4

CALCULATIONS

In this chapter, we'll look at various one-liners for performing calculations, such as finding minimum and maximum elements, counting, shuffling and permuting words, and calculating dates and numbers. You'll also learn about the `-a`, `-M`, and `-F` command-line arguments, the `$,` special variable, and the `@{[ ... ]}` construction that lets you run code inside double quotes.

4.1 Check if a number is a prime

```
perl -lne '(1x$_) !~ /^1?$|^(11+?)\1+$/ && print "$_ is prime"'
```

This one-liner uses an ingenious regular expression by Abigail to detect whether a given number is a prime. (Don't take this regular

expression too seriously; I've included it for its artistic value. For serious purposes, use the `Math::Primality` module from CPAN to see whether a number is prime.)

Here's how this ingenious one-liner works: First, the number is converted into its unary representation by `(1x$_)`. For example, `5` is converted into `1x5`, which is `11111` (`1` repeated `5` times). Next, the unary number is tested against the regular expression. If it doesn't match, the number is a prime; otherwise it's a composite. The `!~` operator is the opposite of the `=~` operator and is true if the regular expression doesn't match.

The regular expression consists of two parts: The first part, `^1?$`, matches `1` and the empty string. The empty string and 1 are clearly not prime numbers, so this part of the regular expression discards them.

The second part, `^(11+?)\1+$`, determines whether two or more `1`s repeatedly make up the whole number. If so, the regular expression matches, which means the number is a composite. If not, it's a prime.

Now consider how the second part of the regular expression would act on the number 5. The number 5 in unary is 11111, so the `(11+?)` matches the first two `1`s, the back-reference `\1` becomes 11, and the whole regular expression now becomes `^11(11)+$`. Because it can't match five `1`s, it fails. Next, it attempts to match the first three `1`s. The back-reference becomes 111, and the whole regular expression becomes `^111(111)+$`, which doesn't match. The process repeats for 1111 and 11111, which also don't match, and as a result the entire regular expression doesn't match and the number is a prime.

What about the number 4? The number 4 is 1111 in unary. The `(11+?)` matches the first two `1`s. The back-reference `\1` becomes 11, and the regular expression becomes `^11(11)+$`, which matches the original string and confirms that the number is not prime.

4.2 Print the sum of all fields on each line

```
perl -MList::Util=sum -alne 'print sum @F'
```

This one-liner turns on field *auto-splitting* with the `-a` command-line option and imports the `sum` function from the `List::Util` module with `-Mlist::Util=sum`. (`List::Util` is part of the Perl core, so you don't need install it.) Auto-splitting happens on whitespace characters by default, and the resulting fields are put in the `@F` variable. For example, the line `1 4 8` would be split on each space so that `@F` would become `(1, 4, 8)`. The `sum @F` statement sums the elements in the `@F` array, giving you `13`.

The `-Mmodule=arg` option imports `arg` from `module`. It's the same as writing

```
use module qw(arg);
```

This one-liner is equivalent to

```
use List::Util qw(sum);
while (<>) {
    @F = split(' ');
    print sum @F, "\n";
}
```

You can change auto-splitting's default behavior by specifying an argument to the `-F` command-line switch. Say you have the following line:

```
1:2:3:4:5:6:7:8:9:10
```

And you wish to find the sum of all these numbers. You can simply specify `:` as an argument to the `-F` switch, like this:

```
perl -MList::Util=sum -F: -alne 'print sum @F'
```

This splits the line on the colon character and sums all the numbers. The output is `55` because that's the sum of the numbers 1 through 10.

4.3 Print the sum of all fields on all lines

```
perl -MList::Util=sum -alne 'push @S,@F; END { print sum @S }'
```

This one-liner keeps pushing the split fields in `@F` to the `@S` array. Once the input stops and Perl is about to quit, the `END { }` block is executed and it outputs the sum of all items in `@F`. This sums all fields over all lines.

Notice how pushing the `@F` array to the `@S` array actually appends elements to it. This differs from many other languages, where pushing array1 to array2 would put array1 into array2, rather than appending the elements of array1 onto array2. Perl performs list flattening by design.

Unfortunately, summing all fields on all lines using this solution creates a massive `@S` array. A better solution is to keep only the running sum, like this:

```
perl -MList::Util=sum -alne '$s += sum @F; END { print $s }'
```

Here, each line is split into `@F` and the values are summed and stored in the running sum variable `$s`. Once all input has been processed, the one-liner prints the value of `$s`.

4.4 Shuffle all fields on each line

```
perl -MList::Util=shuffle -alne 'print "@{[shuffle @F]}"'
```

The trickiest part of this one-liner is the `@{[shuffle @F]}` construction. This construction allows you to execute the code inside the quotation marks. Normally text and variables are placed inside quotation marks, but with the `@{[ ... ]}` construction you can run code, too.

In this one-liner, the code to execute inside of the quotation marks is `shuffle @F`, which shuffles the fields and returns the shuffled list. The `[shuffle @F]` creates an array reference containing the shuffled fields, and the `@{ ... }` dereferences it. You simply create a reference and immediately dereference it. This allows you to run the code inside the quotation marks.

Let's look at several examples to understand why I chose to run the code inside the quotation marks. If I had written `print shuffle @F`, the fields on the line would be concatenated. Compare the output of this one-liner:

```
$ echo a b c d | perl -MList::Util=shuffle -alne 'print "@{[shuffle @F]}"'
b c d a
```

to this:

```
$ echo a b c d | perl -MList::Util=shuffle -alne 'print shuffle @F'
bcda
```

In the first example, the array of shuffled fields (inside the double quotation marks) is interpolated, and the array's elements are separated by a space, so the output is `b c d a`. In the second example, interpolation doesn't happen, and Perl simply dumps out element by element without separating them, and the output is `bcda`.

You can use the `$,` special variable to change the separator between array elements when they're printed. For example, here's what happens when I change the separator to a colon:

```
$ echo a b c d | perl -MList::Util=shuffle -alne '$,=":"; print shuffle @F'
b:c:d:a
```

You can also use the `join` function to join the elements of `@F` with a space:

```
perl -MList::Util=shuffle -alne 'print join " ", shuffle @F'
```

Here, the `join` function joins the elements of an array using the given separator, but the `@{[ ... ]}` construction is the cleanest way to do it.

4.5 Find the numerically smallest element (minimum element) on each line

```
perl -MList::Util=min -alne 'print min @F'
```

This one-liner is somewhat similar to the previous ones. It uses the `min` function from `List::Util`. Once the line has been automatically split by `-a` and the elements are in the `@F` array, the `min` function finds the numerically smallest element, which it prints.

For example, if you have a file that contains these lines:

```
-8  9  10 5
7   0  9  3
5  -25 9  999
```

Running this one-liner produces the following output:

```
-8
0
-25
```

The smallest number on the first line is `-8`; on the second line, the smallest number is `0`; and on the third line, `-25`.

4.6 Find the numerically smallest element (minimum element) over all lines

```
perl -MList::Util=min -alne '@M = (@M, @F); END { print min @M }'
```

This one-liner combines one-liners 4.3 and 4.5. The `@M = (@M, @F)` construct is the same as `push @M, @F`. It appends the contents of `@F` to the `@M` array.

This one-liner stores all data in memory, and if you run it on a really huge file, Perl will run out of memory. Your best bet is to find the smallest element on every line and compare that element with the smallest element on the previous line. If the element on the current line is less than the previous one, it's the smallest element so far. Once all lines have been processed, you can just print the smallest element found through the `END` block:

```
perl -MList::Util=min -alne '
  $min = min @F;
  $rmin = $min unless defined $rmin && $min > $rmin;
  END { print $rmin }
'
```

Here, you first find the minimum element on the current line and store it in `$min`. Then you check to see if the smallest element on the current line is the smallest element so far. If so, assign it to `$rmin`. Once you've looped over the whole input, the `END` block executes and you print the `$rmin`.

Say your file contains these lines:

```
-8  9  10 5
7   0  9  3
5  -25 9  999
```

Running this one-liner outputs `-25` because that's the smallest number in the file.

If you're using Perl 5.10 or later, you can do the same thing with this one-liner:

```
perl -MList::Util=min -alne '$min = min($min // (), @F); END { print $min }'
```

This one-liner uses the `//` operator, which is new to Perl 5.10. This operator is similar to the logical OR operator (`||`), except that it tests the left side's definedness rather than the truth. What that means is it tests whether the left side is defined rather than whether it is true or false. In this one-liner, the expression `$min // ()` returns `$min` if `$min` has been defined, or else it returns an empty list `()`. The `//` operator saves you from having to use `defined` to test definedness.

Consider what happens when this one-liner is run on the previous file. First, Perl reads the line `-8  9  10  5` , splits it, and puts the numbers in the `@F` array. The `@F` array is now `(-8, 9, 10, 5)`. Next, it executes `$min = min ($min // (), @F)`. Because `$min` hasn't been defined, `$min // ()` evaluates to `()`, so the whole expression becomes `$min = min ((), (-8, 9, 10, 5))`.

Perl does list flattening by design, so after flattening the arguments to the `min` function, the expression becomes `$min = min(-8, 9, 10, 5)`. This defines `$min`, setting it to `-8`. Perl proceeds to the next line, where it sets `@F` to `(7, 0, 9, 3)` and again evaluates `$min = min($min // (), @F)`. Because `$min` has now been defined, `$min // ()` evaluates to `$min` and the expression becomes `$min = min(-8, 7, 0, 9, 3)`. At this point, `-8` is still the smallest element, so `$min` remains `-8`. Finally, Perl reads in the last line, and after evaluating `$min = min(-8, 5, -25, 9, 999)`, it finds that `-25` is the smallest element in the file.

4.7 Find the numerically largest element (maximum element) on each line

```
perl -MList::Util=max -alne 'print max @F'
```

This works the same as one-liner 4.5, except that you replace `min` with `max`.

4.8 Find the numerically largest element (maximum element) over all lines

```
perl -MList::Util=max -alne '@M = (@M, @F); END { print max @M }'
```

This one-liner is similar to one-liners 4.6 and 4.7. In this one-liner, each line is auto-split and put in the `@F` array, and then this array is merged with the `@M` array. When the input has been processed, the `END` block executes and the maximum element is printed.

Here's another way to find the maximum element, keeping just the running maximum element instead of all elements in memory:

```
perl -MList::Util=max -alne '
  $max = max @F;
  $rmax = $max unless defined $rmax && $max < $rmax;
  END { print $rmax }
'
```

If you're using Perl 5.10 or later, you can use the `//` operator to shorten this one-liner:

```
perl -MList::Util=max -alne '$max = max($max // (), @F); END { print $max }'
```

This is the same as one-liner 4.6, except you replace `min` with `max`.

4.9 Replace each field with its absolute value

```
perl -alne 'print "@{[map { abs } @F]}"'
```

This one-liner first auto-splits the line using the `-a` option. The split fields end up in the `@F` variable. Next, it calls the absolute value function `abs` on each field with the help of the `map` function. Essentially, the `map` function applies a given function to each element of the list and returns a new list that contains the results of applying the function. For example, if the list `@F` is `(-4, 2, 0)`, mapping `abs` over it produces the list `(4, 2, 0)`. Finally, this one-liner prints the new list of positive values.

The `@{[ ... ]}` construct, introduced in one-liner 4.4, allows you to execute the code inside the quotation marks.

4.10 Print the total number of fields on each line

```
perl -alne 'print scalar @F'
```

This one-liner forces the evaluation of `@F` in the scalar context, which in Perl means "the number of elements in `@F`." As a result, it prints the number of elements on each line.

For example, if your file contains the following lines:

```
foo bar baz
foo bar
baz
```

Running this one-liner produces the following output:

```
3
2
1
```

The first line has three fields, the second line has two fields, and the last line has one field.

4.11 Print the total number of fields on each line, followed by the line

```
perl -alne 'print scalar @F, " $_"'
```

This one-liner is the same as one-liner 4.10, with the addition of `$_` at the end, which prints the whole line. (Remember that `-n` puts each line in the `$_` variable.)

Let's run this one-liner on the same example file that I used in one-liner 4.10:

```
foo bar baz
foo bar
baz
```

Running the one-liner produces the following output:

```
3 foo bar baz
2 foo bar
1 baz
```

4.12 Print the total number of fields on all lines

```
perl -alne '$t += @F; END { print $t }'
```

Here, the one-liner keeps adding the number of fields on each line to variable `$t` until all lines have been processed. Next, it prints the result, which contains the number of words on all lines. Notice that you add the `@F` array to the scalar variable `$t`. Because `$t` is scalar, the `@F` array is evaluated in the scalar context and returns the number of elements it contains.

Running this one-liner on the following file:

```
foo bar baz
foo bar
baz
```

produces the number `6` as output because the file contains a total of six words.

4.13 Print the total number of fields that match a pattern

```
perl -alne 'map { /regex/ && $t++ } @F; END { print $t || 0 }'
```

This one-liner uses `map` to apply an operation to each element in the `@F` array. In this example, the operation checks to see if each element matches */regex/*, and if it does, it increments the `$t` variable. It then prints the `$t` variable, which contains the number of fields that match the */regex/* pattern. The `$t || 0` construct is necessary because if no fields match, `$t` wouldn't exist, so you must provide a default value. Instead of `0`, you can provide any other default value, even a string!

Looping would be a better approach:

```
perl -alne '$t += /regex/ for @F; END { print $t }'
```

Here, each element in `@F` is tested against */regex/*. If it matches, */regex/* returns true; otherwise it returns false. When used numerically, true converts to `1` and false converts to `0`, so `$t += /regex/` adds either `1` or `0` to the `$t` variable. As a result, the number of matches is counted in `$t`. You do not need a default value when printing the result in the `END` block because the `+=` operator is run regardless of whether the field matches. You will always get a value, and sometimes that value will be `0`.

Another way to do this is to use `grep` in the scalar context:

```
perl -alne '$t += grep /regex/, @F; END { print $t }'
```

Here, `grep` returns the number of matches because it's evaluated in the scalar context. In the list context, `grep` returns all matching elements, but in the scalar context, it returns the number of matching elements. This number is accumulated in `$t` and printed in the `END` block. In this case, you don't need to provide a default value for `$t` because `grep` returns `0` in those situations.

4.14 Print the total number of lines that match a pattern

```
perl -lne '/regex/ && $t++; END { print $t || 0 }'
```

Here, */regex/* evaluates to true if the current line of input matches this regular expression. Writing */regex/ && $t++* is the same as writing `if ($_ =~ /regex/) { $t++ }`, which increments the `$t` variable if the line matches the specified pattern. In the `END` block, the `$t` variable contains the total number of pattern matches and is printed; but if no lines match, `$t` is once again undefined, so you must print a default value.

4.15 Print the number π

```
perl -Mbignum=bpi -le 'print bpi(21)'
```

The bignum package exports the bpi function that calculates the π constant to the desired accuracy. This one-liner prints π to 20 decimal places. (Notice that you need to specify n+1 to print it to an accuracy of n.)

The bignum library also exports the constant π, precomputed to 39 decimal places:

```
perl -Mbignum=PI -le 'print PI'
```

4.16 Print the number *e*

```
perl -Mbignum=bexp -le 'print bexp(1,21)'
```

The bignum library exports the bexp function, which takes two arguments: the power to raise *e* to, and the desired accuracy. This one-liner prints the constant *e* to 20 decimal places.

For example, you could print the value of e^2 to 30 decimal places:

```
perl -Mbignum=bexp -le 'print bexp(2,31)'
```

As with π, bignum also exports the constant *e* precomputed to 39 decimal places:

```
perl -Mbignum=e -le 'print e'
```

4.17 Print UNIX time (seconds since January 1, 1970, 00:00:00 UTC)

```
perl -le 'print time'
```

The built-in time function returns seconds since the epoch. This one-liner simply prints the time.

4.18 Print Greenwich Mean Time and local computer time

```
perl -le 'print scalar gmtime'
```

The `gmtime` function is a built-in Perl function. When used in the scalar context, it returns the time localized to Greenwich Mean Time (GMT).

The built-in `localtime` function acts like `gmtime`, except it returns the computer's local time when it's used in the scalar context:

```
perl -le 'print scalar localtime'
```

In the list context, both `gmtime` and `localtime` return a nine-element list (known as `struct tm` to UNIX programmers) with the following elements:

```
($second,             [0]
$minute,              [1]
$hour,                [2]
$month_day,           [3]
$month,               [4]
$year,                [5]
$week_day,            [6]
$year_day,            [7]
$is_daylight_saving   [8]
)
```

You can *slice* this list (that is, extract elements from it) or print individual elements if you need just some part of the information it contains. For example, to print `H:M:S`, slice the elements `2`, `1`, and `0` from `localtime`, like this:

```
perl -le 'print join ":", (localtime)[2,1,0]'
```

To slice elements individually, specify a list of elements to extract, for instance `[2,1,0]`. Or slice them as a range:

```
perl -le 'print join ":", (localtime)[2..6]'
```

This one-liner prints the hour, date, month, year, and day of the week.

You can also use negative indexes to select elements from the opposite end of a list:

```
perl -le 'print join ":", (localtime)[-2, -3]'
```

This one-liner prints elements `7` and `6`, which are the day of the year (for example, the 200th day) and of the week (for example, the 4th day), respectively.

4.19 Print yesterday's date

```
perl -MPOSIX -le '
  @now = localtime;
  $now[3] -= 1;
  print scalar localtime mktime @now
'
```

Remember that `localtime` returns a nine-item list (see one-liner 4.18) of various date elements. The fourth element in the list is the current month's day. If you subtract 1 from this element, you get yesterday.

The `mktime` function constructs the UNIX epoch time from this modified nine-element list, and the `scalar localtime` construct prints the new date, which is yesterday. This one-liner also works in edge cases, such as when the current day is the first day of the month. You need the `POSIX` package because it exports the `mktime` function.

For example, if it's *Mon May 20 05:49:55* right now, running this one-liner prints *Sun May 19 05:49:55*.

4.20 Print the date 14 months, 9 days, and 7 seconds ago

```
perl -MPOSIX -le '
  @now = localtime;
  $now[0] -= 7;
  $now[3] -= 9;
  $now[4] -= 14;
  print scalar localtime mktime @now
'
```

This one-liner modifies the first, fourth, and fifth elements of the `@now` list. The first element is seconds, the fourth is days, and the fifth is months. The `mktime` command generates the UNIX time from this new structure, and `localtime`, which is evaluated in the scalar context, prints the date 14 months, 9 days, and 7 seconds ago.

4.21 Calculate the factorial

```
perl -MMath::BigInt -le 'print Math::BigInt->new(5)->bfac()'
```

This one-liner uses the `bfac()` function from the `Math::BigInt` module in the Perl core. (In other words, you don't need to install it.) The `Math::BigInt->new(5)` construction creates a new `Math::BigInt` object with

a value of 5, after which the bfac() method is called on the newly created object to calculate the factorial of 5. Change 5 to any number you wish to find its factorial.

Another way to calculate a factorial is to multiply the numbers from 1 to *n* together:

```
perl -le '$f = 1; $f *= $_ for 1..5; print $f'
```

Here, I set $f to 1 and then loop from 1 to 5 and multiply $f by each value. The result is 120 (1*2*3*4*5), the factorial of 5.

4.22 Calculate the greatest common divisor

```
perl -MMath::BigInt=bgcd -le 'print bgcd(@list_of_numbers)'
```

Math::BigInt has several other useful math functions including bgcd, which calculates the *greatest common divisor (gcd)* of a list of numbers. For example, to find the greatest common divisor of (20, 60, 30), execute the one-liner like this:

```
perl -MMath::BigInt=bgcd -le 'print bgcd(20,60,30)'
```

To calculate the gcd from a file or user's input, use the -a command-line argument and pass the @F array to the bgcd function:

```
perl -MMath::BigInt=bgcd -anle 'print bgcd(@F)'
```

(I explained the -a argument and the @F array in one-liner 4.2 on page 30.)

You could also use Euclid's algorithm to find the gcd of $n and $m. This one-liner does just that and stores the result in $m:

```
perl -le '
  $n = 20; $m = 35;
  ($m,$n) = ($n,$m%$n) while $n;
  print $m
'
```

Euclid's algorithm is one of the oldest algorithms for finding the gcd.

4.23 Calculate the least common multiple

The *least common multiple (lcm)* function, `blcm`, is included in `Math::BigInt`. Use this one-liner to find the least common multiple of `(35, 20, 8)`:

```
perl -MMath::BigInt=blcm -le 'print blcm(35,20,8)'
```

To find the lcm from a file with numbers, use the `-a` command-line switch and the `@F` array:

```
perl -MMath::BigInt=blcm -anle 'print blcm(@F)'
```

If you know a bit of number theory, you may recall that there is a connection between the gcd and the lcm. Given two numbers `$n` and `$m`, you know that their lcm is `$n*$m/gcd($n,$m)`. This one-liner, therefore, follows:

```
perl -le '
  $a = $n = 20;
  $b = $m = 35;
  ($m,$n) = ($n,$m%$n) while $n;
  print $a*$b/$m
'
```

4.24 Generate 10 random numbers between 5 and 15 (excluding 15)

```
perl -le 'print join ",", map { int(rand(15-5))+5 } 1..10'
```

This one-liner prints 10 random numbers between 5 and 15. It may look complicated, but it's actually simple. `int(rand(15-5))` is just `int(rand(10))`, which returns a random integer from 0 to 9. Adding `5` to it makes it return a random integer from 5 to 14. The range `1..10` makes it draw 10 random integers.

You can also write this one-liner more verbosely:

```
perl -le '
  $n=10;
  $min=5;
  $max=15;
  $, = " ";
  print map { int(rand($max-$min))+$min } 1..$n;
'
```

Here, all variables are more explicit. To modify this one-liner, change the variables $n, $min, and $max. The $n variable represents how many random numbers to generate, and $min-$max is the range of numbers for use in that generation.

The $, variable is set to a space because it's the output field separator for print and it's undef by default. If you didn't set $, to a space, the numbers would be printed concatenated. (See one-liner 4.4 on page 32 for a discussion of $,.)

4.25 Generate all permutations of a list

```
perl -MAlgorithm::Permute -le '
  $l = [1,2,3,4,5];
  $p = Algorithm::Permute->new($l);
  print "@r" while @r = $p->next
'
```

This one-liner uses the object-oriented interface of the module Algorithm::Permute to find all permutations of a list, that is, all ways to rearrange items. The constructor of Algorithm::Permute takes an array reference of elements to permute. In this particular one-liner, the elements are the numbers 1, 2, 3, 4, 5.

The next method returns the next permutation. Calling it repeatedly iterates over all permutations, and each permutation is put in the @r array and then printed. (Beware: The output list gets large really quickly. There are n! (*n* factorial) permutations for a list of n elements.)

Another way to print all permutations is with the permute subroutine:

```
perl -MAlgorithm::Permute -le '
  @l = (1,2,3,4,5);
  Algorithm::Permute::permute { print "@l" } @l
'
```

Here's what you get if you change @l to just three elements (1, 2, 3) and run it:

```
1 2 3
1 3 2
3 1 2
2 1 3
2 3 1
3 2 1
```

4.26 Generate the powerset

```
perl -MList::PowerSet=powerset -le '
  @l = (1,2,3,4,5);
  print "@$_" for @{powerset(@l)}
'
```

This one-liner uses the `List::PowerSet` module from CPAN. The module exports the `powerset` function, which takes a list of elements and returns a reference to an array containing references to subset arrays. You can install this module by running `cpan List::PowerSet` at the command line.

In the `for` loop, you call the `powerset` function and pass it the list of elements of `@l`. Next, you dereference the return value of `powerset`, which is a reference to an array of subsets, and then dereference each individual subset `@$_` and print it.

The *powerset* is the set of all subsets. For a set of n elements, there are exactly 2^n subsets in the powerset. Here's an example of the powerset of `(1, 2, 3)`:

```
1 2 3
2 3
1 3
3
1 2
2
1
```

4.27 Convert an IP address to an unsigned integer

```
perl -le '
  $i=3;
  $u += ($_<<8*$i--) for "127.0.0.1" =~ /(\d+)/g;
  print $u
'
```

This one-liner converts the IP address `127.0.0.1` into an unsigned integer by first doing a global match of `(\d+)` on the IP address. Performing a `for` loop over a global match iterates over all the matches, which are the four parts of the IP address: `127`, `0`, `0`, `1`.

Next, the matches are summed in the `$u` variable. The first bit is shifted 8 × 3 = 24 places, the second is shifted 8 × 2 = 16 places, and the third is shifted 8 places. The last is simply added to `$u`. The resulting integer happens to be `2130706433` (a very geeky number).

Here are some more one-liners:

```
perl -le '
  $ip="127.0.0.1";
  $ip =~ s/(\d+)\.?/sprintf("%02x", $1)/ge;
  print hex($ip)
'
```

This one-liner utilizes the fact that `127.0.0.1` can be easily converted to hex. Here, the `$ip` is matched against `(\d+)`, and each IP part is transformed into a hex number with `sprintf("%02x", $1)` inside the `s` operator. The `/e` flag of the `s` operator makes it evaluate the substitution part as a Perl expression. As a result, `127.0.0.1` is transformed into `7f000001` and then interpreted as a hexadecimal number by Perl's `hex` operator, which converts it to a decimal number.

You can also use `unpack`:

```
perl -le 'print unpack("N", 127.0.0.1)'
```

This one-liner is probably as short as possible. It uses *vstring literals* (version strings) to express the IP address. A vstring forms a string literal composed of characters with the specified ordinal values. The newly formed string literal is unpacked into a number from a string in network byte order (big-endian order) and then printed.

If you have a string with an IP (rather than a vstring), you first have to convert it to byte form with the function `inet_aton`:

```
perl -MSocket -le 'print unpack("N", inet_aton("127.0.0.1"))'
```

Here, `inet_aton` converts the string `127.0.0.1` to the byte form (equivalent to the pure vstring `127.0.0.1`) and then `unpack` unpacks it, as in the previous one-liner.

4.28 Convert an unsigned integer to an IP address

```
perl -MSocket -le 'print inet_ntoa(pack("N", 2130706433))'
```

Here, the integer `2130706433` is packed into a number in big-endian byte order and then passed to the `inet_ntoa` function that converts a number back to an IP address. (Notice that `inet_ntoa` is the opposite of `inet_aton`.)

You can do the same thing like this:

```
perl -le '
  $ip = 2130706433;
  print join ".", map { (($ip>>8*($_))&0xFF) } reverse 0..3
'
```

Here, the `$ip` is shifted 24 bits to the right and then bitwise ANDed with `0xFF` to produce the first part of the IP, which is `127`. Next, it's shifted 16 bits and bitwise ANDed with `0xFF`, producing `0`, and then shifted 8 bits and bitwise ANDed with `0xFF`, producing another `0`. Finally, the whole number is bitwise ANDed with `0xFF`, producing `1`.

The result from `map { ... }` is a list `(127, 0, 0, 1)`. That list is now `join`ed by a dot `"."` to produce the IP address `127.0.0.1`.

You can replace `join` with the special variable `$,`, which acts as a value separator for the `print` statement:

```
perl -le '
  $ip = 2130706433;
  $, = ".";
  print map { (($ip>>8*($_))&0xFF) } reverse 0..3
'
```

Because `reverse 0..3` is the same as `3,2,1,0`, you could also write:

```
perl -le '
  $ip = 2130706433;
  $, = ".";
  print map { (($ip>>8*($_))&0xFF) } 3,2,1,0
'
```