. *Perl* compiles the whole program before executing it.
- The arguments are available via @ARGV, not $1, $2, etc.
- The environment is not automatically made available as variables.

ERRATA AND ADDENDA

The Perl book, *Programming Perl* , has the following omissions and goofs.

On page 5, the examples which read

```
eval "/usr/bin/perl
```

should read

```
eval "exec /usr/bin/perl
```

On page 195, the equivalent to the System V sum program only works for very small files. To do larger files, use

```
undef $/;
$checksum = unpack("%32C*",<>) % 32767;
```

The descriptions of alarm and sleep refer to signal SIGALARM. These should refer to SIGALRM.

The **-0** switch to set the initial value of $/ was added to Perl after the book went to press.

The **-l** switch now does automatic line ending processing.

The qx// construct is now a synonym for backticks.

$0 may now be assigned to set the argument displayed by *ps (1).*

The new @###.## format was omitted accidentally from the description on formats.

It wasn't known at press time that s///ee caused multiple evaluations of the replacement expression. This is to be construed as a feature.

(LIST) x $count now does array replication.

There is now no limit on the number of parentheses in a regular expression.

In double-quote context, more escapes are supported: \e, \a, \x1b, \c[, \l, \L, \u, \U, \E. The latter five control up/lower case translation.

The **$/** variable may now be set to a multi-character delimiter.

There is now a g modifier on ordinary pattern matching that causes it to iterate through a string finding multiple matches.

All of the $^X variables are new except for $^T.

The default top-of-form format for FILEHANDLE is now FILEHANDLE_TOP rather than top.

The eval {} and sort {} constructs were added in version 4.018.

The v and V (little-endian) template options for pack and unpack were added in 4.019.

BUGS

Perl is at the mercy of your machine's definitions of various operations such as type casting, atof() and sprintf().

If your stdio requires an seek or eof between reads and writes on a particular stream, so does *perl*. (This doesn't apply to sysread() and syswrite().)

While none of the built-in data types have any arbitrary size limits (apart from memory size), there are still a few arbitrary limits: a given identifier may not be longer than 255 characters, and no component of your PATH may be longer than 255 if you use -S. A regular expression may not compile to more than 32767 bytes internally.

Perl actually stands for Pathologically Eclectic Rubbish Lister, but don’t tell anyone I said that.