Introduction to Perl

Credits

This note is based on the following materials:
1. Introduction to Perl by Greg Johnson
2. Perl Lessons by Jukka Korpela which is largely based on 1
3. Beginners Introduction to Perl by Doug Sheppard

1. What Is Perl?

Perl is a programming language which can be used for a large variety of tasks. Both Perl interpreters and Perl documentation are freely available for Unix, MVS, VMS, MS/DOS, Macintosh, OS/2, Amiga, and other operating systems. (On Windows systems, Perl can be used in DOS command line mode.)

A typical simple use of Perl would be for extracting information from a text file and printing out a report or for converting a text file into another form. In fact, the name "Perl" was formed from the expression *"**P**ractical **E**xtraction and **R**eport **L**anguage"*. But Perl provides a large number of tools for quite complicated problems, including system administration tasks, which can often be programmed rather portably in Perl.

Perl has powerful string-manipulation functions. On the other hand, it eclectically combines features and purposes of the C language and many command or scripting languages. For such reasons, Perl looks rather odd on first sight. But once you have learned Perl, you will be able to write programs in Perl much faster than in most other languages. This makes Perl especially suitable for writing programs which are used once only.

The following simple Perl program reads a text file consisting people's names like "John Smith", each on a line of its own, and it prints them out so that the first and second name have been swapped and separated with a comma (e.g. "Smith, John").

```
while(<>) {
  split;
  print "$_[1], $_[0]\n"; }
```

As you can see, Perl is compact - and somewhat cryptic, until you learn some basic notations. (There will be a rather similar example, which is annotated in some detail.)

Perl has become popular for programming handlers for World Wide Web forms and generally as glue and gateway between systems, databases, and users.

Perl is typically implemented as an interpreted (not compiled) language. Thus, the execution of a Perl program tends to use more CPU time than a corresponding C program, for instance. On the other hand, computers tend to get faster and faster, and writing something in Perl instead of C tends to save *your* time.

Programs written in Perl are often called **Perl scripts**, especially in the context of CGI programming, whereas the term **the perl program** refers to the system program named `perl` for executing Perl scripts. (What, confused already?)

If you have used Unix shell scripts or awk or sed or similar utilities for various purposes, you will find that you can normally use Perl for those and many other purposes, and the code tends to be more compact. And if you haven't used such utilities but have started thinking you might have need for them, then perlhaps what you really need to learn is Perl instead of all kinds of futilities.

Perl is relatively stable; the current major version, Perl 5, was released in 1994. Extensions to the language capabilities are mainly added though the construction of *modules*, written by different people and made available to other Perl programmers. For major general-purpose Perl applications, particularly CGI scripts and client or server applications, see *CPAN Module documentation*.

The following is the "Sales Pitch" by Doug Sheppard.

Perl is the Swiss Army chainsaw of scripting languages: powerful and adaptable. It was first developed by Larry Wall, a linguist working as a systems administrator for NASA in the late 1980s, as a way to make report processing easier. Since then, it has moved into a large number of roles: automating system administration, acting as glue between different computer systems; and, of course, being one of the most popular languages for CGI programming on the Web.

Why did Perl become so popular when the Web came along? Two reasons: First, most of what is being done on the Web happens with text, and is best done with a language that's designed for text processing. More importantly, Perl was appreciably better than the alternatives at the time when people needed something to use. C is complex and can produce security problems (especially with untrusted data), Tcl can be awkward and Python didn't really have a foothold.

It also didn't hurt that Perl is a friendly language. It plays well with your personal programming style. The Perl slogan is ``There's more than one way to do it,'' and that lends itself well to large and small problems alike.

2. Perl References and Resources

Programming Perl, 2nd Edition, by Larry Wall, Tom Christiansen & Randal L. Schwartz.
The definitive reference. Published by O'Reilly.

The Perl Language Home Page
: Classified information about all aspects of Perl.

The Perl FAQ.
: Answers a large number of Frequently Asked Question. Check this first whenever you have a problem with Perl! Divided into sections:

- The list of questions
- 1 General Questions About Perl
- 2 Obtaining and Learning about Perl
- 3 Programming Tools
- 4 Data Manipulation
- 5 Files and Formats
- 6 Regexps
- 7 General Perl Language Issues
- 8 System Interaction
- 9 Networking

CPAN, the Comprehensive Perl Archive Network

Aims at being *the* Perl archive. The master copy of CPAN is at ftp://ftp.funet.fi/pub/languages/perl/CPAN/ but you can use the address http://www.perl.com/CPAN/CPAN.html to fetch a copy from a "site near you".

Johan Vromans' *Perl 5 Desktop Reference*

An **excellent quick reference** to an experienced Perl programmer who occasionally cannot remember the name of a function or its exact syntax.

Perl pages at Galaxy

A lot of links to information about Perl.

Middle of Nowhere Perl Pages

About CGI programming and object-oriented programming in Perl, and many other things.

"The" Perl manual.

An extensive, but partly rather technical, manual of the language. Available from the CPAN archive and included into the normal Perl distribution.

MU On-line Perl Manual

A version of the above-mentioned Perl manual, with a search function.

3. Creating, Storing and Running Perl Programs

The machines in the laboratories of the School of Computer Science have Perl installed under both UNIX and Windows-NT. For details, please consult the Technical Services Group's Website. To create a Perl program, you will need a text editor. In UNIX, you can use emacs, and in Windows-NT, you can use WordPad (you have to save the document in plain text format). In Windows-NT, you have to run your program in DOS-Shell.

Exercise

Take the following text (using either emacs or WordPad) and put it into a file called first.pl:

```
#!/usr/local/bin/perl

print "Hi there!\n";
```

(Traditionally, first programs are supposed to say Hello world!, but I'm an iconoclast.)

Now, run it with your Perl interpreter. From a command line, go to the directory with this file and type perl first.pl. You should see:

```
Hi there!
```

The \n indicates the ``newline'' character; without it, Perl doesn't skip to a new line of text on its own.

Try this out:

Take the following text and put it into a file.

```
if ($#ARGV >= 0) { $who = join(' ', @ARGV); }
else { $who = 'World'; }
print "Hello, $who!\n";
```

(@ ARGV is a special array which holds the command line parameters. A program is executed as a result of a system command, which consists of the executable program file, followed by a command tail, e.g. :

```
C:> program param1 param2 ... paramn
```

Then $ARGV[0] = "program", $ARGV[1] = "param1", $ARGV[2] = "param2" ... $ARGV[n] = "paramn".

$# ARGV is used to calculate the length of the array, which is 1 less than the number of elements in the array)

Let us assume that the above lines are stored in a Unix file ~/bin/hello. (That's in your home directory, subdirectory bin, file hello.) You can then run the program by entering a command like one of the following:

```
perl ~/bin/hello
perl ~/bin/hello Citizens of Earth
perl hello
```

(The last one works only you're in the ~/bin directory.)

Making it command-like on Unix

If you expect to use this program a lot and want to execute it as a command on Unix, then you need to do five things.

1. Insert a line at the very beginning of the program file, beginning with #! followed by the full pathname of the Perl interpreter (the perl command), #!/usr/local/bin/perl. You may append command-line options like -w (warn about possible inconsistencies), to that line.
2. Set the execute permissions of the program file. To make the file executable (and readable and writable) by only yourself, use a Unix command like:

   ```
   chmod 700 ~/bin/hello
   ```

   To make it executable and readable by all enter a Unix command like the following:

   ```
   chmod a+rx ~/bin/hello
   ```

   You may also need to use chmod a+x on the directories ~ and ~/bin. See man chmod for details and the security implications.
3. Edit your file ~/.cshrc or ~/.login to make directory ~/bin part of the path Unix searches for executables, with a line like this:

   ```
   set path = ($path ~/bin)
   ```

4. This takes effect the next time you start a default tcsh or csh shell (.cshrc file) or login (.login file). If you want it to take effect immediately, enter the above set path command at the Unix prompt or execute your .cshrc or .login file with the source command. If you are using sh, bash, ksh or some other shell, alter ~/.profile or some other file to set the path at login.
5. If a program you want to execute has just been newly created, then issue the csh/tcsh command rehash to rescan the path.

If you perform (1)-(5), then you can execute your program via a command like this:

```
hello
```

4. Functions, Variables and Data Structures

Functions and Statements

Perl has a rich library of *functions*. They're the verbs of Perl, the commands that the interpreter runs. You can see a list of all the built-in functions on the perlfunc main page. Almost all functions can be given a list of *parameters*, which are separated by commas.

The print function is one of the most frequently used parts of Perl. You use it to display things on the screen or to send information to a file (which we'll discuss later). It takes a list of things to output as its parameters.

```
print "This is a single statement.";
print "Look, ", "a ", "list!";
```

A Perl program consists of *statements*, each of which ends with a semicolon. Statements don't need to be on separate lines; there may be multiple statements on one line or a single statement can be split across multiple lines.

```
print "This is "; print "two statements.\n"; print "But this ",
    "is only one statement.\n";
```

Variables in Perl

If functions are Perl's verbs, then variables are its nouns. Perl has three types of variables: *scalars*, *arrays* and *hashes*. Think of them as ``things,'' ``lists,'' and ``dictionaries.'' In Perl, all variable names are a punctuation character, a letter or underscore, and one or more alphanumeric characters or underscores.

The value of a variable in Perl can be a number or a string, among other things. Variables are not typed. You can, for example, assign a string to a variable and later a number to the same variable. Variables are not declared at all. You can simply start using a variable.

An attempt to use an *uninitialized* variable causes a zero or an empty string or the truth value false (depending on the context) to be used. However, using the command-line `switch -w` you can ask the Perl interpreter to issue warnings, such as reporting uses of undefined values.

Perl's Data Structures

In addition to simple (scalar) variables and constants Perl has two kinds of data structures: arrays (lists), and associative arrays ("hashes"). Scalar variable names always begin with the dollar sign, e.g. $myvar. Names for arrays (and array slices) always begin with the commercial at sign, e.g. @myarray. Names for hashes always begin with the percent sign, e.g. %myhash.

Perl allows combinations of these, such as lists of lists and associative arrays of lists. (See *The Perl Data Structures Cookbook* for illustrations of such advanced topics.)

Numbers, Strings and Quotes

There are two basic data types in Perl: numbers and strings.

Numbers are easy; we've all dealt with them. The only thing you need to know is that you never insert commas or spaces into numbers in Perl. always write 10000, not 10,000 or 10 000.

Strings are a bit more complex. A string is a collection of characters in either single or double quotes:

```
'This is a test.'
"Hi there!\n"
```

The difference between single quotes and double quotes is that single quotes mean that their contents should be taken *literally*, while double quotes mean that their contents should be *interpreted*. For example, the character sequence `\n` is a newline character when it appears in a string with double quotes, but is literally the two characters, backslash and `n`, when it appears in single quotes.

```
print "This string\nshows up on two lines.";
print 'This string \n shows up on only one.';
```

(Two other useful backslash sequences are `\t` to insert a tab character, and `\\` to insert a backslash into a double-quoted string.)

Scalars

Scalars can be numeric or character (string) as determined by context:

```
123  12.4  5E-10  0xff (hex)  0377 (octal)

'What you $see is (almost) what \n you get'   'Don\'t Walk'

"How are you?"  "Substitute values of $x and \n in \" quotes."

`date`  `uptime -u`  `du -sk $filespec | sort -n`

$x    $list_of_things[5]    $lookup{'key'}
```

Scalars are single things. This might be a number or a string. The name of a scalar begins with a dollar sign, such as $i or $abacus. You assign a value to a scalar by telling Perl what it equals, like so:

```
$i = 5;

$pie_flavor = 'apple';

$constitution1776 = "We the People, etc.";
```

You don't need to specify whether a scalar is a number or a string. It doesn't matter, because when Perl needs to treat a scalar as a string, it does; when it needs to treat it as a number, it does. The conversion happens automatically. (This is different from many other languages, where strings and numbers are two separate data types.)

If you use a double-quoted string, Perl will insert the value of any scalar variables you name in the string. This is often used to fill in strings on the fly:

```
$apple_count = 5;

$count_report = "There are $apple_count apples.";

print "The report is: $count_report\n";
```

The final output from this code is The report is: There are 5 apples..

Numbers in Perl can be manipulated with the usual mathematical operations: addition, multiplication, division and subtraction. (Multiplication and division are indicated in Perl with the * and / symbols, by the way.)

```
$a = 5;

$b = $a + 10;        # $b is now equal to 15.

$c = $b * 10;        # $c is now equal to 150.

$a = $a - 1;         # $a is now 4, and algebra teachers are cringing.
```

You can also use special operators like ++, --, +=, -=, /= and *=. These manipulate a scalar's value without needing two elements in an equation. Some people like them, some don't. I like the fact that they can make code clearer.

```
$a = 5;

$a++;         # $a is now 6; we added 1 to it.

$a += 10;     # Now it's 16; we added 10.

$a /= 2;      # And divided it by 2, so it's 8.
```

Strings in Perl don't have quite as much flexibility. About the only basic operator that you can use on strings is concatenation, which is a $10 way of saying ``put together.'' The concatenation operator is the period. Concatenation and addition are two different things:

```
$a = "8";     # Note the quotes.  $a is a string.

$b = $a + "1";    # "1" is a string too.

$c = $a . "1";    # But $b and $c have different values!
```

Remember that Perl converts strings to numbers transparently whenever it's needed, so to get the value of $b, the Perl interpreter converted the two strings "8" and "1" to numbers, then added them. The value of $b is the number 9. However, $c used concatenation, so its value is the string "81".

Just remember, the plus sign adds numbers and the period puts strings together.

Arrays

Arrays (also called lists) consist of sequentially-arranged scalars:

```
('Sunday', 'Monday', 'Tuesday', 'Wednesday',
   'Thursday', 'Friday', 'Saturday')

(13,14,15,16,17,18,19)   equivalent to  (13..19)

(13,14,15,16,17,18,19)[2..4]  equivalent to  (15,16,17)

@whole_list
```

A notation like (2, 3, 5, 7) can be called an array constructor. It can be assigned to an array variable in order to initialize it:

```
@primes = (2, 3, 5, 7);
```

Arrays are lists of scalars. Array names begin with @. You define arrays by listing their contents in parentheses, separated by commas:

```
@lotto_numbers = (1, 2, 3, 4, 5, 6);  # Hey, it could happen.
@months = ("July", "August", "September");
```

The contents of an array are *indexed* beginning with 0. (Why not 1? Because. It's a computer thing.) To retrieve the elements of an array, you replace the @ sign with a $ sign, and follow that with the index position of the element you want. (It begins with a dollar sign because you're getting a scalar value.) You can also modify it in place, just like any other scalar.

```
@months = ("July", "August", "September");
print $months[0];   # This prints "July".
$months[2] = "Smarch";  # We just renamed September!
```

If an array doesn't exist, by the way, you'll create it when you try to assign a value to one of its elements.

```
$winter_months[0] = "December";  # This implicitly creates @winter_months.
```

Arrays always return their contents in the same order; if you go through `@months` from beginning to end, no matter how many times you do it, you'll get back `July`, `August` and `September` in that order. If you want to find the length of an array, use the value `$#array_name`. This is one less than the number of elements in the array. If the array just doesn't exist or is empty, `$#array_name` is -1. If you want to resize an array, just change the value of `$#array_name`.

```
@months = ("July", "August", "September");
print $#months;         # This prints 2.
$a1 = $#autumn_months;  # We don't have an @autumn_months, so this is -1.
$#months = 0;           # Now @months only contains "July".
```

Associative arrays

Associative arrays (also called hashes) resemble arrays but can be indexed with strings:

```
$DaysInMonth{'January'} = 31;   $enrolled{'Joe College'} = 1;

$StudentName{654321} = 'Joe College';

$score{$studentno,$examno} = 89;

%whole_hash
```

Hashes are called ``dictionaries'' in some programming languages, and that's what they are: a term and a definition, or in more correct language a *key* and a *value*. Each key in a hash has `one and only one` corresponding value. The name of a hash begins with a percentage sign, like `%parents`. You define hashes by comma-separated pairs of key and value, like so:

```
%days_in_month = ( "July" => 31, "August" => 31, "September" => 30 );
```

You can fetch any value from a hash by referring to `$hashname{key}`, or modify it in place just like any other scalar.

```
print $days_in_month{"September"}; # 30, of course.
$days_in_month{"February"} = 29;   # It's a leap year.
```

If you want to see what keys are in a hash, you can use the `keys` function with the name of the hash. This returns a list

containing all of the keys in the hash. The list isn't always in the same order, though; while we could count on @months to always return July, August, September in that order, keys %days_in_summer might return them in any order whatsoever.

```
@month_list = keys %days_in_summer;
# @month_list is now ('July', 'September', 'August') !
```

The three types of variables have three separate *namespaces*. That means that $abacus and @abacus are two different variables, and $abacus[0] (the first element of @abacus) is not the same as $abacus{0} (the value in %abacus that has the key 0).

Indexing

If you have an **array**, e.g. @myarr, you can form indexed variables by appending an index in brackets (as in the C language) by changing @ into $. The reason for the change is that the indexed variable is a scalar. Indexes are integer numbers, starting with 0 (as in C, but unlike in many other languages). For example, $myarr[5] denotes the 6th element of the array @myarr. And if you have assigned

```
@wday = ('Sun','Mon','Tue','Wed','Thu','Fri','Sat');
```

then $wday[1] equals 'Mon'. (Negative subscripts count from the end so that e.g. $wday[-1] would be 'Sat'.)

Associative arrays are indexed with curly braces enclosing a string. $whatever, @whatever, and % whatever are three different variables.

You can also form array slices, for example @myvar[5..10], which is an array (therefore, denoted using @) consisting of those components of @myvar which have an index between 5 and 10, inclusively.

Hashes, on the other hand, can be indexed e.g. with strings, since indexing method is different. Conceptually, a Perl interpreter performs a *search* from a hash, using the index as the search key. For hashes, the index is in **braces**, e.g. $myhash{'foobar'}. Notice that in this case, too, the indexed variable is a scalar and therefore begins with $.

For example, the predefined hash name %ENV denotes the collection of so-called environment variables. Thus, you could refer e.g. to the value of the environment variable HOST with the expression $ENV{"HOST"}.

Example: using an array for splitting lines into words

In Perl, you can easily split data into fields without coding the details. You simply specify what you want, using the built-in split function, optionally with some arguments.

For instance, the statement

```
split;
```

first splits the current input line into blank-separated fields and then assigns the fields to components of the predefined array variable @_. You can then access the fields using indexed variables. The special variable $#_ contains information about the number of fields: the value of that variable is the number of fields minus one. (More generally, for any array variable @*a*, the variable $#*a* contains the last index of the array.)

Assume, for example, that you have some data where each line consists of blank-separated items (which might be

strings or numbers) and you want to write a Perl script which picks up the second item from each line. (Such filtering is often needed to extract useful information from a large data file.) This is simple:

```
while (<>) {
  split;
  print $_[1], "\n"; }
```

Notice that you must use an index value of **1** to get the **2nd** field, since array indexing begins at 0 in Perl.

Name Conventions

$identifier	simple (scalar) variable
@identifier	list (normal array)
%identifier	associative array
&identifier	subroutine or function
IDENTIFIER	filehandle

`Note: From version 5, it is nolonger necessary to use & in subroutines.`

Every variable kind (scalar, array, hash) has its own *namespace*. This means that `$foo` and `@foo` are two different variables. It also means that `$foo[1]` is a part of `@foo`, not a part of `$foo`. This may seem a bit weird, but that's okay, because it **is** weird. Notice, in particular, that there are two important predefined variables `$_` and `@_` which are quite distinct from each other, and e.g. `$_[2]` is a component of `@_`.

The case of letters is significant in variable names (as in Unix commands and in the C language), e.g. $foo and $Foo are distinct variables.

More examples of arrays and hashes

```
@days = (31,28,31,30,31,30,31,31,30,31,30,31);
           # A list with 12 elements.

$#days         # Last index of @days; 11 for above list

$#days = 7;    # shortens or lengthens list @days to 8 elements

@days          # ($days[0], $days[1],... )

@days[3,4,5]   # = (30,31,30)

@days{'a','c'} # same as ($days{'a'},$days{'c'})

%days          # (key1, value1, key2, value2, ...)
```

Special (predefined) variables

If a letter or underscore is the first character after the $, @, or %, the rest of the name may also contain digits and underscores. If this character is a digit, the rest must be digits. Perl has several dozen special (predefined) variables, recognized from their second character being non-alphanumeric. For example, $/ is the input record separator, newline "\n" by default. See section *Special Variables* in *Perl 5 Reference Guide* for a handy list.

The variable **$_** is presumed (defaulted) by Perl in many contexts when needed variables are not specified. Thus:

```
<STDIN>;          # assigns a record from filehandle STDIN to $_
print;            # prints the current value of $_
chomp;            # removes the trailing newline from $_
@things = split;  # parses $_ into white-space delimited
                  # words, which become successive
                  # elements of list @things.
```

$_, $/, $1, and other implicit variables contribute to Perl Paradox Number Two: *What you don't see can help you or hurt you.*

Subroutines and functions

The words "subroutine" and "function" are used interchangeably when discussing Perl. There really is no difference, but often a subprogram is called "function" if it returns a value and "subroutine" if it does not. On the other hand, quite often "function" means a *builtin* subprogram whereas "subroutine" means a subprogram which is defined in a Perl program.

Subroutines/functions are referenced with names containing an initial &, which is optional if reference is obviously a subroutine/function such as following the sub, do, and sort directives. An example of a simple function (which returns the square of its argument), and a sample invocation:

```
sub square { return $_[0] ** 2; }
print "5 squared is ", &square(5);
```

Inside a function, the special variable @_ contains the list of actual arguments, so $_[0] refers to the first argument (which is the only one in the example above).

A sub is defined with the sub keyword, and adds a new function to your program's capabilities. When you want to use this new function, you call it by name. For instance, here's a short definition of a sub called boo:

```
sub boo {
    print "Boo!\n";
}

boo();
```

(Older versions of Perl required that you precede the name of a sub with the & character when you call it. You no longer have to do this, but if you see code that looks like &bar in other people's Perl, that's why.)

Subs are useful because they allow you to break your program into small, reusable chunks. If you need to analyze a string in four different places in your program, it's much easier to write one &analyze_string sub and call it four times. This way, when you make an improvement to your string-analysis routine, you'll only need to do it in one place, instead of four.

In the same way that Perl's built-in functions can take parameters and can return values, your subs can, too. Whenever you call a sub, any parameters you pass to it are placed in the special array @_. You can also return a single value or a list by using the return keyword.

```
sub multiply {
    my (@ops) = @_;
    return $ops[0] * $ops[1];
}

for $i (1 .. 10) {
     print "$i squared is ", multiply($i, $i), "\n";
}
```

Why did we use the my keyword? That indicates that the variables are private to that sub, so that any existing value for the @ops array we're using elsewhere in our program won't get overwritten. This means that you'll evade a whole class of hard-to-trace bugs in your programs. You don't *have* to use my, but you also don't *have* to avoid smashing your thumb when you're hammering nails into a board. They're both just good ideas.

You can also use my to set up local variables in a sub without assigning them values right away. This can be useful for loop indexes or temporary variables:

```
sub annoy {
    my ($i, $j);
    for $i (1 .. 100) {
        $j .= "Is this annoying yet?\n";
    }
    print $j;
}
```

If you don't expressly use the return statement, the sub returns the result of the last statement. This implicit return value can sometimes be useful, but it does reduce your program's readability. Remember that you'll read your code many more times than you write it!

5. Loops

Control structures

Perl has a rich set of control structures. See section perlsyn in the manual for the full list. Theoretically, and very often practically too, you can use just if statements for branching and while statements for looping.

Within control structures you specify the actions to be conditionally or repeatedly executed as **blocks**. A block is simply a sequence of statements surrounded by braces. Notice that **braces {} are always required** (unlike in C).

The **simplest if statement** is of the form

if(*expression*)*block*

which means that the expression is evaluated, and if the result is true, the block is executed.

For example, the statement if($i < 10) {$j = 100;} sets the value of $j to 100 if the value of $i is less than 10. As mentioned above, braces are required (even if there is a single statement within them), and the parentheses around the

condition expression are obligatory, too.

A **two-branch `if` statement** is of the form

if(*expression*) *block1* else*block2*

which means that the expression is evaluated, and if the result is true, *block1* is executed, otherwise *block2* is executed.

The **`while` statement** is of the form

while(*expression*)*block*

which means that the expression is evaluated, and if the result is true, the block is executed, then the expression is re-evaluated and the process is repeated until the expression evaluates to false.

As a simple example of using the `while` statement is the following script, which splits input lines into fields (in a manner described above) and prints out the fields in reverse order.

```
while (<>) {
  split;
  $i = $#_;
  while($i >= 0) {
    print $_[$i--], " "; }
  print "\n";
}
```

The control in the (inner) `while` loop is based on using an auxiliary variable `$i`, which is initialized to the index of the last field and decremented (using the C-style embedded decrement operator --) within the loop until it reaches zero, i.e. all fields have been processed. The operator `>=` has the obvious meaning 'is greater than or equal to'.

The **`for` statement** is of the form

for(*initialization*;*condition*;*updating*)*block*

If you are familiar with the `for` statement in C, you probably want to use `for` in Perl too, but you might as well use just `while` as the loop construct. However, in Perl there is also a `foreach` statement, which will be illustrated by the next example (and was already used in a previous example).

Example: Command Line Values and Iterative Loops

```
print "$#ARGV is the subscript of the ",
    "last command argument.\n";
# Iterate on numeric subscript 0 to $#ARGV:
for ($i=0; $i <= $#ARGV; $i++) {
    print "Argument $i is $ARGV[$i].\n";
}
# A variation on the preceding loop:
foreach $item (@ARGV) {
    print "The word is:  $item.\n";
}
# A similar variation, using the
# "Default Scalar Variable" $_ :
foreach (@ARGV) {
    print "Say:  $_.\n";
```

```
}
```

Save the above text in a file named " example5.pl" and run this program as follow:

```
> perl example5.pl Gooood morning, Columbia!
2 is the subscript of the last command argument.
Argument 0 is Gooood.
Argument 1 is morning,.
Argument 2 is Columbia!.
The word is:  Gooood.
The word is:  morning,.
The word is:  Columbia!.
Say:  Gooood.
Say:  morning,.
Say:  Columbia!.
```

Example: Standard I/O

The following program illustrates simple interaction with user.

```
print STDOUT "Tell me something: ";
while ($input = <STDIN>) {
   print STDOUT "You said, quote: $input endquote\n";
   chomp $input;
   print STDOUT "Without the newline:  $input endquote\n";
   if ($input eq '') { print STDERR "Null input!\n"; }
   print STDOUT "Tell me more:\n";
}
print STDOUT "That's all!\n";
```

Note 1: The while statement's condition is an assignment statement: assign the next record from standard input to the variable $input. On end of file, this will assign not a null value but an "undefined" value. On keyboard input, end of file can be simulated in different ways on different systems; for example, on Unix the method is traditionally control-D, while on DOS it is control-Z followed by a newline (enter). An undefined value in the context of a condition evaluates to "false". So the while ($input = <STDIN>) does three things: gets a record, assigns it to $input, and tests whether $input is undefined. In other contexts, Perl treats an undefined variable as null or zero. Thus, if $i is not initialized, $i++ sets $i to 1. Perl Paradox Number Three: *Side effects can yield an elegant face or a pain in the rear.*

Note 2: Data records are by default terminated by a newline character "\n" which in the above example is included as the last character of variable $input. The chomp function removes the trailing end-of-line (newline) indicator (if present), which is defined in the special variable $/. (The chomp function was introduced Perl 5. Old programs often use the less safe function chop, which simply removes the last character, whatever it is.)

Save the above text in a file named " example6.pl" and run of this program as follow

```
> perl example6.pl
Tell me something: I'm warm.
You said, quote: I'm warm.
 endquote
Without the newline:  I'm warm. endquote
Tell me more:
Can I have some water?
You said, quote: Can I have some water?
 endquote
Without the newline:  Can I have some water? endquote
```

```
Tell me more:

You said, quote:
 endquote
Without the newline:    endquote
Null input!
Tell me more:
^D
That's all!
```

Example: Perls - A Perl Shell, Calculator, & Learning Tool

Type the following text and save it into a file named "perls.pl"

```
for (;;) {
    print '(',join(', ',@result),")\n? ";
    last unless $input = <STDIN>;
    $? = ''; $@ = ''; $! = '';
    @result = eval $input;
    if ($?) { print 'status=',$?,' ' }
    if ($@) { print 'errmsg=',$@,' ' }
    if ($!) { print 'errno=',$!+0,': ',$!,' ' }
}
```

This reads a line from the terminal and executes it as a Perl program. The `for (;;) {...}` construct makes an endless loop. The `last unless` line might be equivalently specified:

```
$InPuT = <STDIN>;               # Get line from standard input.
if (! defined($InPuT)) {last;}  # If no line, leave the loop.
```

The `eval` function in Perl evaluates a string as a Perl program. The special variable `$@` contains the Perl error message from the last `eval` or `do`.

Demonstration: (note that the statements `system 'date'` and `$x=`date`` invoke a system command named `date` and are therefore *system-dependent* and work (in a useful way) mostly on Unix):

perl perls.pl

```
()
? Howdy
(Howdy)
? 2+5
(7)
? sqrt(2)
(1.4142135623731)
? $x=sqrt(19)
(4.35889894354067)
? $x+5
(9.35889894354067)
? 1/0
errmsg=Illegal division by zero at (eval 6) line 3, <STDIN> chunk 6.
 ()
? system 'date'
Fri Feb  5 15:33:47 EET 1999
(0)
? $x=`date`
```

```
(Fri Feb  5 15:34:06 EET 1999
)
? chomp $x
(1)
? @y=split(' ',$x)
(Fri, Feb, 5, 15:34:06, EET, 1999)
? @y[1,2,5]
(Feb, 5, 1999)
? localtime()
(39, 38, 15, 5, 1, 99, 5, 35, 0)
? foreach (1..3) {print sqrt(),' ';}
1 1.4142135623731 1.73205080756888 ()
? exit
```

6. Operators

Comparison operators

Like all good programming languages, Perl allows you ask questions such as ``Is this number greater than that number?'' or ``Are these two strings the same?'' and do different things depending on the answer.

When you're dealing with numbers, Perl has four important operators: <, >, == and !=. These are the ``less than,'' ``greater than,'' ``equal to'' and ``not equal to'' operators. (You can also use <=, ``less than or equal to,'' and >=, ``greater than or equal to.)

You can use these operators along with one of Perl's *conditional* keywords, such as if and unless. Both of these keywords take a condition that Perl will test, and a block of code in curly brackets that Perl will run if the test works. These two words work just like their English equivalents - an if test succeeds if the condition turns out to be true, and an unless test succeeds if the condition turns out to be false:

```
if ($year_according_to_computer == 1900) {
    print "Y2K has doomed us all!  Everyone to the compound.\n";
}

unless ($bank_account > 0) {
    print "I'm broke!\n";
}
```

Be careful of the difference between = and ==! One equals sign means ``assignment'', two means ``comparison for equality''. This is a common, evil bug:

```
if ($a = 5) {
    print "This works - but doesn't do what you want!\n";
}
```

Instead of testing whether $a is equal to five, you've made $a equal to five and clobbered its old value.

Both if and unless can be followed by an else statement and code block, which executes if your test failed. You can also use elsif to chain together a bunch of if statements:

```
if ($a == 5) {
    print "It's five!\n";
```

```
} elsif ($a == 6) {
    print "It's six!\n";
} else {
    print "It's something else.\n";
}

unless ($pie eq 'apple') {
    print "Ew, I don't like $pie flavored pie.\n";
} else {
    print "Apple!  My favorite!\n";
}
```

while and until

Two slightly more complex keywords are while and until. They both take a condition and a block of code, just like if and unless, but they act like loops similar to for. Perl tests the condition, runs the block of code and runs it over and over again for as long as the condition is true (for a while loop) or false (for a until loop).

Take a look at the following code and try to guess what it will do before reading further:

```
$a = 0;

while ($a != 3) {
    $a++;
    print "Counting up to $a...\n";
}

until ($a == 0) {
    $a--;
    print "Counting down to $a...\n";
}
```

Here's what you see when you run this program:

```
Counting up to 1...
Counting up to 2...
Counting up to 3...
Counting down to 2...
Counting down to 1...
Counting down to 0...
```

String comparisons

So that's how you compare numbers. Now, what about strings? The most common string comparison operator is eq, which tests for *string equality* - that is, whether two strings have the same value.

Remember the pain that is caused when you mix up = and ==? Well, you can also mix up == and eq. This is one of the few cases where it *does* matter whether Perl is treating a value as a string or a number. Try this code:

```
$yes_no = "no";
if ($yes_no == "yes") {
    print "You said yes!\n";
}
```

Why does this code think you said yes? Remember that Perl automatically converts strings to numbers whenever it's necessary; the == operator implies that you're using numbers, so Perl converts the value of $yes_no (``no'') to the number 0, and ``yes'' to the number 0 as well. Since this equality test works (0 is equal to 0), the if block gets run. Change the condition to $yes_no eq "yes", and it'll do what it should.

Things can work the other way, too. The number five is *numerically* equal to the string " 5 ", so comparing them to == works. But when you compare five and " 5 " with eq, Perl will convert the number to the string "5" first, and then ask whether the two strings have the same value. Since they don't, the eq comparison fails. This code fragment will print Numeric equality!, but not String equality!:

```
$a = 5;
if ($a == " 5 ") { print "Numeric equality!\n"; }
if ($a eq " 5 ") { print "String equality!\n"; }
```

More fun with strings

You'll often want to manipulate strings: Break them into smaller pieces, put them together and change their contents. Perl offers three functions that make string manipulation easy and fun: substr(), split() and join().

If you want to retrieve part of a string (say, the first four characters or a 10-character chunk from the middle), use the substr() function. It takes either two or three parameters: the string you want to look at, the character position to start at (the first character is position 0) and the number of characters to retrieve. If you leave out the number of characters, you'll retrieve everything up to the end of the string.

```
$a = "Welcome to Perl!\n";
print substr($a, 0, 7);     # "Welcome"
print substr($a, 7);        # " to Perl!\n"
```

A neat and often-overlooked thing about substr() is that you can use a *negative* character position. This will retrieve a substring that begins with many characters from the *end* of the string.

```
$a = "Welcome to Perl!\n";
print substr($a, -6, 4);      # "Perl"
```

(Remember that inside double quotes, \n represents the single new-line character.)

You can also manipulate the string by using substr() to assign a new value to part of it. One useful trick is using a length of zero to *insert* characters into a string:

```
$a = "Welcome to Java!\n";
substr($a, 11, 4) = "Perl";   # $a is now "Welcome to Perl!\n";
substr($a, 7, 3) = "";        #       ... "Welcome Perl!\n";
substr($a, 0, 0) = "Hello. "; #       ... "Hello. Welcome Perl!\n";
```

Next, let's look at split(). This function breaks apart a string and returns a list of the pieces. split() generally takes two parameters: a *regular expression* to split the string with and the string you want to split. (We'll discuss regular expressions in more detail later; for the moment, we're only going to use a space. Note the special syntax for a regular expression: / /.) The characters you split won't show up in any of the list elements.

```
$a = "Hello. Welcome Perl!\n";
@a = split(/ /, $a);   # Three items: "Hello.", "Welcome", "Perl!\n"
```

You can also specify a third parameter: the maximum number of items to put in your list. The splitting will stop as soon as your list contains that many items:

```
$a = "Hello. Welcome Perl!\n";
@a = split(/ /, $a, 2);   # Two items: "Hello.", "Welcome Perl!\n";
```

Of course, what you can split, you can also join(). The join() function takes a list of strings and attaches them together with a specified string between each element, which may be an empty string:

```
@a = ("Hello.", "Welcome", "Perl!\n");
$a = join(' ', @a);       # "Hello. Welcome Perl!\n";
$b = join(' and ', @a);   # "Hello. and Welcome and Perl!\n";
$c = join('', @a);        # "Hello.WelcomePerl!\n";
```

Example: Numbers and Characters

```
print '007',' has been portrayed by at least ', 004, ' actors. ';
print 7+3, ' ', 7*3, ' ', 7/3, ' ', 7%3, ' ', 7**3, ' ';
$x = 7;
print $x;
print '   Doesn\'t resolve variables like $x and backslashes \n. ';
print "Does resolve $x and backslash\n";
$y = "A line containing $x and ending with line feed.\n";
print $y;
$y = "Con" . "cat" . "enation!\n";
print $y;
```

This produces:

```
007 has been portrayed by at least 4 actors. 10 21
2.33333333333333 1 343 7   Doesn't resolve variables like $x and
backslashes \n. Does resolve 7 and backslash
A line containing 7 and ending with line feed.
Concatenation!
```

In fact, most of the output "runs together", into one line. (The long line has been split above to keep the width of this document reasonable.) Can you guess why?

Example: Comparisons

The following example illustrates, in addition to comparisons, the << mechanism which is very useful when a program has to write out a multi-line string.

```
$x = 'operator';
print <<THATSALL;
A common mistake:  Confusing the assignment $x =
and the numeric comparison $x ==, and the character
comparison $x eq.
THATSALL
$x = 7;
if ($x == 7) { print "x is $x\n"; }
if ($x = 5)  {

    print "x is now $x,",
```

```
    "the assignment is successful.\n";
}
$x = 'stuff';
if ($x eq 'stuff') {
    print "Use eq, ne, lt, gt, etc for strings.\n";
}
```

This produces:

```
A common mistake:  Confusing the assignment operator =
and the numeric comparison operator ==, and the character
comparison operator eq.
x is 7
x is now 5, the assignment is successful.
Use eq, ne, lt, gt, etc for strings.
```

Example: Ordinary Arrays

```
@stuff = ('This', 'is', 'a', 'list.');
print "Lists and strings are indexed from 0.\n";
print "So \$stuff[1] = $stuff[1], ",
    "and \$#stuff = $#stuff.\n";
print @stuff,"\n";
print join('...',@stuff),"\n";
splice(@stuff, 3, 0, ('fine', 'little'));
print join('...',@stuff),"\n";
```

This produces:

```
Lists and strings are indexed from 0.
So $stuff[1] = is, and $#stuff = 3.
Thisisalist.
This...is...a...list.
This...is...a...fine...little...list.
```

Exercise: Validate a date.

```
print "Enter a date numerically:  year-month-dayyear\n";
$_ = <STDIN>;
chomp;
($year,$month,$day) = split('-');
```

Complete this program so that it checks whether the date is valid. Print an error message if the month is not valid. Print an error message if the day is not valid for the given month (31 is ok for January but not for February). See if you can avoid using conditionals (if, unless, ?,...) statements but instead use data structures.

Approach this incrementally. On the first draft, assume that the user enters three numbers separated by hyphens and that February has 28 days. Subsequent refinements should account for bad input and leap year. Finally, find a Perl builtin function that converts a date to system time, and see how to use that to validate time data generally.

Exercise: Play with associative arrays.

Start with a few assignments like:

```
$name{12345} = 'John Doe';
$name{24680} = 'Jane Smith';
```

Print these scalars. What is the value of an associative array element that has never been assigned? What happens if you assign an associative array to a scalar? What happens if you assign an associative array to a normal array?

```
$blunk = %name;
@persons = %name;
print '$blunk=',$blunk,', @persons=',
    join(', ',@persons),"\n";
```

What happens if you assign a normal array to an associative array?

6. File I/O

Filehandles

To read from or write to a file, you have to *open* it. When you open a file, Perl asks the operating system if the file can be accessed - does the file exist if you're trying to read it (or can it be created if you're trying to create a new file), and do you have the necessary file permissions to do what you want? If you're allowed to use the file, the operating system will prepare it for you, and Perl will give you a *filehandle*.

You ask Perl to create a filehandle for you by using the `open()` function, which takes two arguments: the filehandle you want to create and the file you want to work with. First, we'll concentrate on reading files. The following statement opens the file `log.txt` using the filehandle `LOGFILE`:

```
open (LOGFILE, "log.txt");
```

Opening a file involves several behind-the-scenes tasks that Perl and the operating system undertake together, such as checking that the file you want to open actually exists (or creating it if you're trying to create a new file) and making sure you're allowed to manipulate the file (do you have the necessary file permissions, for instance). Perl will do all of this for you, so in general you don't need to worry about it.

Once you've opened a file to read, you can retrieve lines from it by using the `<>` construct. Inside the angle brackets, place the name of your filehandle. What is returned by this depends on what you *want* to get: in a scalar context (a more technical way of saying ``if you're assigning it to a scalar''), you retrieve the next line from the file, but if you're looking for a list, you get a list of all the remaining lines in the file. (One common trick is to use `for $lines (<FH>)` to retrieve all the lines from a file - the `for` means you're asking a list.)

You can, of course, `close` a filehandle that you've opened. You don't always have to do this, because Perl is clever enough to close a filehandle when your program ends or when you try to reuse an existing filehandle. It's a good idea, though, to use the `close` statement. Not only will it make your code more readable, but your operating system has built-in limits on the number of files that can be open at once, and each open filehandle will take up valuable memory.

Here's a simple program that will display the contents of the file `log.txt`, and assumes that the first line of the file is its title:

```
open (LOGFILE, "log.txt") or die "I couldn't get at log.txt";
# We'll discuss the "or die" in a moment.

$title = <LOGFILE>;
print "Report Title: $title";
for $line (<LOGFILE>) {
```

```
    print $line;
}
close LOGFILE;
```

Writing files

You also use `open()` when you are writing to a file. There are two ways to open a file for writing: *overwrite* and *append*. When you open a file in overwrite mode, you erase whatever it previously contained. In append mode, you attach your new data to the end of the existing file without erasing anything that was already there.

To indicate that you want a filehandle for writing, you put a single `>` character before the filename you want to use. This opens the file in overwrite mode. To open it in append mode, use two `>` characters.

```
open (OVERWRITE, ">overwrite.txt") or die "$! error trying to overwrite";
# The original contents are gone, wave goodbye.

open (APPEND, ">>append.txt") or die "$! error trying to append";
# Original contents still there, we're adding to the end of the file
```

Once our filehandle is open, we can use the humble `print` statement to write to it. Specify the filehandle you want to write to and a list of values you want to write:

```
print OVERWRITE "This is the new content.\n";
print APPEND "We're adding to the end here.\n", "And here too.\n";
```

Live free or die!

You noticed that most of our `open()` statements are followed by `or die "some sort of message"`. This is because we live in an imperfect world, where programs don't always behave exactly the way we want them to. It's always possible for an `open()` call to fail; maybe you're trying to write to a file that you're not allowed to write, or you're trying to read from a file that doesn't exist. In Perl, you can guard against these problems by using `or` and `and`.

A series of statements separated by `or` will continue until you hit one that works, or returns a true value. This line of code will either succeed at opening `OUTPUT` in overwrite mode, or cause Perl to quit:

```
open (OUTPUT, ">$outfile") or die "Can't write to $outfile: $!";
```

The `die` statement ends your program with an error message. The special variable `$!` contains Perl's explanation of the error. In this case, you might see something like this if you're not allowed to write to the file. Note that you get both the actual error message (``Permission denied'') and the line where it happened:

```
Can't write to a2-die.txt: Permission denied at ./a2-die.pl line 1.
```

Defensive programming like this is useful for making your programs more error-resistant - you don't want to write to a file that you haven't successfully opened!

Here's an example: As part of your job, you write a program that records its results in a file called `vitalreport.txt`. You use the following code:

```
open VITAL, ">vitalreport.txt";
```

If this `open()` call fails (for instance, `vitalreport.txt` is owned by another user who hasn't given you write permission), you'll never know it until someone looks at the file afterward and wonders why the vital report wasn't written. (Just imagine the joy if that ``someone'' is your boss, the day before your annual performance review.) When you use `or die`, you

avoid all this:

```
open VITAL, ">vitalreport.txt" or die "Can't write vital report: $!";
```

Instead of wondering whether your program wrote your vital report, you'll immediately have an error message that both tells you what went wrong and on what line of your program the error occurred.

You can use or for more than just testing file operations:

```
($pie eq 'apple') or ($pie eq 'cherry') or ($pie eq 'blueberry')
    or print "But I wanted apple, cherry, or blueberry!\n";
```

In this sequence, if you have an appropriate pie, Perl skips the rest of the chain. Once one statement works, the rest are ignored. The and operator does the opposite: It evaluates your chain of statements, but stops when one of them *doesn't* work.

```
open (LOG, "log.file") and print "Logfile is open!\n";
```

This statement will only show you the words *Logfile is open!* if the open() succeeds - do you see why?

File Status

-e	file exists
-s	file has non zero size
-z	file has zero size
-r	file is readable
-w	file is writable
-x	file is executable
-f	file is a plain file, not a directory
-d	file is a directory

Example: File I/O

The following program illustrates reading from a file and writing to a file. It also reads from character from standard input, in order to let the user control what happens. Moreover, it illustrates how "short circuit" operator || can be used so that error processing can be written more conveniently. An expression of the form $A||B$ is evaluated so that A is always evaluated first, and if the result is "true", the expression B is not evaluated at all.

```
# Function:  Reverse each line of a file

# 1: Get command line values:
if ($#ARGV !=1) {
    die "Usage:  $0 inputfile outputfile\n";
}
($infile,$outfile) = @ARGV;
if (! -r $infile) {
    die "Can't read input $infile\n";
```

```
    }
    if (! -f $infile) {
        die "Input $infile is not a plain file\n";
    }

    # 2: Validate files
    open(INPUT,"<$infile") ||
        die "Can't input $infile $!";
    if ( -e $outfile) {
        print STDERR "Output file $outfile exists!\n";
        until ($ans eq 'r' || $ans eq 'a' || $ans eq 'e' ) {
            print STDERR "replace, append, or exit? ";
            $ans = getc(STDIN);
        }
        if ($ans eq 'e') {exit}
    }
    if ($ans eq 'a') {$mode='>>'}
    else {$mode='>'}
    open(OUTPUT,"$mode$outfile") ||
        die "Can't output $outfile $!";

    # 3: Read input, reverse each line, output it.
    while (<INPUT>) {
        chomp $_;
        $_ = reverse $_;
        print OUTPUT $_,"\n";
    }

    # 4: Done!
    close INPUT,OUTPUT;
    exit;
```

Navigating the file system

`opendir` gives you a handle on a directory the same way open gives you a file handle:

```
opendir (TEMPDIR, "/tmp"); #UNIX
opendir (TEMPDIR, "c:\temp"); #Windows
```

`readdir` reads the contents of the director:

In list context

```
@tempfiles = readdir TEMPDIR;#@tempfiles contains all the directory entries
```

or in scalar context

```
$dir="c:\temp";
opendir(TEMPDIR, $dir) or die "Can't open $dir";
foreach $entry (readdir, TEMPDIR) {

..... #process the entry

}
closedir TEMPDIR
```

Example: renaming files

Unix users often get frustrated when they need to rename files e.g. so that all file names ending with some suffix (like .for) are renamed by changing the suffix (e.g. to .f). In some operating systems this is easy, but in normal Unix command interpreters there is no direct way to do it. (A novice user often tries mv *.for *.f but it does not work at all in the way you would like.)

No problem, it's easily done in Perl, for example as follows:

```
while(<*.for>) {
  $oldname = $_;
  s/\.for$/\.f/;
  rename $oldname, $_; }
```

This works on any system with reasonably normal file naming conventions, not just Unix.

The while statement is different from what we have seen before. It means that all file names matching the pattern within the angle brackets (here *.for) are processed and assigned, each in turn, to the variable $_. In fact, the meaning of $_ is not simply 'the current input line' as told before but more generally 'the current data being processed', and the context defines in each case what this exactly means.

Within the loop, the file name is copied to variable $oldname and then modified using a construct which performs a **s**ubstitution.

One might try to use simply s/.for/.f/; instead of s/\.for$/\.f/;. Although the simpler version works in most cases, it is buggy, because the symbol . stands for *any character*, not just the period, and because there is no requirement that the string .for must appear at the end of the file name only. Thus, the code would rename e.g. zapfor.for to za.f.for. To refer to the period character, one must use "escape" notation by prefixing it with a backslash. Moreover, if the trailing $ (denoting *end of line*) is omitted, the code would apply to the *first* appearance of .for in the filename.

Finally, the rename operation is performed using a Perl built-in function, rename, which takes two file names as arguments. Alternatively, we could also use the following:

```
  system "mv $oldname $_";
```

which does the same operation (less efficiently, and in a system-dependent manner) by asking the Unix system to execute its mv command.

7. Data Processing: Grade Book Example

This example produces a score summary report by combining data from a simple file of student info and a file of their scores.

Input file stufile is delimited with colons. Fields are Student ID, Name, Year:

```
123456:Washington,George:SR
246802:Lincoln,Abraham "Abe":SO
357913:Jefferson,Thomas:JR
212121:Roosevelt,Theodore "Teddy":SO
```

Input file scorefile is delimited with blanks. Fields are Student ID, Exam number, Score on exam. Note that Abe is missing exam 2:

```
123456 1 98
212121 1 86
246802 1 89
357913 1 90
123456 2 96
212121 2 88
357913 2 92
123456 3 97
212121 3 96
246802 3 95
357913 3 94
```

The desired report:

```
Stu-ID Name...                  1   2   3   Totals:

357913 Jefferson,Thomas          90  92  94      276
246802 Lincoln,Abraham "Abe"      89      95      184
212121 Roosevelt,Theodore "Teddy"   86  88  96      270
123456 Washington,George         98  96  97      291

                    Totals:  363 276 382
```

The program that made this report:

```
# Gradebook - demonstrates I/O, associative
  # arrays, sorting, and report formatting.
  # This accommodates any number of exams and students
  # and missing data.  Input files are:

  $stufile='stufile';
  $scorefile='scorefile';

  open (NAMES,"<$stufile")
     || die "Can't open $stufile $!";
  open (SCORES,"<$scorefile")
     || die "Can't open $scorefile $!";

  # Build an associative array (%name) of student info
  # keyed by student number
   while (<NAMES>) {
         ($stuid,$name,$year) = split(':',$_);
         $name{$stuid}=$name;
```

```
        if (length($name)>$maxnamelength) {
                $maxnamelength=length($name);
        }
  }
  close NAMES;

# Build an assoc. array (%score) from the test scores:
  while (<SCORES>) {
     ($stuid,$examno,$score) = split;
     $score{$stuid,$examno} = $score;
     if ($examno > $maxexamno) {
        $maxexamno = $examno;
     }
  }
  close SCORES;

  # Print the report from accumulated data!
  printf "%6s %-${maxnamelength}s ",
        'Stu-ID','Name...';
  foreach $examno (1..$maxexamno) {
        printf "%4d",$examno;
  }
  printf "%10s\n\n",'Totals:';

  # Subroutine "byname" is used to sort the %name array.
  # The "sort" function gives variables $a and $b to
  # subroutines it calls.
  # "x cmp y" expression returns -1 if x lt y, 0 if x eq y,
  # +1 if x gt y.  See the Perl documentation for details.

  sub byname { $name{$a} cmp $name{$b} }

  # Order student IDs so the names appear alphabetically:
  foreach $stuid ( sort byname keys(%name) ) {
     # Print scores for a student, and a total:
     printf "%6d %-${maxnamelength}s ",
           $stuid,$name{$stuid};
     $total = 0;
        foreach $examno (1..$maxexamno) {
           printf "%4s",$score{$stuid,$examno};
           $total += $score{$stuid,$examno};
           $examtot{$examno} += $score{$stuid,$examno};
        }
        printf "%10d\n",$total;
  }

  printf "\n%6s %${maxnamelength}s ","","Totals: ";
  foreach $examno (1..$maxexamno) {
     printf "%4d",$examtot{$examno};
  }
  print "\n";
  exit(0);
```

The foreach $stuid ... loop first calls the predefined function sort, passing the name of the ordering subroutine byname as the first parameter. That function returns an array, and the loop iterates over that array so that $stuid gets each of the values in the array in succession.

More advanced applications could be written using the feature that Perl allows an associative array to be "tied" (using the tie function) to a genuine database, such that expressions like $record = $student{$key} use the database.

8. Matching (1 & 2)

The pattern matching and substitution operators are described in detail in section *Regexp Quote-Like Operators* of perlop in the manual.

See also section item *Regular expressions and pattern matching* in perlfunc, although some of the entries there just refer to perlop. See *Regular expressions in Perl* for a tabular summary with examples.

A simple example: tr

Perl has powerful tools for string manipulation. But let us first take **a simple example**. One often wants to *convert letters in input data to lower case*. That's easy:

```
tr /A-Z/a-z/;
```

This can be read as follows: "**tr**anslate all characters in the range from A to Z to the corresponding characters in the range from a to z".

The operation is applied to the value of $_, typically the current input line. If you would like it to be applied to the value of a variable $foo, you should write

```
$foo =~ tr /A-Z/a-z/;
```

Thus, the syntax is odd-looking, but once you get accustomed to it, the Perl string manipulation tools are easy to use.

The matching operator =~

The =~ operator performs pattern matching. For example, if:

```
$s = 'One if by land and two if by sea';
```

then:

```
if ($s =~ /if by la/) {print "YES"}
else {print "NO"}
```

prints YES, because the value of $s matches the simple constant pattern "if by la".

```
if ($s =~ /one/) {print "YES"}
else {print "NO"}
```

prints NO, because the string does not match the pattern. However, by adding the i option to ignore case of letters, we would get a YES from the following:

```
if ($s =~ /one/i) {print "YES"}
else {print "NO"}
```

Regular expressions

Matching involves use of patterns called regular expressions. This, as you will see, leads to Perl Paradox Number Four: *Regular expressions aren't*. See section perlre in the manual.

Patterns can contain a mind-boggling variety of special directions that facilitate very general matching. For example, a period matches any character (except the "newline" \n character).

```
if ($x =~ /l.mp/) {print "YES"}
```

would print YES for $x = "lamp" or "lump" or "slumped", but not for $x = "lmp" or "less amperes".

Parentheses () group pattern elements. An asterisk * means that the preceding character, element, or group of elements may occur zero times, one time, or many times. Similarly, a plus + means that the preceding element or group of elements must occur at least once. A question mark ? matches zero or one times. So:

```
/fr.*nd/  matches "frnd", "friend", "front and back"
/fr.+nd/  matches "frond", "friend", "front and back"
          but not "frnd".
/10*1/    matches "11", "101", "1001", "100000001".
/b(an)*a/ matches "ba", "bana", "banana", "bananananana"
/flo?at/  matches "flat" and "float"
          but not "flooat"
```

Square brackets [] match a class of single characters.

```
[0123456789] matches any single digit
[0-9]        matches any single digit
[0-9]+       matches any sequence of one or more digits
[a-z]+       matches any lowercase word
[A-Z]+       matches any uppercase word
```

[^*class*] matches those characters which do *not* match [*class*] (i.e., ^ denotes negation here - but something quite different outside brackets, see below):

```
[^0-9]       matches any non-digit character.
```

Curly braces {} allow more precise specification of repeated fields. For example [0-9]{6} matches any sequence of 6 digits, and [0-9]{6,10} matches any sequence of 6 to 10 digits.

Patterns float, unless anchored. The circumflex ^ (outside []) anchors a pattern to the beginning, and dollar sign $ anchors a pattern at the end, so:

```
/at/         matches "at", "attention", "flat", & "flatter"
/^at/        matches "at" & "attention" but not "flat"
/at$/        matches "at" & "flat", but not "attention"
/^at$/       matches "at" and nothing else.
/^at$/i      matches "at", "At", "aT", and "AT".
/^[ \t]*$/   matches a "blank line", one that contains nothing
                   or any combination of blanks and tabs.
```

Other characters simply match themselves, but the characters

+?.*^$()[]{}|\

and usually / must be **escaped** with a backslash \ to be taken literally.

```
/10.2/       matches "10Q2", "1052", and "10.2"
/10\.2/      matches "10.2" but not "10Q2" or "1052"
/\*+/        matches one or more asterisks
/A:\\DIR/    matches "A:\DIR"
/\/usr\/bin/ matches "/usr/bin"
```

If a backslash precedes an alphanumeric character, this sequence takes a special meaning, typically a short form of a [] character class. For example, \d is the same as the [0-9] digits character class.

```
/[-+]?\d*\.?\d*/      is the same as
/[-+]?[0-9]*\.?\d*/
```

Either of the above matches decimal numbers: "-150", "-4.13", "3.1415", "+0000.00", etc.

A simple \s specifies "white space", the same as the character class [\t\n\r\f] (blank, tab, newline, carriage return, formfeed). A character may be specified in hexadecimal as a \x followed by two hexadecimal digits which specify the Ascii code of the character; for example, \x1b is the ESC character.

A vertical bar | means "or".

```
if ($answer =~ /^yes|^yeah/i ) {
    print "Affirmative!";
}
```

prints Affirmative! for $answer equal to "yes" or "yeah" (or "YeS", or "yessireebob, that's right", but not "yep").

9. Matching (3)

Perl's most powerful and interesting way of playing with strings, *regular expressions*, or *regexes* for short. (The rule is this: after the 50th time you type ``regular expression,'' you find you type ``regexp'' the next 50 times.)

Regular expressions are complex enough that you could write a whole book on them (and, in fact, someone did - *Mastering Regular Expressions* by Jeffrey Friedl).

Simple matching

The simplest regular expressions are *matching* expressions. They perform tests using keywords like if, while and unless. Or, if you want to be really clever, tests that you can use with and and or. A matching regexp will return a true value if whatever you try to match occurs inside a string. When you want to use a regular expression to match against a string, you use the special =~ operator:

```
$user_location = "I see thirteen black cats under a ladder.";
if ($string =~ /thirteen/) {
    print "Eek, bad luck!\n";
}
```

Notice the syntax of a regular expression: a string within a pair of slashes. The code $string =~ /thirteen/ asks whether the literal string thirteen occurs anywhere inside $string. If it does, then the test evaluates true; otherwise, it evaluates false.

Metacharacters

A *metacharacter* is a character or sequence of characters that has special meaning. We've discussed metacharacters in the context of double-quoted strings, where the sequence \n mean the newline character, not a backslash, and the character n and \t means the tab character.

Regular expressions have a rich vocabulary of metacharacters that let you ask interesting questions such as, ``Does this expression occur at the end of a string?'' or ``Does this string contain a series of numbers?''

The two simplest metacharacters are ^ and $. These indicate ``beginning of string'' and ``end of string,'' respectively. For example, the regexp /^Bob/ will match ``Bob was here,'' ``Bob'' and ``Bobby.'' It won't match ``It's Bob and David,'' because Bob doesn't show up at the beginning of the string. The $ character, on the other hand, means that you are matching the end of a string. The regexp /David$/ will match ``Bob and David,'' but not ``David and Bob.'' Here's a simple routine that will take lines from a file and only print URLs that seem to indicate HTML files:

```
for $line (<URLLIST>) {
        # "If the line starts with http: and ends with html...."
        if (($line =~ /^http:/) and
            ($line =~ /html$/)) {
            print $line;
        }
    }
```

Another useful set of metacharacters is called *wildcards*. If you've ever used a Unix shell or the Windows DOS prompt, you're familiar with wildcards characters like * and ?. For example when you type ls a*.txt, you see all filenames that begin with the letter a and end with .txt. Perl is a bit more complex, but works on the same general principle.

In Perl, the generic wildcard character is .. A period inside a regular expression will match *any* character, except a newline. For example, the regexp /a.b/ will match anything that contains a, another character that's not a newline, followed by b - ``aab,'' ``a3b,'' ``a b,'' and so forth.

If you want to *literally* match a metacharacter, you must escape it with a backslash. The regex /Mr./ matches anything that contains ``Mr'' followed by another character. If you only want to match a string that actually contains ``Mr.,'' you must use /Mr\./.

On its own, the . metacharacter isn't very useful, which is why Perl provides three wildcard *quantifiers*: +, ? and *. Each quantifier means something different.

The + quantifier is the easiest to understand: It means to match the immediately preceding character or metacharacter *one or more times*. The regular expression /ab+c/ will match ``abc,'' ``abbc,'' ``abbbc'' and so on.

The * quantifier matches the immediately preceding character or metacharacter *zero or more times*. This is different from the + quantifier! /ab*c/ will match ``abc,'' ``abbc,'' and so on, just like /ab+c/ did, but it'll also match ``ac,'' because there are zero occurences of b in that string.

Finally, the ? quantifier will match the preceding character *zero or one times*. The regex /ab?c/ will match ``ac'' (zero occurences of b) and ``abc'' (one occurence of b). It won't match ``abbc,'' ``abbbc'' and so on.

We can rewrite our URL-matching code to use these metacharacters. This'll make it more concise. Instead of using two

separate regular expressions (/^http:/ and /html$/), we combine them into one regular expression: /^http:.+html$/. To understand what this does, read from left to right: This regex will match any string that *starts with ``http:''* followed by *one or more occurences of any character*, and *ends with ``html''*. Now, our routine is:

```
for $line (<URLLIST>) {
      if ($line =~ /^http:.+html$/) {
         print $line;
      }
   }
```

Remember the /^something$/ construction - it's very useful!

Character classes

We've already discussed one special metacharacter, ., that matches any character except a newline. But you'll often want to match only specific types of characters. Perl provides several metacharacters for this. <\d> will match a single digit, \w will match any single ``word'' character (which, to Perl, means a letter, digit or underscore), and \s matches a whitespace character (space and tab, as well as the \n and \r characters).

These metacharacters work like any other character: You can match against them, or you can use quantifiers like + and *. The regex /^\s+/ will match any string that begins with whitespace, and /\w+/ will match a string that contains at least one word. (But remember that Perl's definition of ``word'' characters includes digits and the underscore, so whether or not you think _ or 25 are words, Perl does!)

One good use for \d is testing strings to see whether they contain numbers. For example, you might need to verify that a string contains an American-style phone number, which has the form 555-1212. You could use code like this:

```
unless ($phone =~ /\d\d\d-\d\d\d\d/) {
print "That's not a phone number!\n";
   }
```

All those \d metacharacters make the regex hard to read. Fortunately, Perl allows us to improve on that. You can use numbers inside curly braces to indicate a *quantity* you want to match, like this:

```
unless ($phone =~ /\d{3}-\d{4}/) {
print "That's not a phone number!\n";
  }
```

The string \d{3} means to match exactly three numbers, and \d{4} matches exactly four digits. If you want to use a range of numbers, you can separate them with a comma; leaving out the second number makes the range open-ended. \d{2,5} will match two to five digits, and <\w{3,}> will match a word that's at least three characters long.

You can also *invert* the \d, \s and \w metacharacters to refer to anything *but* that type of character. \D matches nondigits; \W matches any character that *isn't* a letter, digit or underscore; and \S matches anything that isn't whitespace.

If these metacharacters won't do what you want, you can define your own. You define a character class by enclosing a list of the allowable characters in square brackets. For example, a class containing only the lowercase vowels is [aeiou]. /b[aeiou]g/ will match any string that contains ``bag,'' ``beg,'' ``big,'' ``bog'' or ``bug''. You use dashes to indicate a range of characters, like [a-f]. (If Perl didn't give us the \d metacharacter, we could do the same thing with [0-9].) You can combine character classes with quantifiers:

```
if ($string =~ /[aeiou]{2}/) {
```

```
 print "This string contains at least
       two vowels in a row.\n";
   }
```

You can also invert character classes by beginning them with the ^ character. An inverted character class will match anything you *don't* list. [^aeiou] matches every character except the lowercase vowels. (Yes, ^ can also mean ``beginning of string,'' so be careful.)

Flags

By default, regular expression matches are case-sensitive (that is, /bob/ doesn't match ``Bob''). You can place *flags* after a regexp to modify their behaviour. The most commonly used flag is i, which makes a match case-insensitive:

```
 $greet = "Hey everybody, it's Bob and David!";
   if ($greet =~ /bob/i) {
       print "Hi, Bob!\n";
   }
```

We'll talk about more flags later.

Subexpressions

You might want to check for more than one thing at a time. For example, you're writing a ``mood meter'' that you use to scan outgoing e-mail for potentially damaging phrases. You can use the pipe character | to separate different things you are looking for:

```
 # In reality, @email_lines would come from your email text,
   # but here we'll just provide some convenient filler.
   @email_lines = ("Dear idiot:",
                   "I hate you, you twit.  You're a dope.",
                   "I bet you mistreat your llama.",
                   "Signed, Doug");

   for $check_line (@email_lines) {
       if ($check_line =~ /idiot|dope|twit|llama/) {
           print "Be careful!  This line might
                  contain something offensive:\n",
                 $check_line, "\n";
       }
   }
```

The matching expression /idiot|dope|twit|llama/ will be true if ``idiot,'' ``dope,'' ``twit'' or ``llama'' show up anywhere in the string.

One of the more interesting things you can do with regular expressions is *subexpression matching*, or grouping. A subexpression is like another, smaller regex buried inside your larger regexp, and is placed inside parentheses. The string that caused the subexpression to match will be stored in the special variable $1. We can use this to make our mood meter more explicit about the problems with your e-mail:

```
 for $check_line (@email_lines) {
       if ($check_line =~ /(idiot|dope|twit|llama)/) {
           print "Be careful!  This line contains the
                  offensive word $1:\n",
                 $check_line, "\n";
```

```
    }
  }
```

Of course, you can put matching expressions in your subexpression. Your mood watch program can be extended to prevent you from sending e-mail that contains more than three exclamation points in a row. We'll use the special {3,} quantifier to make sure we get *all* the exclamation points.

```
for $check_line (@email_lines) {
      if ($check_line =~ /(!{3,})/) {
          print "Using punctuation like '$1'
                 is the sign of a sick mind:\n",
                $check_line, "\n";
      }
  }
```

If your regex contains more than one subexpression, the results will be stored in variables named $1, $2, $3 and so on. Here's some code that will change names in ``lastname, firstname'' format back to normal:

```
$name = "Wall, Larry";
  $name =~ /(\w+), (\w+)/;
  # $1 contains last name, $2 contains first name

  $name = "$2 $1";
  # $name now contains "Larry Wall"
```

You can even nest subexpressions inside one another - they're ordered as they open, from left to right. Here's an example of how to retrieve the full time, hours, minutes and seconds separately from a string that contains a timestamp in hh:mm:ss format. (Notice that we're using the {1,2} quantifier so that a timestamp like ``9:30:50'' will be matched.)

```
$string = "The time is 12:25:30 and I'm hungry.";
   $string =~ /((\d{1,2}):(\d{2}):(\d{2}))/;
   @time = ($1, $2, $3, $4);
```

Here's a hint that you might find useful: You can assign *to* a list of scalar values whenever you're assigning *from* a list. If you prefer to have readable variable names instead of an array, try using this line instead:

```
($time, $hours, $minutes, $seconds) = ($1, $2, $3, $4);
```

Assigning to a list of variables when you're using subexpressions happens often enough that Perl gives you a handy shortcut:

```
($time, $hours, $minutes, $seconds) =
       ($string =~ /((\d{1,2}):(\d{2}):(\d{2}))/);
```

Watch out!

Regular expressions have two traps that generate bugs in your Perl programs: They always start at the beginning of the string, and quantifiers always match as much of the string as possible.

Here's some simple code for counting all the numbers in a string and showing them to the user. We'll use while to loop over the string, matching over and over until we've counted all the numbers.

```
$number = "Look, 200 5-sided, 4-colored pentagon maps.";
   while ($number =~ /(\d+)/) {
      print "I found the number $1.\n";
      $number_count++;
   }
   print "There are $number_count numbers here.\n";
```

This code is actually so simple it doesn't work! When you run it, Perl will print I found the number 200 over and over again. Perl always begins matching at the beginning of the string, so it will always find the 200, and never get to the following numbers.

You can avoid this by using the g flag with your regex. This flag will tell Perl to remember where it was in the string when it returns to it. When you insert the g flag, our code looks like this:

```
$number = "Look, 200 5-sided, 4-colored pentagon maps.";
  while ($number =~ /(\d+)/g) {
     print "I found the number $1.\n";
     $number_count++;
  }
  print "There are $number_count numbers here.\n";
```

Now we get the results we expected:

```
I found the number 200.
  I found the number 5.
  I found the number 4.
  There are 3 numbers here.
```

The second trap is that a quantifier will always match as many characters as it can. Look at this example code, but don't run it yet:

```
$book_pref = "The cat in the hat is where it's at.\n";
  $book_pref =~ /(cat.*at)/;
  print $1, "\n";
```

Take a guess: What's in $1 right now? Now run the code. Does this seem counterintuitive?

The matching expression (cat.*at) is greedy. It contains cat in the hat is where it's at because that's the largest string that matches. Remember, read left to right: ``cat,'' followed by any number of characters, followed by ``at.'' If you want to match the string cat in the hat, you have to rewrite your regexp so it isn't as greedy. There are two ways to do this:

1. Make the match more precise (try /(cat.*hat)/ instead). Of course, this still might not work - try using this regexp against The cat in the hat is who I hate.

2. Use a ? character after a quantifier to specify nongreedy matching. .*? instead of .* means that Perl will try to match the *smallest* string possible instead of the largest:

```
# Now we get "cat in the hat" in $1.
 $book_pref =~ /(cat.*?at)/;
```

Search and replace (1)

Now that we've talked about *matching*, there's one other thing regular expressions can do for you: *replacing*.

If you've ever used a text editor or word processor, you're familiar with the search-and-replace function. Perl's regexp facilities include something similar, the s/// operator, which has the following syntax: s/regex/replacement string/. If the string you're testing matches *regex*, then whatever matched is replaced with the contents of *replacement string*. For instance, this code will change a cat into a dog:

```
$pet = "I love my cat.\n";
  $pet =~ s/cat/dog/;
  print $pet;
```

You can also use subexpressions in your matching expression, and use the variables $1, $2 and so on, that they create. The replacement string will substitute these, or any other variables, as if it were a double-quoted string. Remember our code for changing Wall, Larry into Larry Wall? We can rewrite it as a single s/// statement!

```
$name = "Wall, Larry";
  $name =~ s/(\w+), (\w+)/$2 $1/;  # "Larry Wall"
```

s/// can take flags, just like matching expressions. The two most important flags are g (global) and i (case-insensitive). Normally, a substitution will only happen *once*, but specifying the g flag will make it happen as long as the regex matches the string. Try this code, and then remove the g flag and try it again:

```
$pet = "I love my cat Sylvester, and my other cat Bill.\n";
  $pet =~ s/cat/dog/g;
  print $pet;
```

Notice that without the g flag, Bill doesn't turn into a dog.

The i flag works just as it did when we were only using matching expressions: It forces your matching search to be case-insensitive.

Search and replace (2)

Substitutions

The =~ operator can be used for making substitutions in strings. An expression of the form

$*variable* =~ /*pattern*/

tests whether the value of *variable* matches *pattern*. Normally such an expression is used as a condition (test) in an if statement or other control structure. But an expression of the form

$*variable* =~ s/*pattern*/*pattern2*/

first tests for a match, and if there is a match, **replaces**, within the value of *variable*, the string that matched *pattern* by *pattern2* or, if it contains special notations, by a string generated from *pattern2*.

If you wish to modify the value of the predefined variable $_, you can write simply

s/*pattern*/*pattern2*/

When you include parentheses () in a matched string, the matching text in the parenthesis may subsequently be referenced via variables $1, $2, $3, ... for each left parenthesis encountered. These matches can also be assigned as sequential values of an array.

Example. Assume that we have a text file containing notations like U+0123 which we wish to modify by slapping the strings <code> and </code> around them. The exact format of those notations is U+ followed by one or more hexadecimal characters. Thus, the following program would do the job:

```
while(<>) {
  s?U\+([0-9a-fA-F]+)?<code>U\+$1</code>?g;
  print;}
```

Note: Although we normally use the slash character / when specifying substitutions, here it cannot be used, since the slash occurs in the patterns. We can then pick almost any character which does not occur in the patterns and use it as a separator; here we use the question mark ?. (Alternatively, we could "escape" the / as \/ in the patterns.) Notice that the plus sign + must be "escaped" (as \+) when it needs to stand for itself and not act as a special character in a regular expression.

Parsing

The following program parses dates in a strange old format.

```
$s = 'There is 1 date 10/25/95 in here somewhere.';
print "\$s=$s\n";
$s =~ /(\d{1,2})\/(\d{1,2})\/(\d{2,4})/;
print "Trick 1: \$1=$1, \$2=$2, \$3=$3,\n",
      "         \$\`=",$`," \$\'=",$',"\n";

($mo, $day, $year) =
   ( $s =~ /(\d{1,2})\/(\d{1,2})\/(\d{2,4})/ );
print "Trick 2: \$mo=$mo, \$day=$day, \$year=$year.\n";

($wholedate,$mo, $day, $year) =
   ( $s =~ /((\d{1,2})\/(\d{1,2})\/(\d{2,4}))/ );
print "Trick 3: \$wholedate=$wholedate, \$mo=$mo, ",
   "\$day=$day, \$year=$year.\n";
```

Results of above:

```
$s=There is 1 date 10/25/95 in here somewhere.
Trick 1: $1=10, $2=25, $3=95,
        $`=There is 1 date  $'= in here somewhere.
Trick 2: $mo=10, $day=25, $year=95.
Trick 3: $wholedate=10/25/95, $mo=10, $day=25, $year=95.
```

Note that if patterns are matched in an array context as in Tricks 2 and 3, special variables$1, $2, ..., and $`, $', and $& are not set.

Using a combination of Tricks 1 and 3, we can write the following program which processes its input by replacing notations like 10/25/95 (where the month appears first) by ISO 8601 conformant notations like 1995-10-25. The program uses a conditional operator to add 1900 to the year if it was less than 100.

```
while(<>) {
  while( m/((\d{1,2})\/(\d{1,2})\/(\d{2,4}))/ ) {
    $year = $4 < 100 ? 1900+$4 : $4;
    $newdate = sprintf "%04d-%02d-%02d", $year, $2, $3;
    s/$1/$newdate/; }
  print; }
```

Matching is greedy

Consider the simple regular expression k.* which means 'the letter k followed by a sequence of any characters'. For a string containing k, there would be several possible matches if the language definition did not say how the matching is performed. In Perl, the definition is that the longest possible string is taken; in our example, that means that the expressions matches the substring which extends from the first occurrence of k to the end of the string.

Regular expressions are greedy, seeking the *longest* possible match not the shortest match.

This rule applies to matches involving the repetition specifier * or +. It does *not* apply to selecting between alternatives separated with | in a regular expression. If we have, say, k.*|zap then the *first* substring that matches k.* or zap is taken, so if zap occurs first, it does not matter that a match to k.* would give a longer match. But if a k is found before any zap is found, then matching to k.* is done the normal way, taking the largest among possible matches (i.e. taking the rest of the string).

In the following example we try to match whatever is between "<" and ">":

```
$s = 'Beware of <STRONG>greedy</strong> regular expressions.';
print "\$s=$s\n";
($m) = ( $s =~ /<(.*)>/ );
print "Try 1: \$m=$m\n";
($m) = ( $s =~ /<([^>]*)>/ );
print "Try 2: \$m=$m\n";
```

This results in:

```
$s=Beware of <STRONG>greedy</strong> regular expressions.
Try 1: $m=STRONG>greedy</strong
Try 2: $m=STRONG
```

Thus, by using a more specific match (which says that the string between "<" and ">" must not contain ">") we get the result we want. In Perl 5, it would also be possible to use *? instead of * to request the *shortest* match to be made. That woud mean using /<(.*?)>/ in our case. (This special meaning for the question mark in a specific context is rather strange, but useful.)

Putting it all together

Regular expressions have many practical uses. We'll look at a httpd log analyzer for an example.

(Complete source code.)

```
#!/usr/local/bin/perl

# We will use a command line argument to determine the log filename.
$logfile = $ARGV[0];
```

```
unless ($logfile) { die "Usage: a3-httpd.pl <httpd log file>"; }
analyze($logfile);
report();
sub analyze {
    my ($logfile) = @_;
    open (LOG, "$logfile") or die "Could not open log $logfile - $!";
    while ($line = <LOG>) {
        @fields = split(/\s/, $line);
        # Make /about/ and /about/index.html the same URL.
        $fields[6] =~ s{/$}{/index.html};
        # Log successful requests by file type.  URLs without an extension
        # are assumed to be text files.
        if ($fields[8] eq '200') {
          if ($fields[6] =~ /\.([a-z]+)$/i) {
              $type_requests{$1}++;
          } else {
              $type_requests{'txt'}++;
          }
        }
        # Log the hour of this request
        $fields[3] =~ /:(\d{2}):/;
        $hour_requests{$1}++;
        # Log the URL requested
        $url_requests{$fields[6]}++;
        # Log status code
        $status_requests{$fields[8]}++;
        # Log bytes, but only for results where byte count is non-zero
        if ($fields[9] ne "-") {
            $bytes += $fields[9];
        }
    }
    close LOG;
}
sub report {
    print "Total bytes requested: ", $bytes, "\n";
    print "\n";
    report_section("URL requests:", %url_requests);
    report_section("Status code results:", %status_requests);
    report_section("Requests by hour:", %hour_requests);
    report_section("Requests by file type:", %type_requests);
}
sub report_section {
    my ($header, %type) = @_;
    print $header, "\n";
    for $i (sort keys %type) {
        print $i, ": ", $type{$i}, "\n";
    }
    print "\n";
}
```

First, let's look at a sample line from a httpd log:

```
 127.12.20.59 - - [01/Nov/2000:00:00:37 -0500]
        "GET /gfx2/page/home.gif HTTP/1.1" 200 2285
```

The first thing we want to do is split this into fields. Remember that the split() function takes a regular expression as its first argument. We'll use /\s/ to split the line at each whitespace character:

```
@fields = split(/\s/, $line);
```

This gives us 10 fields. The ones we're concerned with are the fourth field (time and date of request), the seventh (the URL), and the ninth and 10th (HTTP status code and size in bytes of the server response).

First, we'd like to make sure that we turn any request for a URL that ends in a slash (like /about/) into a request for the index page from that directory (/about/index.html). We'll need to escape out the slashes so that Perl doesn't mistake them for terminators in our s/// statement.

```
$fields[6] =~ s/\/$/\/index.html/;
```

This line is difficult to read, because anytime we come across a literal slash character we need to escape it out. This problem is so common, it has acquired a name: *leaning-toothpick syndrome*. Here's a useful trick for avoiding the leaning-toothpick syndrome: You can replace the slashes that mark regular expressions and s/// statements with any other matching pair of characters, like { and }. This allows us to write a more legible regex where we don't need to escape out the slashes:

```
$fields[6] =~ s{/$}{/index.html};
```

(If you want to use this syntax with a matching expression, you'll need to put a m in front of it. /foo/ would be rewritten as m{foo}.)

Now, we'll assume that any URL request that returns a status code of 200 (request OK) is a request for the file type of the URL's extension (a request for /gfx/page/home.gif returns a GIF image). Any URL request without an extension returns a plain-text file. Remember that the period is a metacharacter, so we need to escape it out!

```
if ($fields[8] eq '200') {
        if ($fields[6] =~ /\.([a-z]+)$/i) {
            $type_requests{$1}++;
        } else {
            $type_requests{'txt'}++;
        }
    }
```

Next, we want to retrieve the *hour* each request took place. The hour is the first string in $fields[3] that will be two digits surrounded by colons, so all we need to do is look for that. Remember that Perl will stop when it finds the first match in a string:

```
# Log the hour of this request
    $fields[3] =~ /:(\d{2}):/;
    $hour_requests{$1}++;
```

Finally, let's rewrite our original report() sub. We're doing the same thing over and over (printing a section header and the contents of that section), so we'll break that out into a new sub. We'll call the new sub report_section():

```
sub report {
   print ``Total bytes requested: ", $bytes, ``\n";
print "\n";
   report_section("URL requests:", %url_requests);
   report_section("Status code results:", %status_requests);
   report_section("Requests by hour:", %hour_requests);
   report_section("Requests by file type:", %type_requests);
}
```

The new report_section() sub is very simple:

```
 sub report_section {
    my ($header, %type) = @_;
 print $header, "\n";
    for $i (sort keys %type) {
        print $i, ": ", $type{$i}, "\n";
    }
    print "\n";
}
```

We use the keys function to return a list of the keys in the %type hash, and the sort function to put it in alphabetic order. We'll play with sort a bit more in the next article.

Exercise: Parsing and Reporting

1. See preceding "Grade Book" example. Using the same stufile input, print a list of students ordered by family name, with any quoted nickname listed in place of the given name, and family name last. Produce output like this:

```
Student-ID  Year  Name
357913      JR    Thomas Jefferson
246802      SO    Abe Lincoln
212121      SO    Teddy Roosevelt
123456      SR    George Washington
```

10. Simple CGI

In your UNIX account root directory, create a sub directory, cgi-bin, run your CGI script under cgi-bin, and the url of your cgi script looks like this: www.marian.cs.nott.ac.uk/~smith/cgi-bin/mycgi.cgi, where smith is your user name.

For an introduction to Common Gateway Interface, see the *Introduction to the Common Gateway Interface (CGI)* in the Virtualville Library. For more thorough description, see *How the web works: HTTP and CGI explained* or *An instantaneous introduction to CGI scripts and HTML forms*. See also *The CGI Resource Index*, especially its section *Programming in Perl*. If you intend to do serious work with CGI scripts in Perl, you should read Pankaj Kamthan's *CGI Security : Better Safe than Sorry*.

If you know C, you may wish to take a look at *Getting Started with CGI Programming in C* for comparison, before or after studying how to write CGI scripts in Perl.
Example: Web Form
Here is an example of what a World Wide Web form may look like:

Enter your ID Number:
Enter your Name:

Select favorite Color: (no favorite)

Submit

The HTML source code for the form is the following:

```
<FORM METHOD="POST" ACTION=
 "http://yucca.hut.fi/cgi-bin/formdemo.pl">
<TABLE>
<TR><TD>Enter your ID Number:</TD>
<TD><INPUT NAME=idnum></TD></TR>
<TR><TD>Enter your Name:</TD>
<TD><INPUT NAME=name></TD></TR>
<TR><TD>Select favorite Color:</TD>
<TD><SELECT NAME=color>
<OPTION SELECTED>(no favorite)
<OPTION>red<OPTION>green<OPTION>blue
</SELECT></TD></TR>
</TABLE>
<INPUT TYPE=submit>
</FORM>
```

When something is entered and the submit button pressed, the resulting screen could be e.g. the following:

> **Results of Form**
>
> (From http://yucca.hut.fi/cgi-bin/formdemo.pl)
>
> Your ID Number is 12345, your name is Jukka Korpela, and your favorite color is green.
>
> ---
>
> [Try again]

To test the form, you could a follow a link which invokes a Perl program (as a server-side script) which *generates* the form when invoked with no input data and *processes* the data when some data is passed to it. This is by no means necessary in normal case; you might write the form into a normal ("static") HTML document using an editor, and you might use a server side script just for processing form submissions.

Perl Program to Generate and Process Form

Here is the Perl program that generates both the form and the response. It uses an external module called cgi-lib.pl (There is a more powerful module for the purpose, too: CGI.pm, available from CPAN. See *CGI Programming Made (Relatively) Easy Using Libraries*, which contains nice illustrations of basic use of CGI.pm.)

```
require "cgi-lib.pl";  # Get external subroutines

$script = $ENV{'SCRIPT_NAME'};
$webserver = $ENV{'SERVER_NAME'};

if (! &ReadParse(*input)) { print &PrintHeader; &showform }
else { print &PrintHeader; &readentries }
```

```
exit;

sub showform {
# If there is no input data, show the blank form

print <<EOF;
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<TITLE>Sample form</TITLE>
<FORM METHOD="POST" ACTION=
 "$script">
<TABLE>
<TR><TD>Enter your ID Number:</TD>
<TD><INPUT NAME=idnum></TD></TR>
<TR><TD>Enter your Name:</TD>
<TD><INPUT NAME=name></TD></TR>
<TR><TD>Select favorite Color:</TD>
<TD><SELECT NAME=color>
<OPTION SELECTED>(no favorite)
<OPTION>red<OPTION>green<OPTION>blue
</SELECT></TD></TR>
</TABLE>
<INPUT TYPE=submit>
</FORM>
EOF
} # End of sub showform #

sub readentries {
# Input data was detected.  Echo back to form user.

print <<EOF;
<HTML><HEAD>
<TITLE>Form Example, Part 2</TITLE>
</HEAD><BODY>
<H1>Results of Form</H1>
<P>(From http://$webserver$script)
<P>Your ID Number is $input{'idnum'}, your name is $input{'name'},
and your favorite color is $input{'color'}.
<HR>
[<A HREF="$script">Try again</A>]
EOF
} # end of sub readentries #
```

11. Testing Perl Programs

Use thecommand-line option compiler -w to warn about identifiers that are referenced only once, uninitialized scalars, predefined subroutines, undefined file handles, probable confusion of == and eq, and other things. On Unix, this can be coded in the first line:

```
#!/usr/local/bin/perl -w
```

where you need to replace the path by one that is applicable in your environment. (Cf. to section *Making it command-like on Unix*.) Section perlrun in the manual explains how to simulate #! on non-Unix systems.

As you write your program, put in print statements to display variables as you proceed.Comment them out using # when you feel you don't need to see their output.

CGI scripts require some special attention in testing. In addition to checking server-dependent things, make sure you

know where the problem is; you probably need to use a simple "echoing" script to see whether the problem is on the HTML document containing the form, or in a browser, or in your script. See als *The Idiot's Guide to Solving Perl CGI Problems*.

12. Common Goofs for Novices

Adapted from *Programming Perl*, page 361. For more traps, see See section perltrap in the manual.

1. Testing "all-at-once" instead of incrementally, either bottom-up or top-down.

   Optimistically skipping print scaffolding to dump values and show progress.
   Not running the interpreter with the -w switch.
   Leaving off $ or @ or % from the front of a variable.
   Forgetting the trailing semicolon.
   Forgetting curly braces around a block.
   Unbalanced () or {} or [] or "" or "`` or <>.
   Mixing apostrophes " with quotation marks "" or slash / with backslash \.
   Using == instead of eq, != instead of ne, = instead of ==, etc. ('White' == 'Black') and ($x = 5) evaluate as (0 == 0) and (5) and thus are true!
   Using else if instead of elsif.
   Not chomping the output of something in "backquotes" (like `date`) or not chomping input:

```
print "Enter y to proceed: ";
$ans = <STDIN>;
chomp $ans;
if ($ans eq 'y') { print "You said y\n";}
else { print "You did not say 'y'\n";}
```

   Putting a comma after the file handle in a print statement.
   Forgetting that in Perl indexes start at 0, not 1.
   Using $_, $1, or other side-effect variables, then modifying the code in a way that unknowingly affects or is affected by these.
   Forgetting that regular expressions are greedy.
