

**Studies
in Quantitative Linguistics**

7

Fan Fengxiang

**Quantitative Linguistic
Computing with Perl**

RAM - Verlag

**Quantitative Linguistic
Computing with Perl**

by

Fan Fengxiang

2010

RAM-Verlag

Studies in quantitative linguistics

Editors

Fengxiang Fan (fanfengxiang@yahoo.com)
Emmerich Kelih (emmerich.kelih@uni-graz.at)
Reinhard Köhler (koehler@uni-trier.de)
Ján Mačutek (jmacutek@yahoo.com)
Eric S. Wheeler (wheeler@ericwheeler.ca)

1. U. Strauss, F. Fan, G. Altmann, *Problems in quantitative linguistics 1*. 2008, VIII + 134 pp.
2. V. Altmann, G. Altmann, *Anleitung zu quantitativen Textanalysen. Methoden und Anwendungen*. 2008, IV+193 pp.
3. I.-I. Popescu, J. Mačutek, G. Altmann, *Aspects of word frequencies*. 2009, IV +198 pp.
4. R. Köhler, G. Altmann, *Problems in quantitative linguistics 2*. 2009, VII + 142 pp.
5. R. Köhler (ed.), *Issues in Quantitative Linguistics*. 2009, VI + 205 pp.
6. A. Tuzzi, I.-I. Popescu, G. Altmann, *Quantitative aspects of Italian texts*. 2010, IV+161 pp.
7. F. Fan, *Quantitative linguistic computing with Perl*. 2010, VIII + 205 pp.

© Copyright 2011 by RAM-Verlag, D-58515 Lüdenscheid

RAM-Verlag
Stüttinghauser Ringstr. 44
D-58515 Lüdenscheid
RAM-Verlag@t-online.de
<http://ram-verlag.de>

Preface

Empirical research in linguistics, in particular in quantitative linguistics, relies to a high degree on the acquisition of large amounts of appropriate data and, as a matter of course, on sometimes intricate computation. The last decades with the advent of faster and faster electronic machinery and at the same time growing storage capacities at falling prices contributed, together with advances in linguistic theory and analytic methods, to the availability of suitable linguistic material for all kinds of investigations on all levels of analysis.

Ideally, a researcher in quantitative linguistics has enough programming knowledge to acquire the data needed for his/her specific study. This is, however, not always the case. If professional programmers can be asked for help, most problems may be overcome; however, also this way is not always possible. And sometimes, it may be more awkward to explain a task than do perform it.

The selection of the appropriate programming language should not be conducted by taste or familiarity (as it is, unfortunately, very often even among programmers); instead, at least the following criteria should be taken into account:

1. Quality of the language. There is a number of quality-related properties of a programming language such as ability of preventing programming mistakes, readability of the code, changeability, testability, learnability. Unfortunately, these and other properties are not independent of each other; some of them compete (e.g. efficiency is always a competitor of most other properties) whereas others co-operate (e.g. readability advances most of the others). A programmer has to decide on priorities of the quality properties with respect to the individual task and application.
2. Nature of the problem. Every programming language has advantages and disadvantages also with respect to the task to be performed. One of the criteria often cited is the old distinction between low-level (close to the basic processor instructions and to the memory organisation of the computer) and high-level languages (with concepts close to the problems or algorithms). However, matters of efficiency etc. do not play a role any more (at least in the overwhelming majority of applications) since the compilers and their optimisers produce better code than most human programmers would be able to do. But there are still concepts and tasks that can be expressed in one language better than in another one, e.g. only few programmers would prefer a scripting language for the programming a data base.
3. Size and complexity of the problem. The larger a problem and the higher its complexity the more relevant become the quality properties of the language. If, e.g. several persons work on the same project, readability of the code is of fundamental importance but even if a single programmer

II

does all the coding of a complex problem he/she will encounter problems with his own code after some time if the programming language allows code in a less readable form. In any case, corrections, changes, and maintenance of a program depend crucially on some of the quality properties.

4. Security aspects. This is a simple matter: If the application you write is supposed to run in an environment that is accessible to potential attacks (e.g., the Internet or a computer network or if other users have access to the computer) and if the data your program works with should be protected, then a special focus should be put on security properties of the programming language. Scripting languages, e.g., are known to be frail, as a rule. You should make sure that the language you use has at least protection mechanisms you can switch on. Surprisingly, this aspect is very often neglected – even by institutions such as banks (Internet banking).
5. Reliability of the compiler/interpreter/libraries. Many popular languages are not *defined*. Instead, ‘reference implementations’ are offered. However, to be sure how a language element works you would have to try it out in every possible form of use and combination with other elements – an unrealistic idea. Another aspect is even more significant in practice: Some languages, among them some of the currently most popular ones, are subject to substantial changes every now and then. The user of such a language is witness and victim of a ripening process (or mere experimentation): If your program will work with the next version of the language, is more or less a matter of chance. You should consider how much harm such a situation would do to your project if you decide to use a language that is not defined.
6. Frequency of application. It matters whether your program is an ad-hoc solution and will be used just once or a few times to evaluate some data and then will be discarded or whether it is meant to be useful for a longer time and may be changed and adapted for varying conditions. In the first case, not so much value is to be set on quality properties of the language; immediate availability of a practical solution may then come into foreground. In the latter case, however, readability, changeability and other properties play a bigger role.
7. Intended users of the software product. Similarly, if you alone will use a program, some disadvantages such as missing robustness or a bad user interface would not constitute a serious problem as you will exactly know which behaviour you have to expect and how to circumvent inelegance or even mistakes. If, on the other hand, an unknown number of unknown persons will use it you should base your product on reliable tools, among them the language you formulate your solution in.

The main problem, however, quantitative linguists will have to face – independent of how they are inclined to weight the criteria discussed above – is probably

that they fail to have an overview about programming languages and their pro's and con's. Whenever a programming layman is asked for advice the probability is high that the answer will depend on personal taste and familiarity with a language and possibly on its current popularity. You should, at least, know what criteria to base your decision on; with an idea of your priorities at hand and after discussing them with a programming expert, you can increase the chance to obtain a good hint.

Perl belongs to the so-called scripting languages. To run a program written in Perl you need the Perl interpreter; it has to be installed on your computer before the program can be executed. The reason is that such a program is interpreted, line by line, each time it is started as opposed to compiled programs which can run without any interpreter. There are, again, pro's and con's of either solution. Scripting languages have, e.g. the advantage that a program can change its own logic and easily adapt its data structures while it runs, to an extent which is impossible with compiled programs – a comfortable but also potentially dangerous facility.

Linguists fancy, in particular, the powerful language elements of Perl, which enable a programmer to write powerful programs in very short time. This property is especially useful for string and text manipulation and analysis because many ready-made tools for string handling are 'innate' to the language. Advanced programmers will find it even more useful for Internet programming. A clear disadvantage is the not so readable program text which makes finding and correcting of mistakes sometimes awkward in long and complex programs. Therefore, careful formulations and exhaustive comments within the program code are strongly recommended. If these caveats are taken into account Perl can be used with much success with little effort – and make a lot of fun.

Reinhard Köhler

Table of Contents

Preface	I
1 Introduction	1
1.1 Quantitative linguistics and Perl.....	1
1.2 Characteristics of this book and its intended readers	2
1.3 Downloading and installation.....	3
1.4 Program editor	7
1.5 Conventions used in this book.....	8
2 Perl variables and operators	11
2.1 Perl variables	11
2.2 Value assignment to variables	11
2.3 Perl numeric operators and functions	13
2.3.1 Math operators.....	13
2.3.2 Math functions.....	16
2.3.3 Numeric comparison operators	19
2.4 String operators and string comparison operators	21
2.4.1 String operators	21
2.4.2 String comparison operators.....	22
2.5 The logical operator.....	23
Exercises	24
3 Input and output	26
3.1 Input at the command line	26
3.1.1 The use of @ARGV	26
3.1.2 The use of STDIN	27
3.1.3 Command line file input.....	28
3.2 Inputting files inside a program.....	29
3.3 Some string manipulation functions	31
3.4 Applications.....	34
Exercises	40
4 Regular expressions: basic structure	41
4.1 Operators for regular expressions.....	41
4.1.1 =~ and m//	41
4.1.2 s///	42
4.1.3 tr///	45
4.2 Regular expression quantifiers and other operators	49
4.2.1 The general quantifiers and wild card.....	49
4.2.2 The greediness of the quantifiers * and +	52
4.2.3 The alternative operator, anchors and the escape operator	53
4.3 Applications.....	55
4.3.1 Text tokenizer	55
4.3.2 Computing syllabic word length	56
4.3.3 Removal of HTML codes in texts	57
Exercises	60

5 Regular expressions: advanced topics	61
5.1 Metacharacters for regular expressions	61
5.2 Special variables	63
5.3 Back referencing	65
5.4 Quantifying expressions	66
5.5 String manipulation functions and the <i>for</i> program control structure	68
5.6 Applications	71
5.6.1 Extraction of POS tags	71
5.6.2 Making concordance for a text.....	72
5.6.3 Extraction of lexical bundles from texts.	73
5.6.4 A Chinese tokenizer.....	75
Exercises	77
6 Arrays	79
6.1 Array creation	79
6.1.1 One dimensional arrays.....	79
6.1.2 Multi-dimensional arrays	82
6.1.3 Converting texts into arrays	84
6.2 Functions for array operations.....	85
6.2.1 Functions for array input and output.....	85
6.2.2 Array insertion, truncation and deletion.....	88
6.2.3 Sorting an array	89
6.2.4 The anonymous variable and the <i>join</i> , <i>map</i> and <i>grep</i> functions... 91	
6.3 Combining identical array elements and random sampling from an array	94
6.4 Applications	97
6.4.1 Selecting words from a wordlist	97
6.4.2 Turning a text into bigrams	98
6.4.3 Turning a text into a list of word types with frequencies.....	100
6.4.4 Computing sentence length distribution.....	101
Exercises	102
7 Hash tables	103
7.1 Hash input and output.....	103
7.1.1 Manual input and output	103
7.1.2 Hash input and output using arrays and functions	106
7.1.3 The use of <i>values()</i> , <i>each()</i> , <i>exist()</i> and <i>delete()</i>	110
7.2 Hash operations	112
7.2.1 Converting hash elements into an array	112
7.2.2 Combining two or more hashes together	113
7.2.3 Hash comparisons	115
7.2.4 Computing value frequencies.....	117
7.3 Applications	118
7.3.1 Computing per word entropy of English.....	118
7.3.2 Making a word frequency spectrum.....	119

7.3.3	Lemmatization.....	120
7.3.4	Lexical comparison between two texts	122
	Exercises	124
8	Subroutines and modules	126
8.1	Subroutines	126
8.1.1	The basic structure	126
8.1.2	Parameters of subroutines	127
8.1.3	The use of <i>return()</i> in subroutines.....	129
8.1.4	Localization of variables in subroutines	130
8.2	Modules	131
8.3	References.....	135
8.3.1	Making references	135
8.3.2	Dereferencing for scalar variables and references	136
8.3.3	Dereferencing for arrays	137
8.3.4	Dereferencing for hashes.....	139
8.4	Use of references in subroutines and modules	140
8.5	Applications	142
8.5.1	Computing arc length.....	143
8.5.2	A module for removing non-alphanumeric characters.....	144
8.5.3	A lexical comparison program	145
	Exercises	149
9	Directory and file management	150
9.1	Directory management	150
9.2	File management.....	151
9.3	Formatting output files	154
9.3.1	Outputting data in the original format.....	154
9.3.2	Arranging data in left-justified columns	155
9.3.3	Arranging data in right-justified columns	156
9.3.4	Arranging data in centre-justified columns	157
9.3.5	Formatting data that has line breaks.....	159
9.3.6	Producing page heading and paginating output files	161
9.4	Applications	164
9.4.1	A page-formatting program	164
9.4.2	Computing vocabulary growth and number of hapax legomena.....	166
9.4.3	A program for computing word range.....	171
	Exercises	175
Appendix	Model answers to the exercises	177
	Exercises of Chapter 2.....	177
	Exercises of Chapter 3.....	177
	Exercises of Chapter 4.....	181
	Exercises of Chapter 5.....	183
	Exercises of Chapter 6.....	186
	Exercises of Chapter 7.....	188

VIII

Exercises of Chapter 8.....	192
Exercises of Chapter 9.....	195
Index	202

1 Introduction

1.1 Quantitative linguistics and Perl

Once the venerable quantitative linguist Professor Gabriel Altmann was asked by a physicist as to the line of his work; when he replied quantitative linguistics, the inquirer's observation was: Oh, you count letters. This remark is not far off the point. Quantitative linguists do count letters, for example, in the computation of letter graphemic load, letter phonemic load, letter frequency, letter utility and so on. Apart from letter counting, quantitative linguists practically count anything else in language, from phoneme, morpheme, syllable, word, phrases and so on up to clause and sentence, measuring their length, the possible senses they signify etc. The results of the counting are used to develop mathematic models or laws describing the distributions of these components of language or their interrelationships, or test such models, laws or interrelationships, characterizing a text, language etc. Take the computation of Yule's K as an example, which can be used as a measure of vocabulary richness and of author identification. It's computed as follows:

$$K = 10000 \frac{\sum_m m^2 V(m, N) - N}{N^2}$$

where m is the word frequency class, $V(m, N)$ the number of words with frequency m , and N the number of words in the text in question. Suppose we want to compute K for Lewis Carroll's *Alice's Adventures in Wonderland*, which has 27,285 word tokens and 2,570 word types. It would take quite a long while to get K manually, and the process would be very tedious and error prone. Imagine doing this by hand to a longer text, e.g., Dickens' 184,282-word *Great Expectations*!

In this day and age, apart from a pencil, an eraser and a wad of paper, quantitative linguists also have a powerful weapon under their belt: the computer. There are several powerful computer software packages that can be used for language studies, e.g., *WordSmith*, *Tact*, *OCP*, *Lexa*, *Antconc*, and so on. However, software packages like the above were designed for specific tasks and can't fully satisfy the needs of quantitative linguists. Possibly a linguist would have difficulty finding a software package to compute Yule's K for a million-word text automatically at one sitting, to say nothing of the many ad hoc data- and computation-intensive research inspirations popping out just off the top of the head of a linguist. In such cases, a programming language is needed.

There are many languages that can be used for linguistic purposes. For example, VB, C, C++, SNOBOL4/SPITBOL, PYTHON, JAVA, ICON, the computer language attached to Foxpro, Perl and so on. Of these, Perl is a very

good candidate. Perl, short for Practical Extraction and Report Language, was created by Larry Wall in the mid-1980s for string manipulation and text processing purposes. It has the following advantages.

Perl is free and easy to obtain; it can be downloaded from many websites all over the world. It's also easy to install; after downloading, it's fully functional with only a few mouse clicks.

Perl is trans-platform, that is, it can be used under nearly all the major computer operating systems, such as Macintosh, VMS, OS/2, MS/DOS, Unix, Linux, Windows and so on. Programs written in Perl can be run under all of the above systems with practically no change.

Perl is very versatile and powerful. Perl programs for language processing purposes are usually short, hence are easy to write and quick to run. One of the most useful functions of Perl is its powerful regular expressions, which greatly simplify complicated pattern matching in large texts or mega-corpora. In addition to string manipulation and text processing, Perl is also very good for number crunching, that is, it can be used for math operations with efficiency.

Everything has two sides. Despite Perl's so many advantages, it has one downside. Perl programs written by other people are not easy to read. However, this won't be of much trouble to the reader of this book since many of the programs have notes or explanations to them.

The following is a Perl program for extracting words with 4 or more consecutive vowels from a 95,132-word wordlist of a 10,000,000-word sample from the BNC (the British National Corpus) and storing the result to a file called *vcluster.txt*.

```
open(F,"bncwordlist.txt");
open(W,">vcluster.txt");
while($word=<F>){
  print (W $word) if($word=~m/[aeiou]{4,}.*);
}
```

This program has only five lines but can get all the target words in a file in a fraction of a second. If it were written in other languages, it would be much longer.

1.2 Characteristics of this book and its intended readers

This book is mainly on Perl programming for quantitative linguistics; other applications, such as CGI (Common Gateway Interface) programming for handling internet web pages, graphics etc, are not covered in this book. But this book is a good stepping stone to learning these functions with other Perl books that introduce these functions. The programs in the book were all written by the authors and computer-tested, and the majority of them are immediately useful for

serious research, after changing only the input and output file names and their path. This book can be used as a course book that takes roughly 36-lab hours to complete; it can also be used for self-study. The intended readership is broad-spectrum: students, teachers, researchers and other people related to or interested in language studies, natural language processing, literature, language teaching, information retrieval and so on; no previous computer programming experience is required for this book.

The learner of this book is not expected to become a professional programmer who can write large commercial software packages immediately after learning Perl using this book. But those who read this book carefully from cover to cover and do all the exercises in it will be able to use Perl as a handy tool in their everyday routine work or research, so that they won't often hassle their computer programmer friend for help, in the worse case calling him or her in the middle of the night for a little program when a brilliant research idea suddenly pops up but can't be tested with just a wad of paper, an eraser and a pencil. Take a student in literature as an example, if he or she's doing a term project writing a paper on word length distribution of *Great Expectations* by Dickens, with a very short Perl script, data can be collected in a fraction of a second and the remaining task is writing up on these data.

There are many Perl communities on the Internet, one of the well-known is CPAN (<http://www.cpan.org>), where one can ask for help when problems arise concerning Perl and no solutions are offered in the book, or where we can download Perl programs or modules. For detailed documentation on Perl, go to <http://www.perldoc.com>. Perl can process any language in the world; however, in this book, Perl is used mainly to deal with English, occasionally Chinese. With some changes, the programs in the book can also be adapted to process other languages.

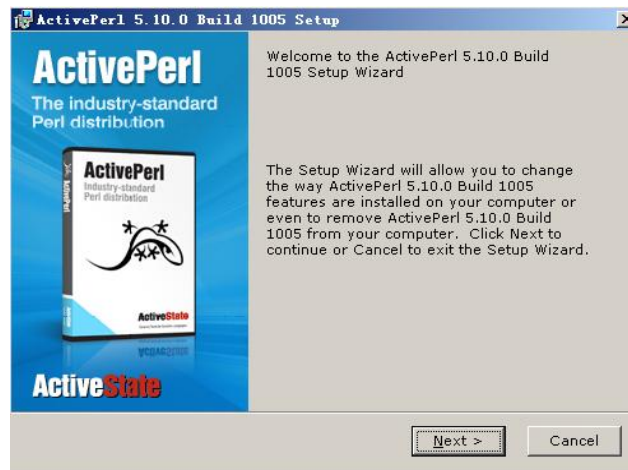
1.3 Downloading and installation

The Perl used in this book is Perl 5.10. It can be downloaded from the following website:

<http://www.activestate.com/activeperl>

For the Windows operating system, select *Windows (X86)*; for other systems, select *Other Systems and Versions*. Click *Windows (X86)* to start downloading and select a drive and folder on your computer for the downloaded file. After the downloading completes, go to the drive and folder where the downloaded file is stored (the downloaded file is *ActivePerl-5.10*) and double click it. The setup starts. It goes through the following seven steps:

(1) Start the Wizard.



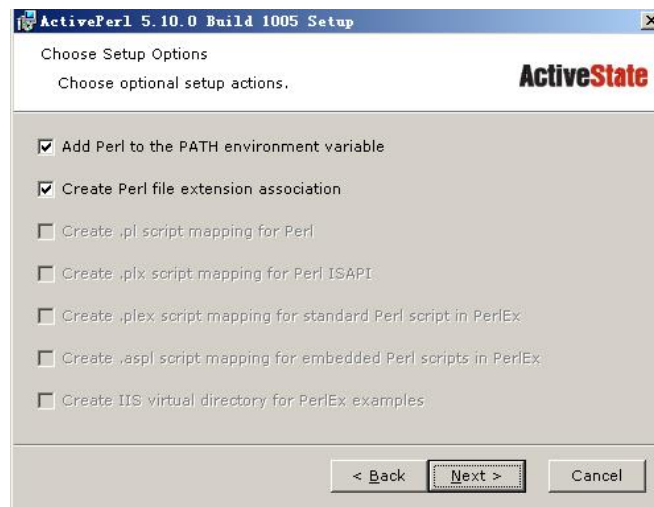
(2) Accept the terms in the License Agreement.



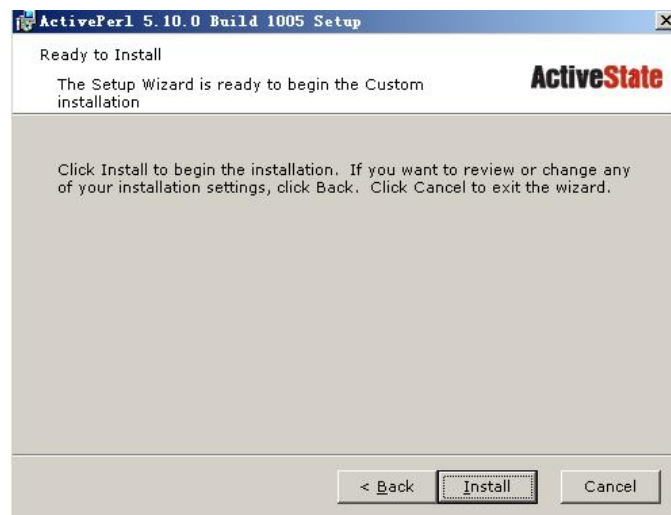
(3) Select drive and folder (we suggest putting Perl in *D:\Perl*. Click *Browse* for drive and folder selection).



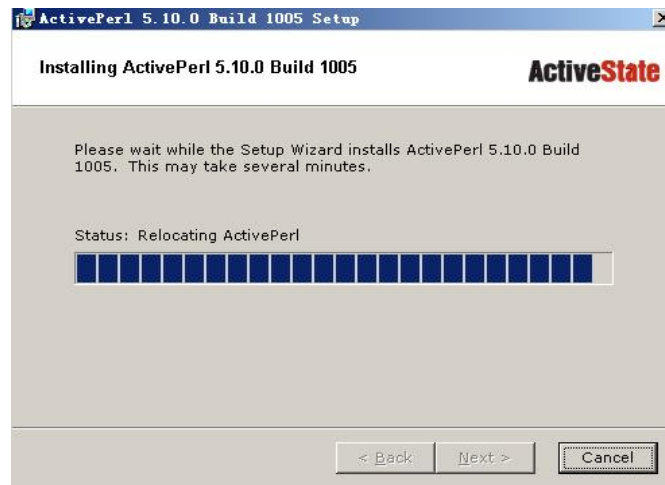
(4) Choose Setup Options.



(5) Start the custom installation.



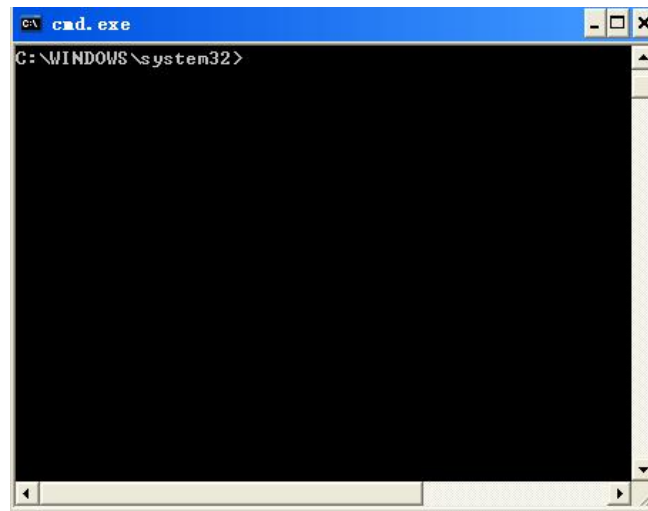
(6) Install Perl.



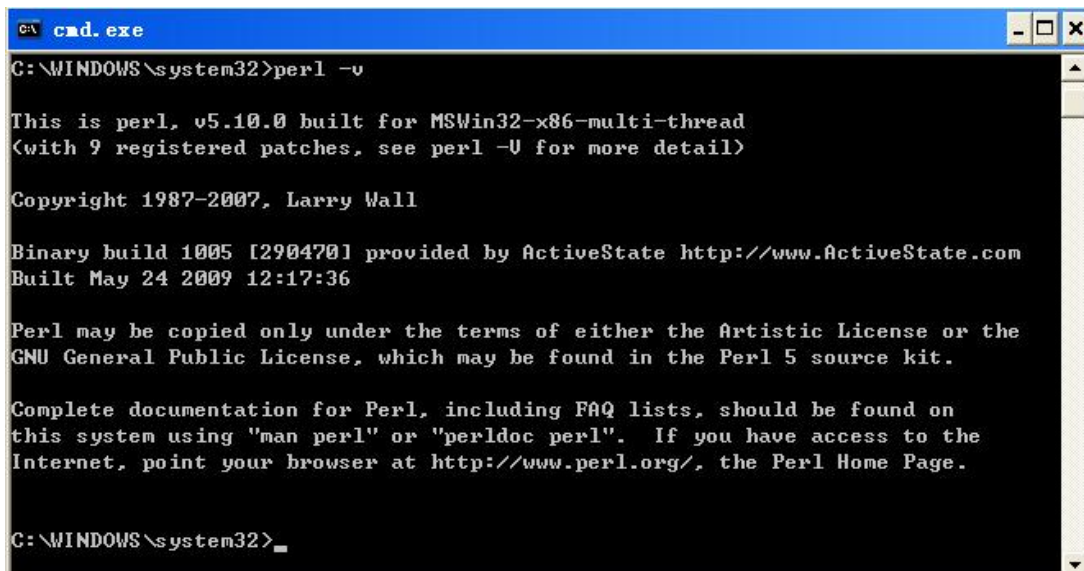
(7) Finish installation.



To check whether Perl has been successfully installed, run *cmd.exe* to get to the MS/DOS environment. *cmd.exe* is located in *C:\WINDOWS\system32*. Once in the directory, point the mouse to the file and then right-click the mouse, a pop-up menu appears. Select *create shortcut* and then press the left button and pull the newly created shortcut icon to desktop. Double click the *cmd.exe* shortcut icon to get to the MS/DOS environment, as shown below:



Enter *Perl -v* at the cursor and press *Enter*, the following should appear:



This means Perl has been successfully installed in your computer and is ready to function.

1.4 Program editor

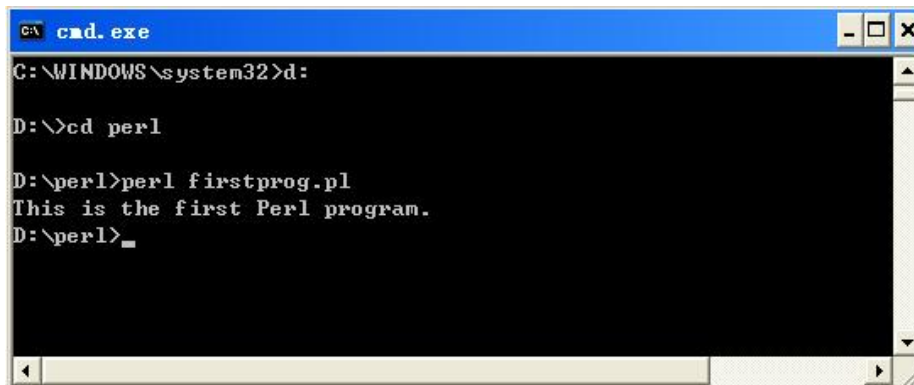
Perl programs can be written with any text editor. We recommend Windows *Wordpad* or *Notepad* since they are attached to Windows and are easy to handle. Whatever text editor is used, a Perl program must be saved as text only, with *pl* as the file extension. From now on we assume *Wordpad* is used as the Perl program editor. Create a short cut for it and drag it to Windows desktop. Start it and type the following:

```
print "This is the first Perl program. ";
```

Save it as *firstprog.pl* in *d:\perl*. To run a Perl program, start *cmd.exe*, go to the directory where the program is stored and enter *Perl* followed by the program name. Now type the following at the command line after *cmd.exe* is started, ↵ means *press Enter*:

```
d: ↵  
cd \perl ↵  
perl firstprog.pl ↵
```

The result is shown below:



```
cmd.exe  
C:\WINDOWS\system32>d:  
D:\>cd perl  
D:\perl>perl firstprog.pl  
This is the first Perl program.  
D:\perl>
```

1.5 Conventions used in this book

In explaining Perl commands and functions, this book uses the following conventions:

(1) The commands and functions are written in bold except the brackets. The user-specified components of a command or a function are written in plain italics. For example, in the function **open**(*filehandle*,"*filename*"), which is for opening a text file, *filehandle* and *filename* are specified by the user. *filehandle* can be any of the 26 English letters or a cluster of such letters, while *filename* can be any text files to be processed. If we name *filehandle* *W* and the text to be processed is *adventure.txt*, this function is written as **open**(*W*, "*adventure.txt*").

(2) For ease of explanation, statement numbers are added to programs that have more than four statements as shown below:

1. **open**(*F*,"*bncwordlist.txt*");
2. **open**(*W*,">*vcluster.txt*");
3. **while**(*\$word*=<*F*>){
4. **print** (*W* *\$word*) **if**(*\$word*=~**m**/[*aeiou*]{4,}.***/**);
5. **}**

When writing programs, which are also called scripts, don't use these numbers, or the program won't run. Perl does not allow numbers put before a statement.

(3) Commands to be entered at the DOS command line are in italics; the symbol ↵ means *press Enter*.

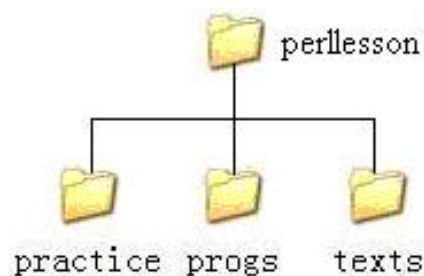
(4) The optional part of a Perl command or function is enclosed between a pair of bold square brackets [and]. For example, the **if** program control structure has the following form:

```
if(condition){
  statements to be carried out if condition is true.
[]}else{
  statements to be carried out if condition is not true.
}
}
```

The elements between the bold square brackets are optional.

Perl provides the character # that makes the computer ignore whatever follows it. It can be used to put notes within a program or make a statement inactive.

All the programs, model answers to the exercises and data used in this book are in the CD of this book, which has the following structure:



The contents of the folders are as follows:

practice: for holding programs and data created by the reader during practice, currently empty.

progs: containing all the Perl programs in this book, including the model programs for exercises at the end of each chapter.

texts: containing Lewis Carroll's *Alice's Adventures in Wonderland* (*adventure.txt*), *Through the Looking-glass* (*lookingglass.txt*), 20 text chunks from *adventure.txt* (*alice1.txt* to *alice20.txt*), a short passage in Chinese (*chinese.txt*), and other text files (including wordlists of BNC samples) etc.

Put the entire *perllesson* folder to a drive on your computer, say, *d* and make *d:\perllesson\practice* your default folder, from where you run your Perl programs and store the results. Copy all the files in the *texts* folder to the *practice*

folder. Each time you start a Perl session, type on the DOS command line the following:

```
d: ↵  
cd perllesson\practice ↵
```

To run a Perl program here, type *perl* followed by the program name, with the file extension *pl*, and the program will start to run.

2 Perl variables and operators

2.1 Perl variables

A variable is a temporary storage that stores whatever it is given. There are three types of variables in Perl:

- scalar variables,
- array variables,
- hash variables.

The latter two are simply called arrays and hashes. In this chapter we'll focus on scalar variables, which are referred to just as variables. The name of a variable can be a single or a cluster of alphanumeric characters and the underscore character `_`, with the symbol `$` placed at the initial position. Arabic numerals can be used with these characters in variable names as long as they are not placed immediately after `$`.

The following are valid variable names: `$v`, `$c_34`, `$counter`, `$word`, `$line_1`, `$word_list`, `$text_b`, `$get_sent_length`, etc.

Punctuation marks and characters such as `*`, `-`, `+`, `=`, `(`, `)`, `[`, `]`, `{`, `}`, `%`, `@`, `&`, `^`, `~`, `\`, `/`, `|`, `>`, `<`, `%`, `#`, etc, and Arabic numerals used alone or placed immediately after `$`, are not allowed in variable names. The following are invalid variable names: `$1`, `$284`, `$6v2`, `$@`, `$*w`, `$b-44&text`, `$get- word-length`, etc.

Perl variables are case sensitive; `$word` is not the same as `$Word` or `$WORD`. The maximum length of a variable name is 255 characters. However, in practical programming, nobody would use such a long variable. Generally, when writing a program, we should keep variable names as short as possible, but they should be suggestive. For example, if a variable stores word frequencies, names such as `$word_freq`, `$word_frequency` or `$wordfrequency` etc are better than `$a`, `$bb`, `$wf` and so on.

2.2 Value assignment to variables

Perl variables can be assigned any string or numeric values with the assignment operator `=`. String values should be enclosed either between a pair of single quotes or double quotes, but this is not necessary for numeric values. A complete Perl statement must end with a semicolon. To display the result of a Perl statement on the screen, use the Perl function **print**. Now start *Windows Wordpad* or any another text editor and enter the following:

```
$words = "This is an example of value assignment.";
print $words;
$number=22.34;
print $number;
```

Save this short script as *test.pl* and run it by typing the following on the MS DOS command line:

```
perl test.pl ↵
```

The following is displayed on the screen:

```
This is an example of value assignment. 22.34
```

From now on, demonstration Perl scripts like the above one are assumed to be saved as *test.pl* in *d:\perllesson\practice* and then run by typing *perl test.pl* on the DOS command line; the results of these statements are given where necessary right below these statements in italics. If you wish to keep these demonstration scripts, give them different names so that they won't be overwritten.

To put the two results of the above script on separate lines, Perl's line breaking character `\n` should be used; however, `\n` can function only within a pair of double quotes. Now modify *test.pl* as follows:

```
$words = "This is an example of value assignment.";
print "$words\n";
$number=22.34;
print "$number\n";
This is an example of value assignment.
22.34
```

This time the two results are placed in two separate lines.

The following is allowed in Perl:

```
$number1=$number2=$number3=5;
print "$number1\n";
$word1=$word2=$word3="Perl";
print "$word1\n";
5
Perl
```

To assign to a variable strings that have quotation marks in them, the following methods are used.

a. use of the escape character `\`:

```
$words="\Perl is very useful.\" She said.";
print $words;
"Perl is very useful." She said.
```

b. use of `qq(string)`:

```
$words=qq("Perl is very useful. " She said.);
print $words;
"Perl is very useful." She said.
```

c. use of `q(string)`:

```
$words=q("Perl is very useful." She said.);
print $words;
"Perl is very useful." She said.
```

The difference between *b* and *c* is that `qq` stands for double quote and `q` a single one. Remember the line breaking character `\n` that can function only within a pair of double quotes? Now do the following:

a.

```
$words=qq("Perl is very useful." \nShe said.);
print $words;
"Perl is very useful."
She said.
```

b.

```
$words=q("Perl is very useful." \nShe said.);
print $words;
"Perl is very useful." \nShe said.
```

2.3 Perl numeric operators and functions

2.3.1 Math operators

The following are math operators used in Perl.

- + Addition.
- ++ Add one.
- Subtraction.
- Subtract one.
- * Multiplication.
- / Division.
- ** Exponentiation.
- % Modulo.

Now do the following:

```
$a=4*5;
```



```
print $a;  
20
```

```
$a=(34+45-20)*3;  
print $a;  
177
```

```
$a=4457/226;  
print $a;  
19. 7212389380531
```

```
$a=32**(1/4);  
print $a;  
2. 37848423000544
```

```
$a=(2**3)**2;  
print $a;  
64
```

Unlike some other computer languages, Perl does a series of exponential computation from the right:

```
$a=2**3**2  
print $a;  
512
```

```
$a= 13%10;  
print $a;  
3
```

```
$a=6;  
$a++;  
print $a;  
7
```

```
$a=10;  
$a--;  
print $a;  
9
```

```
$a=20;  
$a=$a+2;  
print $a;
```

22

```
$a=20;  
$a=$a-2;  
print $a;  
18
```

```
$a=20;  
$a=$a*2;  
print $a;  
40
```

```
$a=20;  
$a=$a/2;  
print $a;  
10
```

Look at the following:

```
$a=4;  
$a+=20;  
print $a;  
24
```

In the above, `$a+=20` is the same as `$a=$a+20`. Now do the following:

```
$a=4;  
$a-=20;  
print $a;  
-16
```

```
$a=4;  
$a*=20;  
print $a;  
80
```

```
$a=4;  
$a/=20;  
print $a;  
0.2
```

```
$a=4;  
$a**=20;
```

```
print $a;
1099511627776
```

For extremely large integers, Perl automatically converts them into floating point values:

```
$bignumber=4294967295**4;
print $bignumber;
3. 40282366604026e+038
```

In case the actual integer is needed, however large it is, the **Math::BigInt** package in Perl can be invoked by putting *use bigint* in the program:

```
use bigint;
$bignumber=4294967295**4;
print $bignumber;
340282366604025813516997721482669850625
```

2.3.2 Math functions

Perl has the following math functions:

sqrt(n) The square root of n .
int(n) Convert n to integer.
log(n) The natural logarithm n (to the base e).
exp(n) Raising e to the n^{th} power.
rand(n) Producing a random number between 0 and n .
abs(n) Absolute value of n .

Now compute the square root of 334:

```
$a=sqrt(334);
print $a;
18. 2756668824971
```

To get to the k^{th} root of n , use the following:

```
$n**(1/k)
```

For example, $\sqrt[5]{32}$ can be expressed in Perl as $32^{1/5}$.

```
$a= 32**(1/5);
```

```
print $a;  
2
```

```
$a=int(3.665);  
print $a;  
3
```

```
$a=-445.6;  
print abs($a);  
445.6
```

```
$a=log(2.71828);  
print $a;  
0.999999327347282
```

To get the logarithm of n to the base 2 and base 10, use the following:

log(n)/log(2) the logarithm to the base 2.

log(n)/log(10) the logarithm to the base 10.

For example:

```
$a= log(4)/log(2);  
print $a;  
2
```

```
$a= log(100)/log(10);  
print $a;  
2
```

```
$a=exp(2);  
print $a;  
7.38905609893065
```

```
$a=rand(10);  
print $a;  
0.857646487375
```

To reduce decimal places, we can use the following:

sprintf "%.nf", n

```
$a=sprintf "%.4f", 3.1415926;
```

```
print $a;
3.1416
```

```
$a=sprintf "%.0f",3.1415926;
print $a;
3
```

atan2(n,x) Returning in radians the arc tangent of n/x in the range $-\pi$ to π .

cos(n) returning in radians the cosine of n .

sin(n) returning in radians the sine of n .

Now do the following:

```
$a=atan2(1,1);
print $a;
0.785398163397448
```

```
$a=cos(3.1415926);
print $a;
-0.9999999999999999
```

```
$a=sin(30);
print $a;
-0.988031624092862
```

More trigonometry functions are provided in Perl's **Math::Trig** package. For example, the following gets the tangent of 1 by invoking the **Math::Trig** package:

```
use Math::Trig;
print tan(1);
1.5574077246549
```

Next, we'll use some of the math operators and functions to write a program to solve the following problem (keep 6 decimal places):

$$\left[\sqrt[5]{23.412 \times \frac{4453.78}{9221}} \div \sqrt[4]{3321 - (12 + 31)^3} \right] \div \log_2 23867$$

The program is as follows:

```
math1.pl
$mmath_result=sprintf "%.6f",(((23.412*(4453.78/9221))**(1/5))/
```

```
(33210000-(12+31)**3)**(1/4)/(log(23867)/log(2));
print $math_result;
0.001472
```

This program has three lines but only two statements. Since the first statement is too long, we can break it into two lines and Perl permits this way of writing programs since a statement ends with a semicolon.

2.3.3 Numeric comparison operators

The following are operators used for numeric comparison:

```
== Equal.
!= Not equal.
< Less than.
> Greater than.
<= Less than or equal to.
>= Greater than or equal to.
```

Before practicing using these operators, let's learn the *if* structure control in Perl. It's in the following form (Note that the part between the pair of bold square brackets is optional):

```
if (condition){
  statements to be carried out if condition is true.
[else{
  Statements to be carried out if condition is not true.
}]
}
```

There is another way to use the *if* structure where more than two *conditions* exist:

```
if (condition1){
  statements to be carried out if condition1 is true.
[elsif(condition2){
  Statements to be carried out if condition2 is true.
}elsif(condition3){
  Statements to be carried out if condition3 is true.
}elsif(condition4){
  Statements to be carried out if condition4 is true.
}elsif(condition5){
  Statements to be carried out if condition5 is true.
}
```

```
.....  
}else{  
  Statements to be carried out if all the above conditions are not true.  
}  
]  
}
```

Now do the following:

```
$a=4;  
$b=6;  
if($a>$b){  
  print "$a is greater than $b!\n";  
}else{  
  print "$a is not greater than $b.\n";  
}  
4 is not greater than 6.
```

```
$a=14;  
$b=14;  
if($a!=$b){  
  print "$a is not equal to $b!\n";  
}else{  
  print "$a is equal to $b.\n";  
}  
14 is equal to 14.
```

```
$a=2239;  
$b=2239;  
if($a==$b){  
  print "$a is equal to $b.\n";  
}else{  
  print "$a is no equal to $b!\n";  
}  
2239 is equal to 2239.
```

Pay attention to the double equal signs used in *if(condition)*. Here if only a single equal sign is used, error may result. Do the following:

```
$a=2239;  
$b=44;  
if($a=$b){  
  print "2239 is equal to 44.\n";  
}else{
```

```
print "2239 is not equal to 44!\n";
}
2239 is equal to 44.
```

The result is wrong; this was caused by the use of a single equal sign in *if(condition)*. This is because in Perl, as well as in several other programming languages such as C, C++, Java etc, one uses = as a symbol for assignment, not as an equal sign.

The following example shows the use of the *if...elsif* structure.

```
$a=120;
$b=150;
$c=160;
if ($a>$b){
print ("a is larger than b.\n");
}elsif($b>$c){
print ("b is larger than c.\n");
}else{
print ("Neither a nor b is larger than c.\n");
}
Neither 120 nor 150 is larger than 160.
```

If(condition) has a simplified form as shown below :

statement to be carried out **if**(*condition*)

```
$a=34;
$b=50;
print "$a is less than $b.\n" if($a<$b);
34 is less than 50.
```

2.4 String operators and string comparison operators

2.4.1 String operators

Perl has the following string operators:

- x Repetition.
- . Concatenation.
- .. Range operator

Now do the following:


```

$letter="t" x 10;
print $letter;
tttttttt
$string1="This is";
$space=" ";
$string2="string concatenation.";
$string3=$string1.$space.$string2;
print $string3;
This is string concatenation.

```

We can also use `.=` for string concatenation:

```

$string1="conca";
$string2="tenation";
$string1.= $string2;
print $string1;
concatenation

```

Here `$string1.= $string2` is the same as `$string1= $string1.$string2`.

In the following, the range operator assigns the small case alphabet to `@letters` (`@letters` is an array, which will be dealt with in Chapter 6):

```

@letters=a..z;
print @letters;
abcdefghijklmnopqrstuvwxyz
@digits=3..20;
print @digits;
34567891011121214151617181920

```

2.4.2 String comparison operators

The following are operators for string comparison:

lt Less than.
gt Greater than.
eq Equal to.
le Less than or equal to.
ge Greater than or equal to.
ne Not equal to.

```

$letter1="a";
$letter2="c";

```

```

if ($letter1 gt $letter2){
print "$letter1 is greater than $letter2!\n";
}else{
print "$letter1 is not greater than $letter2.\n";
}
a is not greater than c.

```

```

$word1="Perl";
$word2="perl";
if ($word1 ne $word){
print "$word1 is not equal to $word2.\n";
}else{
print "$word1 is equal to $word2.\n";
}
Perl is not equal to perl.

```

Next, we'll write a short program to simulate coin casting. If a fair coin is thrown up into the air, the possibility of getting heads or tails is 0.5.

```

castcoin.pl
1. $message="Now let's flip a coin. H: Heads, T: Tails.\n";
2. print $message;
3. $randomnumber=rand(1);
4. if($randomnumber<0.51){
5. $cast="T";
6. }else{
7. $cast="H";
8. }
9. print "Your cast is: $cast.";

```

In statement 3 *rand(1)* generates a random number between 0 and 1, which is assigned to *\$randomnumber*. In statements 4—7, if *\$randomnumber* is less than 0.51, it's tails, otherwise it's heads. Statement 9 prints out the results.

2.5 The logical operator

The following are the logical operators:

- and** Expressing the AND relationship; some times **&&** is used.
- or** Expressing the OR relationship; some times **||** is used.
- not** Expressing the NOT relationship; some times **!** is used.

Now do the following:

```
$a=12;
$b=10;
$d=4;
$c=6;
if($c<$a and $b and $c >$d){
print "$c is less than $a and $b but greater than $d.\n";
}else{
print "$c is greater than $a and $b but less than $d.\n";
}
6 is less than 12 and 10 but greater than 4.
```

```
$a=12;
$b=10;
$d=4;
$c=6;
if($b<$a or $c <$d){
print "$b is less than $a.\n";
}else{
print "$b is greater than $a.\n";
}
10 is less than 12.
```

```
$a=12;
$b=10;
if(not $a>$b){
print "$a is not greater than $b.\n";
}else{
print "$a is greater than $b.\n";
}
12 is greater than 10.
```

In all the above, AND, OR and NOT can all be respectively replaced with **&&**, **||** and **!**.

Exercises

1. Assign the following two sentences to two variables respectively, then concatenate the two variables and output them on the screen.
 - a. "I'm learning Perl."
 - b. Sally said to her friend.

2. The following was proposed by Altmann:

$$CN = bL^{-a}$$

where CN is the number of compounds, L the word length measured in syllables, and b, a are parameters. If $a = 2.3212$, $b = 30.2693$, check the fit of the above relationship to the following empirical data:

Word syllable length	Observed mean number of compounds
1	30.29
2	5.86
3	2.01

3. Tuldava proposes that the relationship between vocabulary size V and the length of text is $V = Ne^{-\alpha(\ln N)^\beta}$; if $\alpha = 0.009152$, $\beta = 2.3057$, $N = 1000000$, compute V .

4. Popescu, Mačutek and Altmann explored the possibility of using the arc length of rank-frequency distributions in text characterization and language typology. The arc length of rank-frequency distribution L is expressed as follows:

$$L = \sum_{r=1}^{V-1} \{[f(r) - f(r+1)]^2 + 1\}^{1/2},$$

where V = vocabulary size of a text; r = rank of word frequency, with the highest frequency being $r = 1$; $f(r)$ = word frequency at rank r . Write a program called *arclength.prg* to compute the arc length of the following imagined word rank-frequencies ($V = 3$):

Rank	Frequency
1	1635
2	872
3	825
4	730

5. Compute the following:

$$100938.3 \times 2248^3 - 7754 \div 5.56 + \sqrt[4]{\frac{3400}{2578}} \times (1102 - 331)^{(12-8)}.$$

3 Input and output

In the preceding chapter, we learned Perl variables and operators and practiced using them in test programs. The data used for these programs were all contained within the practice programs, and the results were displayed only on the screen. Apart from that, data or texts to be processed can also be inputted outside the program. Very short data can be entered on the DOS command line and can be either displayed on the screen or stored in a file that can be accessed again. For large data sources such as the entire 100-million-word BNC, Perl can also use external files as the data source and store results in a re-accessible file. In this chapter we'll look at how these are achieved.

3.1 Input at the command line

3.1.1 The use of @ARGV

Perl has a special in-built array @ARGV that keeps records of whatever is entered at the DOS command line, except for the command to run a program, such as *perl test.pl*. @ARGV must be in upper case. An array can be thought of as a table that has individual cells numbered starting from 0. That is, the first cell is numbered by convention 0, the second 1, the third 2 etc. Values are stored in these numbered cells and can be accessed by referring to the number of the cells. In Perl arrays, these cells are called array elements. An array element is expressed as follows:

\$arrayname[element number]

For example, the first element of @ARGV is expressed as \$ARGV[0]. Now enter the following in *Wordpad* or whatever text editor you use:

```
print("$ARGV[0]\n");
print("$ARGV[1]\n");
print("$ARGV[2]\n");
print("$ARGV[3]\n");
print("$ARGV[4]\n");
print("$ARGV[5]\n");
print("$ARGV[6]\n");
```

Again save it as *test.pl* and enter the following on the DOS command line:

```
perl test.pl This is input at the command line. ↵
This
```

*is
input
at
the
command
line.*

Next we'll compute $(10*2)/4$ using the command line input:

```
$result=$ARGV[0]*$ARGV[1]/$ARGV[2];  
print $result;  
perl test.pl 10 2 4 ↵  
5
```

Note that $\$ARGV[0]*\$ARGV[1]/\$ARGV[2]$ must be assigned to a variable, or error will result.

3.1.2 The use of STDIN

STDIN is used to capture the keyboard input. It's often used for interactive programming and must be enclosed between a pair of pointed brackets < and >. Look at the following program:

```
interactive.pl  
1. print("Please enter a number:");  
2. $number1=<STDIN>;  
3. chomp $number1;  
4. print("The number is $number1. Please enter another number:\n");  
5. $number2=<STDIN>;  
6. chomp $number2;  
7. print("The second number is $number2. The result of $number1 +  
   $number2 is:\n");  
8. print $number1+$number2;
```

The program asks for two numbers, adds them together and outputs the result. Note the use of the Perl character function **chomp** in statements 3 and 6. It's a function for removing carriage returns. When a number is given and *Enter* is pressed, STDIN records the number as well as the *Enter* key. Try putting # before these two statements and see what happens.

3.1.3 Command line file input

Now we'll look at how to specify a file as the source of input at the command line. Before doing that, we'll first learn the open file function and the *while...* program control structure.

open(*FILEHANDLE*, *\$ARGV[0]*) [**or die** ("*expression\n*")]

This function is put inside a program, usually at the beginning. *FILEHANDLE* is given by the user; it represents the name of the input file. *FILEHANDLE* can be either a single letter or several letters. The file given at the command line as the first argument is captured by *\$ARGV[0]*, which is then processed by the program. The contents within the bold square brackets are optional, telling the program to stop in case the file can't be opened. **die** is a Perl command that stops a program from where it's issued. *expression* can be a message such as *file can't be opened* etc.

At the end of a program, files opened must be closed. The function to do this is as follows:

close(*FILEHANDLE*) This closes the file represented by *FILEHANDLE*.

The **while** program control structure tells the computer to execute the statements between the pair of curly brackets as long as *condition* holds true:

```
while(condition){
  statements to be carried out while condition holds.
}
```

The following is a short program taking a file as input specified at the command line and output it line by line to the screen.

```
printpoem1.pl
1. open(F,$ARGV[0]) or die("File does not exist!\n");
2. while($line=<F>){
3. print($line);
4. }
5. close(F);
```

Then type the following at the DOS command line (assuming you are in *d:\perllesson\practice*):

```
perl printpoem1.pl poem.txt ↵
```

The program starts to output the poem line by line to the screen. In the file open function, the file handle is *F*. To input the file, the file handle must be put be-

tween two pointed brackets, which are called the file input operator; `$ARGV[0]` contains the file path and name. The *while* control structure loops between the two curly brackets; as long as there are lines in the input file, input one and output it to the screen. It should be emphasized that in Perl assignments have a truth value which depends on the value assigned, and that truth values are identified with numerical values and that, thus, in case of an unsuccessful assignment the truth value `FALSE` is obtained.

The result produced by *printpoem1.pl* is outputted to the screen and is not reusable. To store the result to a file that can be accessed any time we wish, use the following functions:

```
open(FILEHANDLE,">$ARGV[1]") [or die ("expression\n");
print(FILEHANDLE variable)
```

The first function opens a file for storing the output. The file name is given at the DOS command line and captured by `$ARGV[1]` if this filename is entered as the second argument and `.` Note the double quotes and the single right pointed bracket; If two right pointed brackets `>>` are used here, each time the program is run, the result is added to the old result. The second function prints the result in the file. Note there is no comma after the file handle.

```
printpoem2.pl
1. open(F,$ARGV[0]) or die("File does not exist!\n");
2. open(W,">$ARGV[1]") or die("File can't be opened.\n");
3. while($line=<F>){
4. print(W $line);
5. }
6. close(F);
7. close(W);
```

Now type the following on the DOS command line:

```
perl printpoem2.pl poem.txt result.txt ↵
```

The result is stored in *result.txt* in *d:\perllesson\practice*.

3.2 Inputting files inside a program

Input and output files can also be specified within a program. This is very easy to do; instead of `$ARGV[0]` and `$ARGV[1]`, we just use the names of the input file and output file as the following:

```
open(FILEHANDLE,"inputfilename") [or die ("expression\n");
```

```
open(FILEHANDLE,">outputfilename") [or die ("expression\n")];
```

We can also specify the drive and folder of the file to be opened. Look at the following example.

```
printpoem3.pl
1. open(F,"d:\\perllesson\\texts\\poem.txt") or die("File does not exist!\n");
2. open(W,">result.txt") or die("File can't be opened.\n");
3. while($line=<F>){
4. print(W $line);
5. }
6. close(F);
7. close(W);
```

Note that in statements 1—2 \\ is used. This is because \ is an escape character in Perl. The first \ is used to tell Perl that the following \ is a path separator, not an escape character. If two single quotes are used to enclose the file path and name, only one \ is necessary as shown below:

```
open(F,'d:\perllesson\texts\poem.txt') or die("File does not exist!\n");
```

Now type on the DOS command line the following:

```
perl printpoem3.pl ↵
```

The result is stored in *d:\perllesson\practice\result.txt*. Statement 1 is only for practice and is not necessary here since we have copied every file from *texts* to *practice*. So statement 1 can be simplified as follows:

```
open(F,"poem.txt") or die("File does not exist!\n")
```

Instead of reading the poem line by line, we can print out the entire file all at once using the following read file function:

```
read(FILEHANDLE,variable,filelength)
```

FILEHANDLE here must be the one specified in the open file function; *variable* is given by the user; it stores the contents of the entire file. *filelength* is length of the file in terms of number of bytes. A safe margin should be given to ensure that the file is not truncated. If a file is about 200000 bytes in length, 210000 would be safe for *filelength*.

```
printpoem4.pl
1. open(F,"poem.txt") or die("File does not exist!\n");
```

2. `read(F,$text,1500);`
3. `open(W,">result.txt") or die("File can't be opened.\n");`
4. `print(W $text);`
5. `close(F);`
6. `close(W);`

Because of the use of the `read` file function, the *while* loop is not used. Statement 2 puts the contents of the input file to the variable *\$text*; it can be any other name as long as it's suggestive. The length of the poem is only 1000 bytes, so 1500 is quite safe.

In Chapter 2 we wrote *math1.pl* to solve a complicated math problem as shown below:

$$\left[\sqrt[5]{23.412 \times \frac{4453.78}{9221}} \div \sqrt[4]{3321 - (12 + 31)^3} \right] \div \log_2 23867$$

Now we'll modify *math1.pl* and store the result in a file instead of displaying it on the screen.

```
math2.pl
open(W,">mathresult.txt") or die ("Can't create file.\n");
$math_result=sprintf "%.6f",(((23.412*(4453.78/9221))**(1/5))/(33210000-
(12+31)**3)**(1/4))/(log(23867)/log(2));
print (W $math_result);
close(W);
```

The result is stored in *mathresult.txt* and can be opened and closed any time it's needed.

3.3 Some string manipulation functions

There are several built-in string manipulation functions in Perl. We'll look at some of them in this section.

lc *variable* This converts the contents stored in *variable* into lower cases.

```
$words="THIS IS A DEMONSTRATION OF THE USE OF LC.";
$words= lc $words;
print $words;
this is a demonstration of the use of lc.
```

uc *variable* This function converts *variable* into upper case.

```

$words="this is a demonstration of the use of uc.";
$words= uc $words;
print $words;
THIS IS A DEMONSTRATION OF THE USE OF UC.

```

ucfirst *variable* This function puts the first letter of the first word in *variable* into upper case and the rest into lower case.

```

$words="this is a demonstration of the use of ucfirst.";
$words= ucfirst $words;
print $words;
This is a demonstration of the use of ucfirst.

```

However, if all the words in *\$variable* are in upper case, this function has no effects on it.

length(*variable*) This function measures the length of *variable* in number of characters.

```

$words="this is a demonstration of the use of length.";
$words_length=length($words);
print $words_length;
45

```

Note that the result is 45 instead of 37; this is because the white space is also a character, represented by the decimal ASCII character code 32. The following is the ASCII character code table:

0 NUL	16 DLE	32 SP	48 0	64 @	80 P	96 `	112 p
1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
2 STX	18 DC2	34 "	50 2	66 B	82 R	98 b	114 r
3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
4 EOT	20 DC4	36 \$	52 4	68 D	84 T	100 d	116 t
5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
7 BEL	23 ETB	39 '	55 7	71 G	87 W	103 g	119 w
8 BS	24 CAN	40 (56 8	72 H	88 X	104 h	120 x
9 HT	25 EM	41)	57 9	73 I	89 Y	105 i	121 y
10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
11 VT	27 ESC	43 +	59 ;	75 K	91 [107 k	123 {
12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124
13 CR	29 GS	45 -	61 =	77 M	93]	109 m	125 }
14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
15 SI	31 US	47 /	63 ?	79 O	95 _	111 o	

The ASCII character codes 0—31 are control characters and unprintable. For example 7 is the bell, 9 stands for the tab key, 13 carriage return. The English alphabet is represented by the ASCII character codes 65—90 and 97—122. The expanded ASCII character codes now have 256 characters.

Next, we'll look at two Perl functions for the ASCII character codes.

chr *asciicodenumber* This function converts ASCII codes into characters.

```
print chr 33;
!
```

```
print chr 56;
8
```

```
print chr 72;
H
```

```
print chr 104;
h
```

There is *asciicodes.txt* in *practice* containing the 256 ASCII code numbers. The following program converts them into characters. Note some are not printable, and a bell sound can be heard when the ASCII code number 7 is processed. Some computers are unable to print out ASCII character codes higher than 125.

asciicharacter.pl

1. open(F,"asciicodes.txt") or die ("File can't be opened.\n");
2. while (\$asciicodes=<F>){
3. print chr \$asciicodes;
4. print "\n";
5. }
6. close(F);

ord *character* This function converts a character into its corresponding ASCII code number. For character clusters, it only gives the ASCII code of the first character in the cluster.

There is a file called *characters.txt* containing 92 characters. The following short program converts these characters into their corresponding ASCII code values.

asciicodes.pl

1. open(F,"characters.txt") or die ("File can't be opened.\n");
2. while (\$character=<F>){

3. print ord \$character;
4. print "\n";
5. }
6. close(F);

getc filehandle This function cuts one character off *filehandle*. The following short program cuts characters one at a time from *adventure.txt* and turns them into their corresponding ASCII codes:

- ```
getc.pl
1. open(F,"adventure.txt") or die("Can't open file.\n");
2. open(W,">result.txt") or die("Can't create file.\n");
3. while($char=ord getc F){
4. print W "$char ";
5. }
6. close(F);
7. close(W);
```

Note the space after *\$char* in statement 4. Part of the result is shown below:

```
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 65 76 73
67 69 39 83 32 65 68 86 69 78 84 85 82 69 83 32 73 78 32
87 79 78 68 69 82 76 65 78 68 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 10 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 67 72 65 80 84 69 82 32 73 10 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 68 111 119 110 32
116 104 101 32 82 97 98 98 105 116 45 72 111 108 101 10
32
```

### 3.4 Applications

Now we'll write programs using some of the functions we've learned. First we'll write a short program that removes all the line breaks in *bncwordlst.txt* and puts all the words in a long line instead of in a one-word column.

(1)

- ```
removebreak.pl
1. open(F,"bncwordlist.txt") or die("File can't be opened.\n");
2. open(W,">result.txt") or die("File can't be created.\n");
3. while ($word=<F>){
4. chomp $word;
```

-
5. \$line.=lc \$word.", ";
 6. }
 7. print W \$line;
 8. close(F);
 9. close(W);

In this program, statements 1—2 do file input and output. Statements 3—6 create a loop, in which the word is inputted and processed one by one, until all the words in the wordlist are exhausted. Now *\$line* stores all the words in a long line, with the words separated with a comma and a space. Statement 7 prints contents of *\$line* to *result.txt*. The most important statement of this program is statement 4, which removes the line break after each word using the *chomp* function. Statement 5 puts the words in a long line one by one, separated by a comma and a space. Note the use of *.=*.

The following program converts all the words in *bncwordlist.txt* into upper case using the *uc* function.

(2)

- ```
uppercase.pl
1. open(F,"bncwordlist.txt")or die("File can't be opened\n");
2. open(W,">result.txt")or die("File can't be created.\n");
3. read(F,$text,9000000);
4. $convert=uc $text;
5. print(W "$convert");
6. close(F);
7. close(W);
```

Statement 3 reads the contents of *bncwordlist.txt* into the file. The actual file length of *bncwordlist.txt* is 889000 bytes, so 9000000 is long enough to get the entire file.

Next, we'll write a program to centre-justify all the lines in Keats' poem *To Autumn* in *poem.txt*. The poem is left-justified, as shown below:

```
To Autumn
Season of mists and mellow fruitfulness
Close bosom-friend of the maturing sun
Conspiring with him how to load and bless
With fruit the vines that round the thatch-eaves run;
To bend with apples the moss'd cottage-trees,
And fill all fruit with ripeness to the core;
To swell the gourd, and plump the hazel shells
With a sweet kernel; to set budding more
And still more, later flowers for the bees,
Until they think warm days will never cease,
For summer has o'er-brimm'd their clammy cells.
```

--John Keats—

Assuming the width of a common page is 80 characters excluding the page margins, then the vertical centre line of such a page is 40 characters from the left edge and 40 from the right. Suppose a line of the poem has 36 characters (including spaces), to centre-justify this line, the first half of the line should be placed to the left of the vertical centre line, with 22 white spaces from the left edge of the page to the start of the first half. Bearing the above in mind, we can now write the program.

(3)

*cjustify.pl*

```
1. open(F,"poem.txt")or die("File can't be opened.\n");
2. open(W, ">result.txt")or die("File can't be created.\n");
3. while ($line=<F>){
4. $linelength=length($line);
5. $leftspaces=" "x(40-$linelength/2);
6. $line=$leftspaces.$line;
7. print(W $line);
8. }
9. close(F);
10.close(W);
```

Statement 4 measures the length of each line. Statement 5 places the first half of a line to the left of the vertical centre line of the page, with  $40 - \text{\$linelength}/2$  white spaces placed between the left edge of the page and the beginning of the poem line. A white space can be produced by a pair of double quotes with a space in between, or use the *chr* function. The result is as follows:

*To Autumn*

*Season of mists and mellow fruitfulness  
 Close bosom-friend of the maturing sun  
 Conspiring with him how to load and bless  
 With fruit the vines that round the thatch-eaves run;  
 To bend with apples the moss'd cottage-trees,  
 And fill all fruit with ripeness to the core;  
 To swell the gourd, and plump the hazel shells  
 With a sweet kernel; to set budding more  
 And still more, later flowers for the bees,  
 Until they think warm days will never cease,  
 For summer has o'er-brimm'd their clammy cells.*

--John Keats--

The next program adds line numbers to the poem. The title of the poem *To Autumn* should have no line number, neither should the last line, which is the name of the author. In addition, all the lines of the poem should start three characters from the left edge of the page.

(4)

*linenumber.pl*

```

1. open(F,"poem.txt")or die("File can't be opened.\n");
2. open(W, ">result.txt")or die("File can't be created.\n");
3. while ($line=<F>){
4. $linenumber++;
5. $threespaces=(chr 32)x3;
6. if ($linenumber==1){
7. print (W $threespaces.$line);
8. }elsif(ord $line==45){
9. print(W $threespaces.$line);
10. }else{
11. $width=(chr 32)x(3-length($linenumber-1));
12. $line=($linenumber-1).$width.$line;
13. print(W $line);
14. }
15. }
16. close(F);
17. close(W);

```

In this program, statements 3—15 form a loop, in which the lines of the poem are inputted one by one and processed. Statement 4 generates line numbers, and statement 5 produces three white spaces. The spaces can also be generated by putting three spaces between a pair of single quotes or double quotes; but it would be difficult for the reader to tell the exact number of spaces this way. Statements 6—9 are for the first and last lines, which do not need line numbers (the last line begins with two hyphens, whose ASCII character code is 45), but they should be three characters away from the left edge of the page, which is done by statements 7 and 9. Statements 10—14 are for lines that need line numbers. Statement 11 generates spaces. If the length of the line number is 1, two spaces are produced; if the length is 2, one space is produced. Statement 12 adds the space or spaces thus produced to the right of the line numbers, followed a line of poem. Since the first line does not need a line number, the second line should be given line number 1, the third line number 2 and so on, which is done by *\$linenumber-1*. The result is as follows.

*To Autumn*

```

1 Season of mists and mellow fruitfulness
2 Close bosom-friend of the maturing sun
3 Conspiring with him how to load and bless
4 With fruit the vines that round the thatch-eaves run;
5 To bend with apples the moss'd cottage-trees,
6 And fill all fruit with ripeness to the core;
7 To swell the gourd, and plump the hazel shells
8 With a sweet kernel; to set budding more

```



9 And still more, later flowers for the bees,  
 10 Until they think warm days will never cease,  
 11 For summer has o'er-brimm'd their clammy cells.  
 --John Keats--

The following program picks out words beginning with *a*, *e*, *i*, *o*, *u* in *bncwordlist.txt* and stores them in separate files.

(5)

*pickwords.pl*

```

1. open(F,"bncwordlist.txt")or die("File can't be opened.\n");
2. open(A,">aword.txt")or die("File A can't be created.\n");
3. open(E,">eword.txt")or die("File E can't be created.\n");
4. open(I,">iword.txt")or die("File I can't be created.\n");
5. open(O,">oword.txt")or die("File O can't be created.\n");
6. open(U,">uword.txt")or die("File U can't be created.\n");
7. while ($word=<F>){
8. if (ord $word==65){
9. print(A $word);
10. }elseif(ord $word==69){
11. print(E $word);
12. }elseif(ord $word==73){
13. print(I $word);
14. }elseif(ord $word==79){
15. print(O $word);
16. }elseif(ord $word==85){
17. print(U $word);
18. }
19. }
20. close(F);
21. close(A);
22. close(E);
23. close(I);
24. close(O);
25. close(U);

```

Although this program has 25 statements, it's mainly simple repetitions of the *open*, *close* functions and *print* function. Statements 1—6 open *bncwordlist.txt* and create five files for storing separately words beginning with *A*, *E*, *I*, *O* and *U*. Statements 7—19 create a loop, in which words are inputted one by one and selected by statements 8—17 using the *ord* function. Note the use of the *if...elsif* statements, in which the ASCII value of the first letter of every word is checked. If it's one of the specified letters, the word is sent to the target file; otherwise the word is ignored and the next word is inputted and checked.

The following is an interactive program that counts the number of words of a wordlist, the total length of the words and the average length of the words, and stores the result to another file. The program first asks the reader for the name of the file to be processed, its path, and where to store the results.

(6)

*countword.pl*

1. print "This program computes the number of words of the file you specify,\n";
2. print "and stores the result in another file.\n";
3. print "Now enter the name of file to be processed below:\n";
4. \$input=<STDIN>;
5. open(F,\$input) or die ("File can't be opened.\n");
6. print "The file to be processed is:\n".uc \$input;
7. print "Please enter the name of the result file below:\n";
8. \$output=<STDIN>;
9. print "The result will be stored in: \n".uc \$output;
10. print "Processing. Please wait...\n";
11. print "Number Length\n";
12. open(W,">\$output") or die("File can't be created.\n");
13. while (\$word=<F>){
14. chomp \$word;
15. \$wordnumber++;
16. \$wordlength+=length(\$word);
17. if(\$wordnumber%10000==0){
18. print "\$wordnumber \$wordlength\n";
19. }
20. }
21. \$average=\$wordlength/\$wordnumber;
22. print (W "The total number of words is: \$wordnumber\n");
23. print (W "The total number of characters in file is: \$wordlength\n");
24. print (W "The average word length is: \$average");
25. print ("The total number of words is: \$wordnumber\n");
26. print ("The total number of characters in file is: \$wordlength\n");
27. print ("The average word length is: \$average");
28. close(F);
29. close(W);

Statements 1—11 are the interactive part of the program. The names of the input file and output file are captured by the STDIN function. The main part of the program are statements 13—20. Statement 15 counts the number of words; statement 16 adds the length of each of the words; statement 17 displays on the screen the number of words that have been counted every 10,000 words using the *modulo* operator %. Statements 21—24 send the obtained data with messages

about the data to the output file; statements 25—27 display the results to the screen.

### Exercises

1. Write an interactive program to combine all the lines of *poem.txt* into one paragraph.
2. Rewrite *cjustify.pl* program so that it can do right-justification to the words in *bncwordlist.txt* with word numbers added to the left of the words.
3. Write a program so that it can do central-justification to *poem.txt*, add line numbers to the left of each of the justified lines except for the first and the last lines, and compute the average line length (including spaces) of the poem (again excluding the first and the last lines). Output the result to a new file.
4. Write a program to divide words in *bncwordlist.txt* into the following groups:
  - a. words with length less than or equal to 3 and put these words in a file;
  - b. words with length 4—10 and put these words to different files according to their lengths;
  - c. words with length over 10 and put them in a file;
  - d. compute the total number of the words, the average length of all the words and the number of words in the different word length groups in a file.
5. In 3.3 we wrote *getc.pl* that turned all the characters in *adventure.txt* into ASCII codes stored in *result.txt*. Now write a program to turn all the ASCII codes back into characters.

## 4 Regular expressions: basic structure

In natural language processing, language teaching, quantitative linguistics and other language-related research, pattern matching is a run-of-the-mill task, boring but necessary, done quite often. The regular expressions provided by Perl are very efficient for such tasks. In *Speech and Language Processing, An Introduction to natural Language Processing, Computational Linguistics, and Speech Recognition* by Zurafsky and Marin, a substantial part of the second chapter is devoted to Perl's regular expressions. In this chapter we'll look at regular expressions and the application of regular expressions.

### 4.1 Operators for regular expressions

There is a set of operators used to form Perl regular expressions. They are `=~`, `m//`, `s///` and `tr///`.

#### 4.1.1 `=~` and `m//`

`=~` This is the most important regular expression operator; it applies the regular expression on its right to the string or variable on its left.

`m/pattern/` This operator matches *pattern* within a string or variable; *m* is optional.

Look at the following examples:

```
$phrasea="regular expression";
$phraseb="Regular Expression";
$phrasec="regular expression";
if ($phrasea=~m/$phraseb/){
print "phrasea equals phraseb";
}
if ($phrasea=~m/$phrasec/){
print "phrasea equals phrasec";
}
phrasea equals phrasec
```

```
$sentence="We know that the regular expression is very powerful.";
$string="pressio";
if ($sentence=~m/$string/){
print "The string pressio exists in the sentence";
}
```

```
}
The string pressio exists in the sentence.
```

*m* can be omitted:

```
$phrasea="regular expression";
$phraseb="Regular Expression";
$phrasec="regular expression";
if ($phrasea=~/$phraseb/){
print "phrasea equals phraseb";
}
if ($phrasea=~/$phrasec/){
print "phrasea equals phrasec";
}
phrasea equals phrasec
```

#### 4.1.2 s///

The *s///* operator is used as a pattern substitution operator; the syntax of its usage is as follows:

*s/pattern/substitution/*

```
$sentence="I like regular expressions.";
$sentence=~s/I like/Everybody likes/;
print $sentence;
Everybody likes regular expressions.
```

*substitution* can be a variable:

```
$sentence="I like regular expressions.";
$change="Everybody likes";
$sentence=~s/I like/$change/;
print $sentence;
Everybody likes regular expressions.
```

*s/pattern/substitution/* uses the following switches.

**e** This switch carries out Perl functions or math operations in *substitution*.

```
$sentence="I like regular expressions.";
$change="Everybody likes";
```

```
$sentence=~s/I like/uc $change/e;
print $sentence;
EVERYBODY LIKES regular expressions.
```

```
$phrase="Several Perl regular expression operators";
$phrase=~s/Several/3*2/e;
print $phrase;
6 Perl regular expression operators
```

```
$phrase="The length of the word antidisestablishmentarianism is x.";
$phrase=~s/x/length antidisestablishmentarianism /e;
print $phrase;
The length of the word antidisestablishmentarianism is 28.
```

**g** It's a global switch. With this switch, every instance of *pattern* is substituted by *substitution* in *s/pattern/substitution/*. This switch can also be used in *m/pattern/* to search for every instance of *pattern*.

```
$phrase="The length of antidisestablishmentarianism and the length of
about.";
$phrase=~s/length of/number of letters in/;
print $phrase;
The number of letters in antidisestablishmentarianism and
the length of about.
```

In the above, only the first instance of *length of* is replaced with *number of letters in*. The following uses the *g* switch:

```
$phrase="The length of antidisestablishmentarianism and the length of
about.";
$phrase=~s/length of/number of letters in/g;
print $phrase;
The number of letters in antidisestablishmentarianism and
the number of letters in about.
```

```
$sentence="Perl is soooooo powerful";
while($sentence=~m/o/g){
$number++;
print "Found $number O.\n";
}
Found 1 O.
Found 2 O.
```

*Found 3 0.*

*Found 4 0.*

*Found 5 0.*

*Found 6 0.*

Without the *g* switch, the program will go into a dead loop.

Perl automatically records the number of substitutions the *s///* operator has made. The following structure gets this number:

```
$n=($variable=~s/pattern/substitution/g)
$phrase="student of linguistics, teacher of linguistics and researcher of
linguistics.";
$number=($phrase=~s/linguistics/computer science/g);
print $number;
3
```

**i** This switch makes pattern matching and substitution case insensitive. It can also be used in *m//*.

```
$phrase="student of Linguistics, teacher of Linguistics and researcher of
Linguistics.";
$phrase=~s/linguistics/computer science/g;
print $phrase;
student of Linguistics, teacher of Linguistics and
researcher of Linguistics
```

No substitution has been made. Look at the following:

```
$phrase="student of Linguistics, teacher of Linguistics and researcher of
Linguistics.";
$phrase=~s/linguistic/computer science/gi;
print $phrase;
student of computer science, teacher of computer science and
researcher of computer science.
```

```
$sentence="Perl is so powerful.";
if($sentence=~m/perl/){
print "There is perl in sentence.\n";
}else{
print "Cannot find perl in sentence.\n";
}
Cannot find perl in sentence.
```

```

$sentence="Perl is so powerful.";
if($sentence=~m/perl/i){
print "There is perl in sentence.\n";
}else{
print "Cannot find perl in sentence.\n";
}

```

*There is perl in sentence.*

The following program counts the total number of *A* and *a* in *adventure.txt*.

```

counta.pl
open(F,"adventure.txt") or die ("Can't open file.\n");
read(F,$text,150000);
$number=($text=~s/a/a/gi);
print "The number of A's and a's in Alice's Adventures in Wonderland is:
$number.";

```

*The number of A's and a's in Alice's Adventures in Wonderland is: 8772.*

The counting is done in statement 3, in which *a* is substituted by itself, and *\$number* gets the number of such substitutions. Note that here instead of *a* any other letter, a space or even nothing can do.

**x** It ignores white spaces and the unprintable characters such as tabs, line breaks, and so on. It's not of much use except for putting notes within a pattern. This can also be used in *m//*.

```

$phrase="student of Linguistics, teacher of Linguistics and researcher of
Linguistics.";
$phrase=~s/li
n g
ui

s ti
cs/computer science/gix;
print $phrase;

```

*student of computer science, teacher of computer science and researcher of computer science.*

### 4.1.3 *tr//*

The syntax of *tr//* is as follows:



**tr/characterset1/characterset2/**

This operator does character substitution. It replaces characters in *characterset1* with those in *characterse2* one by one from the left, i.e., the first character of *characterset1* is replaced by the first character of *characterset2*, the second character of *characterset1* is replaced by the second character of *characterset2*, etc. Look at the following example:

```
$phrase="cat, cat, jumps on mat.";
$phrase=~tr/act/odg/;
print $phrase
dog, dog, jumps on mog.
```

In the above example, *a*, *c* and *t* in *cat, cat, jumps on mat* are respectively replaced by *o*, *d* and *g*.

If *characterset2* has more characters than *characterset1*, those extra characters in *characterset2* are ignored:

```
$phrase="cat, cat, jumps on mat.";
$phrase=~tr/act/odgh/;
print $phrase;
dog, dog, jumps on mog.
```

However, if *characterset2* has fewer characters than *characterset1*, after all the characters in *characerset2* are used up, the remaining characters of *characterset1* will all be replaced by the last character of *characterset2*:

```
$phrase="cat, cat, jumps on mat.";
$phrase=~tr/act/od/;
print $phrase;
dod, dod, jumps on mod.
```

In the above example, *a* is replaced by *o* and *c* by *d*; since there are only two characters in *characterset2* whose last character is *d*, *t* in *characterset1* is replaced by *d*.

The *tr///* operator can use the range operator”-“ for character replacement:

```
$phrase="cat, cat, jumps on mat.";
$sentence=~tr/a-z/A-Z/;
print $sentence;
CAT, CAT, JUMPS ON MAT.
```

The following program picks out words that have numbers in them, such as *A4*, *AK47*, etc.

```
numberword1.pl
```

```
1. open(F,"bncwordlist.txt") or die ("Can't open file.\n");
2. open(W,">result.txt") or die ("Can't create file.\n");
3. while ($word=<F>){
4. if($word=~tr/0-9/0-9/){
5. $number++;
6. print (W $word);
7. }
8. }
9. print (W "The number of words that have digits: $number");
```

Statement 4 checks whether a word has numbers. If it does, the character substitution succeeds, *\$number* is increased by 1 and the word that has numbers in it is put to *result.txt*.

As in the case of the *s///* operator, Perl records the number of character substitutions the *tr///* operator has made. The following structure gets this number:

```
$n=($variable=~tr/character1/character2/)
$phrase="cat, cat, jumps on mat.";
$number=($phrase=~tr/act/odg/);
print "$phrase\n";
print "Number of characters substituted: $number";
dog, dog, jumps on mog.
Number of characters substituted: 8
```

The *tr///* operator uses the following switches:

**c** This keeps the characters of a string unchanged if these characters are in *character1*, while all other characters, including spaces, of the string will be replaced by the last character of *character2*.

```
$phrase="cat, cat, jumps on mat.";
$phrase=~tr/act/!/?/c;
print $phrase;
cat??cat????????????at?
```

In the above example, except for *a*, *c* and *t* in *\$phrase*, which are in *character1*, all the rest, including spaces and punctuation marks, are replaced by the last character *?* in *character2*. Here, if we use *\$phrase=~tr/act/?/c* instead of *\$phrase=~tr/act/!/?/c*, the result would be the same.

**d** This deletes the unmatched characters in *characterSet1*.

```
$phrase="cat, cat, jumps on mat.";
$phrase=~tr/actm/odg/d;
print $phrase;
dog, dog, jups on og.
```

In the above example, *m* of *characterSet1* has no corresponding substitution character in *characterSet2*, so it's deleted from *\$phrase*.

**s** This switch keeps only one of the consecutive identical substitutions and deletes the rest.

```
$phrase="caat, caaat, jumps on maaaaat.";
$phrase=~tr/act/odg/;
print $phrase;
doog, dooog, jumps on moooog.
```

```
$phrase="caat, caaat, jumps on maaaaat.";
$phrase=~tr/act/odg/s;
print $phrase;
dog, dog, jumps on mog.
```

The *s* switch is particularly useful in removing extra spaces in texts. In the following, there are 12 spaces after the first *cat* and comma and 13 spaces after the second *cat* and comma; the *s* switch keeps only one space for each of the two consecutive space clusters:

```
$phrase="cat, cat, jumps on mat.";
$phrase=~tr/ / /s;
print $phrase;
cat, cat, jumps on mat.
```

*s///* can't be used in such situations:

```
$phrase="cat, cat, jumps on mat.";
$phrase=~s/ / /g;
print $phrase;
cat, cat, jumps on mat.
```

## 4.2 Regular expression quantifiers and other operators

### 4.2.1 The general quantifiers and wild card

In the previous section we looked at three regular expression operators: *m//*, *s///* and *tr///*. However, these operators are not very useful without regular expression quantifiers and other devices. Next, we'll look at four such quantifiers: *\**, *+*, *?* and *.*

*\** This is also called the Kleene star. It's used in regular expressions to stand for zero or more consecutive occurrences of the immediate preceding character, e.g. *a\** : zero or more *a*; *CC\**;: one or more *C* etc. The following program picks out words with two or more consecutive *e*'s in *bncwordlist.txt*:

*eeword.pl*

1. `open(F,"bncwordlist.txt") or die ("Can't open file.\n");`
2. `open(W,">result.txt") or die ("Can't create file.\n");`
3. `while($word=<F>){`
4. `if($word=~m/eee*/i){`
5. `$number++;`
6. `print (W "$word");`
7. `}`
8. `}`
9. `print (W "There are $number words with two or more e's in file.");`

Statement 4 searches for two *e*'s followed by zero or more *e*.

*+* This quantifier is used in regular expressions to stand for one or more consecutive occurrences of the preceding character. For example, *a+* means one or more *a*, *oo+* means two or more *o*'s etc. In *lookinglass.txt*, each paragraph is terminated with two line breaks, while within the paragraph each line of text ends with a line break. We'll separate paragraphs with only one line break and remove all other line breaks within a paragraph.

*makeparagraph.pl*

1. `open(F,"lookinglass.txt") or die ("Can't open file.\n");`
2. `open(W,">result.txt") or die ("Can't create file.\n");`
3. `read(F,$text,170000);`
4. `$text=~s/\n/ /g;`
5. `$text=~s/ +/\n/g;#there two spaces before+`
6. `print (W $text);`

In this program, statement 4 replaces each line break with a space. Statement 5 replaces two or more consecutive line breaks with only one line break.

? This quantifier is used in regular expressions to stand for zero or one occurrence of the preceding character, e.g. *b?*: zero or one *b*; *er?*: *e* followed by zero or one *r*, etc. The following searches for *colour* and *color* in *bncwordlist.txt* using ?:

*colour.pl*

1. open(F,"bncwordlist.txt") or die ("Can't open file.\n");
2. open(W,">result.txt") or die ("Can't create file.\n");
3. while (\$word=<F>){
4. if(\$word=~m/colou?r/i){
5. \$number++;
6. print (W \$word);
7. }
8. }
9. print (W "The number of colour or color is: \$number");

In statement 4 *u?* means zero or one occurrence of *u*.

· It is a wild card that can stand for any character except the line break. It's often used with \* and ?. In the following the wild card · is used together with ? to return the different word forms of *begin* to its stem, leaving other words unchanged:

```
$word="begin begins beginning began begun beginner beginners begging
beggar";
$number=($word=~s/beg.n+e?r?s?i?n?g?/begin/g);
print "$word: There are $number words whose stem is begin.";
begin begin begin begin begin begin begin begging beggar:
There are 7 words whose stem is begin.
```

In the regular expression */beg.n+e?r?s?i?n?g? /* the wild card between *g* and *n* matches *i*, *a* and *u* in *begin*, *began* and *begun*, while *n+* matches one or more *n*, in this case *nn* in *beginning*, *beginner* and *beginners*. *e?r?* matches either zero *er* or just one *er*; it's the same with *s?i?n?g?*. So this regular expression matches all the variant forms of *begin* in *\$word*, which are then replaced by *begin*, while *begging* and *beggar* remain unchanged.

Next, we'll use the wild card and + to write a program that picks words ending in *able*, *ous*, *ial*, *ious*, *less* or *ful* and stores the results in separate output files.

*adjword.pl*

```

1. open(F,"bncwordlist.txt") or die ("Can't open file.\n");
#Statement 2—7 create output files for storing words with the specified
endings
2. open(G,">ble.txt") or die ("Can't create file.\n");
3. open(H,">ious.txt") or die ("Can't create file.\n");
4. open(I,">less.txt") or die ("Can't create file.\n");
5. open(J,">ful.txt") or die ("Can't create file.\n");
6. open(K,">ous.txt") or die ("Can't create file.\n");
7. open(L,">ial.txt") or die ("Can't create file.\n");
#The following statement creates an output file for storing the number of
#words with the specified word endings.
8. open(W,">result.txt") or die ("Can't create file.\n");
9. while ($word=<F>){
10.if($word=~m/.+ble\n/){
11.$number_ble++;
12.print G $word;
13.}elseif($word=~m/.+ious\n/){
14.$number_ious++;
15.print H $word;
16.}elseif($word=~m/.+less\n/){
17.$number_less++;
18.print I $word;
19.}elseif($word=~m/.+ful\n/){
20.$number_ful++;
21.print J $word;
22.}elseif($word=~m/.+ous\n/){
23.$number_ous++;
24.print K $word;
25.}elseif($word=~m/.+ial\n/){
26.$number_ial++;
27.print L $word;
28.}
29.}
30.print (W "Number of words ending in _ble, _ious, _less, _full, _ous and
 _ial\n");
31.print (W "The number of words ending in ble is: $number_ble\n");
32.print (W "The number of words ending in ious is: $number_ious\n");
33.print (W "The number of words ending in less is: $number_less\n");
34.print (W "The number of words ending in ful is: $number_ful\n");
35.print (W "The number of words ending in ous is: $number_ous\n");
36.print (W "The number of words ending in ial is: $number_ial\n");
37.$totalnumber=$number_ble+$number_ious+$number_less+$number_f
 ul+$number_ous+$number_ial;

```

```
38. print (W "The total number of these words is: $totalnumber");
```

This program has 38 statements, but many of them are simple repetitions. The regular expression used to get words with the specified endings is *m/.+wordending\n/*. *.+* stands for one or more letter before the specified word endings, while *\n* ensures that the specified endings occur at the end of the words, not in the middle, since the words in *bncwordlist.txt* are arranged in one vertical column, each with a *\n* immediately after it.

#### 4.2.2 The greediness of the quantifiers \* and +

The quantifier *\** and *+* are “greedy”; that is, they try to extend their effect as far as possible. In the sentence *It is the so called greediness, isn't it?*, if we want to replace *It is* with *It's*, it appears that the following would work:

```
$line="It is the so called greediness, isn't it?";
$line=~s/.*is/It's/;
print "$line\n";
```

It seems the combination of the wild card and the Kleene star would stand for *It* and the white space before *is*, which would all be replaced by *That's*. However, the result is *It'sn't it?* If we use *+* instead of *\**, the result is the same. This is the so called greediness of *\** and *+*. There are two *is*'s in the sentence. The combination of the wild card and *\** or *+* here means everything from the beginning to the rightmost instance of *is*, which would all be replaced by *It's*, hence the result. To reduce the greediness, we can use the *?* quantifier put immediately after *\** or *+* to confine the effect of *.\** or *.+* from the start to the first *is*:

```
$line="It is the so called greediness, isn't it?";
$line=~s/.?*is/It's/;
print "$line\n";
It's the so called greediness, isn't it?
```

Here's another example to show the greediness of *\** and *+*. The following script attempts to replace *The* with *Those* using the pattern *.\*e*:

```
$line="The greedy Perl quantifiers";
$line=~s/.*e/Those/;
print "$line\n";
Thosers
```

This is because `.*e` gets everything from the start to the last `e`, leaving only `rs`, and everything preceding `rs` is replaced by `Those`. Now we'll use `?` to confine `.*e` within the start and the first `e`:

```
$line="The greedy Perl quantifiers";
$line=~s/.*?e/Those/;
print "$line\n";
Those greedy Perl quantifiers
```

### 4.2.3 The alternative operator, anchors and the escape operator

Now let's look at the alternative operators `[ ]` and `|`.

`[character set]` This means any single character of *character set*. The following example removes vowel letters from the list of words:

```
$word="about cat educate bed dog Perl unit";
$word=~s/[aeiou]//g;
print $word;
bt, ct, dt, bd, dg, Prl, nt
```

`|` This is the pipe line alternative operator with function of OR, e.g. `a|b|c|d` means *a* or *b* or *c* or *d*.

```
$word="about cat educate bed dog Perl unit";
$word=~s/a|e|i|o|u//g;
print $word;
bt, ct, dt, bd, dg, Prl, nt
```

Perl has two regular expression position anchors `^` and `$`, the former for specifying the initial position of a string, while the latter the end position of a string. The syntax of the position anchors is as follows:

`s/^string/pattern/`

`m/^pattern/`

`s/string$/pattern/`

`m/string$/`

Look at the following examples



```
$word="antelope aardvark llama";
$word=~s/[aeiou]//g;
print $word;
ntlpr rdvrk llm
```

The above removes all the vowel letters from *\$word*. Now we'll use the `^` anchor:

```
$word="antelope aardvark llama";
$word=~s/^[aeiou]//g;
print $word;
ntelope aardvark llama
```

`^[aeiou]` means any of the five vowel letters at the beginning of *\$word*. So this time only *a* in *antelope* is removed. Now look at the use of the end position anchor `$`:

```
$word="antelope aardvark llama";
$word=~s/[aeiou]$//g;
print $word;
antelope aardvark llam
```

Here `[aeiou]$` means any of the five vowel letters at the end of *\$word*. So *Llama* now becomes *llam*.

However, placed inside the alternative operator `[ ]`, `^` means **NOT**:

```
$word="about cat educate bed dog Perl unit";
$word=~s/[^(aeiou)]//g;
print $word;
aouaeuaeeoeui
```

In the above, the regular expression `s/[^(aeiou)]//g` removes letters except *a*, *e*, *i*, *o*, *u*, accomplished by the use of `^` inside `[ ]`.

Next, we'll consider the use of the escape operator `\`. Suppose we want to remove the punctuation marks from the following sentence with a regular expression:

*Processing is under way. Please wait...*

The escape operator `\` must be used:

```
$sentence="Processing is under way. Please wait...";
$sentence=~s/\./g;
print $sentence;
Processing is under way Please wait
```

Without the escape operator, error will result because the full stop will be regarded as the wild car. Of the non-alphanumeric characters listed below:

```
,.?!;:~'><~`!@#$%^&*()-_+={}[]\|/
```

the following need the escape operator when treated as literals in *s///* or *m//*:

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

The following example removes the non-alphanumeric characters from *\$string* using the pipe line alternative operator *|*. Note the escape operator used before *.*?

```
$^*+-(){ } [] \| \ /.
```

```
$string=q(Remove these marks:.,?!"*%|<>+=$_@()&~^#\[\]\{\}\$-);
$string=~s/\.|?|\$|\-|\^|*|\+|\(|\)\|{|}\|_|!|,|#|@|_|=|<|>|/|g;
print $string;
Remove these marks
```

However, when inside the alternative operator *[ ]*, only the following non-alphanumeric characters need the escape operator when treated as literals:

```
^ \ / [] $ -
```

Look at the following example:

```
$string=q(Remove these marks:.,?!"*%|<>+=$_@()&~^#\[\]\{\}\$-);
$string=~s/[^\[\]\|\$-\|:.,?!"*%|<>+=$_@()&~#\{\}]/g;
print $string;
Remove these marks
```

## 4.3 Applications

So far we have covered a lot of ground in Perl programming. Next, we'll try to use what we have learned so far to write some practical programs.

### 4.3.1 Text tokenizer

First, we'll write a program that tokenizes a text, i.e. breaking a text into individual words arranged in a single column. The program is as follows:

*tokenizer.pl*

```
1. open(F,'adventure.txt') or die("File cant be opened.\n");
2. open(W,'>result.txt') or die("Can't create file.\n");
3. read(F,$text,150000);
#The following statement coverts non-alphanumeric characters, as well as
#spaces, into spaces. Note the space after \n. The s switch turns two or
more consecutive spaces into only one space.
```

4. `$text=~tr/[,:.?!"*%|<>+=_@(){}&~^#\[\]\$\\-\n ]/ /s;`
5. `$text=~s/^\s|$/g; #remove initial and end space`
6. `$text=~s/ \n/g; #turn spaces into line breaks`
7. `print (W "$text");`
8. `close(F);`
9. `close(W);`

### 4.3.2 Computing syllabic word length

There are two ways to measure word length, one in number of letters, the other in number of syllables. The former can be easily done with the *length()* function, but the latter is not very straight forward. It's difficult to separate words into syllables even manually. There are rules for separating the syllables of a word. The syllable structure of the English words is as follows: (nV)nCnV[nC(nV)]. nV is a vowel or a vowel cluster and nC is a consonant or a consonant cluster. Here V and C can also mean vowel letters and consonant letters. nV within the round brackets are optional; while those in the square brackets can be re-duplicated. Generally, the number of syllables of a word is actually the number of nV's in it. However, there are exceptions and the following are some of them:

- A. a consonant plus *e* at the end of a word does not form a syllable, e.g., *live, like*, etc, except in a few words such as *simile, recipe*, etc ;
- B. word final *ble, gle, ple, sm* etc constitute a syllable, e.g., *people, syllable, strangle, isolationism*, etc;
- C. vowel clusters such as *ea, io, ia, uo* can constitute either one syllable, or two syllables, e.g., *peasant, creation, ratio, biology, quote, duo, India, special*, etc.

The above covers the majority of the constitution of word syllables, hence is enough for our purpose. Now we'll write a program to compute word length in syllables. For the sake of simplicity, *ia* in medial position and *io* will be regarded as two syllables while *ea, uo* and other vowel clusters as one syllable.

#### *countsyllable.pl*

1. `open(F,"bncwordlist.txt")or die("Can't open file.\n");`
2. `open(R,">wordsyl.txt") or die("Can't create file.\n");`
3. `open(W,">syllainfo.txt") or die("Can't create file.\n");`
4. `while($word=<F>){`
5. `$wordnumber++;`
- `#The following statement assigns $word to $word2 because $word2 will be`
- `#destroyed in statement 7 for counting the number of a, e, i, o, u, y.`
6. `$word2=$word;`
- `#The following statement roughly estimates number of syllables by`
- `#counting number of a, e, i, o, u, y. This number will be adjusted in later`

```

#statements. Here z can be any other letter.
7. $syllable=(($word2=~s/[aeiouy]+/z/gi);
#The following is for strings consisting only of consonant letters such as St,
#Sqrt etc or word ending in sm.
8. $syllable++ if($syllable==0 or $word=~m/sm$/);
#In statements 9 and 10, e in tively is not a syllable, word final ial onstitute
#one syllable, and word final e preceded by letters other than e, i, o, a is not
#a syllable. So $syllable should be reduced by 1, with the exception of
#word final ple, ble, gle or words such as He, She, He, Be.
9. if($word=~m/tively$/ or $word=~m/ial$/ or $word=~m/[^eioa]e$/){
10. unless($word=~m/ple$/ or $word=~m/ble$/ or $word=~m/gle$/ or
 $word eq "The\n" or $word eq "She\n" or $word eq "He\n" or $word eq
 "Be\n"){
11. $syllable--;
12. }
13. }
14. print (R "$syllable\t$word");
15. $cumusyllable+=$syllable;#for total number of syllables in
 wordlist
16. $syllable=0; #return $syllable to 0 for the next word
17. }
18. $average=$cumusyllable/$wordnumber;
19. print (W "The total number of words is: $wordnumber\n");
20. print (W "The average word length in number of syllables is:
 $average\n");
21. close(F);
22. close(R);
23. close(W);

```

This program uses rule based syllable segmentation, but, as the saying goes, rules are made to be broken. There are cases that are not dealt with in this program such as *e* in words like *tasteful*, *likely*, *simile*, *recipe* etc; these are left to the reader as an exercise. Unlike some branches of science that require zero error, automatic natural language processing is notoriously error prone (e.g. machine translation), so the result of a program should always be manually checked.

### 4.3.3 Removal of HTML codes in texts

Quite often texts to be dealt with are not clean ones—they have codes in them. Many files have HTML (Hyper Text Markup Language) codes, such as web pages, text files to be read with web browsers such Internet Explorer. The file *msndream.htm* containing Shakespeare's *A Midsummer Night's Dream* is marked

with the HTML codes. To see these HTML codes, open it with *Wordpad*, not a web browser where these codes are invisible. The following is an extract:

```

<HTML>
<HEAD>
 <META HTTP-EQUIV="Content-Type" CONTENT="text/html;
 charset=iso-8859-1">
 <META NAME="GENERATOR" CONTENT="Mozilla/4.03 [en]
 (Win95; I) [Netscape]">
<TITLE>Midsummer Night's Dream: Entire Play</TITLE>
<!-- saved from
url=(0056)http://www.chemicool.com/shakespeare/midsummer/
full.html -->
<LINK
href="Midsummer Night's Dream.files/shake.css"
media=screen rel=stylesheet
type=text/css>
</HEAD>
<BODY TEXT="#000000" BGCOLOR="#FFFFFF">

<TABLE WIDTH="100%" BGCOLOR="#CCF6F6" >
<CAPTION><TBODY>

</TBODY></CAPTION>

<TR>
<TD ALIGN=CENTER class="play">A
Midsummer Night's Dream </TD>
</TR>

<TR>
<TD ALIGN=CENTER class="nav">by William
Shakespeare </TD>
</TR>
</TABLE>

<H3>
ACT I</H3>

<H3>
SCENE I. Athens. The palace of THESEUS. </H3>

```

```
<BLOCKQUOTE><I>Enter THESEUS, HIPPOLYTA, PHILOSTRATE, and
Attendants</I></BLOCKQUOTE>
```

```
THESEUS
```

```
<BLOCKQUOTE>Now, fair Hippolyta, our
nuptial hour
```

```

Draws on apace; four happy days
bring in
```

```

Another moon: but, O, methinks,
how slow
```

```

This old moon wanes! she lingers
my desires,
```

```

Like to a step-dame or a dowager
```

```

Long withering out a young man
revenue. </BLOCKQUOTE>
```

```
HIPPOLYTA
```

```
<BLOCKQUOTE>Four days will quickly
steep themselves
in night;
```

In the above the HTML codes are all enclosed between < and >; the rest is the actual text. Now we'll write a program to remove these HTML codes to get the clean text.

```
cleanhtm.pl
```

1. open(F,"msndream.htm") or die ("File can't be opened.\n");
2. open(R,">result.txt") or die ("Can't create file.\n");
3. while (\$line=<F>){
4. if(\$line!~m/<LINK|type=|href=|<TITLE>|\&nbsp;/){
5. \$line=~s/<.\*?>//g and \$line=~s/^ +//;
6. print (R \$line) if(\$line!~/^\n/);
7. }
8. }

The program loops between statements 3 and 8. In statement 4 lines with codes such as <LINK, type=, href= and <TITLE> are not considered since lines with those codes have no original text. Note the use of the NOT operator !, the pipe line alternative operator | and the escape operator \. Statement 5 removes non-textual codes, which are enclosed between pairs of pointed brackets. ? is used to prevent the greediness of \*. The following is the above extract with the HTML codes removed:

*ACT I*

*SCENE I. Athens. The palace of THESEUS.*

*Enter THESEUS, HIPPOLYTA, PHILOSTRATE, and Attendants*

*THESEUS*

*Now, fair Hippolyta, our nuptial hour*

*Draws on apace; four happy days bring in*

*Another moon: but, O, methinks, how slow*

*This old moon wanes! she lingers my desires,*

*Like to a step-dame or a dowager*

*Long withering out a young man revenue.*

*HIPPOLYTA*

*Four days will quickly steep themselves*

*in night;*

## Exercises

1. Write a program to extract words that have five or more consecutive consonant letters from *bncwordlist.txt* and count the number of such words.

2. Write a program using regular expressions to count the number of the different word forms of *make* in *adventure.txt*.

3. There is *tagged.txt* which contains a section of *Alice's Adventures in Wonderland* with POS tags. Write a program to remove these POS tags and other non-textual codes, and then compute its average syllabic word length.

4. *text.xml* contains part of *Alice's Adventures in Wonderland* in the XML format. Write a program to remove the XML codes, and then compute the syllabic word length.

5. *poswords.txt* contains a list of words with their POS tags, which are enclosed between a pair of double quotes as shown below:

"By No Means AV0"

"By PRP"

"By PRP-AVP"

"Can VM0"

"Capital NN1-AJ0"

"Capitalism NN1"

Write a program to remove the double quotes and the POS tags, separate phrases such as *By No Means* into individual words and compute the average word length in letters of all the words.

## 5 Regular expressions: advanced topics

In Chapter 4, we looked at the basic structures of regular expressions, regular expression operators, quantifiers, etc. We also used regular expressions in some practical programs and saw the advantage and power of regular expressions. In this chapter, we'll deal with some advanced topics on regular expressions. We'll examine the use of metacharacters, special variables, numbered variables and back references. In addition, we'll also look at three string handling functions. Finally we'll put what we learn in this chapter in some practical programs.

### 5.1 Metacharacters for regular expressions

The metacharacters for regular expressions are used to specify what sort of characters, whether printable or non-printable, to be matched in the target string. This makes pattern matching more flexible and convenient.

`\w` This metacharacter matches letters, numbers and the underscore `_`.

The following example removes alphanumeric characters from `$string`:

```
$string=qq((1).This is an example: _/[[]{} \n|^#\t&%$@=+-<>?!~`,';");
$string=~s/\w//g;
print $string;
O. : /[]{}
/*^# &%=+-<>?!~`,';"
```

In the above example, `\w` stands for all the alphanumeric characters and the underscore in `$string`, which are replaced by nothing, leaving only the non-alphanumeric characters, including the white space, the line break and the tab.

`\W` This is the opposite of `\w`; it stands for all the non-alphanumeric characters, including the white space, line break and tab, except the underscore `_`.

```
$string=qq((1).This is an example: _/[[]{} \n|^#\t&%$@=+-<>?!~`,';");
$string=~s/\W//g;
print $string;
1Thisisanexample_
```

The following is a short program that computes the total number of the 26 letters in `adventure.txt`.

```
letternumber.pl
```



1. `open(F, "adventure.txt") or die("Can't open file.\n");`
2. `read(F,$text,160000);`
3. `$text=~s/\W//g;#remove non-alphanumeric characters`
4. `$text=~s/[0-9]_|_//g;#remove digits and the underscore`
5. `$letternumber=length($text);`
6. `print "The total number of letters used is: $letternumber.";`
7. `close(F);`

*The total number of letters used is: 107450.*

**\s** This matches white space.

```
$string="This is an example of the use of metacharacters.";
$string=~s/\s//g;
print $string;
Thisisanexampleoftheuseofmetacharacters.
```

**\S** This matches anything but white space.

```
$string="This is 1 example of the use of metacharacters.";
$string=~s/\S*/g;
print $string;
**** * * ***** ** *** ** * ****
```

**\d** This matches a number.

In Chapter 4 there is a program called *numberword1.pl* that extracts words with numbers within them. This was done with `$word=~tr/0-9/0-9/`. Now we'll use `\d` to do the same thing instead.

*numberword2.pl*

1. `open(F, "bncwordlist.txt") or die("Can't open file.\n");`
2. `open(R, ">result.txt") or die ("Can't create file.\n");`
3. `while($word=<F>){`
4. `if($word=~m/\d/g){`
5. `$wordnumber++;`
6. `print R $word;`
7. `}`
8. `}`
9. `print(R "The total number of alphanumeric words is: $wordnumber.\n");`
10. `close(F);`
11. `close(R);`

**\D** This matches anything but a number.

```
$string="The following are not letters: 0123456789,./';&()*", but the
following are: abcde...";
$string=~s\D//g;
print $string;
0123456789
```

**\b** This matches a word boundary. The following is an example without the use of `\b`:

```
$string="Is this the title of his thesis? Yes. It is. ";
$string=~s/is/was/gi;
print $string;
was thwas the title of hwas theswas? Yes. It was.
```

In the above, *is* in `$string`, whether it's a word or part of a word, is all replaced by *was*. The following replaces *is* with *was* only if *is* is an independent word because of the use of `\b`:

```
$string="Is this the title of his thesis? Yes. It is.";
$string=~s/\bis\b/was/gi;
print $string;
was this the title of his thesis? Yes. It was.
```

**\B** This matches anything but a word boundary.

```
$string="Is this the title of his thesis? Yes. It is.";
$string=~s/\B*/gi;
print $string;
I*s t*h*i*s t*h*e t*i*t*l*e o*f h*i*s t*h*e*s*i*s?* Y*e*s.*
I*t i*s.*
```

In the above, all the non-word-boundaries are replaced by `*`.

## 5.2 Special variables

The following are the special variables used in regular expressions.

**\$&** This stands for the pattern to be matched in a string.

In the following, we separate *It's useful, isn't it? Yes, it is! He replied enthusiastically.* into three sentences and print the sentences on different lines. We'll use `.?!` as sentence delimiters.

```

$sentence="It's useful, isn't it? Yes, it is! He replied enthusiastically.";
$sentence=~s/[.?!]\s/$&\n/g;
print $sentence;
It's useful, isn't it?
Yes, it is!
He replied enthusiastically.

```

In the above, the special variable `$&` in the second statement matches one of the punctuation marks within the alternative operator followed by a space, which are then replaced by the punctuation mark and a line break. Without the special variable, the sentences would all be without sentence final punctuation marks.

In the following, we'll use `$&` in a program to mark the different word forms of *make* with four `*`'s on either side and count the total number of *make* and its different word forms.

```

markmake.pl
1. open(F,"adventure.txt") or die ("File can't be opened.\n");
2. read(F,$text,160000);
3. open(R,">result.txt") or die ("Can't create file.\n");
4. $number=($text=~s/\bma(de|k(e|es|ing))\b/****$&****/gi);
5. print "The number of MAKE in text is: $number\n";
6. print R $text;
7. close(F);
8. close(R);

```

The total number of *make* and its different word forms are 76. Check *result.txt* to see whether these different word forms of *make* are marked with `*`.

`$`` This stands for the string before the pattern to be matched.

```

$sentence="They all thought that warm days would never end.";
$sentence=~m/thought/;
print $`;
They all

```

`$'` This stands for the string following the pattern to be matched.

```

$sentence="They all thought warm days would never end.";
$sentence=~m/thought/;
print $';
that warm days would never end.

```

These special operators can be used together:

```
$sentence="They all thought that warm days would never end.";
$sentence=~m/thought/;
print $&.$'.'.$`;
thought that warm days would never end. They all
```

### 5.3 Back referencing

Suppose we want to search for the following pattern ABCDDCBA, such as the string *accbdddmbcca* in a text, back referencing is very suitable for such tasks. Back referencing uses back reference variables in the form of  $\backslash n$  or  $\$n$  to refer back to the specified strings. The string to be thus referred must be enclosed in round brackets in the form of (*stringtoberferred*). If there are three back referenced strings, then  $\backslash 1$  refers back to the first back referenced string,  $\backslash 2$  the second back referenced string, and  $\backslash 3$  the third back referenced string. Instead of  $\backslash n1$ ,  $\backslash n2$  and  $\backslash n3$ , we can also use  $\$1$ ,  $\$2$  and  $\$3$ . The difference between  $\backslash n$  and  $\$n$  is that both can be used in *substitution* in *s/pattern/substitution/* while only  $\backslash n$  can be used in *pattern* in *m/pattern/* or *s/pattern/substitution/*.

The following script tests whether the string *kccxjj jxxcck* is of the pattern ABCD DCBA:

```
$pattern="The string kccxjj jxxcck is of the pattern ABCD DCBA";
if($pattern=~k(c+)x(j+) \2x\1k/){
print "The string kccxjj jxxcck is of the pattern ABCD DCBA";
}
The string kccxjj jxxcck is of the pattern ABCD DCBA
```

Here  $\backslash 1$  refers back to the pattern in the first pair of brackets, and  $\backslash 2$  to the pattern in the second. If there are more back referenced strings enclosed in brackets, more back reference variables should be used, e.g.  $\backslash 3$ ,  $\backslash 4$  and so on. In the above case only  $\backslash n$  can be used.

In the following, the back reference variables are used in *substitution* in *s/pattern/substitution/*:

```
$line="ONOMATOPEAIA Frequency 1 Length 12 This word
is rare.";
$line=~s/(\w+)\s+\w+\s+(\d)\s+\w+\s+(\d+)\s+./$2 $3 $1/g;
print $line;
1 12 ONOMATOPEAIA
```

In the above,  $(\w+)$  stands for *ONOMATOPEAIA* and is back referenced by  $\$1$ , the following  $\backslash s+\backslash w+\backslash s+$  stands for *Frequency* with spaces on either side.  $(\d)$  stands for *1*, back referenced by  $\$2$ , while the following  $\backslash s+\backslash w+\backslash s+$  stands for

*Length* with spaces on either side. (*d+*) stands for *12*, back referenced by *\$3*. Here we can also use *\1*, *\2* and *\3* as back reference variables.

## 5.4 Quantifying expressions

In the preceding chapter, we learned the use of general quantifiers; now we'll look at the quantifying expressions that are used to specify the occurrence of a pattern.

*{n}* This expression means *n* occurrences of a pattern.

The following is a program that uses this expression to get words with seven consecutive consonant letters in *bncwordlist.txt*

```
consonant2.pl
```

1. open(F,"bncwordlist.txt") or die ("File can't be opened.\n");
2. open(W,">result.txt") or die ("Can't create file.\n");
3. while(\$word=<F>){
4. if(\$word=~m/[bcdfghjklmnpqrstvwxyz]{7}/i){
5. \$number++;
6. print W \$word;
7. }
8. }
9. print(W "The total number of words that have seven consecutive  
consonant letters or more is: \$number.\n");
10. close(F);
11. close(W);

```
Aaaaaaaaaahhhhhhhhhh
```

```
Aaaaarrrggghhh
```

```
Aberystwyth
```

```
Argyllshire
```

```
Arrhythmia
```

```
Arrhythmic
```

```
Blythswood
```

```
Brachyrhynchus
```

```
Cccckkk
```

```
Cylchgrawn
```

```
Dyffryn
```

```
Eglwyswrw
```

```
Gcggatcttggtgaccaggg
```

```
... ..
```

---

*The total number of words that have seven consecutive consonant letters or more is: 43.*

**{*n*,}** This means at least *n* occurrences of a pattern.

The following gets words containing at least two consecutive *o*'s.

*doubleo.pl*

```
1. open(F,"bncwordlist.txt") or die ("File can't be opened.\n");
2. open(W,">result.txt") or die ("Can't create file.\n");
3. while($word=<F>){
4. if($word=~m/o{2,}/i){
5. $number++;
6. print W $word;
7. }
8. }
9. print(W "The total number of words that have at least two consecutive
 O's is: $number.\n");
10. close(F);
11. close(W);
... ..
```

*Zoologist*

*Zoology*

*Zoolympics*

*Zoom*

*Zoonomia*

*Zoonoses*

*Zooplankton*

*Zoot*

*Zooxanthellae*

*Zzaperoonies*

*The total number of words that have at least two consecutive O's is: 1718.*

**{*m*,*n*}** This expression means the number of a pattern is between *m* and *n* inclusive. The following program gets words with 2—3 initial consecutive vowel letters.

*vowel2\_3.pl*

```
1. open(F,"bncwordlist.txt") or die ("File can't be opened.\n");
2. open(W,">result.txt") or die ("Can't create file.\n");
3. while($word=<F>){
```

```

4. if($word=~m/^[aeiou]{2,3}/i){
5. $number++;
6. print W $word;
7. }
8. }
9. print(W "The total number of words that begin with 2--3 vowel letters is:
 $number.\n");
10.close(F);
11.close(W);

```

*Aa*

*Aaa*

*Aaaaa*

*Aaaaaaaaaahhhhhhhhhh*

*Aaaaaarrrrrgh*

*Aaaaah*

*Aaaaarrrrggghhh*

*Aaaaaw*

*Aaaaw*

*Aaaeeeyaaa*

*Aaah*

*Aaargh*

*Aachen*

*Aacr*

*Aad*

*... ..*

*The total number of words that begin with 2--3 vowel letters is: 1553.*

## 5.5 String manipulation functions and the *for* program control structure

Perl has the following string manipulation functions.

**index(*string,character*)** This function gets the position of the first *character* in *string*. If *character* occurs more than once, *index(string,character)* gets the position of the first occurrence. The position is measured from the left of *string* in number of characters (including white spaces).

```

$line="Peter ran to the door, and yelled at the dog in his pajamas, and the
dog ran away yelping.";
$position=index($line,'dog');

```

```
print $position;
41
```

**rindex**(*string,character*) This function measures the position of the last occurrence of *character* in *string*. However, if *character* occurs only once, *rindex(string,character)* and *index(string,character)* are the same.

```
$line="Peter ran to the door, and yelled at the dog in his pajamas, and the
dog ran away yelping.";
$position=rindex($line,'dog');
print $position;
69
```

**substr**(*string,m,n*) This function cuts *n* characters off *string* from position *m* measured from the left of *string* in number of characters.

```
$line="This is a demonstration of SUBSTR.";
$line1=substr($line,0,4);
$line2=substr($line,27,7);
print ("$line1\t$line2");
This SUBSTR.
```

Next, we'll look at the *for* program control structure. This structure is like the *while* control structure in that it creates a loop within which statements are executed. The syntax of the *for* structure is as follows:

```
for(n=m; n<x; n++){
 statements to be executed
}
```

Here we can also use *n<x*, *n--*, *n+=k*, *n-=k* etc. *m* is the initial value assigned to *n*; *n<x* or *n>x* is the condition: as long as *n<x* or *n>x*, statements between the curly brackets are executed, after which *n* is increased or decreased by 1; or by any other value, such as 2, 3, 4...etc. As soon as *n* equals *x*, the program goes out of the loop. In the following example, the initial value of *\$i* is set to 1 and is auto-increased by 1 until *\$i* equals 11:

```
for($i=0;$i<11;$i++){
 print $i;
}
012345678910
```



In the following example, the initial value of *\$i* is set to 20, and it's auto-decreased by 2 until it equals 10:

```
for($i=20;$i>10;$i-=2){
 print $i;
}
201816141210
```

The following script repeatedly cuts a character from *\$string* and outputs it to the screen until the value of *\$i* equals 0:

```
$string="This is a demonstration";
$stringlength=length($string);
for($i=$stringlength;$i>0;$i--){
 $cut=substr($string,0,1);
 $string=substr($string,1,$stringlength);
 print $cut."*";
}
T*h*i*s* *i*s* *a* *d*e*m*o*n*s*t*r*a*t*i*o*n*
```

Next we'll use *substr*, *rindex* and the *for* structure to write a program to divide *adventure.txt* into 20 text chunks of roughly equal length, and each of the chunks ends in a complete sentence.

```
dividefile.pl
1. open(F,"adventure.txt") or die ("File can't be opened.\n");
2. read(F,$text,150000);
3. $textlength=length($text);
4. $chunklength=$textlength/20;
5. for($number=1;$number<21;$number++){
6. $chunk=substr($text,0,$chunklength+110);
#The following statement gets the position of the last full stop in $chunk.
7. $laststop=rindex($chunk,")+1;
#The following assigns a chunk $lastop in length to $chunk.
8. $chunk=substr($text,0,$laststop);
#The following removes this chunk from $text.
9. $text=substr($text,$laststop);
#Statements 10--11 create output file names alice1.txt, alice2.txt, alice3.txt
#etc.
10. $output="alice".$number.".txt";
11. open(W,">$output")or die ("Can't create file.\n");
12. print W $chunk;
13. }
```

14. close(F);
15. close(W);

In this program, in statement 6 *\$chunk* is assigned a text chunk with a length of *\$chunklength+110*. Why 110 is added? This is because we can not ensure the chunk ends in a complete sentence, or worse, not even ends in a complete word. Suppose a chunk ends in *I think I could, if I only know how to begin.' For, you see, so many out-of-the-way things had happened lately, that Alice had begun to think that very few things indeed were really impossible. There seemed to be no use in waiting by the little door, so sh*, then the characters after the last full stop of this chunk should be removed so as to end this chunk in a complete sentence. And instead of assigning exactly *\$chunklength* characters to *\$chunk* from *\$text*, each chunk would be more or less 110 characters shorter than *\$chunklength*, and the last chunk cut would be much longer than *\$chunklength*. That's the logic behind the addition of 110; this value is a rough estimation.

## 5.6 Applications

In this section, we'll try to use what we've learned in this chapter to write some practical programs.

### 5.6.1 Extraction of POS tags

In the study of word class distribution, we need to extract POS tags from tagged texts. There are several tag sets used for marking the part of speech of a word. The CLAWS POS tag set is one of them. The following is a sentence of *adventure.txt* and is tagged with the CLAWS-5 tag set:

```
<s>
ALICE_NP1 'S_GE ADVENTURES_NN2 IN_II WONDERLAND_NP1
CHAPTER_NN1 I_ZZ1 Down_II the_AT Rabbit-Hole_NP1
Alice_NP1 was_VBDZ beginning_VVG to_TO get_VVI very_RG
tired_JJ of_IO sitting_VVG by_II her_APPGE sister_NN1
on_II the_AT bank_NN1 ,_, and_CC of_IO having_VHG
nothing_PN1 to_TO do_VDI :_: once_RR or_CC twice_RR
she_PPHS1 had_VHD peeped_VVN into_II the_AT book_NN1
her_APPGE sister_NN1 was_VBDZ reading_VVG ,_, but_CCB
it_PPH1 had_VHD no_AT pictures_NN2 or_CC
conversations_NN2 in_II it_PPH1 ,_, `_" and_CC what_DDQ
is_VBZ the_AT use_NN1 of_IO a_AT1 book_NN1 ,_, ' _GE
```

```

thought_NN1 Alice_NP1 `_" without_IW pictures_NN2 or_CC
conversation_NN1 ?_? ' _"
</s>

```

Now we'll write a program to extract the POS tags from the sentence.

```

getpostag.pl
1. open(F,"tagged.txt") or die("File does not exist.\n");
2. read(F,$text,7000);
3. open(W,">result.txt") or die("Can't create file.\n");
4. $text=~s/[.?!"':()_][.?!"':()_]/g;
5. $text=~s/<.+?>/g;
6. $text=~s/.*?_(\w+)\s.*?/1 /g;
7. $text=~s/\n/g;
8. print W $text;
9. close(F);
10.close(W);

```

In this program, statement 4 removes punctuation marks, which are in the form of `._, ?_?, !_!` etc. Statement 5 removes the non-textual sentence marker `<s>` and `</>`. Note the use of `.+?` to avoid greediness. Statement 6 extracts the POS tags, which are in the form of `_ NP1, _ NN2, _ VVD` etc, followed by a white space. Note the use of `.*?` to avoid greediness. `(\w+)\s` gets POS tags back-referenced by `\1`. Statement 7 removes line breaks. The following is part of the result, which are the POS tags of the above tagged sentence:

```

NP1 GE NN2 II NP1 NN1 ZZ1 II AT NP1 NP1 VBDZ VVG TO VVI
RG JJ IO VVG II APPGE NN1 II AT NN1 CC IO VHG PN1 TO VDI
RR CC RR PPHS1 VHD VVN II AT NN1 APPGE NN1 VBDZ VVG CCB
PPH1 VHD AT NN2 CC NN2 II PPH1 CC DDQ VBZ AT NN1 IO AT1
NN1 GE NN1 NP1 IW NN2 CC NN1

```

### 5.6.2 Making concordance for a text

The concordance of a word is the natural context on either side of the word. Usually the context, often referred to as the span, contains 3—5 words on either side, and the word with such contexts is called the key word. The key words are normally centre-justified. The following program makes concordance of every word in *adventure.txt*. The key words are all centre-justified, with a five-word span on either side.

```

concordance.pl
1. open(F,"adventure.txt") or die ("Can't open file!\n");

```

```

2. open(R, ">result.txt") or die("File does not exist!\n");
3. read(F,$text,150000);
4. $text=~s/[\n\-\-]/g;
5. $text=~tr/ / /s; #turn two or more consecutive spaces into one
6. $text='* * * * *.$text.' * * * * *';
7. $textlength=length($text);
8. while (length($text)>40){
9. $concordance=$text;
10. $concordance=~s/^(\\S+\\s){5})(\\S+)\\s(\\S+\\s){5}).*/$1$3$4/;
11. $leftconcordlength=length($1);
12. $spaceposition=index($text,' ');
13. $text=substr($text,$spaceposition+1,$textlength);
14. $centrejustify=' 'x(45-$leftconcordlength);
15. $concordance=$centrejustify.$1.(uc $3)'.$4;
16. print (R "$concordance\n");
17. }
18. close(F);
19. close(R);

```

In this program, statement 6 adds 5 asterisks as dummy words on both sides of *\$text* because the first 4 words and the last four words of *\$text* don't have enough words in their spans. Statements 8—17 are a loop, within which words are taken from *\$text* one by one, with a five-word span on either side. The concordance of a word is made by `^(\\S+\\s){5})(\\S+)\\s(\\S+\\s){5}` in statement 10. `^(\\S+\\s){5}` gets the first five words as the left span of the key word; the following `(\\S+)` gets the key word, and the rest the right five-word span. The left span, the key word and the right span are back-referenced by *\$1*, *\$3* and *\$4*. Note that in back-referencing, for nested bracketing, the sequence is in the following order:  $^1(2(2)^1)^3(3)^4(5(5)^5)^4$ , etc. Statement 11 measures the length of the left span; statements 12 gets the position of the space after the key word, and statement 13 removes this word together with the space from *\$text*. Statements 14—15 centre-justify the key word. i.e. *\$3*, which is converted into upper case. The concordance and the key word is outputted to *result.txt* in statement 16, after which the program goes back to statement 9 to make the concordance of the next word, until all the words in the text are exhausted.

### 5.6.3 Extraction of lexical bundles from texts.

In conversation and written discourse we often see word sequences such as *at the same time, it used to be, the end of the* and so on. These lexical sequences are called lexical bundles. Lexical bundles are recurring sequences of word forms in natural discourse. In *bundle.txt* there are some of the lexical bundles commonly

used in conversation. The following program extracts these lexical bundles from *adventure.txt* and arranges the extracted lexical bundles in the following form:

9. THE END OF THE

(1) and this time it vanished quite slowly, beginning with **THE END OF THE** tail, and ending with the grin, which remained some time after the rest of it had gone.

(2) `Now at OURS they had at **THE END OF THE** bill, "French, music, AND WASHING--extra.

(3) ' cried the Gryphon, and, taking Alice by the hand, it hurried off, without waiting for **THE END OF THE** song.

(4) ' `They're putting down their names,' the Gryphon whispered in reply, `for fear they should forget them before **THE END OF THE** trial.

*getbundle.pl*

```

1. open(F,"bundle.txt")or die ("bundle.txt can't be opened\n");
2. open(G,"adventure.txt")or die ("adventure.txt can't be opened\n");
3. read(G,$text,150000);
4. open(R,">result.txt") or die("File can't be created.\n");
5. $text=~s/\n//g;
6. $text=~tr// /s;
7. while ($bundle=<F>){
8. $getbundle=$textb=$text;
9. chomp $bundle;
10. if ($textb=~m/\b$bundle\b/i){
11. $bundletype++;
12. $bundleb=uc $bundle;
13. print (R "\n$bundletype. $bundleb\n");
14. }
15. while ($textb=~m/\b$bundle\b/i){
16. $bundletype++;
17. $getbundle=~s/(.*?\b$bundle\b.*?[.?!]).*\1/i;
18. $getbundle=~s/.*?[.?!](.*\b$bundle\b.*)\1/i;
19. $textb=~s/.*?\b$bundle\b.*?[.?!]/i;
20. $getbundle=~s/\b$bundle\b/ **$bundle** /i;
21. $getbundle=~s/\b$bundle\b/uc $bundle/e;
22. print (R " ($bundletype) $getbundle\n");
23. $getbundle=$textb;
24. }

```

```
25. $bundlefreq=0;
26. }
27. close(F);
28. close(G);
29. close(R)
```

This program opens two input files: *adventure.txt* and *bundle.txt*, and one output file, *result.txt*, and has two loops, the outer loop and the inner loop. The outer loop is between statements 7 and 26, in which lexical bundles are taken one by one from *bundle.txt* and searched for in *adventure.txt*; the inner loop is between statements 15 and 24, in which the occurrences of *\$bundle* is computed. Statement 8 assigns the value of *\$text* to *\$getbundle* and *\$textb*. The use of *\$textb* is to reserve the original value of *\$text*. Statement 9 removes the line break of *\$bundle*. Statement 10 checks whether *\$textb* has *\$bundle*. Statement 11 counts the number of different lexical bundles in *\$textb*. Statement 12 assigns the uppercase *\$bundle* to *\$bundleb*, which serves as the heading of a lexical bundle in the output file, under which the sentences that contains the bundle are listed. In statement 17 `~s/(.*?\b$bundle\b.*?[.?!]).*\n/` gets a text chunk from the beginning of *\$getbundle* to the sentence that contains the bundle. This chunk is further processed in statement 18, where only the sentence containing the bundle is kept, done by back-referencing, the rest all discarded. Statement 19 removes this bundle from *\$textb*. Statements 20—21 mark the extracted bundle with two \*'s on either side and turn it into upper case, both for easy viewing. Statement 23 assigns the value of *\$textb* to *\$getbundle* for the next instance of *\$bundle* in what remains of *\$textb*. After the entire contents of *\$textb* have been searched, the program goes out of the inner loop, and *\$bundlefreq* is reset to 0 in statement 25 for the next bundle. Then the program goes to statement 8 for a new bundle.

#### 5.6.4 A Chinese tokenizer

The Chinese language is different from the English language as far as the computer is concerned in that it's a double byte language; that is, a Chinese character has two bytes while an English letter has one. In addition, in a Chinese text, there are no word delimiters, while in English texts generally words are separated by a white space on either side. To make matters worse, many texts in Chinese have foreign words composed of one-byte characters, one-byte Arabic numerals, one byte punctuation marks, etc. The following is an example of a text in Chinese:

Perl是一种功能强大但简单易学的计算机语言， Perl现有5.6到Perl 5.10版本。Perl有些函数与C语言相似。Perl可用如WIN32、Macintosh、Linux、VMS等不同操作平台。Perl在语言和文学研究、语言和文学教学、字典编撰等诸多领域中有着极其广泛的应

用，例如统计词频，排序、计算语篇平均词长、句长、词汇密度，单词查询、研究单词搭配、覆盖率、出现概率、文体比较、句子结构、语法等等。可以进行极其复杂的语言处理，但编程非常简单，可处理任何自然语言。

Texts like the above are difficult to tokenize. One way to tokenize a Chinese text is to make continuous two-byte cuts from the start to the end. But, as shown above, many Chinese texts have one-byte characters. In the above example, there is the letter *C*, which is one byte long. If a two-byte cut is made, the result would be *C* plus the first half of the following Chinese character, and if this continues, the result would be a mess of garbled codes.

The following program takes the above into consideration and can properly tokenize Chinese texts mixed with one-byte characters.

*tokenchinese.pl*

```

1. open(F,"chinese.txt")or die("File does not exist.\n");
2. read(F,$text,2000);
3. open(R,">result.txt") or die("Can't create file.\n");
4. $text=~s/[\s\t\n]//g;
5. $text=~tr/ //s;
6. $linelength=length($text);
7. while(length($text)>0){
8. $getcharacter=substr($text,0,1);
9. if(ord($getcharacter)<127){
10. $text=substr($text,1,$linelength);
11. $characters.=$getcharacter;
12. }else{
13. $characters.=' ';
14. $getcharacter=substr($text,0,2);
15. $text=substr($text,2,$linelength);
16. $characters.=$getcharacter;
17. }
18. }
19. print(R $characters);
20. close(F);
21. close(R);

```

In the above program, statement 6 measures the length of *\$text*. Statements 7—18 are a loop, in which tokenization is done. Statement 8 assigns one byte from *\$text* to *\$getcharacter*. Statement 9 checks if *\$getcharacter* now is a one-byte character whose ASCII code value is less than 127, or half of a two-byte character, whose ASCII code value is greater than 127. If it's a one byte character, i.e. an English letter, statement 10 removes this one byte from *\$text*. Statement 11

puts this character to *\$character*, and then the program goes to statement 8 for the next character. However, if it's half of a Chinese character, its ASCII code value is greater than 127, the program goes to statement 13, which adds a white space after what have been stored so far in *\$character*. Statement 14 assigns a two-byte character to *\$getcharacter*, and this two-byte character is removed from *\$text* in statement 15. Statement 16 adds this two-byte character to *\$characters*. Then the program goes to statement 8 again to start the next round. The result is shown below:

*Perl 是一种功能强大但简单易学的计算机语言，Perl 现有 5.6 到 Perl5.10 版本。Perl 有些函数与 C 语言相似。Perl 可用如 WIN32、Macintosh、Linux、VMS 等不同操作平台。Perl 在语言和文学研究、语言和文学教学、字典编撰等诸多领域中有着极其广泛的应用，例如统计词频，排序、计算语篇平均词长、句长、词汇密度，单词查询、研究单词搭配、覆盖率、出现概率、文体比较、句子结构、语法等等。可以进行极其复杂的语言处理，但编程非常简单，可处理任何自然语言。*

Now all the Chinese words are separated by white spaces, but the one-byte characters are kept unchanged. Tokenized Chinese texts are extremely useful because nearly all the programs for processing English can be used for tokenized Chinese texts as well.

## Exercises

1. Write a program that marks all the word forms of *be* (including *be*) in *adventure.txt* with *\*\*\*\** on either side, and count the total number of it.
2. Write a program to pick out *as soon as* from *adventure.txt* and count its frequency.
3. Write a program to get all the collocations of *go* and its different word forms from *adventure.txt* with a five-word span on either side.
4. In *bncwordlist.txt* there are many out-of-dictionary words such as *Arimaddeyya*, *P400ps* etc. Write a program to remove these non-words as many as possible and put them into a file, and output in-dictionary words to another file. Check the results for errors and then improve the program to reduce the errors.



5. Write a program that can turn *adventure.txt* into a list of different word types with frequencies in the following form:

1630	the
4	adventure
46	happy

# 6 Arrays

An array is a structured data storage device. It's like a table in that it has sequenced rows in which to store data. Such rows are called array elements. The number of elements an array can have depends on the memory size of your computer. In this chapter we'll learn array creation, manipulation and application.

## 6.1 Array creation

In Perl, the name of an array is prefixed by `@`. For example, if we want to create an array to store the different chapters of a book, we can name it `@bookchapter` and then put the first chapter in its first element, the second in its second element and so on. Perl has a strange way of numbering the elements of an array. That is, element numbers start from 0 and these numbers are enclosed between a pair of square brackets. The elements of an array are prefixed by `$` followed by the array name and element number. So the first element of `@bookchapter` is `$bookchapter[0]`, the second `$bookchapter[1]`, the third `@bookchapter[2]` and so on. To change this default setting, use the special variable `$[` and assign the desired starting number to it. If we want the first element of an array to be 1 instead of 0, then simply assigns 1 to it: `$[=1`.

Arrays like `@bookchapter` are called one-dimensional arrays because the elements have no substructures. Arrays whose elements have hierarchical structures are called multi-dimensional arrays. Elements of a multi-dimensional array are in the form of `$arrayname[x][y]`, `$arrayname[x][y][z]` and so on.

### 6.1.1 One dimensional arrays

Now we'll create a one-dimensional array called `@shortarray` to hold the following words:

```
This
is
a
short
array
```

We can assign these words to the array manually one by one:

```
$shortarray[0]='This';
$shortarray[1]='is';
$shortarray[2]='a';
$shortarray[3]='short';
$shortarray[4]='array';
```

We can access any of the words stored in the array by referring to their element number:

```
print $shortarray[3]
short
print $shortarray[0]
This
print $shortarray[4]
array
```

We can also print out all the elements of the array:

```
print @shortarray
Thisisashortarray
```

We can input data to an array using round brackets:

```
@shortarray=('This','is','a','short','array');
print $shortarray[1];
is
print $shortarray[0];
This
print $shortarray[2];
a
print @shortarray;
Thisisashortarray
```

We can use the range operator within the brackets for consecutive values in the ascending order:

```
@shortarray=(a..z,A..Z);
print $shortarray[15];
p
print $shortarray[31];
F
print @shortarray
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
@shortarray=(80..90);
print $shortarray[1];
81
print $shortarray[6];
86
```

---

There is a special operator `$#` that returns the largest element number of an array. It is used as follows:

```
$#arrayname

@shortarray=(a..z,A..Z);
print $#shortarray;
51
```

Since the first element of an array is given the number of 0 by default, the number of elements in `@shortarray` is 52. If an array is empty, `$#` returns -1:

```
print $#emptyarray;
-1
```

Two or more arrays can be combined together using the following expressions:

```
@newarray=(@array1,@array2...@arrayn)
```

Look at the following examples:

```
@letters=(a..z);
@numbers=(0..9);
@alphanumeric=(@letters,@numbers);
print $alphanumeric[24];
y
print $alphanumeric[30];
4
```

The following function can reverse the elements of an array:

```
reverse(@arrayname)

@letters=(a..z);
@letters_reverse=reverse(@letters);
print $letters_reverse[0];
z
print $letters[2];
x
```

Of course this function can also reverse a string:

```
$reverseword=reverse("Perl");
```

```
print $reverseword;
 lreP
```

### 6.1.2 Multi-dimensional arrays

Multi-dimensional arrays are not as useful as one-dimensional arrays in language research. Next, we'll create a three-dimensional array. Suppose we have the following set of words:

Set 1:

Subset 1: apple, peach, apricot, pear

Subset 2: cabbage, onion, lettuce, spinach

Subset 3: pine, birch, elm, acacia

Subset 4 rose, carnation, tulip, daisy

Set 2:

Subset 1 bee, hornet, fly, mosquito

Subset 2 dove, eagle, owl, sparrow

Subset 3 chicken, duck, goose, turkey

Subset 4 carp, salmon, tuna, cod

This set of data has a three-level hierarchical structure: set->subset->word. Now we'll write a program in which to create a three-dimensional array called *@wordset* to store this set of data while keeping its structures unchanged.

*multiarray.pl*

```
1. open(W,">result.txt");
2. $wordset[0][0][0]='apple';
3. $wordset[0][0][1]='peach';
4. $wordset[0][0][2]='apricot';
5. $wordset[0][0][3]='pear';
6. $wordset[0][1][0]='cabbage';
7. $wordset[0][1][1]='onion';
8. $wordset[0][1][2]='lettuce';
9. $wordset[0][1][3]='spinach';
10. $wordset[0][2][0]='pine';
11. $wordset[0][2][1]='birch';
12. $wordset[0][2][2]='willow';
13. $wordset[0][2][3]='acacia';
14. $wordset[0][3][0]='rose';
15. $wordset[0][3][1]='carnation';
16. $wordset[0][3][2]='tulip';
17. $wordset[0][3][3]='daisy';
18. $wordset[1][0][0]='bee';
19. $wordset[1][0][1]='hornet';
```

```

20. $wordset[1][0][2]='butterfly';
21. $wordset[1][0][3]='mosquito';
22. $wordset[1][1][0]='dove';
23. $wordset[1][1][1]='eagle';
24. $wordset[1][1][2]='peacock';
25. $wordset[1][1][3]='sparrow';
26. $wordset[1][2][0]='chicken';
27. $wordset[1][2][1]='duckling';
28. $wordset[1][2][2]='goose';
29. $wordset[1][2][3]='turkey';
30. $wordset[1][3][0]='carp';
31. $wordset[1][3][1]='salmon';
32. $wordset[1][3][2]='barracuda';
33. $wordset[1][3][3]='shark';
34. $[=1;
35. for($i=1;$i<3;$i++){
36. print W "Set $i\n";
37. for($j=1;$j<5;$j++){
38. print W "Subset ($j) ";
39. for($k=1;$k<5;$k++){
40. print W "$k. $wordset[$i][$j][$k]\t";
41. }
42. print W "\n";
43. }
44. print W "\n";
45. }
46. close(W);

```

In the program, statements 2—17 input the data of Set 1, while statements 18—33 input the data of Set 2. Statement 34 changes the default starting element number from 0 to 1. Statements 35—45 output the contents of the array, keeping their original hierarchical structure. Note the use of the three *for* program control structures. The first *for* structure is between statement 35 and statement 45, for processing the sets, subsets and words. The second is between statement 37 and statement 43 for processing the subsets, and the third is between statement 39 and statement 41 for outputting the words. The result is shown below:

*Set 1*

```

Subset (1) 1. apple 2. peach 3. apricot 4. pear
Subset (2) 1. cabbage 2. onion 3. lettuce 4. spinach
Subset (3) 1. pine 2. birch 3. willow 4. acacia
Subset (4) 1. rose 2. carnation 3. tulip 4. daisy

```

*Set 2*

---

```

Subset (1) 1. bee 2. hornet 3. butterfly 4. mosquito
Subset (2) 1. dove 2. eagle 3. peacock 4. sparrow
Subset (3) 1. chicken 2. duckling 3. goose 4. turkey
Subset (4) 1. carp 2. salmon 3. barracuda 4. shark

```

### 6.1.3 Converting texts into arrays

Suppose we want to put *adventure.txt* into an array called *@aliceword*, with each word as an array element, and then output these words, none of the above ways are practical because the time and labour involved. In such cases, the following functions should be used for automatic array input and output.

**split(*delimiter*,*\$variable*)** This function splits the contents stored in *\$variable* into individual array elements at *delimiter*.

```

$line="This is a short array.";
@line=split(/ /,$line);

```

In the above, the variable *\$line* contains five words; the *split* function separates *\$line* by the white space into individual array elements of *@line*. The following outputs the specified elements of *@line*:

```

print ("$line[0]\n");
This
print ("$line[4]\n");
short

```

However, in the above example, outputting the elements of *@line* was still done manually. The following *foreach* function can be used for automatic array output:

```

foreach $variable(@arrayname){
 statements to be executed
}

```

The *foreach* function gets each of the elements of an array from the top to the bottom, and the element is then handled by the statements between the pair of curly brackets. Now we'll use the *split* function and the *foreach* function together for automatic array input and output:

```

$line="This is a short array.";
@line=split(/ /,$line);

```

```
foreach $word(@line){
print "$word\n";
}
This
is
a
short
array
```

In the above example, the contents of *\$line* are split at the white space and then assigned to *\$word* by the *split* function; *\$word* is then printed followed by a line break.

Next we'll write a program to turn each of the words in *adventure.txt* into an array element, count the total number of the words and then output these words to a file.

```
split.pl
1. open(F,'adventure.txt') or die("File cant be opened.\n");
2. open(W,'>result.txt') or die("Can't create file.\n");
3. read(F,$text,150000);
4. $text=tr/[,?";`!()\n\t*\-\]/ /g;
5. $text=~s/^ | $//;
6. @textarray=split(/ /,$text);
7. foreach $word(@textarray){
8. $number++;
9. print (W "$word\n");
10. }
11. print (W "The total number of words in text is: $number");
12. close(F);
13. close(W)
```

## 6.2 Functions for array operations

In this section, we'll look at some useful functions for array manipulations.

### 6.2.1 Functions for array input and output

There are functions for inputting data to and outputting data from an array. They are the following:



**push(@arrayname,string)** This function adds *string* as an element to *@arrayname*. The *push* function always adds an element to the bottom of an array.

```
push(@shortarray,'This');
push(@shortarray,'is');
push(@shortarray,'a');
push(@shortarray,'short');
push(@shortarray,'array');
print $shortarray[4];
array
```

**unshift(@arrayname,string)** This function adds *string* to the top of *@arrayname*:

```
unshift(@shortarray,'This');
unshift(@shortarray,'is');
unshift(@shortarray,'a');
unshift(@shortarray,'short');
unshift(@shortarray,'array');
print "$shortarray[0]\n";
array
print "$shortarray[1]\n";
short
print "$shortarray[2]\n";
a
print "$shortarray[3]\n";
is
print "$shortarray[4]\n";
This
```

**shift(@arrayname)** This function gets an element from the top of *@arrayname* and removes it from the array.

```
push(@shortarray,'This');
push(@shortarray,'is');
push(@shortarray,'a');
push(@shortarray,'short');
push(@shortarray,'array');
print shift(@shortarray)."\n";
This
print "$#shortarray\n";
3
```

```
print shift(@shortarray)."n";
is
print "$#shortarray\n";
2
print shift(@shortarray)."n";
a
print "$#shortarray\n";
1
print shift(@shortarray)."n";
short
print "$#shortarray\n";
0
print shift(@shortarray)."n";
array
print "$#shortarray";
-1
```

**pop(@arrayname)** This function gets an element of *@arrayname* from the bottom and removes it from the array.

```
push(@shortarray,'This');
push(@shortarray,'is');
push(@shortarray,'a');
push(@shortarray,'short');
push(@shortarray,'array');
print "$#shortarray\n";
4
print pop(@shortarray)."n";
array
print "$#shortarray\n";
3
print pop(@shortarray)."n";
short
print "$#shortarray\n";
2
print pop(@shortarray)."n";
a
print "$#shortarray\n";
1
print pop(@shortarray)."n";
is
print "$#shortarray\n";
0
```

```
print pop(@shortarray)."\n";
This
print $#shortarray;
-1
```

## 6.2.2 Array insertion, truncation and deletion

Array insertion, truncation and deletion are mainly done with the *splice* function.

**splice(@arrayname[,elementnumber[,n[,substituteelement]])** This function, when used without the optional arguments in the square brackets, empties *@arrayname*

```
$word ='dog cat mouse';
@wordarray=split(/ /,$word);
print $#wordarray;
2
splice(@wordarray);
print $#wordarray;
-1
```

Here originally *@wordarray* has three elements. The *splice* function removes everything from it, resulting in -1, meaning empty.

Used with *elementnumber*, the *splice* function truncates *@arrayname* at *elementnumber*:

```
$word ='dog cat mouse';
@wordarray=split(/ /,$word);
print @wordarray;
dogcatmouse
splice(@wordarray,1);
print @wordarray;
dog
```

In the above example, the *splice* function truncates *@wordarray* at *@wordarray[1]*, leaving only *@wordarray[0]*, which is *dog*.

The *splice* function, used with *elementnumber*, *n*, removes *n* elements of *@arrayname* starting from *elementnumber*:

```
$word ='dog cat mouse';
@wordarray=split(/ /,$word);
splice(@wordarray,0,2);
```

```
print @wordarray;
mouse
```

Here, the *splice* function removes two elements from *@wordarray*, that is, from *\$wordarray[0]* to *\$wordarray[1]*, leaving only *\$wordarray[2]*, which is *mouse*.

The *splice* function, used with *elementnumber*, *n*, *substituteelement*, replaces *n* elements of *@arrayname* starting from *elementnumber* with one element *substituteelement*.

```
$word='dog cat mouse';
@wordarray=split(/ /,$word);
splice(@wordarray,0,2,'duck');
print @wordarray;
duckmouse
print $wordarray[0];
duck
print $wordarray[1];
mouse
print $#wordarray;
1
```

In the above, the *splice* function replaces the two elements *\$wordarray[0]* and *\$wordarray[1]* with *duck*, which is *\$wordarray[0]* now.

### 6.2.3 Sorting an array

Perl has a built-in function for sorting strings and numerals either in the ascending or descending order. The following is the sort function for sorting string elements of an array in the ascending order:

```
sort(@arrayname)
```

Look at the following:

```
$string='this is a demonstration of the sort function';
@array=split(/ /,$string);
@array=sort(@array);
foreach $word(@array){
print "$word\n";
}
```

The result is as follows:

```

a
demonstration
function
is
of
sort
the
this

```

The above sorting example can also be written as follows and the result is the same:

```

$strings='this is a demonstration of the sort function';
@array=split(/ /,$strings);
foreach $word(sort @array){
print "$word\n";
}

```

The following is for sorting strings in the descending order:

**sort({\$b cmp \$a} @arrayname)**

```

$strings='this is a demonstration of the sort function';
@array=split(/ /,$strings);
foreach $word(sort{$b cmp $a} @array){
print "$word\n";
}
this
the
sort
of
is
function
demonstration
a

```

The above two forms of the *sort* function are for sorting strings and can't be use to sort numbers. Look at the following example:

```

$numeral='1,2,3,4,10,12,21,35';
@array=split(/,/, $numeral);
foreach $number(sort @array){

```

```
print "$number\n";
}
1
10
12
2
21
3
35
4
```

This is because these numbers were treated as strings and sorted according to their ASCII code value. To sort numbers, the following form of the sort function should be used:

`sort({$a<=>$b}@arrayname)` This is for sorting numbers in ascending order.

`sort({$b<=>$a}@arrayname)` This is for sorting numbers in descending order.

```
$numeral='1,2,3,4,10,12,21,35';
@array=split(/,/, $numeral);
foreach $number(sort{$a<=>$b} @array){
print "$number\n";
}
1
2
3
4
10
12
21
35
```

#### 6.2.4 The anonymous variable and the *join*, *map* and *grep* functions

In Perl, there is a special variable `$_`. It's known as the anonymous variable because it can be used to stand for some variables such as the one getting input from a file handle, the one in the *foreach* function, and those in some functions that use a single argument. In such situations, `$_` is often omitted. However, the

use of the anonymous variable often makes a program difficult to understand. So avoid it whenever possible. The following example uses the anonymous variable:

```
open(F,"bncwordlist.txt") or die ("File can't be opened.\n");
while(<F>){
print "$_\n";
}
close(F);
```

In the above example, Perl automatically assigns the contents of the file handle *F* to *\$\_* in *while(<F>)* behind the scenes. In the following example, no overt *\$\_* is seen.

```
$string='This shows the use of the anonymous variable';
@array=split(/ /,$string);
foreach (@array){
print;
print "\n";
}
This
shows
the
use
of
the
anonymous
variable
```

Here, *foreach (@array)* is actually *foreach \$\_(@array)*, and the statement *print* is actually *print \$\_*.

Next, let's look at the *join*, *map* and *grep* functions.

**join**(*joiningcharacter,@arrayname*) This function joins the individual elements of *@arrayname* with *joiningcharacter*. Look at the following examples:

```
$string='Perl for quantitative linguistics';
@array=split(/ /,$string);
print @array;
Perlforquantitativelinguistics
```

```
$string='Perl for quantitative linguistics';
@array=split(/ /,$string);
```

```
print join("-",@array);
Perl-for-quantitative-linguistics
```

```
$string='Perl for quantitative linguistics';
@array=split(/ /,$string);
print join("\n",@array);
Perl
for
quantitative
linguistics
```

**map**(*{expression}*,@*arrayname*) This function uses *expression* to manipulate the elements of *@arrayname*. The anonymous variable *\$\_* is often used to stand for each of the elements.

```
$string='Perl for quantitative linguistics';
@array=split(/ /,$string);
print map ({uc $_."\n"} @array);
PERL
FOR
QUANTITATIVE
LINGUISTICS
```

In the above example, the expression used in the *map* function is *uc \$\_."\n"* , which capitalizes *\$\_* standing for each of the array elements, and puts a line break after each of the elements.

```
$string='Perl for quantitative linguistics';
@array=split(/ /,$string);
print map ({substr($_,0,1)." "} @array);
P f q l
```

The expression in the above *map* function is *substr(\$\_,0,1)." "*. It first uses the *substr* function to get the first letter of each of the elements of *@array*, and then puts a space after each of the elements.

In the following example, the *map* function is nested within the *join* function which separates the result of the *map* function with */*. The expression within the *map* function multiplies each of the array elements by 4:

```
$numeral='1, 2, 3, 4, 5, 6, 7, 8, 9, 10';
@array=split(/,,$numeral);
print join("/",map({4*$_} @array));
```



---

4/8/12/16/20/24/28/32/36/40

**grep**(*{pattern}*,@*arrayname*) This function searches for *pattern* in @*arrayname*. In the following, the *grep* function gets the element of @array whose length is 12 letters. Here the double equal signs must be used:

```
$string='Perl for quantitative linguistics';
@array=split(/ /,$string);
print grep({length($_)==12}@array);
quantitative
```

In the following, the *grep* function picks the word that has four letters:

```
$string='Perl for quantitative linguistics';
@array=split(/ /,$string);
print grep({/^b\w{4}\b/} @array);
Perl
```

### 6.3 Combining identical array elements and random sampling from an array

In this section we'll look at how to combine identical elements of an array and calculate their frequencies, and how to do random sampling from an array.

Suppose we have the following string *Perl is good for Quantitative Linguistics and Perl is good for general linguistics too* and want to put the words in an array, and then get the occurrences of each of the words. The following does this:

*combineelement1.pl*

```
1. $string='Perl is good for Quantitative Linguistics and Perl is good for
 general linguistics too';
2. @temp=split(/ /,lc $string);
3. @array=sort(@temp);
4. $freq=1;
5. for($i=0;$i< $#array+1;$i++){
6. if($array[$i+1]eq $array[$i]){
7. $freq++;
8. }else{
9. $word_freq=$array[$i]."\t".$freq;
10. print "$word_freq\t\n";
11. $freq=1;
12. }
```

---

```
13. }
```

The combination of identical elements and the calculation of their frequency are done by statements 6—7. The result is shown below:

```
And 1
For 2
General 1
Good 2
Is 2
Linguistics 2
Perl 2
Quantitative 1
```

However, the result doesn't look very tidy. Perl has the **Text::Tabs** module that can set tab length. We can call it in a program as shown below:

```
use Text::Tabs;
$tabstop=n;
... ..
print FILEHANDLE expand(expression);
```

Now we'll set the tab length to 15 using the above and see the result:

```
combineelement2.pl
1. use Text::Tabs;
2. $tabstop=15;
3. $string='Perl is good for Quantitative Linguistics and Perl is good for
 general linguistics too';
4. @temp=split(/ /,lc $string);
5. @array=sort(@temp);
6. $freq=1;
7. for($i=0;$i< $#array+1;$i++){
8. if($array[$i+1]eq $array[$i]){
9. $freq++;
10. }else{
11. $word_freq=$array[$i]."\t".$freq;
12. print W expand("$word_freq\t\n");
13. $freq=1;
14. }
15. }
And 1
```

---

<i>For</i>	2
<i>General</i>	1
<i>Good</i>	2
<i>Is</i>	2
<i>Linguistics</i>	2
<i>Perl</i>	2
<i>Quantitative</i>	1
<i>Too</i>	1

The following expression randomly draws an element from an array:

```
(@arrayname)[arraylength*rand]
```

*arraylength* is the number of elements in *@arrayname*.

```
$string='Perl is good for Quantitative Linguistics and Perl is good for
general linguistics too';
@array=split(/ /,lc $string);
$word=(@array)[14*rand];
print "$word";
is
```

Each time we run the above, an element is randomly picked from *@array*. Next, we'll randomly select 10 words from *bncwordlist.txt*.

```
randomsample.pl
1. open(F,"bncwordlist.txt") or die("Can't open file.\n");
2. read(F,$wordlist,900000);
3. open(W,">result.txt") or die ("Can't create file.\n");
4. @wordarray=split(/\n/,$wordlist);
5. $length=$#wordarray+1;
6. for($i=0;$i<10;$i++){
7. $word=(@wordarray)[$length*rand];
8. print(W "$word\n");
9. }
10.close(F);
11.close(W);
Reselection
Costumier
Heroism
Flannan
Shatt
```

*Nastier*  
*Bestial*  
*Palau*  
*Granada*  
*Bonce*

## 6.4 Applications

In this section we'll look at four practical programs that use arrays and some of the functions we've learned.

### 6.4.1 Selecting words from a wordlist

First we'll use the *grep* function to select words between length 2 and 5 in letters from *bncwordlist.txt* and output these words to a file arranged in the following format:

```
WORDS BEGINNING WITH A
Length 2
Worda, wordb, wordc...
... ..
The total number of words beginning with A with length 2 is:...
Length3
worda, wordb, wordc...
... ..
The total number of words beginning with A with length 3 is:...
... ..
WORDS BEGINNING WITH B
... ..
Length 5
worda, wordb, wordc...
... ..
The total number of words beginning with B with length 5 is:...
... ..
grepwords.pl
```

1. open(F,'bncwordlist.txt') or die("File does not exist.\n");
2. open(W,'>result.txt') or die ("Unable to create file.\n");
3. read(F,\$text,900000);
4. @temp=split(/\n/,\$text);
5. for(\$i=65;\$i<91;\$i++){ #for A to Z
6. \$char=chr(\$i);
7. print W "WORDS BEGINNING WITH \$char:\n";

```

8. for($j=2;$j<6;$j++){ #for words with length 2—5
9. print W "LENGTH $j\n";
#Note the use of the double equal signs used in the following statement.
10. $words=join(" ",grep({length($_)==$j and /^$char/} @temp))."\n";
11. $wordnumber=(($words=~s//g)+1); #counting number of such words
12. print W "$words";
13. print W "Total number of word beginning with $char with length $j is:
 $wordnumber\n\n";
14. }
15. print W "\n";
16. }
17. close(F);
18. close(W);

```

Part of the result is shown below:

```

WORDS BEGINNING WITH A:
LENGTH 2
A1 A2 A3 A4 A5 A6 A7 A8 A9 Aa Ab Ac Ad Ae Af Ag Ah Ai Aj Ak
Al Ao Ap Ar As At Au Av Aw Ax Ay Az
Total number of word beginning with A with length 2 is: 32

```

### 6.4.2 Turning a text into bigrams

In language studies, language teaching and natural language processing, we often need to separate a text or a corpus into  $N$ -grams, i.e., bigrams, trigram and so on. To turn a text into  $N$ -grams, we just pair every word in the text with its immediate following  $N-1$  words. For example, we can turn *Perl is good for Quantitative Linguistics* into the following bigrams:

```

Perl is
is good
good for
for Quantitative
Quantitative Linguistics.

```

Now we'll write a program to turn *adventure.txt* into a set of bigrams. Punctuation marks are considered as words so that we can see what words are often associated with them.

```

bigram.pl
1. open(F,"adventure.txt")or die("file can't be opened.\n");
2. read(F,$text,150000);
3. $text=~s/([.,`?!";])+/ $&/g; #note the space preceding $&
4. $text =~s/\n+/ /g;

```

```
5. $text =~tr/ / /s;
6. $text=~s/^ //g; #remove initial space in $text
7. @wordlist=split(/ /,$text);
8. for($i=0;$i<$#wordlist;$i++){
9. for($j=0;$j<2;$j++){
10. $bigram.=" ".$wordlist[$i]; #separate two words with a space
11. $i++;
12. }
13. push(@bigramarray,$bigram);
14. $i-=2;
15. $bigram="";
16. }
17. open(W,">bigram.txt") or die("Can't create file.\n");
18. while($#bigramarray>=0){
19. $getbigram=shift(@bigramarray);
20. print(W "$getbigram\n");
21. }
22. close(F);
23. close(W);
```

In this program, statements 7—16 get pairs of consecutive array elements to make bigrams, which are stored in *@bigramarray*. Note *\$i-=2* in statement 14, which decreases *\$i* by 2 because the value of *\$i* has been increased by 2 between statements 9—12 that make bigrams. Statement 15 empties *\$bigram* to make room for the next bigram. Statements 17—21 output the bigrams stored in *@bigramarray* to *bigramtext.txt*. Part of the result is shown below:

```
ALICE'S ADVENTURES
ADVENTURES IN
IN WONDERLAND
WONDERLAND CHAPTER
CHAPTER I
I Down
Down the
the Rabbit-Hole
Rabbit-Hole Alice
Alice was
was beginning
beginning to
to get
get very
very tired
```

### 6.4.3 Turning a text into a list of word types with frequencies

The following program turns *adventure.txt* into a list of word types with their frequencies, and computes the total number of word types in it.

```
wordtype.pl
1. use Text::Tabs;
2. $tabstop=30;
3. open(F,'adventure.txt') or die("File does not exist!\n");
4. open(W,'>wordlist.txt') or die ("Unable to create file!\n");
5. read(F,$text,150000);
6. $text=~tr/["?";!"*_()\n\-\[\]]/ /s;
7. $text=~s/^ | $//g;
8. @temp=split(/ /,lc $text);
9. $wordnumber=$#temp+1;
10. @words=sort(@temp);
11. $freq=1;
12. for($i=0;$i<$#words+1;$i++){
13. if($words[$i+1]eq $words[$i]){
14. $freq++;
15. }else{
16. $word_freq=$words[$i]."\t".$freq;
17. $typenumber++;
18. print W expand("$word_freq\t\n");
19. $freq=1;
20. }
21. }
22. print (W " _____\n\n");
23. print (W "The total number of word tokens is: $wordnumber\n");
24. print (W "The total number of word types is: $typenumber\n");
25. close(F);
26. close(W);
```

The following is part of the result:

```
... ..
yesterday 3
yet 25
you 410
young 5
your 63
yours 3
yourself 10
youth 6
```

---

<i>zealand</i>	<i>1</i>
<i>zigzag</i>	<i>1</i>

---

*The total number of word tokens is: 27285*

*The total number of word types is: 2570*

#### 6.4.4 Computing sentence length distribution

Sentence length refers to the number of words a sentence has. The following program divides *adventure.txt* into sentences and then computes the distribution of sentence length.

*sentlength.pl*

```

1. open(F,'adventure.txt') or die("File does not exist.\n");
2. open(R,'>sentence.txt') or die("Can't create file.\n");
3. open(W,'>slength.txt') or die ("Unable to create file.\n");
4. read(F,$text,150000);
5. $text=~s/(Mr|Mrs)\.\/1/g; #remove full stop after Mr and Mrs
6. $text=~tr\/.\/s; #turn two or more consecutive full stops into one
7. $text=~tr\/ / /s; # turn two or more consecutive spaces into one
#In statement 9, the punctuation marks such as .?! and their combinations
#with other mark, such as ." etc are regarded as sentence delimiters and are
#replaced by such marks followed by \n, which is also regarded here as a
#sentence delimiter. \n will be used in statement 11 to split $text into indi-
#vidual sentences.
8. $text=~s\/"."|\.|.\`|\.\)\|?"|\?|\?|\?)\|!"\!|\!|\.\|?\|!/$&\n/g;
#The following statement changes line break followed by one or more
#spaces and another line break into a single line break.
9. $text=~s\/\n\s*\n\/n/g;
#The following statement puts individual sentences into @sentence.
10. @sentence=split(\n/,$text);
11. for($i=0;$i<$#sentence+1;$i++){
12. $sentence[$i]=~s\/^ | $//g; #remove space at beginning and end of
 sentence
#The following statement counts number of words in sentence.
13. $sentlength=($sentence[$i]=~tr\/ / /s)+1;
#Sentence length is put in @lengtharray for later use.
14. push(@lengtharray,$sentlength);
15. $sentnumber++;
16. print R "$sentnumber\t$sentlength\t$sentence[$i]\n";
17. $wordnumber+=$sentlength;
18. }

```



```

#In statement 19 @lengtharray_sort contains the sorted sentence length.
19. @lengtharray_sort=sort({$a<=>$b})(@lengtharray);
#Statements 20—29 compute sentence length distribution.
20. $freq=1; # $freq holds the frequency of a sentence length
21. for($i=0;$i< $#lengtharray_sort+1;$i++){
22. if($lengtharray_sort[$i+1]eq $lengtharray_sort[$i]){
23. $freq++;
24. }else{
25. $sentlength_freq=$lengtharray_sort[$i]."\t".$freq;
26. print W "$sentlength_freq\t\n";
27. $freq=1;
28. }
29. }
30. $average=$wordnumber/$sentnumber;
31. print W "The total number of words in text is: $wordnumber\n";
32. print W "The total number of sentences is: $sentnumber\n";
33. print W "The average sentence length is: $average";
34. close(F);
35. close(R);
36. close(W);

```

The results are stored in two files: *sentence.txt* and *length.txt*. The former stores individual sentences preceded with sentence number and sentence length, the latter the distribution of sentence length.

## Exercises

1. Assign each of the following words *Perl for quantitative linguistics* to elements of an array and then output the contents of these elements using *split*, *push*, *shift*, *pop* and *unshift*.
2. Put all the words in *bncwordlist.txt* in an array and then output all the words in descending order.
3. Modify *bigram.pl* so that it can make trigrams from *adventure.txt*.
4. Write a program to turn *adventure.txt* into bigrams and then compute the frequency of each of the bigrams. Set the tab length to 30.
5. Write a program to compute word length (measured in number of letters) distributions of *adventure.txt* and the average word length.

# 7 Hash tables

In Perl there is a very useful data structure called hash tables, also known as hashes. A hash is like an array in that it has cells that can store different types of data. These cells are called hash elements. However, unlike the elements of an array, hash elements are named, not numbered. Hashes are prefixed with the % sign, e.g., %wordlist, %frequency etc, and the individual hash elements are prefixed with the dollar sign \$, followed by the hash name and the element names. Hash element names are called keys and are put between a pair of curly brackets, e.g. \$wordlist{apple}, %frequency{bed}, etc, and the contents of a hash key are called values. An array can be compared to a chest of drawers whose drawers are numbered 0, 1, 2, ..., and we can put different things in these numbered drawers and access them by using the drawer numbers. A hash can be seen as a chest of drawers whose drawers have names instead of numbers.

## 7.1 Hash input and output

In this section we'll consider ways of inputting data to and outputting data from a hash. Data can be inputted either manually, or automatically with the help of an array or the *split* function.

### 7.1.1 Manual input and output

One of the ways to create a hash is by manually assigning values to its elements. For example, the following creates a hash called *profession*:

```
$profession{Peter}="lecturer";
$profession{Sally}="student";
$profession{John}="professor";
$profession{Mary}="secretary";
$profession{Tom}="student";
$profession{Joe}="assistant";
```

The entire hash is called %*profession*. The following output the contents of the individual elements of %*profession*.

```
print $profession{Peter};
lecturer
print $profession{Sally};
student
print $profession{John};
```

```

professor
print $profession{Tom};
student
print $profession{Joe};
assistant

```

Note how the entire hash is outputted:

```

print %profession;
JoeassistantSallystudentJohnprofessorMarysecretaryPeterlectu
rerTomstudent

```

We can use the *join* function to separate the keys and values:

```

$profession{Peter}="lecturer";
$profession{Sally}="student";
$profession{Peter}="professor";
$profession{Mary}="secretary";
$profession{Peter}="student";
$profession{Joe}="assistant";
$folks=join(" ",%profession);
print $folks;
Joe assistant Sally student John professor Mary secretary
Peter lecturer Tom student

```

If we create two or more identical keys in a hash, only one will be kept in the hash, the rest will be automatically discarded, and we don't know beforehand which one will be kept.

```

$profession{Peter}="lecturer";
$profession{Sally}="student";
$profession{Peter}="professor";
$profession{Mary}="secretary";
$profession{Peter}="student";
$profession{Joe}="assistant";
print join(" ",%profession);
Joe assistant Sally student Mary secretary Peter student

```

Here, only *\$profession{Peter}* and its value *student* is kept; *\$profession{Peter}* and its value *lecturer* was discarded. Note that here the order of the keys with their values is randomly arranged by Perl.

We can also use the following way to create a hash:

```
%profession=(Peter,lecturer,Sally,student,John,professor,Mary,secretary,
Tom, student,Joe, assistant);
print $profession{Tom};
 student
print $profession{Mary};
 secretary
```

The following create a hash with a list of words as its keys and word frequencies as the values.

```
$frequency{apple}=6;
$frequency{apricot}=1;
$frequency{banana}=2;
$frequency{peach}=12;
$frequency{papaya}=3;
$frequency{grape}=20;
print $frequency{banana};
 2
print $frequency{apple};
 6
print $frequency{peach};
 12
```

In the above examples, the keys and values are all manually created. If we want to turn an entire text into a hash, the *split* function is in order. Note the white space between the two slashes:

```
$wordclass="Book Noun Large Adjective The Article Slowly Adverb But
Conjunction";
%hash=split(/ /,$wordclass);
print join(" ",%hash);
 But Conjunction Book Noun Slowly Adverb The Article Large
 Adjective
```

When the *split* function is used to turn a text into a hash, the text is cut into pairs of words from the beginning to the end, with the first word of a pair serving as the key, the second the value. If the number of words of a text is odd, then the last word is the key without a value.

```
$wordclass="Book Noun Large Adjective The Article Slowly Adverb But";
%hash=split(/ /,$wordclass);
print join(" ",%hash);
 Book Noun Large Adjective The Article Slowly Adverb But
```

As mentioned above, a hash key is unique within a hash; that is, each key must be different from others. So when using *split* to turn a text into a hash, some words may be automatically discarded by Perl to avoid identical keys. Look at the following:

```
$wordclass="Book Noun Book Verb Large Adjective The Article Slowly
Adverb But Conjunction";
%hash=split(/ /,$wordclass);
print join(" ",%hash);
But Conjunction Book Verb Slowly Adverb The Article Large
Adjective
```

The pair *Book Noun* is discarded because this *Book* would be the key of *Noun*, but the hash has chosen the other *Book* whose value is *Verb*.

### 7.1.2 Hash input and output using arrays and functions

Another way to create a hash is using an array. The following example automatically turns each of the elements of *@array* into a key of *%wordlist*, and each of the keys is given the value of 1:

```
$array[0]="apple";
$array[1]="banana";
$array[2]="orange";
$array[3]="grape";
foreach $word(@array){
 $wordlist{$word}=1;
}
print join(" ",%wordlist);
banana 1 apple 1 orange 1 grape 1
```

Here, we can also use *\$wordlist{\$word}++* instead of *\$wordlist{\$word}=1*:

```
$array[0]="apple";
$array[1]="banana";
$array[2]="orange";
$array[3]="grape";
foreach $word(@array){
 $wordlist{$word}++;
}
print join(" ",%wordlist);
banana 1 apple 1 orange 1 grape 1
```

Look at the following example:

```
$array[0]="apple";
$array[1]="banana";
$array[2]="orange";
$array[3]="apple";
$array[4]="orange";
$array[5]="grape";
$array[6]="banana";
$array[7]="orange";
$array[8]="orange";
foreach $word(@array){
$wordlist{$word}++;
}
print $wordlist{apple};
2
print $wordlist{grape};
1
print join(" ",%wordlist);
banana 2 apple 2 orange 4 grape 1
```

Here `$wordlist{$word}++` counts the occurrence of each of the keys of `%wordlist` and assigns the occurrence to the corresponding key as its value.

To get all the keys of a hash without their values, Perl provides the `keys()` function, which is in the following form:

**keys(%hashname)**

```
$array[0]="apple";
$array[1]="banana";
$array[2]="orange";
$array[3]="apple";
$array[4]="orange";
$array[5]="grape";
$array[6]="banana";
$array[7]="orange";
$array[8]="orange";
foreach $word(@array){
$wordlist{$word}++;
}
print join(" ",keys(%wordlist));
banana apple orange grape
```

To get the individual keys and their corresponding values, we can use the *foreach()* function together with *keys()*:

```
$array[0]="apple";
$array[1]="banana";
$array[2]="orange";
$array[3]="apple";
$array[4]="orange";
$array[5]="grape";
$array[6]="banana";
$array[7]="orange";
$array[8]="orange";
foreach $word(@array){
 $wordlist{$word}++;
}
foreach $word(keys(%wordlist)){
 print "$word\t$wordlist{$word}\n";
}
banana 2
apple 2
orange 4
```

In the above, *foreach \$word(keys(%wordlist))* assigns the keys of *%wordlist* one by one to *\$word*, while *print "\$word\t\$wordlist{\$word}\n"* prints out the keys and their corresponding values, separated with a line break.

We can also sort the keys:

```
... ..
foreach $word(sort(keys(%wordlist))){
 print "$word\t$wordlist{$word}\n";
}
apple 2
banana 2
grape 1
orange 4
```

The *reverse()* function we learned in the last chapter can also be used to turn values into keys and keys into values:

```
... ..
%wordlist=reverse(%wordlist);
foreach $word(keys(%wordlist)){
 print "$word\t$wordlist{$word}\n";
}
4 orange
```

```

1 grape
2 banana

```

Note 2 *apple* are missing because after reversion, 2 became a key, and there is another key that uses 2 as its name, so 2 *apple* were discarded.

Using a hash and the *keys* function, it would be very easy to turn a text into a list of individual word types with their frequencies. The following short program does this for *adventure.txt*:

```

wordtype_hash.pl
1. use Text::Tabs;
2. $tabstop=30;
3. open(F,'adventure.txt') or die("File does not exist.\n");
4. open(W,'>wordlist.txt') or die ("Unable to create file.\n");
5. read(F,$text,150000);
#The following statement replaces non-alphanumeric characters with
#spaces. Note the space after -; this ensures only one space on either side of
#a word.
6. $text=~tr/[.,?";`':!()><+&^%*{ }_~\|/\\n\t[\]\@#\$\-]/ /s;
7. $text=~s/^ | $//g;
8. @temp=split(/ /,lc $text);
9. foreach $word(@temp){
10. $wordnumber++; #count total number of word tokens
11. $wordtype{$word}++; # get word frequency
12. }
13. foreach $type(sort(keys(%wordtype))){
14. $typenumber++; #count number of word types
15. print W expand"$type\t$wordtype{$type}\n";
16. }
17. print (W " _____\n\n");
18. print (W "The total number of word tokens is: $wordnumber\n");
19. print (W "The total number of word types is: $typenumber\n");
20. close(F);
21. close(W);

```

In this program, statements 9—12 turn the array elements into keys of a hash, with the frequencies as the values. Statement 13 sorts the keys and assigns them one by one to *\$type*. Statement 15 prints out the keys, followed by a 30-space tab, and the frequencies. The following is part of the result:

```

you 410
young 5
your 63
yours 3

```



---

<i>yourself</i>	<i>10</i>
<i>youth</i>	<i>6</i>
<i>zealand</i>	<i>1</i>
<i>zigzag</i>	<i>1</i>

---

*The total number of word tokens is: 27285*

*The total number of word types is: 2570*

### 7.1.3 The use of *values()*, *each()*, *exist()* and *delete()*

**values(%hashname)** This function gets the value of a hash key. The following prints out the value of each of the elements of a hash:

```
$wordlist{apple}=2;
$wordlist{grape}=1;
$wordlist{orange}=4;
print join("\n",values(%wordlist));
2
4
1
```

The *values()* function can also be used with *foreach()* in the following form:

```
foreach $variable(values(%hashname)){
 statements to be executed
}
```

```
$wordlist{apple}=2;
$wordlist{grape}=1;
$wordlist{orange}=4;
foreach $freq(values(%wordlist)){
 print "$freq\n";
}
2
4
1
```

**each(%hashname)** This function gets one of the keys of a hash and its value.

```
$wordlist{apple}=2;
$wordlist{grape}=1;
```

```
$wordlist{orange}=4;
print join(" ",each(%wordlist));
apple 2
```

We can use *each()* to assign a key and its value respectively to two variables:

```
$wordlist{apple}=2;
$wordlist{grape}=1;
$wordlist{orange}=4;
($word,$freq)=each(%wordlist);
print "$word\t$freq\n";
apple 2
```

The following outputs every key and its value of *%wordlist*:

```
$wordlist{apple}=2;
$wordlist{grape}=1;
$wordlist{orange}=4;
while(($word,$freq)=each(%wordlist)){
print "$word\t$freq\n";
}
apple 2
orange 4
grape 1
```

**exists(\$hashname{hashelement})** This function checks whether a specified element exists in a hash.

```
$wordlist{apple}=2;
$wordlist{grape}=1;
$wordlist{orange}=4;
if(exists($wordlist{grape})){
print "The element exists; it occurred $wordlist{grape} time(s).";
}
The element exists; it occurred 1 time(s).
```

**delete(\$hashname{hashelement})** This function deletes the specified element of a hash.

```
$wordlist{apple}=2;
$wordlist{grape}=1;
$wordlist{orange}=4;
delete($wordlist{apple});
```

```

foreach $word(keys(%wordlist)){
print "$word\t$wordlist{$word}\n";
}
orange 4
grape 1

```

To empty a hash, use the following expression:

```

%hashname=()

$wordlist{apple}=2;
$wordlist{grape}=1;
$wordlist{orange}=4;
%wordlist=();
print %wordlist;

```

Nothing is printed out because *%wordlist* is now empty.

## 7.2 Hash operations

In this section we'll look at how to convert hash elements into an array and the manipulation of hash elements within one or more hashes.

### 7.2.1 Converting hash elements into an array

In *wordtype\_hash.pl* we saw how to put array elements into a hash. This process can be reversed, that is, a hash can be turned into an array. One of the applications of putting a hash into an array is for sorting the values of a hash.

```

hash_to_array1.pl
1. $word{book}="N";
2. $word{hot}="Adj";
3. $word{desk}="N";
4. $word{read}="V";
5. $word{good}="Adj";
6. $word{at}="Prep";
7. while(($word,$pos)=each(%word)){
8. push(@word,"$pos\t$word");
9. }
10. foreach $word(sort @word){
11. print "$word\n";

```

---

12. }

This program puts six words and their respective parts of speech in a hash. The hash is then converted into an array by statement 8, which combines *\$pos*, a tab, and *\$word* as an array element and is then pushed into an array called *@word*. Statements 10—11 print out the array elements sorted in ascending order. The result is as follows:

```

Adj good
Adj hot
N book
N desk
Prep at
V read

```

The above method can be used to output a wordlist stored in a hash with the frequencies sorted in descending order.

```

hash_to_array2.pl
1. $word{book}=20;
2. $word{hot}=45;
3. $word{desk}=12;
4. $word{read}=24;
5. $word{good}=67;
6. $word{at}=80;
7. while(($word,$freq)=each(%word)){
8. push(@word,"$freq\t$word");
9. }
10. foreach $freq_word(sort{$b<=>$a} @word){
11. print "$freq_word\n";
12. }
80 at
67 good
45 hot
24 read
20 book
12 desk

```

### 7.2.2 Combining two or more hashes together

Quite often, we may need to combine two or more hashes into one. The following expression is for doing do this:

```
%newhash=(%hash1,%hash2,%hash3...%hashn)
```

```

$wordset1{cow}=2;
$wordset1{horse}=4;
$wordset1{mule}=3;
$wordset2{dog}=4;
$wordset2{cat}=5;
%combineset=(%wordset1,%wordset2);
foreach $word(keys %combineset){
print "$word\t$combineset{$word}\n";
}
cat 5
cow 2
dog 4
mule 3
horse 4

```

The combination of hashes, together with other functions we've learned, is used in the following program that counts the occurrences of *go*.

*countgo.pl*

```

1. $go="goes go going go went go gone go";
2. $text="I am going to school she went to school he goes to school they
 have gone to school we go to school Peter and Sally go to school";
3. @wordarray=split(/ /,$text);
4. foreach $word(@wordarray){
5. $wordhash{$word}++;
6. }
#In the following, the different word forms of go are put in the hash %go,
#with go as their value.
7. %go=split(/ /,$go);
#In statements 8—10 the different word forms of go are taken one by one
#from %go and assigned to $wordform, and are then checked for their
#existence in %wordhash. If a word form of go exists in %wordhash, its
#lemma go is assigned to $lemma.
8. foreach $wordform(keys %go){
9. if(exists($wordhash{$wordform}))){
10. $lemma=$go{$wordform}; #this assigns go to $lemma
#The following creates a new hash %go_freq whose function is to count the
#occurrences of go and its variants.
11. $go_freq{$lemma}+=$wordhash{$wordform};
#In the following statement, after the frequency of a word form of go is
#added to the frequency of go, the word form is deleted from %wordhash.
12. delete($wordhash{$wordform});
13. }

```

```

#In statements 14—16 , if go itself exists in %wordhash, its frequency is
#added to that in %go_freq, and this go is then deleted from %wordhash. In
#the end, all the different word forms of go and go itself are deleted from
#%wordhash, leaving only other words.
14. if(exists($wordhash{$lemma})) {
15. $go_freq{$lemma}+=$wordhash{$lemma};
16. delete($wordhash{$lemma});
17. }
18. }
#Statement 19 combines %wordhash and %go_freq into a new hash
#%wordlist.
19. %wordlist=(%wordhash,%go_freq);
#The following output the sorted %wordlist.
20. foreach $word(sort keys %wordlist){
21. print "$word\t$wordlist{$word}\n";
22. }

```

The result is shown below:

```

I 1
Peter 1
Sally 1
am 1
and 1
go 6
have 1
he 1
school 6
she 1
they 1
to 6
we 1

```

### 7.2.3 Hash comparisons

Apart from combining hashes together, we can also compare two hashes, picking out identical keys and keys unique to each of the hashes compared. The following program deletes the words that both hashes have and send them to a new hash, keeping their original frequencies; so the remaining words in each of the hashes are all unique ones.

*comparehash.pl*

```
1. $wordlist1="apple 2 banana 2 apricot 7 grape 1 orange 4 pear 1 plum
```

```

12";
2. $wordlist2="apricot 2 banana 5 grape 3 papaya 4 peach 8 plum 1";
3. %hash1=split(/ /,$wordlist1);
4. %hash2=split(/ /,$wordlist2);
#In statements 5—10, words are taken one by one from %hash2 and
#checked in %hash1 for their existence. If they exist, they are deleted from
#%hash1 and %hash2, and put in %hash3, with their respective frequencies
#separated with :.%hash3 stores words that both %hash1 and %hash2 have.
5. foreach $word(keys %hash2){
6. if(exists($hash1{$word})) {
7. $hash3{$word}=$hash1{$word}." ".$hash2{$word};
8. delete($hash1{$word});
9. delete($hash2{$word});
10. }
11. }
#Statements 12—19 respectively output words unique to %hash1 and
#%hash2.
12.print "Words unique to wordlist1: \n";
13.foreach $word(sort keys %hash1){
14.print "$word\t$hash1{$word}\n";
15. }
16.print "Words unique to wordlist2: \n";
17.foreach $word(sort keys %hash2){
18.print "$word\t$hash2{$word}\n";
19. }
#The following output words shared by %hash1 and %hash2.
20.print "Words occurring in both: \n";
21.foreach $word(sort keys %hash3){
22.print "$word\t$hash3{$word}\n";
23. }

```

The following are the results:

```

Words unique to wordlist1:
apple 2
orange 4
pear 1
Words unique to wordlist2:
papaya 4
peach 8
Words occurring in both:
apricot 7:2
banana 2:5

```

---

```

grape 1:3
plum 12:1

```

## 7.2.4 Computing value frequencies

Suppose we have the following short wordlist:

```

apple 2
banana 2
apricot 7
grape 1
orange 4
pear 1
plum 12
peach 1

```

and we want to compute the frequency of frequencies, i.e., how many times frequency 1 occurs, how many times frequency 2 occurs, etc. We can turn this wordlist into a hash, with the words as its keys and the frequencies as its values. Then we can use the *values* function to get these values and compute their frequencies, i.e., the frequency of the frequency classes.

*countvalue.pl*

```

1. $wordlist="apple 2 banana 2 apricot 7 grape 1 orange 4 pear 1 plum 12
 peach 1";
2. %wordhash=split(/ /,$wordlist);
3. foreach $wordfreq(values(%wordhash)){
4. $wordfreqhash{$wordfreq}++;
5. }
6. foreach $freqclass(sort({$a<=>$b}keys%wordfreqhash)){
7. print"$freqclass\t$wordfreqhash{$freqclass}\n";
8. }

```

In the above, statement 3 gets the values, i.e., word frequencies, of *%wordhash* and assigns them one by one to *\$wordfreq*. Statement 4 puts *\$wordfreq* in *%wordfreqhash*, and computes their occurrences. Now the keys of *%wordfreqhash* are actually the word frequency classes, and the values the frequencies of such classes. Statements 6—8 get the sorted frequency classes and the frequencies of these classes. The result is shown below. The first column is the frequency class, the second its frequency. It's a miniature frequency spectrum.

```

1 3
2 2
4 1
7 1

```



### 7.3 Applications

In this section, we'll look at some practical programs that use hashes. These programs are very useful in collecting data for linguistic research and natural language processing.

#### 7.3.1 Computing per word entropy of English

Entropy is a concept widely used in information processing, quantitative linguistics, natural language processing, etc. It's computed with the following:

$$H = - \sum_{x \in X} p(x) \log_2 p(x)$$

where  $H$  is entropy,  $p(x)$  the probability of a variable  $x$ . If  $x$  stands for a word, then  $p(x)$  is the probability of the occurrence of  $x$ , obtained with the frequency of  $x$  divided by the size of the text where  $x$  is in. *bncwordlist2.txt* contains the entire vocabulary of a 10,000,000-word sample from BNC (the British National Corpus). In the wordlist, words are listed in the first column and their frequencies in the second column, with white spaces between the two columns. The following program computes the per word entropy of the English language based on *bncwordlist2.txt*.

*entropy.pl*

```

1. open(F,'bncwordlist2.txt') or die("File does not exist!\n");
2. read(F,$wordlist,3600000);
#The following statement converts \n and consecutive spaces into one
#space
3. $wordlist=~tr/ \|n/ /s;
4. %wordlist=split(/ /,$wordlist);
#Statements 5—7 compute the size of the sample from which the wordlist
#was made by adding the frequency of each word together.
5. foreach $freq(values %wordlist){
6. $cumufreq+=$freq;
7. }
8. foreach $word(keys %wordlist){
9. $probability=$wordlist{$word}/$cumufreq;
10. $logprob=log($probability)/log(2);
#In the following statement, %entropy takes $word as its keys and
```

```

#probability*log probability of $word as its values.
11. $entropy{$word}=$probability*$logprob;
12. }
13. foreach $value(values %entropy){
14. $cumuvalue+=$value;
15. }
16. $entropy=-$cumuvalue;
17. print $entropy;
18. close(F);

```

Statement 9 computes  $p(x)$  and statement 10  $\log_2 p(x)$ . Statement 14 gets  $\sum p(x) \log_2 p(x)$ . The result is 10.0112544220046.

### 7.3.2 Making a word frequency spectrum

A word frequency spectrum is the distribution of word frequencies, i.e., the occurrences of frequency 1, frequency 2, frequency 3 etc. It's also called frequency of frequencies. The following program makes the frequency spectrum for *adventure.txt*. It's similar to *countvalue.pl* in 7.2.4.

```

spectrum.pl
1. open(F,"adventure.txt") or die("Can't open file.\n");
2. read(F,$text,150000);
3. open(W,">spectrum.txt") or die("Can't create file.\n");
4. use Text::Tabs;
5. $stabstop=30;
6. $text=~tr/[,;?";'!:()><+&^%*{ }_=-~\|/\\n\t[\]\@#\$\-]/ /s;
7. $text=~s/^ | $//g; #remove initial and end spaces of $text
8. @temp=split(//,lc $text);
9. foreach $word(@temp){
10. $wordlist{$word}++;
11. }
12. foreach $freq(values %wordlist){
13. $spectrum{$freq}++;
14. }
15. foreach $freqclass(sort({$a<=>$b}keys %spectrum)){
16. print(W expand "$freqclass\t$spectrum{$freqclass}\n");
17. }
18. close(F);
19. close(W);

```

The logic of this program is the same as *countvalue.pl* in 7.2.4. Part of the result is shown below:

---

1	1118
2	392
3	229
4	144
5	91
6	63
7	61
8	55
9	33
10	39
...	...
1635	1

In the above, *1 1118, 2 392, 1635 1* mean there are 1118 words occurring once, 392 words occurring twice, one word occurring 1635 times, which is *the*.

### 7.3.3 Lemmatization

Lemmatization is the process of turning word forms into its base form, e.g., turning words such as *goes, going, went, gone* into *go*. The base form of a set of words with the same major part-of-speech and the same word-sense is called a lemma. One of the most well-known lemmatization algorithms is the Porter stemmer. The following program lemmatizes *adventure.txt* using a dictionary *lemmadic.txt* that contains 47,726 word forms and their corresponding lemmas. The word forms and their lemmas are arranged in the follow form:

```
Accelerated Accelerate
Accelerates Accelerate
Accelerating Accelerate
```

The basic logic of the program is the same as *countgo.pl* in 7.2.2. The program turns *adventure.txt* into a hash *%tempwordlist* and *lemmadic.txt* into another hash *%lemmadic*, in which the word forms are the keys and the lemmas the values. The keys of *%lemmadic* are searched one at a time in *%tempwordlist*; if a match is hit, the word form is deleted from *%tempwordlist* and its corresponding lemma is put to *%lemmatemp* together with the word form's frequency. If the lemma already exists in *%lemmatemp*, only the frequency is added. If instead of the word form, its lemma exists in *%tempwordlist*, it's also deleted and put to *%lemmadic*, with its frequency added.

```
lemmatizer.pl
```

1. `open(F,"adventure.txt")or die("File can't be opened.\n");`
2. `read(F,$text,150000);`
3. `open(W,">wordlist.txt") or die("Can't create file.\n");`

```

4. $text=~tr/[,?";`':!()><+&^%*{} _=~\|/\\n\t[\]\@#\$\-]/ /s;
5. $text=~s/^\s|^ \s$/g; #remove initial and end spaces
6. @tempwordlist=split(/ /,lc $text);
7. foreach $word(@tempwordlist){
8. $tokennumber++;
9. $tempwordlist{$word}++;
10. }
11. open(G,"lemmadic.txt")or die("File does not exist.\n");
12. read(G,$dinput,900000);
13. %lemmadic=split(/[\n]/,lc $dinput); #note the space before \n
14. foreach $wordform(keys %lemmadic){
15. if(exists($tempwordlist{$wordform}))){
16. $lemma=$lemmadic{$wordform};
17. $lemmatemp{$lemma}+=$tempwordlist{$wordform};
18. delete($tempwordlist{$wordform});
19. if(exists($tempwordlist{$lemma}))){
20. $lemmatemp{$lemma}+=$tempwordlist{$lemma};
21. delete($tempwordlist{$lemma});
22. }
23. }
24. }
25. %wordlist=(%tempwordlist,%lemmatemp);
26. foreach $word(sort(keys(%wordlist))){
27. $vocsize++; #compute vocabulary size
28. use Text::Tabs;
29. $stabstop=30;
30. print (W expand("$word\t$wordlist{$word}\n"));
31. }
32. print(W " _____\n");
33. print(W "Total number of word tokens: $tokennumber\n");
34. print(W "Vocabulary size: $vocsize\n");
35. close(F);
36. close(G);
37. close(W);

```

Part of the result is as follows:

<i>a</i>	<i>687</i>
<i>abe</i>	<i>1</i>
<i>able</i>	<i>1</i>
<i>about</i>	<i>93</i>
<i>above</i>	<i>3</i>
<i>absence</i>	<i>1</i>
<i>absurd</i>	<i>2</i>

---

<i>acceptance</i>	1
<i>accident</i>	2
<i>accidentally</i>	1
<i>account</i>	3
<i>accusation</i>	1
<i>accustom</i>	1
<i>ache</i>	1
<i>across</i>	5
... ..	

---

*Total number of word tokens: 27285*

*Vocabulary size: 1939*

### 7.3.4 Lexical comparison between two texts

In 7.2.3 we looked at how to make lexical comparisons between two hashes. Now we'll write a practical program that can make lexical comparisons between two large wordlists. *comparewords.pl* makes lexical comparisons between two lemmatized wordlists *wordlista.txt* and *wordlistl.txt*, made respectively from *adventure.txt* and *lookingglass.txt*. It picks out words shared between the two wordlists and those unique to *wordlista.txt* and *wordlistl.txt*. Although the program has 42 statements, the logic is simple and is the same as that of *comparehash.pl* in 7.2.3.

*comparewords.pl*

```

1. open(F,"wordlista.txt")or die("File can't be opened.\n");
2. read(F,$wordlista,70000);
3. open(G,">aliceword.txt") or die("Can't create file.\n");
4. open(H,"wordlistl.txt") or die("File can't be opened.\n");
5. read(H,$wordlistb,70000);
6. open(I,">lglassword.txt") or die("Can't create file.\n");
7. open(J,">sharedwords.txt") or die("Can't create file.\n");
8. use Text::Tabs;
9. $stabstop=20;
10. $wordlista=~tr// /s;
11. $wordlistb=~tr// /s;
12. %alice=split(/ \n/, $wordlista);
13. %lglass=split(/ \n/, $wordlistb);
14. foreach $word(keys %alice){
15. if(exists($lglass{ $word})) {
16. $sharedwords{ $word}=$alice{ $word}.".".$lglass{ $word};

```

---

```

17. delete($alice{$word});
18. delete($lglass{$word});
19. }
20. }
21. print G "Words unique to Alice in Wonderland: \n";
22. foreach $word(sort keys %alice){
23. $wordnumber1++;
24. print G expand "$word\t$alice{$word}\n";
25. }
26. print G "The number of words unique to Alice in Wonderland is:
 $wordnumber1\n";
27. print I "Words unique to Through the Looking_glass: \n";
28. foreach $word(sort keys %lglass){
29. $wordnumber2++;
30. print I expand "$word\t$lglass{$word}\n";
31. }
32. print I "The number of words unique to Through the Looking_glass is:
 $wordnumber2\n";
33. print J "Words occurring in both: \n";
34. foreach $word(sort keys %sharedwords){
35. $wordnumber3++;
36. }
37. print J "The number of shared words is: $wordnumber3\n";
38. close(F);
39. close(G);
40. close(H);
41. close(I);
42. close(J);

```

The program can be divided into four sections. The first section is from the start to statement 9, in which wordlists are opened and output files created. Tab length is also set in this section. The second section is between statements 10—13, which turn the two wordlists into hashes. The third section is between statements 14—20, where lexical comparisons are made. The last section is from statement 21 to the end, which outputs the results to the files created in the first section. Part of the results is shown below:

```

Words unique to Alice in Wonderland:
abide 1
absence 1
absurd 2
acceptance 1
accident 2

```

---

<i>accidentally</i>	1
... ..	
<i>Words unique to Through the Looking_glass:</i>	
1	2
364	1
365	1
<i>accent</i>	1
<i>acre</i>	1
... ..	
<i>Words occurring in both:</i>	
<i>a</i>	687:718
<i>able</i>	1:6
<i>about</i>	93:60
<i>above</i>	3:2
<i>account</i>	3:1
<i>across</i>	5:8
<i>actually</i>	1:2
<i>add</i>	24:20
... ..	

## Exercises

1. Put the following two short wordlists into two hashes, `%hash1` and `%hash2`, with the words as the keys and frequency as values.

Wordlist 1: *people 6 book 14 read 40 linguistics 13 Perl 12 journal 14 student 6 program 20*

Wordlist 2: *journal 12 student 12 teacher 6 Perl 10 program 18 book 5 do 40 computer 20*

Then do the following:

- 1) Compare the two hashes, outputting the words occurring in both hashes while keeping their respective frequencies.
- 2) Combine the two hashes and output the words with frequency sorted in descending order.
- 3) Reverse the combined hash, turning keys as values and values as keys. Output the hash thus combined with the new keys sorted in ascending order. Make sure nothing is discarded.
- 4) Make a sorted frequency spectrum for the combined hash.

2. The following short text has different forms of *be*. Put the text in a hash, lemmatize the variants of *be* and compute its frequency.

*The word be has the following word forms: am, is, are, was, were, and being.*

3. Write a program using a hash to make a wordlist for *adventure.txt*, with words grouped according to their length in letters, and compute the total number of word types and average word length in letters. The result should be arranged as shown below:

*2-letter words:*

*ah,5; am,16; an,57; as,263; at,211...*

*The total number of words with length 2 is: 41*

*3-letter words:*

*act,1; ada,1; age,4; ago,2; air,15;...*

*The total number of words with length 3 is: 171*

*... ..*

*THE TOTAL NUMBER OF WORD TYPE IS: 2570*

*THE AVERAGE WORD LENGTH IS: 3.93806120579073*

4. In Chapter 1 we mentioned briefly about Yule's  $K$ , which is computed with the following:

$$K = 10000 \frac{\sum_m m^2 V(m, N) - N}{N^2},$$

where  $m$  is the word frequency class,  $V(m, N)$  the number of words with frequency  $m$ , and  $N$  the number of words in the text in question. Now write a program to compute Yule's  $K$  for *adventure.txt*.



## 8 Subroutines and modules

A subroutine is a sub-program within a program, which can be called wherever it's needed to perform certain functions in the program; while a module is a stand-alone program that can be called by other programs to perform certain functions. Suppose we want to write a program to turn four different texts into four lemmatized wordlists, instead of writing bulky blocks of statements for making lemmatized wordlists four times, we can put just one block of statements for making a lemmatized wordlist in a sub-program and call it from the main program. We can also turn it into a stand-alone program and call it whenever it's needed by other programs. Subroutines and modules can make a program more concise, tidy and readable.

### 8.1 Subroutines

Subroutines are sometimes called user-defined functions. Some people say that there is a difference between a subroutine and a user-defined function; namely, a user-defined function returns values, while a subroutine doesn't. However, we don't make such distinctions. In this section, we'll look at the subroutine's basic structure and the different ways to pass information between a subroutine and the main program.

#### 8.1.1 The basic structure

A subroutine has the following basic structure:

```
sub subroutinename{
 statements
 return(resultofsubroutine)
}
```

The function of *return(resultofsubroutine)* is that, upon the completion of the subroutine, it tells the program to return with the result to the place where the subroutine is called. This expression can be omitted when it's not necessary to pass the result back to the main program, or when the main program can get the result by default. We'll discuss the use of *return()* in detail later.

A subroutine can be placed anywhere within a program. In this book, it's placed at the end of a program. The following expression calls a subroutine in the main program:

```
subroutinename($variable1, $variable2, $variable3...)
```

Here, *\$variable1*, *\$variable2*, *\$variable3* etc are called parameters of a subroutine.

Look at the following program that uses a short subroutine. The subroutine turns the words of a text into the upper case.

```
casechange_sub.pl
```

```
1. $sentence="This is a demonstration of the use of subroutines.";
2. uppercase($sentence);
3. $sentence="Its use can make a program more concise and readable.";
4. uppercase($sentence);
5. sub uppercase{
6. print uc "$sentence\n";
7. print "Case change completed.\n";
8. }
```

Statements 1—4 constitute the main program. The subroutine *uppercase* is between statements 5—8. Statements 2 and 4 call it respectively and pass the parameter *\$sentence* to the subroutine. Note the omission of *return* ().

### 8.1.2 Parameters of subroutines

The subroutine in *casechange\_sub.pl* is not of much use since the parameter is passed directly to the subroutine, and the subroutine can accept only one and the same parameter, which can't change its name in the subroutine. To write more versatile and powerful subroutines, we need to know the other way in which parameters are passed to the subroutines.

When a subroutine is called, the parameters are automatically stored in an array called *@\_*, also known as the anonymous array; the first element stores the first parameter, the second element the second parameter, etc. This way, it's extremely easy to retrieve the parameters in the subroutine, and the names of the retrieved parameters don't have to be the same with the ones in the main program. Look at the following example:

```
parameter_sub1.pl
```

```
1. $word1="Perl";
2. $word2="program";
3. $word3="parameter";
4. $word4="subroutine";
5. parameters($word1,$word2,$word3,$word4);
6. sub parameters{
7. print join("\n",@_);
8. }
```

```
perl
program
parameter
subroutine
```

```
parameter_sub2.pl
1. $word1="Perl";
2. $word2="program";
3. $word3="parameter";
4. $word4="subroutine";
5. parameters($word1,$word2,$word3,$word4);
6. sub parameters{
7. for ($i=1;$i<5;$i++){
8. $string=shift();
9. print "$string\t";
10. }
11. }
```

The subroutine loops between statements 7—10 four times, and each time *shift()* gets an element, i.e., a parameter, from the anonymous array *@\_*; the parameter is assigned to *\$string*, which is printed by statement 9. The result is shown below:

```
Perl programparameter subroutine
```

Apart from variables, parameters can also be arrays and hashes. Look at the following two examples:

```
array_sub.pl
1. $sentence="Subroutines and user-defined functions: what is the
 difference?";
2. @array=split(/ /,lc $sentence);
3. printarray(@array);
4. sub printarray{
5. @subarray=@_;
6. foreach $word(sort @subarray){
7. print "$word\n";
8. }
9. }
```

```
hash_sub.pl
1. $sentence="Subroutines and user-defined functions: what is the
 difference?";
2. %hash=split(/ /,lc $sentence);
3. printhash(%hash);
```

```

4. sub printhash{
5. %subhash=@_;
6. foreach $word(sort keys %subhash){
7. print "$word\t$subhash{$word}\n";
8. }
9. }

```

### 8.1.3 The use of *return()* in subroutines

In the above examples, no values were passed back from the subroutines to the main programs, so none of the subroutines used *return()*. Quite often we need to pass back the result of a subroutine to the main program. Without *return()*, the value of the last variable of a subroutine is passed back to the main program by default. In *getlength\_sub.pl*, the subroutine computes two types of text length: length in number of characters, including white spaces, and length in number of words. The subroutine without *return()* passes back the length in number of characters since it's the value of the last variable of the subroutine:

*getlength\_sub1.pl*

```

1. $sentence="Subroutines and user-defined functions: what is the
 difference?";
2. $sentlength=getlength($sentence);
3. print $sentlength;
4. sub getlength{
5. $string=shift();
6. $wordlength=(($string=~s/ /g))+1;
7. $characterlength=length($string);
8. }
63

```

If we need the text length in number of words, then *return()* must be used:

*getlength\_sub2.pl*

```

1. $sentence="Subroutines and user-defined functions: what is the
 difference?";
2. $sentlength=getlength($sentence);
3. print $sentlength;
4. sub getlength{
5. $string=shift();
6. $wordlength=(($string=~s/ /g))+1;
7. $characterlength=length($string);
8. return($wordlength);

```

```
9. }
8
```

In the following program, the subroutine adds, subtracts, multiplies and divides 20 by 4, and returns the results to the main program:

*mathoperation.pl*

```
1. $number=20;
2. ($a,$b,$c,$d,$e)=compute($number);
3. print "$a\n$b\n$c\n$d";
4. sub compute{
5. $num=shift();
6. $add=$num+4;
7. $subtract=$num-4;
8. $divide=$num/4;
9. $multiply=$num*4;
10. return ($add,$subtract,$divide,$multiply);
11. }
24
16
5
80
```

#### 8.1.4 Localization of variables in subroutines

The variables used in all the subroutines so far are global; that is, their values are recognized both in the subroutines and the main program. In the following example, the value of the variable *\$message* is changed in the subroutine, and the new value is recognized in the main program.

*global.pl*

```
1. $sentence="Subroutines and user-defined functions: what is the
 difference?";
2. $message="Text is changed into upper case.";
3. $newtext=uppercase($sentence);
4. print "$newtext\n";
5. print $message;
6. sub uppercase{
7. $strings=uc shift();
8. $message="Task completed successfully.";
9. return($strings);
10. }
```

---

*SUBROUTINES AND USER-DEFINED FUNCTIONS: WHAT IS THE DIFFERENCE?*

*Task completed successfully.*

In a long program with many variables and several subroutines, the use of global variables in subroutines can result in serious errors, such as inadvertently changing the values of some variables in the main program or in other subroutines. To avoid such situation from occurring, we can localize the variables used in a subroutine by the use of *my* the first time the variables are used. Variables thus treated is recognized only in its own subroutine, not in the main program or other subroutines.

*local\_my.pl*

1. \$sentence="Subroutines and user-defined functions: what is the difference?";
2. \$message="Text is changed into upper case.";
3. \$newtext=uppercase(\$sentence);
4. print "\$newtext\n";
5. print \$message;
6. sub uppercase{
7. \$strings=uc shift();
8. my \$message="Task completed successfully.";
9. print "\$message\n";
10. return(\$strings);
11. }

*Task completed successfully.*

*SUBROUTINES AND USER-DEFINED FUNCTIONS: WHAT IS THE DIFFERENCE?*

*Text is changed into upper case.*

Because of the use of *my* in statement 8, although *\$message* is given a new value in the subroutine, it's effective only within the subroutine; outside the subroutine this value is not recognized. *my* can also be used to localize arrays and hashes in subroutines.

## 8.2 Modules

Subroutines can be called anywhere within a program, but it can't be used outside the program. To do this, we need modules. A module is a stand-alone subroutine that can be called by other programs. There are two types of modules, the

common module and the exporter module. All Perl modules have the file extension *pm*.

The common module has the following structure:

```
package modulename
sub subroutinename{
 statements
return(results);
}
1;
```

As with subroutines, *return()* can be omitted depending on the situation. The number *1* tells the calling program a true value is returned to it. It can be any other number; without this number an error message would result and the program would stop.

To call a module, the calling program should put the following expressions where the module is needed:

```
use modulename;
modulename::subroutinename($variables);
```

Here *\$variables* are the target to be handled by a module; it can be scalar variables, arrays or hashes. Now we'll write a program that turns a short text into a wordlist with frequencies. This program calls two modules: *tokenizer.pm* and *printer.pm*, the first one turning the text into individual words stored in an array, and the second printing out the wordlist.

*wordlist\_module.pl*

1. use tokenizer;
2. use printer;
3. \$text="Alice was beginning to get very tired of sitting by her sister on the bank and of having nothing to do";
4. @wordtoken=tokenizer::tokenize(\$text);
5. foreach \$word(@wordtoken){
6. @wordtype{\$word}++;
7. }
8. printer::printwords(%wordtype);

Statement 4 calls *tokenizer.pm* to tokenize *\$text* and put the result in *@wordtoken*. Statement 8 calls the module *printer.pm* to print out the wordlist. The following are the two modules.

*tokenizer.pm*

1. package tokenizer;

```
2. sub tokenize{
3. my($text,@temp);
4. $text=shift();
5. @temp=split(/ /,lc $text);
6. }
7. 1;
```

Note the localization of *\$text* and *@temp* in statement 3.

*printer.pm*

```
1. package printer;
2. sub printwords{
3. my($word,%wordlist);
4. %wordlist=@_;
5. foreach $word(sort keys %wordlist){
6. print "$wordlist{$word}\t$word\n";
7. }
8. }
9. 1;
```

The result is shown below:

```
1 alice
1 and
1 bank
1 beginning
1 by
1 do
1 get
1 having
1 her
1 nothing
2 of
1 on
1 sister
1 sitting
1 the
1 tired
2 to
1 very
1 was
```



The two modules can also be called by other programs that need to turn a text into words stored in an array or to print out a wordlist stored in a hash. This really saves a lot of time and labour. However, it's a bit inconvenient that each time we call a module of this type, we have to use expressions like *tokenizer::tokenize(\$text)*, *printer::printwords(%wordtype)* etc. Perl's exporter modules reduce the labour by using only the part after the two colons. The structure of an exporter module is as follows:

```
package modulename;
use Exporter;
@ISA=("Exporter");
@EXPORT=("modulename");
sub subroutinename{
 statements
 return(results);
}
1;
```

Here *return(results)* can be omitted. The following are expressions used in the calling program:

```
use modulename;
subroutinename($variables);
```

Here *\$variables* can be scalar variables, arrays or hashes.

Now we'll test the use of an exporter module. The following is a program turning a sentence into a short wordlist with frequencies using an exporter module.

*test\_exporter.pl*

1. \$text="Alice was beginning to get very tired of sitting by her sister on the bank and of having nothing to do";
2. use test\_exporter;
3. getword(\$text);

Statement 3 specifies the use of the module *wordlistmodule*, and statement 4 calls the module by its subroutine name *makewordlist*. The following is the exporter module called by *wordlist.pl*:

*test\_exporter.pm*

1. package test\_exporter;
2. use Exporter;
3. @ISA=("Exporter");
4. @EXPORT=("getword");

```

5. sub getword{
6. my($text,$word,$wordnumber,$wordtype,$type,@temp,%wordtype);
7. $text=shift();
8. @temp=split(/ /,lc $text);
9. foreach $word(@temp){
10.$wordnumber++;
11.$wordtype{$word}++;
12. }
13.foreach $type(sort(keys(%wordtype))){
14.print "$wordtype{$type}\t$type\n";
15. }
16. }
17.1;

```

### 8.3 References

A reference is a variable that stores other variables, arrays, hashes or other references. To be exact, a reference stores in the computer's memory the address of a variable, an array, a hash or another reference, instead of their actual values. In this sense, it's a pointer towards the address of a variable, an array or a hash it references. References are often used in subroutines, modules as well as in main programs. In this section we'll examine the structure and applications of references.

#### 8.3.1 Making references

The structure of a reference is as follows:

$$\$reference = \backslash \$variable\ to\ be\ referenced$$

*\$variable to be referenced* can be scalar variables, arrays or hashes. The scalar variables, arrays or hashes thus referenced must be preceded with a backward slash \.

The following stores a scalar variable called *\$word* in a reference called *\$wordref*:

```

$word="Perl reference variable dereference";
$wordref=\$word;

```

The same applies to making a reference to an array:

```

@array=(Perl,reference,variable,dereference);

```

```
$arrayref=\@array;
```

We can make a reference to a hash the same way:

```
%hash=(Perl,reference,variable,dereference);
$hashref=%hash;
```

However, to access the value of a variable through its reference is not strait forward. We can't output a reference the usual way we output a scalar variable, an array or a hash. Try the following:

```
$word="Perl reference variable dereference";
$wordref=\$word;
print $wordref;
SCALAR(0x2829dac)
```

```
@array=(Perl,reference,variable,dereference);
$arrayref=\@array;
print $arrayref;
ARRAY(0x2829e1c)
```

```
%hash=(Perl,reference,variable,dereference);
$hashref=%hash;
print $hashref;
HASH(0x2829e1c)
```

Instead of the values of *\$word*, *\$array* and *%hash*, the results are the addresses of *\$word*, *\$array* and *\$hash* in RAM. There are special expressions for outputting the contents of scalar variables, arrays and hashes through their references. This is called dereferencing.

### 8.3.2 Dereferencing for scalar variables and references

To dereference a reference storing scalar variables we put an extra *\$* in front of the reference as follows:

```
$$refvariable
```

Look at the following:

```
$word="Perl reference variable dereference";
$wordref=\$word;
```

```
print $wordref;
SCALAR (0x2829dac)
print $$wordref;
Perl reference variable dereference
```

The following shows how to dereference a reference storing another reference:

```
$word="Perl reference variable dereference";
$wordref1=\$word;
$wordref2=\$wordref1;
$wordref3=\$wordref2;
print "$$wordref1\n";
print "$$$wordref2\n";
print "$$$$wordref3";
Perl reference variable dereference
Perl reference variable dereference
Perl reference variable dereference
```

The value of a variable can be changed through its reference. Look at the following example:

```
$text="Subroutines and user-defined functions: what is the difference?";
$refvariable=\$text;
print $$refvariable;
$$refvariable="Sometimes subroutines are also called user-defined
functions.";
print "\n";
print $text;
Subroutines and user-defined functions: what is the
difference?
Sometimes subroutines are also called user defined functions.
```

By assigning a new value to `$$refvariable`, the original value of `$text` is changed to the new value.

### 8.3.3 Dereferencing for arrays

There are several ways to dereference for arrays. To dereference for an array, we put `@` before the reference:

```
@array=(Perl,reference,variable,dereference);
$arrayref=\@array;
```

```
print join("-", @$arrayref);
Perl-reference-variable-dereference
```

To dereference for the individual elements of an array, we can use one of the following expressions:

1. `@$referencename[elementnumber]`

```
@array=(Perl,reference,variable,dereference);
$arrayref=\@array;
print @$arrayref[2];
print "\n";
print @$arrayref[1];
variable
reference
```

2. `$$referencename[elementnumber]`

```
@array=(Perl,reference,variable,dereference);
$arrayref=\@array;
print $$arrayref[0];
print "\n";
print $$arrayref[3];
Perl
Dereference
```

3. `${$referencename}[elementnumber]`

```
@array=(Perl,reference,variable,dereference);
$arrayref=\@array;
print ${$arrayref}[1];
print "\n";
print ${$arrayref}[2];
reference
variable
```

4. `@{ $referencename } [ elementnumber ]`

```
@array=(Perl,reference,variable,dereference);
$arrayref=\@array;
print @ { $arrayref } [2];
print "\n";
```

```
print @{$arrayref}[0];
variable
Perl
```

### 8.3.4 Dereferencing for hashes

To dereference for a hash, we put % before the hash reference, as shown below:

```
%hash=(Perl,reference,variable,dereference);
$hashref=%hash;
foreach $word(keys %$hashref){
print "$word:$hashref{$word}\n";
}
variable:dereference
Perl:reference
```

To dereference for the individual keys of a hash, we can use one of the expressions shown below:

#### 1. `$$referencename{keyname}`

```
%hash=(Perl,reference,variable,dereference);
$hashref=%hash;
print $$hashref{Perl};
print "\n";
print $$hashref{variable};
reference
dereference
```

#### 2. `$referencename->[keyname]`

```
%hash=(Perl,reference,variable,dereference);
$hashref=%hash;
print $hashref->{Perl};
print "\n";
print $hashref->{variable};
reference
dereference
```

## 8.4 Use of references in subroutines and modules

In this section we'll look at how to use references for communication between the main program and its subroutine or module. Program *arrayref.pl* uses references to pass two arrays to its subroutine for further processing.

*arrayref.pl*

```

1. @array1=(subroutines,modules,Perl,program);
2. @array2=(array,hash,variable,functions);
3. getarrays(\@array1,\@array2);
4. sub getarrays{
5. my ($arr1,$arr2)=@_;
6. print "Contents of arr1: \n";
7. print join("\n",@$arr1);
8. print "\n";
9. print "Contents of arr2: \n";
10. print join("\n",@$arr2);
11. }
```

In statement 3 *@array1* and *@array2* are turned into two references, which are in turn put in the anonymous array *@\_*. Statement 5 gets the two references, which are respectively outputted by statement 7 and 10.

Program *arrayhashref.pl* uses references to return an array and a hash from its subroutine to the main program.

*arrayhashref.pl*

```

1. $sentence1="Subroutines and user-defined functions: what is the
 difference?";
2. $sentence2="Sometimes subroutines are also called user defined
 functions.";
3. ($array,$hash)=process($sentence1,$sentence2);
4. print join(" _ ",@$array);
5. print "\n";
6. print join(" ",%$hash);
7. sub process{
8. @string1=split(/ /,uc shift());
9. @string2=split(/ /,uc shift());;
10. foreach $word(@string2){
11. $string2{$word}++;
12. }
13. return(\@string1,%string2);
14. }
```

Note how *@string1* and *%string2* are returned as references to the main program. The result is shown below:

```
SUBROUTINES _ AND _ USER-DEFINED _ FUNCTIONS: _ WHAT _ IS _
THE _ DIFFERENCE?
SUBROUTINES 1 DEFINED 1 SOMETIMES 1 CALLED 1 ARE 1 FUNCTIONS.
1 ALSO 1 USER 1
```

Program *moduleref.pl* uses an exporter module called *cleantext.pm* to clean sentences of non-textual characters and extra spaces and punctuation marks, and gets sentence length. References are used to pass scalar variables from the main program to the module.

*moduleref.pl*

1. use cleantext;
2. \$sentence1="Subroutines and &^% user-defined functions: ~~what is the difference?";
3. processtext(\\$sentence1);
4. print \$sentence1;
5. \$sentence2="Sometimes \\ \\ \*\* subroutines () are also... called user-defined +functions.";
6. processtext(\\$sentence2);
7. print \$sentence2;
8. \$sentence3="Some^ people &&claim that there is a \differrnce between >a subroutine and a user-fined function.";
9. processtext(\\$sentence3);
10. print \$sentence3;
11. \$sentence4="Namely, a user-defined functions returns values, ]while [a --- ##subroutine @@@@ does not.";
12. processtext(\\$sentence4);
13. print \$sentence4;
14. \$sentence5="However, we do not make ==such `fine \_\_ distinctions between ~subroutines and user-defined functions.";
15. processtext(\\$sentence5);
16. print \$sentence5;

Note how *\$sentence1*, *\$sentence2*, *\$sentence3* and *\$sentence4* are passed to the module *cleantext.pm* as references. The advantage of using the references is that if the references are assigned new values, these new values will be passed back to the variables thus referred.

*cleantext.pm*

1. package cleantext;
2. use Exporter;
3. @ISA=("Exporter");



```

4. @EXPORT=("processtext");
5. sub processtext{
6. $string=shift();
7. $i++;
8. $$string=~tr/[.,?";`':!()><+&^%*{} }_ =~\|\\|\\n\t\[\]\@#\$\-]/ /s;
9. $$string=~s/^ | $//g;
10. $sentlength=($$string=~s/ //g)+1;
11. $$string="This is the cleaned sentence ".$i.". It has ".$sentlength ."
 words:\n".$$string."\n\n";
12. }
13. 1;

```

In the module, statement 8 uses a regular expression to clean sentences of the non-alphanumeric characters, and the result is passed back to the referenced variables in the main program, i.e., `$sentence1`, `$sentence2`, `$sentence3`, `$sentence4` and `$sentence5`. Statement 10 measures sentence length in number of words. The result is shown below:

```

This is the cleaned sentence 1. It has 9 words:
Subroutines and user defined functions what is the
difference
This is the cleaned sentence 2. It has 8 words:
Sometimes subroutines are also called user defined functions
This is the cleaned sentence 3. It has 16 words:
Some people claim that there is a difference between a
subroutine and a user defined function
This is the cleaned sentence 4. It has 12 words:
Namely a user defined functions returns values while a
subroutine does not
This is the cleaned sentence 5. It has 14 words:
However we do not make such fine distinctions between
subroutines and user defined functions

```

## 8.5 Applications

In this section, we'll put what we've learned in this chapter into practice. We'll write three programs. The first one uses a subroutine and the second and third use a module each. Parameters are passed from the main program to the module with references in the last two programs.

### 8.5.1 Computing arc length

The arc length of rank-frequency distributions can be used in text characterization and language typology. The arc length of rank-frequency distribution  $L$  is expressed as follows:

$$L = \sum_{r=1}^{V-1} \{ [f(r) - f(r+1)]^2 + 1 \}^{1/2},$$

where  $V$  = vocabulary size of a text;  $r$  = rank of word frequency, with the highest frequency being  $r = 1$ ;  $f(r)$  = word frequency at rank  $r$ . The following program computes the arc length of rank-frequency distributions of *adventure.txt* and *lookingglass.txt*. The main program opens the two texts and the subroutine *getarclength* computes the arc lengths of the two texts.

*arclength.pl*

```

1. open(F,'adventure.txt') or die("File does not exist.\n");
2. read(F,$text,150000);
3. getarclength($text);
4. open(F,'lookingglass.txt') or die("File does not exist.\n");
5. read(F,$text,170000);
6. getarclength($text);
7. sub getarclength{
8. $string=shift(); #assign the texts opened in the main program to $string
9. $string=~tr/[.,?";`'!:()><+&^%*{} _=~\|\\|\\n\t\[\]\@#\$\-]/ /s;
10. @wordtoken=split(/ /,lc $string);
#Statements 11—13 turn the array into a wordlist with word frequencies.
11. foreach $word(@wordtoken){
12. $wordlist{$word}++;
13. }
#Statement 14 assigns word frequencies sorted in descending order to $freq
#using the values function.
14. foreach $freq(sort{$b<=>$a}values %wordlist){
15. $typenumber++; #computes number of word types
#Statements 16—18 assign the highest frequency to $freqr and no
#computation is done; this ensures that $freqr, which serves as f(r), remains
#larger than $freq, which serves as f(r+1).
16. if($typenumber<2){
17. $freqr=$freq;
18. }else{
#Statement 19 computes the arc length.
19. $arclength+=(($freqr-$freq)**2+1)**(1/2);
#Statement 20 is for the next round of computing arc length, in which the

```

```

#present value of $freq serving as f(r-1) now will become f(r) then and the
#new value of $freq will serve as f(r-1).
20. $freqr=$freq;
21. }
22. }
23. print "$arclength\n";
#Statements 24—26 empty $arclength, $typenumber and %wordlist for the
#next text.
24. $arclength=0;
25. $typenumber=0;
26. %wordlist=();
27. }

```

### 8.5.2 A module for removing non-alphanumeric characters

In turning a text into a wordlist, a necessary step is to remove non-alphanumeric characters from the text. *ridcharacter.pm* does this and can be called by any program that needs it. The program calling this module must turn the variable storing the text to be processed into a reference, which is then dereferenced in the module. *wordlist.pl* turns *adventure.txt* into a wordlist. It also computes the type-token ratio, i.e., TTR. *ridcharacter.pm* is called to remove non-alphanumeric characters.

```

wordlist.pl
1. open(F,'adventure.txt') or die("File does not exist!\n");
2. read(F,$text,150000);
3. open(W,">result.txt") or die("Can't create file.\n");
4. use Text::Tabs;
5. $stabstop=30;
6. use ridcharacter;
7. cleantext(\$text);
8. @words=split(/ /,lc $text);
9. $wordnumber=$#words+1;
10. foreach $word(@words){
11. $wordhash{$word}++;
12. }
13. foreach $word(sort keys %wordhash){
14. $typenumber++;
15. print (W expand "$word\t$wordhash{$word}\n");
16. }
17. $ttr=$typenumber/$wordnumber; #compute type-token ratio
18. print (W " _____\n\n");
19. print (W "The total number of word tokens is: $wordnumber\n");

```

```

20. print (W "The total number of word types is: $stypenumber\n");
21. print (W "The TTR ratio is : $ttr");
22. close(F);
23. close(W);

```

*ridcharacter.pm*

```

1. package ridcharacter;
2. use Exporter;
3. @ISA=("Exporter");
4. @EXPORT=("cleantext");
5. sub cleantext{
6. my($text);
7. $text=shift();
8. $$text=~tr/[.,?";`!()><+&^%*{} _=~\|\\|\\n\t\[\]\@#\$\-]/ /s;
9. $$text=~s/^ | $//g;
10. }
11. 1;

```

### 8.5.3 A lexical comparison program

*comparetexts.pl* makes lexical comparisons between *adventure.txt* and *lookingglass.txt*. Its logic is similar to that of *comparehash.pl* in 7.2.3. Words unique to each of the texts are picked out and put in *aliceonly.txt* and *glassonly.txt* respectively, while those occurring in both texts are picked out and stored in *sharedwords.txt*. A lemmatization module *lemmatizer.pm* is called in the program to lemmatize the words in the two texts. This module can be called by any program that needs it; it passes back to the calling program a referenced hash containing the lemmatized wordlist and the total number of word tokens of the text. The program also has a subroutine *printwords* that creates output files and prints out the results to them.

*comparetexts.pl*

```

1. use Lemmatizer;
2. use Text::Tabs;
3. $stabstop=30;
4. open(F,"adventure.txt")or die("File can't be opened.\n");
5. open(G,"lookingglass.txt")or die("File can't be opened.\n");
6. read(F,$text1,150000);
7. read(G,$text2,170000);
#In statements 9—10, $text1 and $text2 are turned into two hashes contain-
#ing lemmatized wordlists in the Lemmatizer module and assigned to the
#two references $wordlist1 and $wordlist2.
8. ($wordlist1)=lemmatize($text1);

```

```

9. ($wordlist2)=lemmatize($text2);
10. foreach $word(keys %$wordlist1){
11. if (exists($$wordlist2{$word}))
12. $$sharedwords{$word}=$$wordlist1{$word}.".".$$wordlist2{$word};
13. delete($$wordlist1{$word});
14. delete($$wordlist2{$word});
15. }
16. }
17. $wordlist3=\%sharedwords;
18. printwords($wordlist1,$wordlist2,$wordlist3);
19. close(F);
20. close(G);
21. sub printwords{
22. open(H,">aliceonly.txt")or die("File can't be opened.\n");
23. open(I,">glassonly.txt")or die("File can't be opened.\n");
24. open(J,">sharedwords.txt")or die("File can't be opened.\n");
25. $filehandle=72;#the ASCII value of H
26. while ($wordlist=shift()){
#In the following, the first time printwords is called, the value of $file-
#handle is H, while the second time it's I.
27. $filehandle=chr $filehandle;
28. foreach $word(sort(keys(%$wordlist))){
29. $wordnumber++;
30. print ($filehandle expand("$word\t$$wordlist{$word}\n"));
31. }
32. print ($filehandle
"_____ \n");
33. print ($filehandle "The total number of words in file is:
$wordnumber.\n");
34. $filehandle++;
35. $wordnumber=0;
36. }
37. close(H);
38. close(I);
39. close(J);
40. }
41.

```

#### *lemmatizer.pm*

```

1. package Lemmatizer;
2. use Exporter;
3. @ISA=("Exporter");
4. @EXPORT=("lemmatize");
5. sub lemmatize{

```

```

6. my($textinput,$word,$dicinput,$wordform, $tokennumber, $lemma,
 @tempwordlist,%dichash,%wordlist,%lemmatemp);
7. $textinput=shift();
8. $textinput=~tr/[.,?";`'!:()><+&^%*{ }_~\|/\\n\t\[\]\@#\$\-]/ /s;
9. $textinput=~s/^ | $//g;
10. @tempwordlist=split(/ /,$textinput);
11. foreach $word(@tempwordlist){
12. $tokennumber++;
13. $word=lc $word;
14. $word=ucfirst($word);
15. $tempwordlist{$word}++;
16. }
17. open(LEMMAFILEHANDLE,"lemmadic.txt")or die("File does not
 exist.\n");
18. read(LEMMAFILEHANDLE,$dicinput,900000);
19. %dichash=split(/[\n]/,$dicinput);
20. foreach $wordform(sort(keys(%dichash))) {
21. $wordform=ucfirst($wordform);
22. if(exists($tempwordlist{$wordform})) {
23. $lemma=$dichash{$wordform};
24. $lemmatemp{$lemma}+=$tempwordlist{$wordform};
25. delete($tempwordlist{$wordform});
26. if(exists($tempwordlist{$lemma})) {
27. $lemmatemp{$lemma}+=$tempwordlist{$lemma};
28. delete($tempwordlist{$lemma});
29. }
30. }
31. }
32. %wordlist=(%tempwordlist,%lemmatemp);
33. %tempwordlist=();
34. %lemmatemp=();
35. %dichash=();
#The following statement passes back a referenced wordlist hash and a
#scalar variable containing number of word tokens in text.
36. return(\%wordlist,$tokennumber);
37. close(LEMMAFILEHANDLE);
38. }
39. 1;

```

Part of the results is as follows.

*aliceonly.txt:*

... ..

*Worry*

2

<i>Wow</i>	6
<i>Wretch</i>	2
<i>Writhe</i>	1
<i>Yelp</i>	1
<i>Yer</i>	4
<i>Youth</i>	6
<i>Zealand</i>	1
<i>Zigzag</i>	1

---

*The total number of words in file is: 705.*

*glasssonly.txt:*

... ..

<i>Worst</i>	7
<i>Wough</i>	1
<i>Wring</i>	2
<i>Wrist</i>	1
<i>Yellow</i>	2
<i>Yhtils</i>	1
<i>Ykcowrebbaj</i>	1
<i>Yonder</i>	2
<i>Ysmim</i>	1

---

*The total number of words in file is: 836.*

*sharedwords.txt:*

... ..

<i>Ye</i>	1:3
<i>Year</i>	3:6
<i>Yes</i>	13:14
<i>Yesterday</i>	3:4
<i>Yet</i>	25:17
<i>You</i>	410:613
<i>Young</i>	5:5
<i>Your</i>	63:71
<i>Yours</i>	3:1
<i>Yourself</i>	10:11

---

---

*The total number of words in file is: 1234.*

### Exercises

1. Assign `$sentence1` and `$sentence2` to two references and then output the contents of `$sentence1` and `$sentence2` from the references.

```
$sentence1= "Alice was beginning to get very tired of sitting by her sister on
the bank and of having nothing to do";
$sentence2="In another moment down went Alice after it, never once con-
sidering how in the world she was to get out again. ";
```

2. Assign the following array to a scalar variable and output the array contents in ascending order from the variable using reference.

```
$array[0]="apple";
$array[1]="peach";
$array[2]="orange";
$array[3]="banana";
$array[4]="apricot";
$array[5]="grape";
```

3. Assign the above array to a reference that is an element of a hash, and output the contents of the array in ascending order from the reference.

4. In exercise 4 of Chapter 3 we wrote a program called *wordlength.pl* (see model answers for Chapter 3 in Appendix). Now modify this program by using three subroutines to respectively replace the statements for creating the nine output files (from the file handles *G* to *O*), the statements for outputting words of different length to files holding words of specified length, and those for outputting word length information to *wordinfo.txt*.

5. In *wordtype.pl* in 6.4.3 there are some statements for combining identical array elements and computing their frequency. Turn these statements into an exporter module called *processarray.pm*, adding a sorting function in it. Then modify *wordtype.pl* by putting *ridcharacter.pm* in 8.5.2 and *processarray.pm* in it.



# 9 Directory and file management

In this chapter, we'll look at directory and file management within a Perl program. Directory and file management includes the following topics: creation or removal of directories; changing file path; collecting the file names within a directory; deleting files and renaming files; changing file attributes; formatting files etc.

## 9.1 Directory management

The following functions are used for directory management.

**mkdir**(*drive:\directoryname*) This function sets up a new directory with specified drive and directory name. The following makes a directory called *newdirectory* in *d*.

```
mkdir ('d:\newdirectory')or die ("Can't create directory.");
```

**rmdir**(*drive:\directoryname*) This function removes the specified directory from the specified drive. The directory to be removed must be empty.

```
rmdir ('d:\newdirectory') or die ("Can't remove directory.");
```

The following three functions respectively open a directory, get the file names in it and then close the directory:

```
opendir(directoryhandle,drive:\directoryname)
```

```
readdir(directoryhandle)
```

```
closedir(directoryhandle)
```

Now we'll access the directory *perllesson* in *d* and output all the file names to a file called *dirfiles.txt*.

```
dirfile1.pl
```

1. `open(F,">dirfiles.txt") or die ("Can't open file.\n");`
2. `opendir (D,'d:\perllesson') or die ("Can't open directory.");`
3. `print F join("\n", readdir(D));`
4. `close(F);`
5. `closedir(D);`

The *opendir* function also has the following form:

```
opendir(directoryhandle, 'drive:\numberofdots')
```

To get the file names of the current directory, use one dot and *drive* is optional.

```
dirfile2.pl
```

1. `open(F, ">dirfiles.txt") or die ("Can't open file.\n");`
2. `opendir (D, '.') or die ("Can't open directory.");`
3. `print F "Contents of the current directory:\n";`
4. `print F join("\n", readdir(D));`
5. `close(F);`

To get all the directory names of the current drive and the file names outside these directories in the drive, use two dots, and *drive* is optional.

```
dirfile3.pl
```

1. `open(F, ">dirfiles.txt") or die ("Can't open file.\n");`
2. `opendir (D, 'e:\..') or die ("Can't open directory.");`
3. `print F "\nContents of drive E:\n";`
4. `print F join("\n", readdir(D));`
5. `close(F);`
6. `closedir(D);`

**glob**('drive\directory\\*.\*') This function gets all the file names in the specified directory.

```
print join("\n", glob('d:\perllesson*.*'));
```

This prints out all the file names in *d:\perllesson* separated with line breaks. We can also get all the file names with the file extension of *txt* in the current directory and put them in an array:

```
@filename=glob('*.*txt');
print join("\n", @filename);
```

## 9.2 File management

The following functions are for file management operations.

**rename**('oldfilename', 'newfilename') This function changes *oldfilename* to *newfilename*. The following changes *result.txt* to *testresult.txt*:

```
rename("result.txt","testresult.txt") or die ("Can't rename the file.");
```

**chmod**(*attribute*,*filename*) This function changes file attributes. The following numbers are for setting *attribute*: 0666: read/write; 0444: read only. The following changes the attribute of *result.txt* as read only:

```
chmod(0444,'result.txt');
```

The following two functions respectively get and set file attributes using the Win32::File module.

**Win32::File::GetAttributes**("filename", *variable*) This function gets the attributes of *filename* and puts the result to *variable*. File attributes are expressed in the following numbers: 1, read only; 2, hidden; 32; archive.

**Win32::File::SetAttributes**("filename",*attribute*) This sets file attributes. *attribute* has the following settings: ARCHIVE, READONLY, HIDDEN.

To use the above two functions, the **Win32::File** module must be called first. The following program sets the file attribute of *result.txt* as hidden and then outputs the file attribute number:

```
setattribute.pl
use Win32::File;
Win32::File::SetAttributes("result.txt",HIDDEN);
Win32::File::GetAttributes("result.txt",$attricode);
print $attricode;
2
```

**utime**(*variable*,*variable*,*filename*) This function changes the creation time of *filename*. *variable* must be assigned the intended creation time of the file in seconds. This seems a complicated thing to do. Actually it's not as difficult as it sounds. In Perl, time starts on January 1, 1970, and there is a function to get the time between now and then in seconds: **time**(). As the time of writing this section, 1,260,016,947 seconds have elapsed since then. So to change the creation time of a file, we just use **time**() plus or minus the desired number of days in terms of seconds and assign this value to a variable. Suppose we want to change the creation time of *result.txt* two days earlier than its actual creation time, use the following:

```
$when=time()-2*24*3600;
utime($when,$when,"result.txt");
```

**truncate**(“*filename*”, *length*) This function truncates the specified file by *length* (in number of bytes).

```
truncate("result.txt",2000);
```

This truncates 2,000 bytes from *result.txt*.

**copy**(“*file1*”, “*file2*”) This function copies *file1* to *file2*. To use this function, the **File::Copy** module must be called.

```
use File::Copy;
copy("adventure.txt","adventurecopy.txt");
```

The above copies *adventure.txt* to a file called *adventurecopy.txt*.

**eof** *filehandle* This function checks whether the end of a file is reached. The following repeatedly prints a chunk of text 100 bytes in length from *alice1.txt* until the end of the file is reached:

```
open(F,"alice1.txt") or die ("File can't be opened.\n");
until(eof F){
 read(F,$text,100);
 print "$text\n";
}
```

**compare**(“*file1*”, “*file2*”) This function checks whether the contents of *file1* and the contents of *file2* are identical. To use this function, the **File::Compare** module must be called first.

```
use File::Compare;
if(compare("alice1.txt","alice2.txt")){
 print "Different\n";
}else{
 print "Same";
}
Different
```

**unlink**(<*filename*>) This function deletes a specified file or files. The wild card \* can be used in the function. Files thus deleted can’t be recovered. So this function should be used with care. The following deletes all files whose names begin with *word* with the extension of *txt*:

```
unlink(<word*.txt>);
```

### 9.3 Formatting output files

When writing data to output files, we may need to arrange the data in certain format, e.g., left-justification, right-justification, centre-justification etc. If the output file is very long, perhaps we may also want to divide the file into pages of certain length with a page heading on each page. In this section we'll learn ways for doing this.

The basic structure for formatting data and send them to an output file without a page heading and page division is as follows:

```
format outputfilehandle =
formatting expression
.
write outputfilehandle
```

The basic structure for dividing an output file into pages of certain length with a page heading on each page is as follows:

```
format outputfilehandle_TOP =
page setting expression
.
```

In the following sections we'll learn how to use the data formatting structure and the page division structure to format data and output files.

#### 9.3.1 Outputting data in their original format

The formatting expression can be any character on the keyboard, including tabs, line breaks and white spaces, except @, ~, ^ and #. They can also be a string or a text, which are outputted in its original format. Look at the following text:

```

* *
* PERL FOR *
* QUANTITATIVE LINGUISTICS *
* *

```

December 7, 2009

The following program outputs this text in its original format:

```

asterisktitle.pl
1. open(W,">result.txt") or die ("Can't create file.\n");
2. format W =
3. *****
4. * *
5. * PERL FOR *
6. * QUANTITATIVE LINGUISTICS *
7. * *
8. *****
9.
10. December 7, 2009
11.
12. write W;
13. close(W);

```

The formatting expression is from statement 3 to statement 10, between which everything, including white spaces and line breaks, is outputted to *result.txt*. Note that statement 9 only has a line break, which is sent to the output file.

### 9.3.2 Arranging data in left-justified columns

To arrange data in left-justified columns, we use the following formatting expression:

```
@<+
variable
```

Here @ represents the contents of *variable*, <, as well as @, stands for the width of one character, and + means one or more < (in the rest of this chapter we'll use the plus sign in formatting symbols to represent one or more the preceding character). Suppose we want to set the line width of the left-justified contents of a variable to 25 characters, then 24 <'s following @ should be used since @ also stands for the width of one character. More than one set of @<+ and *variable* can be used to format data, as shown below:

```
@<+@<+@<+...
variable1, variable2, variable3...
```

Look at the following data, which consist of words, word frequency and word length: *Grammar, 33, Linguistics, 9, Syntax, 45, Word, 154, Onomatopoeia, 1*. We'll output this line of data in three left-justified columns, one column containing words, 20 characters in width; the second column frequencies, eight







```

3. foreach $word(keys %wordlist){
4. format F=
5. @||||||||||||||||@||||||||||||||||
6. $word,$wordlist{$word}
7. .
8. write (F);
9. }
10.close(F);

```

The result is shown below:

<i>Onomatopoeia</i>	<i>Letter</i>
<i>Syntax</i>	<i>Word</i>
<i>Grammar</i>	<i>Linguistics</i>
<i>Semantics</i>	<i>Morph</i>

The following formatting expression centre-justifies decimal numbers by the decimal point:

```
@#+.#+
```

To format decimal numbers stored in several variables, more than one set of @#+.#+ can be used, as shown below:

```
@#+.#+@#+.#+@#+.#+...
variable1, variable2, variable3...
```

Each # represents a digit. @ also stands for a digit place on the left of the decimal point. If there are fewer digits than # on the right of the decimal point, 0's are added so that the number of digits is equal to that of # on the right. Look at the following example:

```

justifydecimal.pl
1. open(F,">result.txt") or die ("Can't create file.\n");
2. @decimal=(0.0445,324.5579,11.2213,24.31,4,1189.22137);
3. foreach $number(@decimal){
4. format F=
5. @#####.#####
6. $number
7. .
8. write(F);
9. }

```

The result is shown below:





2. \$text="Perl for quantitative\nlinguistics.";
3. format F=
4. @\*
5. \$text
6. .
7. write(F);
8. close(F);

The result is shown below:

```
Perl for quantitative
linguistics.
```

### 9.3.6 Producing page heading and paginating output files

If the output file is very long, we may want to divide the output file into pages, each with a page heading and page number. In this case, the page setting structure is used:

```
format outfilehandle_TOP =
page setting expression
.
```

The default page length generated by this page setting structure is 60 lines. In *page setting expression* we can use any character on the keyboard, including tabs, line breaks and white spaces, except @, ~, ^ and #. We can also use a string or a text formatted with the formatting symbols. *page setting expression* can also consist of page number generating symbol, which is in the following form:

```
"string $%"
```

The page setting structure can be used together with the data formatting structure. The following script uses the two structures to add a page heading and page number to the output file and format the data.

```
formatpage.pl
1. open(F,">result.txt") or die ("Can't create file.\n");
2. %wordinfo=(Come,332,Linguistics,9,Study,45,Work,154,Onomatopaei
 a,1);
3. $pagetitle="WORDLIST";
4. format F_TOP=
5.
6. @||||||||||||||||||||||||||||||||
```





---

<i>leisurely</i>	9	1
<i>lend</i>	4	1
<i>length</i>	6	1
<i>less</i>	4	1
<i>lessons</i>	7	6
<i>let</i>	3	30
<i>letter</i>	6	1
<i>letters</i>	7	1
<i>licking</i>	7	1
<i>lid</i>	3	3
<i>lie</i>	3	2
<i>lies</i>	4	1
<i>life</i>	4	8
<i>lifted</i>	6	5
<i>light</i>	5	3
<i>lighted</i>	7	1
<i>lighting</i>	8	1
<i>lightly</i>	7	1
<i>lightning</i>	9	3

## 9.4 Applications

This section contains three practical programs. The first one divides *adventure.txt* into 45-line, 2-column pages. The second computes vocabulary growth. The last one gets word range in a collection of texts. Word range here refers to the number of different texts a word occurs in.

### 9.4.1 A page-formatting program

The following program divides *adventure.txt* into pages containing two columns of text. Each page is 45 lines in length.

```
twocolumn.pl
1. open(F,"adventure.txt") or die ("Can't open file.\n");
2. read(F,$text,150000);
3. open(W,">result.txt") or die ("Can't create file.\n");
4. $text=~tr/ / /s;
5. @textarray=split(/ /,$text);
6. for($i=1;$i< $#textarray+1;$i++){
```





---

*WONDERLAND CHAPTER I when the Rabbit actually*  
*Down the Rabbit-Hole TOOK A WATCH OUT OF ITS*  
*Alice was beginning to get WAISTCOAT- POCKET, and*  
*very tired of sitting by looked at it, and then*  
*her sister on the bank, hurried on, Alice started*  
*and of having nothing to to her feet, for it*  
*do: once or twice she had flashed across her mind*  
*peeped into the book her that she had never before*  
*sister was reading, but it see a rabbit with either a*  
*had no pictures or waistcoat-pocket, or a*  
*conversations in it, `and watch to take out of it,*  
*what is the use of a and burning with*  
*book,` thought Alice curiosity, she ran across*  
*`without pictures or the field after it, and*  
*conversation?` So she was fortunately was just in*  
*considering in her own time to see it pop down a*  
*mind (as well as she large rabbit-hole under*  
*could, for the hot day the hedge. In another*  
*made her feel very sleepy moment down went Alice*  
*and stupid), whether the after it, never once*  
*pleasure of making a considering how in the*  
*daisy-chain would be worth world she was to get out*  
*the trouble of getting up again. The rabbit-hole*  
*and picking the daisies, went straight on like a*  
*when suddenly a White tunnel for some way, and*  
*Rabbit with pink eyes ran then dipped suddenly down,*  
*close by her. There was so suddenly that Alice had*  
*nothing so VERY remarkable not a moment to think*

---

#### 9.4.2 Computing vocabulary growth and number of hapax legomena

The following program computes vocabulary growth of the 20 text chunks of *adventure.txt: alice1.txt—alice20.txt*. It turns the twenty text chunks into 20 lemmatized wordlists and outputs two formatted files, the first containing a wordlist for the 20 text chunks, the second vocabulary growth, the growth of hapax legomena and the ratio between the number of hapax legomena and vocabulary size. The program uses three subroutines: *makewordlist*, *getvocgrowth* and *printcumuwordlist*. The first subroutine produces a wordlist

for each of the text chunks; the second the vocabulary growth from *alice1.txt* to *alice20.txt*; and the third the overall wordlist of the 20 text chunks. The program uses the **system(ctl)** function, which clears the screen of the old output.

```
vocgrowth.pl
```

```
1. use Lemmatizer;
2. use Text::Tabs;
3. $abstop=30;
4. open(R,">wordlist.txt") or die("Can't create file\n");
5. open(W,">vocgrowth.txt") or die("Can't create file\n");
6. $|=1; #set the first array element number to 1 instead of 0
#Statement 7 puts all the text names in @filename.
7. @filenames=glob("alice*.txt");
8. for($i=1;$i< $#filenames+1;$i++){
#Statements 9—12 open the text chunks and create the output files called
#wordlist1.txt, wordlist2.txt...etc.
9. $inputfile=@filenames[$i];
10. $outputfile='>wordlist'. $i.'.txt';
11. open(F,$inputfile)or die("File can't be opened!\n");
12. open(G,$outputfile) or die("Cant' create files.\n");
13. makewordlist();
14. getvocgrowth();
15. }
16. printcumuwordlist();
17. close(G);
18. close(R);
19. close(W);
20. sub makewordlist{
21. read(F,$text,10000);
#In statement 22, the lemmatizer module returns a referenced hash storing
#a wordlist and a scalar variable storing number of word tokens, which
#are respectively assigned to $wordlist and $wordnumber.
22. ($wordlist,$wordnumber)=lemmatize($text);
#In statement 23 $wordlist is dereferenced which is assigned to @wordlist.
#The purpose is to get vocabulary size of each text chunk. Since a word list
#consists of words and their frequencies, when assigned to an array, words
#and frequencies all become array elements. In statement 24 the total
#number of elements divided by 2 gets vocabulary size.
23. @wordlist=% $wordlist;
24. $textvocsiz=$#wordlist/2;
25. $pagetitle="WORDLIST";
#The following generates subtitles alice1.txt, alice2.txt etc with text size
#and vocabulary size.
26. $subtitle="Text: alice". $i.".txt\nText size: $wordnumber\nVocabulary
```







---

9618	1094	429	0.3921
10926	1179	469	0.3978
12200	1277	518	0.4056
13583	1352	537	0.3972
14938	1413	545	0.3857
16347	1481	569	0.3842
17644	1543	589	0.3817
19082	1624	629	0.3873
20411	1699	651	0.3832
21726	1747	662	0.3789
23180	1792	673	0.3756
24537	1854	699	0.3770

### 9.4.3 A program for computing word range

Word range refers to how many different texts a word occurs in. The following program *wordrange.pl* computes word range in 20 texts: *alice1.txt*—*alice20.txt*, and the vocabulary growth. In addition, it also produces a wordlist for each of the texts, and an overall wordlist for these texts. It uses the lemmatization module *lemmatizer* and four subroutines: *getwordlist*, *getrange*, *getvocgrowth* and *prinrange*, which respectively produce a wordlist for each of the texts, compute word range and vocabulary growth, and send the word range data to an output file. A new Perl function **eval** is used. This function can turn a variable into a Perl command. For example, if we assign *open(F,'adventure.txt')* to a variable called *\$openfile* in the following statement

```
$openfile="open(F,'adventure.txt)";
```

then

```
eval $openfile
opens the file adventure.txt whose file handle is F.
```

*wordrange.pl*

1. use Lemmatizer;
2. mkdir "data" or die "Can't create directory";
3. open(Q,'>data\wordrange.txt');
4. open(T,'>data\vocgrowth.txt');
5. for (\$i=1;\$i<21;\$i++){
6. \$filename='alice'.'. \$i.'.txt'; #input files alice1.txt, alice2.txt etc
7. \$textname='Alice'.'. \$i';

---

```

8. $outfile="data\wordlist.$i.txt";#for outputting wordlists
9. $openfile=q/open(W,"$infile") or die("Can't open files.\n");
10. $writefile=q/open(R,"$outfile") or die("Can't create files.\n");
11. eval $openfile;
12. eval $writefile;
13. read(W,$text,10000);
14. ($wordlist,$textsize)=lemmatize($text);
15. getwordlist();
16. getrange();
17. getvocgrowth();
18. }
19. printrange();
20. close(R);
21. close(Q);
22. close(T);
23. close(W);
24. sub getwordlist{
25. system(cls);
26. print("Processing file: $i\n"); #send processing info to screen
27. use Text::Tabs;
28. $stabstop=25;
29. $textvocsiz=0;
#Statements 30—34 get wordlists for each of the texts.
30. foreach $word(sort(keys(%$wordlist))){
31. $textvocsiz++; #get vocabulary size of each of the texts
32. print(R expand("$word\t$wordlist->{$word}\n"));
33. }
34. print (R " _____\nVocabulary size of $textname:
 $textvocsiz");
35. }
36. sub getrange{
37. use Text::Tabs;
38. $stabstop=5;
39. foreach $word(sort(keys(%$wordlist))){
40. $wordfreq=$wordlist->{$word};
41. $cumuvoc{$word}+=$wordfreq;
#%cumuvoc contains cumulative vocabulary and cumulative word
frequency
42. $cumufreq=$cumuvoc{$word};
#The following compute word range.
43. if(exists($cumuvoc{$word}))){
44. $wordrange{$word}++;
#In the following %textinfo stores word, in which text it occurs and its
#frequency in the text.

```







---

```

15, 33;Alice16, 33;Alice17, 36;Al
ice18, 28;Alice19, 28;Alice20, 22;
Abide 1 1 Alice9, 1;
Able 1 1 Alice2, 1;

```

## Exercises

1. Do the following:
  - a. Create a directory called *test* with a Perl statement and then removes the directory with another Perl statement.
  - b. Put all the files with the *txt* extension in *d:\perllesson\texts* in an array and then output the names of these files to a file.
  - c. Use the *opendir*, *readdir* and *closedir* functions to output all the file names in *d:\perllesson\progs* to a file.
  
2. Centre-justify *poem.txt* using the formatting symbol for centre-justification.
  
3. Write a program to compute vocabulary coverage of *bncwordlist2.txt* over each of the 20 *alice* texts. The output file *coverage.txt* should contain the following for each of the texts: text name, text length, vocabulary size, number of lemmas covered, number of lemmas uncovered, coverage rate, and the uncovered lemmas. These data should be arranged in seven left-justified columns. At the end of *coverage.txt* there should be the average coverage. The output file should contain a page head. In addition, the program should also make a wordlist of covered words with frequency for each of the text, and a wordlist of all the uncovered words.
  
4. Write a program to get the following information on each of the 20 *alice* text chunks: text name, text size, vocabulary size, number of hapaxes, number of dis legomena and number of tris legomena, hapax/vocabulary ratio, dis legomena/vocabulary ratio, tris legomena/vocabulary ratio, TTR and average word length in letters. Arrange these data in 11 columns, with character data and integer numbers left-justified and decimal numeric data centre-justified around the decimal point.
  
5. Vocabulary overlap refers to the number of identical lemmas shared between two texts. Write a program that divides the 20 *alice* text chunks into 10 random pairs and computes the vocabulary overlaps of these ten text pairs. The results of the program should be put in the following files: ten wordlists containing shared words of each text pair; a file containing the following: name of the texts of each text pair and their respective length and vocabulary size, and the vocabulary overlap. The file should have a page top, and at the bottom of the page give the

average vocabulary overlap and the standard deviation of the overlap, which is given by the following:

$$S = \sqrt{\frac{\sum (x - \bar{x})^2}{N}},$$

where  $S$  is the standard deviation,  $x$  the vocabulary overlap of a text pair,  $\bar{x}$  the average vocabulary overlap of the 10 text pairs,  $N$  the number of pairs.

# Appendix

## Model answers to the exercises

### Exercises of Chapter 2

2.

```
compound.pl
print "\n";
$cn1=(30.2693*(1**2.3212));
$cn2=30.2693*2**2.3212;
$cn3=30.2693*3**2.3212;
$cn4=30.2693*4**2.3212;
$cn5=30.2693*5**2.3212;
print ("$cn1\n$cn2\n$cn3\n$cn4\n$cn5\n");
```

3.

```
tuldava.pl
$v=1000000*2.71828**(-0.009152*log(1000000)**2.3057);
print("$v\n");
```

4.

```
arclength.pl
$arclength=((1635-872)**2+1)**(1/2)+((872-825)**2+1)**(1/2)+((825-730)**2+1)**(1/2);
print $arclength;
```

5.

```
math2.pl
$result=(100938.3*2248**3-7754/5.56+(3400/2578)**(1/4)*(1102-331)**(12-8));
print("$result\n");
```

### Exercises of Chapter 3

1.

```
combinelines.pl
#This program combine lines into a paragraph
open(F,"poem.txt")or die("File can't be opened.\n");
open(W, ">result.txt")or die("File can't be created.\n");
while ($line=<F>){
 chomp $line;
 $cumulines.=$line.' ';
}
}
```

```
print (W $cumulines);
close(F);
close(W);
```

2.

*rjustify.pl*

```
#This program right-justifies bncwordlist.txt
open(F,"bncwordlist.txt")or die("File can't be opened.\n");
open(W, ">result.txt")or die("File can't be created.\n");
while ($word=<F>){
$wordnumber++;
$digitlength=length($wordnumber);
$wordlength=length($word);
$leftspaces=" "x(30-($wordlength+$digitlength));
$word=$wordnumber.$leftspaces.$word;
print(W $word);
}
close(F);
close(W);
```

3.

*cjustnumber.pl*

```
#This program justifies poem.txt and add line numbers
open(F,"poem.txt")or die("File can't be opened.\n");
open(W, ">result.txt")or die("File can't be created.\n");
while ($line=<F>){
$linenumber++;
$linelength=length($line);
$cumulinelength+=$linelength;
$leftspaces=" "x(40-$linelength/2);
$cline=$leftspaces.$line;
if ($linenumber==1){
print (W $cline);
}elseif(ord $line==45){
print(W $cline);
}else{
$cline=($linenumber-1).$cline;
print(W $cline);
}
}
$average=$cumulinelength/($linenumber-2);
print(W "\n"." \n"."The average line length of the poem is: $average");
close(F);
close(W);
```

4.

*wordlength.pl*

```
#This program computes word length in letters
open(F,"bncwordlist.txt")or die("File can't be opened.\n");
open(G, ">length3.txt")or die("File can't be created.\n");
open(H, ">length4.txt")or die("File can't be created.\n");
open(I, ">length5.txt")or die("File can't be created.\n");
open(J, ">length6.txt")or die("File can't be created.\n");
open(K, ">length7.txt")or die("File can't be created.\n");
open(L, ">length8.txt")or die("File can't be created.\n");
open(M, ">length9.txt")or die("File can't be created.\n");
open(N, ">length10.txt")or die("File can't be created.\n");
open(O, ">over10.txt")or die("File can't be created.\n");
open(P, ">wordinfo.txt")or die("File can't be created.\n");
while ($word=<F>){
 $wordnumber++;
 chomp $word;
 $wordlength=length($word);
 $cumulength+=$wordlength;
 if ($wordlength<=3){
 $wordlength3++;
 print(G "$word\n");
 }elseif($wordlength==4){
 $wordlength4++;
 print(H "$word\n");
 }elseif($wordlength==5){
 $wordlength5++;
 print(I "$word\n");
 }elseif($wordlength==6){
 $wordlength6++;
 print(J "$word\n");
 }elseif($wordlength==7){
 $wordlength7++;
 print(K "$word\n");
 }elseif($wordlength==8){
 $wordlength8++;
 print(L "$word\n");
 }elseif($wordlength==9){
 $wordlength9++;
 print(M "$word\n");
 }elseif($wordlength==10){
 $wordlength10++;
 print(N "$word\n");
 }else{
```

```

$over10++;
print(O "$word\n");
}
}
$average=$cumulength/$wordnumber;
print (P " LEXICAL INFORMATION ON BNCWORDLIST\n");
print (P "The total number of words: $wordnumber\n");
print (P "The average word length: $average\n");
print (P "The total number of words with length 1--3: $wordlength3\n");
print (P "The total number of words with length 4: $wordlength4\n");
print (P "The total number of words with length 5: $wordlength5\n");
print (P "The total number of words with length 6: $wordlength6\n");
print (P "The total number of words with length 7: $wordlength7\n");
print (P "The total number of words with length 8: $wordlength8\n");
print (P "The total number of words with length 9: $wordlength9\n");
print (P "The total number of words with length 10: $wordlength10\n");
print (P "The total number of words with length over 10: $over10\n");
close(F);
close(G);
close(H);
close(I);
close(J);
close(K);
close(L);
close(M);
close(N);
close(O);
close(P);

```

5.

```

reversegetc.pl
#This program turns ASCII codes into characters
open(F,"result.txt") or die("Can't open file.\n");
read(F,$text,620000);
open(W,">test.txt") or die("Can't create file.\n");
$textlength=length($text);
while(length($text)>0){
$position=index($text,'')+1;
$code=substr($text,0,$position);
$text=substr($text,$position,$textlength);
$code=chr $code;
print W "$code";
}
close(F);

```

```
close(W);
```

### Exercises of Chapter 4

1.

*Consonant1.pl*

```
#This program computes the number of words with five or more consecutive
#consonant letters
open(F,"bncwordlist.txt") or die ("File can't be opened.\n");
open(W,">result.txt") or die ("Can't create file.\n");
while($word=<F>){
if($word=~m/[bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz]
[bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz]/x)
{
$number++;
print W $word;
}
}
print(W "The total number of words that have five consecutive consonant letters
or more is: $number.\n");
close(F);
close(W);
```

2.

*getmake.pl*

```
#This program counts the number of make and its variants.
open(F,"adventure.txt") or die ("File can't be opened.\n");
read(F,$text,150000);
open(W,">result.txt") or die ("Can't create file.\n");
$text=~tr/[,;`!()><+&^%*{}_~\|\\n\t[\]\@#\$\-]/ /s;
$number=(($text=~s/ma((de)|(k(e|es|ing))) //gi);
print (W "The total number of different word forms of MAKE is: $number");
close(F);
close(W);
```

3.

*removepos.pl*

```
#This program removes POS tags
open(F,"tagged.txt") or die ("File can't be opened.\n");
read(F,$text,200000);
open(R,">result.txt") or die ("File can't be created.\n");
$text=~s/<.*?>\n//g;
$text=~s/_.+? //g;
```



```
print (R $text);
close(F);
close(R);
```

4.

*wlengthxml.pl*

```
#This program removes the xml tags and computes word length in syllables
open(F,"text.xml")or die("File does not exist!\n");
read(F,$text,23000);
open(R,">result.txt") or die("File can't be created!\n");
$text=~s/.*ALICE.*ALICE/ALICE/;
$text=~s/<.*?>/ /g;
$text=~s/ \n//g;
$text=~tr/[,;?";`!()><+&^%*{} _~\|\\n\t[\]\@#\$\-]/ /s;
$text=~s/ $//g;
$text=~s/ ^n/g;
print R $text;
close(F);
close(R);
#The following is similar to countsyllable.pl
open(S,"result.txt") or die("Can't create file.\n");
open(W,">syllainfo.txt") or die("Can't create file.\n");
while($word=<S>){
 $word=lc($word);
 $wordnumber++;
 $word2=$word;
 $sylnumber=(($word2=~s/[aeiouy]+/z/g);
 $sylnumber++ if($sylnumber==0 or $word=~m/sm$/);
 if($word=~m/tively$/ or $word=~m/ial$/ or $word=~m/[^eioa]e/){
 unless($word=~m/ple$/ or $word=~m/ble$/ or $word=~m/gle$/ or $word eq
 "the\n" or $word eq "she\n" or $word eq "he\n" or $word eq "be\n"){
 $sylnumber--;
 }
 }
 print (W "$sylnumber\t$word");
 $cumusylnumber+=$sylnumber;
 $sylnumber=0;
 }
 $average=$cumusylnumber/$wordnumber;
 print (W "The total number of words is: $wordnumber\n");
 print (W "The average word length in syllables is: $average\n");
 close(S);
 close(W);
```

5.

*removetag.pl*

```
#This program removes POS tags, double quotes and separates phrases in
#poswords.txt
open(F,"poswords.txt") or die ("File can't be opened.\n");
read(F,$word,12000);
open(R,">result.txt") or die ("File can't be created.\n");
$word=~s/ \n/g; #replace spaces with line break
$word=~s/.+?"//g; #remove POS tags, which all have " on the right
$word=~s/"//g;#remove remaining " on the left
$word=~s/\n+\n/g; #keep only one linebreak between each word
$word=~s/\n$/; #remove the empty line at the end of $word
$breaknumber=($word=~s/\n/g);#count number of linebreaks
$wordnumber=$breaknumber+1; #the last word of $word has no line break, so 1
must be added
$totallength=length($word)-$breaknumber;
$average=$totallength/$wordnumber;
print R $word;
print (R "The total number of words is: $wordnumber.\n");
print (R "The average word length in letters is: $average.");
close(F);
close(R);
```

**Exercises of Chapter 5**

1.

*markbebe.pl*

```
#This program extracts BE and its variants from a text
open(F,"adventure.txt") or die ("File can't be opened.\n");
read(F,$text,150000);
open(W,">result.txt")or die ("Can't create file.\n");
$wordnumber=($text=~s/\bbe(ing)?\b|\bwass\b|\bwere\b|\bam\b|\bis\b|\bare\b/**
&/gi);
print (W "The number of the different word forms of BE is:
$wordnumber\n$text");
close(F);
close(W);
```

2.

*getassoon.pl*

```
#This programs gets AS SOON AS from a text.
open(F,"adventure.txt")or die ("File can't be opened\n");
read(F,$text,150000);
```

```

open(R,">result.txt") or die("File can't be created!\n");
$text=~s/\n//g;
$text=~tr/ / /s;
$string="AS SOON AS";
print (R "$string\n");
while ($text=~m/\b$string\b/i){
$getassoon=$text;
$number++;
$getassoon=~s/(.*?\b$string\b.*?[.?!]).*\1/i;
$getassoon=~s/.*?[.?!](.*\b$string\b.*)\1/i;
$text=~s/.*?\b$string\b.*?[.?!]/i;
$getassoon=~s/\b$string\b/ **$string** /i;
print (R " $number\t $getassoon\n");
$getassoon=$text;
}
close(F);
close(R)

```

3.

*collocation.pl*

#This program makes collocation of *go* and its variants in a text using regular #expressions.

```

open(F,"adventure.txt");
open(R,">result.txt") or die("File does not exist!\n");
read(F,$text,150000);
$text=~s/[\t\r\n\(\)]//g;
$text=~tr/ / /s;
$text='* * * * *.$text.* * * * *';
while ($text=~m/\bwent\b|\bgone\b|\b(go(ing|es)?)\b/i){
$collo=$text;
$collonumber++; #number of collocations
#In the following statement the left context is in obtained with ((\S+\s){3,5}).
#Note the use of {3,5} here because sometimes there are only 3 or 4 words left to
#form the left context. The key word is extracted with
#((\bwent\b|\bgone\b|\b(go(ing|es)?)\b). {0,2} is used because there are cases of 2
#puncs together after the key word.The right context is in the 7th brackets, back
#referenced by $7. $1$3$7 gets the left context, the key word and the right
#context.
$collo=~s/.*?((\S+\s){3,5})((\bwent\b|\bgone\b|\b(go(ing|es)?)\b)[.,?;:"!]{0,2})\s(
(\S+\s){3,5})/$1$3$7/i;
$leftlength=length($1);
$centrejustify=' 'x(45-$leftlength);
$collo=$centrejustify.$1.' '(uc$3)'. '$7;
#Pay attention to the outer brackets. Without it only a fraction of the keywords

```

```

#would be picked
$text=~s/.*?((\S+\s){3,5}\bwent\b\bgone\b\b(go(ing|es)?)\b)[.,?:"!]{0,2};//i;
print (R "$collo\n");
}
print (R "The number of GO and its different word forms are: $collonumber\n");
close(F);
close(R);

```

4.

*picknonwords.pl*

```

#This program picks out-of-dictionary words and puts them into non-word.txt,
#and outputs in-dictionary words in result.txt. Manual check should be done and
#more statements can be added to reduce errors.
open(F,"bncwordlist.txt") or die ("Can't open file.\n");
open(R,">result.txt") or die("File can't be created.\n");
open(W,">nonword.txt") or die("File can't be created.\n");
#The following statements pick non-words:
while ($line=<F>){
if ($line=~m/[0-9]+/g){
print W $line;
}elsif($line=~m/^[aeiouy]\1|([aeiouy])\1\1|[aeouy]{4}/gi){
print W $line;
}elsif($line=~/^[bcdfghjklmnpqrstvwxyz]\1|([bcdfghjklmnpqrstvwxyz])\1\1/ig){
print W $line;
}elsif($line=~/([bcdfghjkmnpqrtvwxyz])\1$/ig){
print W $line;
}elsif($line=~/([bcdfghjklmnpqrtvwxz]){5,}/ig){
print W $line;
}elsif($line=~/^[bcdfgjkmnqtvxz]{2,}/ig){
print W $line;
}elsif($line=~/[plhryw]([bcdfgjkmqtvxz])/ig){
print W $line;
}elsif($line=~/[bcdfgjmqtvxzlywhr][psn]|([bcdfghjklmnpqrstvwxyzl]){4,}$/ig){
print W $line;
}else{ #the following prints out in-dictionary words
print R $line;
}
}
close(F);
close(R);
close(W);

```

5.

*getwordtype.pl*

```

#This program makes an unsorted word type list. But it's very slow.
open(F,"adventure.txt");
open(R,">result.txt") or die("file does not exist!\n");
read(F,$text,150000);
$text=~s/[\n.?,\[\]\(\)*!:;`"]/ /g;
$text=~s/\-/ /g;
while (length($text)>1){
$text=~s/^ //;
$text=~tr/ / /s;
$textb=$text;#$textb is destroyed in the following statement
$textb=~s/(\b\w+\b).*/$1/i;
$word=$1;#$1 is changed into nothing in the following statement
#In the following all the words identical with $1 are removed from $text, making
#it shorter and shorter.
$wordfreq=($text=~s/\b$1\b//gi);
print (R "$wordfreq\t$word\n");
}
close(F);
close(R)

```

### Exercises of Chapter 6

4.

*bigramfreq.pl*

```

#This program makes frequencied bigrams
open(F,"adventure.txt") or die("file can't be opened.\n");
read(F,$text,150000);
use Text::Tabs;
$stabstop=30;
$text=~s/([\.,`?!";])+ $&/g;
$text =~s/\n+/ /g;
$text =~tr/ / /s;
$text=~s/^ //g;
@wordlist=split(/ /,$text);
for($i=0;$i< $#wordlist;$i++){
for($j=0;$j<2;$j++){
$bigram.=" ".$wordlist[$i];
$i++;
}
push(@temp,$bigram);
$i-=2;
$bigram="";
}

```

```

@bigramarray=sort(@temp);
$freq=1;
for($i=0;$i< $#bigramarray+1;$i++){
if($bigramarray[$i+1]eq $bigramarray[$i]){
$freq++;
}else{
$bigram_freq.=expand($bigramarray[$i]."\t".$freq."\n");
$freq=1;
}
}
open(W,">bigramfreq.txt") or die("Can't create file.\n");
print(W "$bigram_freq\n");
close(F);
close(W);

```

5.

*wordlengthfreq.pl*

```

#This program computes word length distribution
use Text::Tabs;
$stabstop=30;
open(F,'adventure.txt') or die("File does not exist!\n");
open(W,'>wordlength.txt') or die ("Unable to create file!\n");
read(F,$text,150000);
$text=~s/[.?:";!'*_\n \(\)\-\[\]]//g;
$text=~tr// /s;
$text=~s/^\s|$/g;
@temp1=split(/ /,$text); #get words
$wordnumber=$#temp1+1;
for($i=0;$i<$wordnumber;$i++){
push(@temp2,length(@temp1[$i]));
}
@wordlength=sort({$a<=>$b}@temp2);
$freq=1;
for($i=0;$i< $#wordlength+1;$i++){
$cumulength+=$wordlength[$i]; #get cumulative word length
if($wordlength[$i+1]eq $wordlength[$i]){
$freq++;
}else{
$wordlength_freq=$wordlength[$i]."\t".$freq;
print W expand("$wordlength_freq\t\n");
$freq=1;
}
}
$average=$cumulength/$wordnumber;

```

```
print (W " _____\n\n");
print (W "The total number of word tokens is: $wordnumber\n");
print (W "The average word length is: $average");
close(F);
close(W);
```

## Exercises of Chapter 7

1.

(1)

*getsamewords.pl*

#This program turns \$text1 and \$text2 into two hashes, outputs the words shared  
#by the two texts with their respective frequencies

```
$text1="people 6 book 14 read 40 linguistics 13 Perl 12 journal 14 student 6
program 20";
```

```
$text2="journal 12 student 12 teacher 6 Perl 10 program 18 book 5 do 40
computer 20";
```

```
%hash1=split(/ /,$text1);
```

```
%hash2=split(/ /,$text2);
```

```
foreach $word(keys %hash1){
```

```
if(exists($hash2{$word})){
```

```
$hash3{$word}=$hash1{$word}." ".$hash2{$word};
```

```
delete($hash1{$word});
```

```
delete($hash2{$word});
```

```
}
```

```
}
```

```
foreach $word(keys %hash3){
```

```
print "$word\t$hash3{$word}\n";
```

```
}
```

(2)

*descendfreq.pl*

#This program turns \$text1 and \$text2 into two hashes, combines the two hashes  
#and then outputs the words with the frequencies sorted in the descending order.

```
$text1="people 6 book 14 read 40 linguistics 13 Perl 12 journal 14 student 6
program 20";
```

```
$text2="journal 12 student 12 teacher 6 Perl 10 program 18 book 5 do 40
computer 20";
```

```
%hash1=split(/ /,$text1);
```

```
%hash2=split(/ /,$text2);
```

```
foreach $word(keys %hash1){
```

```
$hash2{$word}+= $hash1{$word};
```

```
}
```

```

while(($word,$freq)=each(%hash2)){
push(@wordlist,"$freq\t$word");
}
foreach $freq_word(sort{$b<=>$a}@wordlist){
print "$freq_word\n";
}

```

(3)

*reverse.pl*

```

#This program combines two hashes and then reserves the new hash, turning its
#keys as values and values as keys.

```

```

$text1="people 6 book 14 read 40 linguistics 13 Perl 12 journal 14 student 6
program 20";

```

```

$text2="journal 12 student 12 teacher 6 Perl 10 program 18 book 5 do 40
computer 20";

```

```

%hash1=split(/ /,$text1);

```

```

%hash2=split(/ /,$text2);

```

```

foreach $word(keys %hash1){

```

```

$hash2{$word}+= $hash1{$word};
}

```

```

}

```

```

foreach $word(keys %hash2){

```

```

$value=$hash2{$word};

```

```

$reversehash{$value}.= $word.' ';
}

```

```

}

```

```

foreach $word(sort{$a<=>$b} keys %reversehash){

```

```

print "$word\t$reversehash{$word}\n";
}

```

```

}

```

(4)

*freespectrum.pl*

```

#This programs combines two hashes into a new one and then makes a frequency
#spectrum.

```

```

$text1="people 6 book 14 read 40 linguistics 13 Perl 12 journal 14 student 6
program 20";

```

```

$text2="journal 12 student 12 teacher 6 Perl 10 program 18 book 5 do 40
computer 20";

```

```

%hash1=split(/ /,$text1);

```

```

%hash2=split(/ /,$text2);

```

```

foreach $word(keys %hash1){

```

```

$hash2{$word}+= $hash1{$word};
}

```

```

}

```

```

foreach $freq(values %hash2){

```

```

$spectrum{$freq}++;
}

```

```

}

```



```
foreach $freq(sort{$a<=>$b}keys %spectrum){
print "$freq\t$spectrum{$freq}\n";
}
```

2.

*lemmatizebe.pl*

#This program lemmatizes the different word forms of be.

\$text="The word be has the following word forms: am, is, are, was, were, and being";

\$lemma="am be is be are be was be were be being be";

@wordarray=split(/[:,.] | /,lc \$text);#there's a space after ] and before |

%lemmahash=split(/ /,\$lemma);

foreach \$word(@wordarray){

\$wordhash{\$word}++;

}

foreach \$wordform(keys %lemmahash){

if(exists(\$wordhash{\$wordform}))){

\$lemma=\$lemmahash{\$wordform};

\$lemmago{\$lemma}+=\$wordhash{\$wordform};

delete(\$wordhash{\$wordform});

}

if(exists(\$wordhash{\$lemma}))){

\$lemmago{\$lemma}+=\$wordhash{\$lemma};

delete(\$wordhash{\$lemma});

}

}

%wordlist=(%wordhash,%lemmago);

foreach \$word(sort keys %wordlist){

print "\$word\t\$wordlist{\$word}\n";

}

3.

*lengthgroup.pl*

#This program makes a wordlist for adventure.txt, with words grouped according #to their length in letters.

open(F,'adventure.txt') or die("File does not exist!\n");

open(W,'&gt;wordlist.txt') or die ("Unable to create file!\n");

read(F,\$text,150000);

use Text::Tabs;

\$stabstop=20;

\$text=~tr/["'`!()&gt;&lt;+&amp;^%\*{}\_~\|\\/\n\t[\]\@#\\$\- ]/ /s;

\$text=~s/^ | \$//g;

@temp=split(/ /,lc \$text);

foreach \$word(@temp){

```

$wordnumber++;
$scumulength+=length($word);
$wordlist{$word}++;
}
#The following assign word and its frequency to $word_freq, separated by a
#comma, then put it to a new hash $word_freq_length, with $word_freq as key
#and $wordlength as value.
while(($word,$freq)=each(%wordlist)){
$wordlength=length($word);
$word_freq=$word.'.$freq.';
$word_freq_length{$word_freq}=$wordlength;
}
#The following create a new hash %lengthgroup, in which the keys are length
#and values words with such length.
foreach $key(sort keys %word_freq_length){
$keylength=$word_freq_length{$key};#assign word length to $keylength
#The following statement combines words of the same length as values, with
#the length as key
$lengthgroup{$keylength}.= $key;
}
foreach $lengthclass(sort{$a<=>$b}keys %lengthgroup){
#The following statement counts number of words with the same length:
$wordnumber2=($lengthgroup{$lengthclass}=~s/;/g);
$scumunumber+=$wordnumber2;
print W $lengthclass."-LETTER WORDS". "\n";
print W $lengthgroup{$lengthclass}.";". "\n";
print W "The total number of words with length $lengthclass is:
$wordnumber2\n";
print W "\n";
}
$average=$scumulength/$wordnumber;
print W "THE TOTAL NUMBER OF WORD TYPE IS: $scumunumber\n";
print W "THE AVERAGE WORD LENGTH IS: $average";
close(F);
close(W);

```

4.

*yulesk.pl*

```

#This program computes Yule's K for adventure.txt
open(F,"adventure.txt") or die("Can't open file.\n");
read(F,$text,150000);
$text=~tr/[.,?";`!()><+&^%*{} _=~\|/\\\\n\t[\]\[@\#\$\-]/ /s;
$text=~s/^\s|$/g;
@temp=split(/ /,lc $text);

```

```

foreach $word(@temp){
$wordnumber++;
$wordlist{$word}++;
}
#The following get frequency spectrum
foreach $freq(values %wordlist){
$spectrum{$freq}++;
}
#The following compute Yule's K
foreach $freqclass(sort({$a<=>$b}keys%spectrum)){
$cumu+=$freqclass**2*$spectrum{$freqclass};
}
$k=10000*(($cumu-$wordnumber)/$wordnumber**2);
print $k;
close(F);

```

## Exercises of Chapter 8

1.

*scalarrefex.pl*

```

#This program uses references to output the contents of $sentence1 and
$sentence2.
$sentence1= "Alice was beginning to get very tired of sitting by her sister on the
bank and of having nothing to do";
$sentence2="In another moment down went Alice after it, never once
considering how in the world she was to get out again.";
$ref1=\$sentence1;
$ref2=\$sentence2;
print "$$ref1\n";
print "$$ref2";

```

2.

*arrayrefex.pl*

```

#This program uses a reference to output the contents of an array.
$array[0]="apple";
$array[1]="peach";
$array[2]="orange";
$array[3]="banana";
$array[4]="apricot";
$array[5]="grape";
$ref=\@array;
foreach $word(sort @$ref){
print("$word\n");
}

```

}

3

*hashrefex.pl*

#This program uses a key of a hash as a reference for an array, and outputs the  
#contents of the array using the reference.

```
$array[0]="apple";
$array[1]="peach";
$array[2]="orange";
$array[3]="banana";
$array[4]="apricot";
$array[5]="grape";
$hash{fruit}=\@array;
foreach $word(sort @{$hash{fruit}}){
print "$word\n";
}
```

4.

*wordlengthsub.pl*

#This program uses three subroutines to create nine output files (from the file  
#handles *G* to *O*) and outputs words of different length to files holding words of  
#specified length, and word length information to *wordinfo.txt*.

```
open(F,"bncwordlist.txt")or die("File can't be opened.\n");
```

```
openoutput();
```

```
while ($word=<F>){
```

```
 $wordnumber++;
```

```
 chomp $word;
```

```
 $wordlength=length($word);
```

```
 $cumulength+=$wordlength;
```

```
 for($i=3;$i<11;$i++){
```

```
 if ($wordlength<=$i and $i==3){
```

```
 $wordlength{$i}++;
```

```
 printword();
```

```
 }elseif($wordlength>=$i and $i==10){
```

```
 $wordlength{10}++;
```

```
 printword();
```

```
 }elseif($wordlength==$i and $i>3 and $i<10){
```

```
 $wordlength{$i}++;
```

```
 printword();
```

```
 }
```

```
 }
```

```
}
```

```
printwordinfo();
```

```
sub openoutput{
```

```

for($j=3;$j<11;$j++){
$character=chr (68+$j);
$filehandle=$character;
$filename="length".$j.".txt";
open($filehandle, ">$filename")or die("File can't be created.\n");
}
open(P, ">wordinfo.txt")or die("File can't be created.\n");
}
sub printword{
$character=chr (68+$i);
$filehandle=$character;
print $filehandle "$word\n";
}
sub printwordinfo{
$average=$cumulength/$wordnumber;
print (P " LEXICAL INFORMATION ON BNCWORDLIST\n");
print (P "The total number of words: $wordnumber\n");
print (P "The average word length: $average\n");
for($i=3;$i<11;$i++){
print (P "The total number of words with length $i: $wordlength{$i}\n");
$character=chr (68+$i);
$filehandle=$character;
close($filehandle);
}
close(P);
}

```

5.

*wordtypemodule.pl*

#This program combines identical array elements and computes their frequency  
#using an exporter module processarray.pm. The program also calls  
#ridcharacter.pm in 8.5.2.

```

use processarray;
use ridcharacter;
open(F,'adventure.txt') or die("File does not exist!\n");
open(W,'>wordlist.txt') or die ("Unable to create file!\n");
read(F,$text,150000);
cleantext(\$text);
@wordarray=split(/ /,lc $text);
$wordnumber=$#wordarray+1;
total(\@wordarray);
foreach $word(@wordarray){
$typenumber++;
print W "$word\n";
}

```

```

}
print (W " _____\n\n");
print (W "The total number of word tokens is: $wordnumber\n");
print (W "The total number of word types is: $typenumber\n");
close(F);
close(W);

```

*processarray.pm*

```

package processarray;
use Text::Tabs;
$tabstop=30;
use Exporter;
@ISA=("Exporter");
@EXPORT=("total");
sub total{
my(@array,$temp,$i,$word,$word_freq,$freq);
$temp=shift();
@array=sort @$temp;
$freq=1;
for($i=0;$i< $#array+1;$i++){
if($array[$i+1]eq @array[$i]){
$freq++;
}else{
$word_freq=expand @array[$i]."\t".$freq;
push(@arrayfreq,$word_freq);
$freq=1;
}
}
@$temp=@arrayfreq;
}
1;

```

## Exercises of Chapter 9

2.

*cjustify2.pl*

```

#This program centre-justifies poem.txt using centre-justification symbols.
open(F,"poem.txt") or die ("Can't open file.\n");
open(W,">result.txt") or die ("Can't create file.\n");
while($line=<F>){
format W=
@|
$line

```







4.

*lexinfo.pl*

```
#This program gets the following information on each of the 20 alic text
#chunks: text name, text size, vocabulary size, number of hapaxes, number of dis
#legomena and number of tris legomena, hapax/vocabulary ratio, dis legomena/
#vocabulary ratio, tris legomena/vocabulary ratio, TTR and average word length
#in letters.
```

```
use Lemmatizer;
```

```
open(T,'>lexinfo.txt') or die ("Can't create file.\n");
```

```
for ($i=1;$i<21;$i++){
```

```
$infilename='alice'.'.txt';
```

```
$textname='Alice'.'.i';
```

```
system(cls);
```

```
print("Processing file: $i\n");
```

```
$openfile=q/open(R,"$infilename") or die("Can't open files.\n");
```

```
eval $openfile;
```

```
read(R,$text,10000);
```

```
($wordlist,$textsize)=lemmatize($text);
```

```
$pageheading="LEXICAL INFORMATION";
```

```
format T_TOP=
```

```
@||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
$pageheading
```

```
TEXT TSIZE VSIZE H DIS TRIS H/V DIS/V TRIS/V TTR
AVERLENGTH
```

```

```

```
.
```

```
foreach $word(sort keys %$wordlist){
```

```
$vocsize++;
```

```
#In the following, $cumuwordlength gets the cumulative word length of every
#word in text.
```

```
$cumuwordlength+=length($word)*$wordlist->{$word};
```

```
if ($wordlist->{$word}==1){
```

```
$hapnum++;
```

```
}elsif($wordlist->{$word}==2){
```

```
$disnum++;
```

```
}elsif($wordlist->{$word}==3){
```

```
$trisnum++;
```

```
}
```

```
}
```

```
$hv=$hapnum/$vocsize;
```

```
$disv=$disnum/$vocsize;
```

```
$trisiv=$trisnum/$vocsize;
```





```
}
}
#The following compute standard deviation.
$moverlap=$cumuoverlap/10;
foreach $overlap(@overlaparray){
$sigma+=$(overlap-$moverlap)**2;
}
$s=sqrt($sigma/10);
print T "-----\n";
print T "Average lexical overlap: $moverlap\n";
print T "Overlap standard Deviation: $s";
close(F);
close(T);
close(W);
```

# Index

- , 13
- , 13
- != , 19
- \$# , 81
- \$\$ , 136
- \$\$referencename[elementnumber], 138
- \$\$referencename{keyname} , 139
- \$% , 162
- \$& , 63
- \$[ , 79
- \$\_ , 91
- \$` , 64
- \${\$referencename}[elementnumber], 138
- \$' , 64
- \$1 , 65
- \$a<=>\$b , 91
- \$b cmp \$a , 90
- \$b<=>\$a , 91
- \$referencename->[keyname] , 139
- \$tabstop , 95
- % , 13
- \* , 13, 49
- \*\* , 13
- .. , 21, 50
- ... , 21
- / , 13
- ? , 50
- @\$referencename[elementnumber], 138
- @{\$referencename}[elementnumber ], 138
- @ARGV , 26
- @EXPORT , 134
- @ISA , 134
- [ ] , 53
- {m,n} , 67
- {n,} , 67
- {n} , 66
- | , 53
- ~~^followed by any number of < or > , 159
- + , 13, 49
- ++ , 13
- < , 19
- <= , 19
- =~ , 41
- == , 19
- > , 19
- >= , 19
- 1 , 65
- 2 , 65
- 3 , 65
- abs(n) , 16
- ActivePerl-5.10 , 3
- alternative operators , 53
- and , 23
- anonymous array , 140
- anonymous variable , 91, 93
- Antconc , 1
- arc length , 25, 143
- ARCHIVE , 152
- array , 79, 128
- array insertion, truncation and deletion , 88
- ascending order , 89
- ASCII , 32
- atan2 , 18
- b , 63
- back reference , 65
- bigram , 98
- BNC , 2, 9, 26, 118, 180, 194
- C , 1, 47
- C++ , 1
- centre-justification , 154
- centre-justified , 72
- centre-justify , 35, 36, 175
- CGI , 2
- chmode , 152
- chomp , 27
- chr , 33

- 
- CLAWS, 71
  - close, 28
  - closedir, 150
  - command line file input, 28
  - compare, 153
  - concatenation, 21
  - concordance, 72
  - consonant cluster, 56
  - copy, 153
  - corpus, 2, 98
  - cos, 18
  - d, 48, 62, 63
  - delete, 111
  - dereference, 136
  - dereference for a hash, 139
  - dereference for arrays, 137
  - dereference for the individual elements of an array, 138
  - dereference for the individual keys of a hash, 139
  - dereferencing, 136
  - descending order, 90
  - die, 28
  - directory and file management, 150
  - directory management, 150
  - dis legomena, 175, 198
  - distribution of word frequencies, 119
  - DOS, 6, 9, 10, 12, 26, 28, 29, 30
  - double byte language, 75
  - e, 42
  - each, 110
  - else, 19, 20
  - elsif, 19
  - entropy, 118
  - eof, 153
  - eq, 22
  - escape character, 12, 30, 59
  - eval, 171
  - exists, 111
  - exp(n), 16
  - expand, 95
  - exponential computation, 14
  - exporter, 132
  - file handle, 28
  - File::Compare, 153
  - File::Copy, 153
  - floating point, 16
  - for, 69
  - foreach, 84, 110
  - format filehandle =, 154, 155
  - format filehandle\_TOP =, 154, 161
  - Foxpro, 1
  - frequency spectrum, 119
  - g, 43
  - ge, 22
  - getc, 34
  - glob, 151
  - global, 130
  - global variables, 131
  - greediness, 52
  - grep, 94
  - gt, 22
  - hapax legomena, 166
  - hash, 103, 128
  - hash elements, 103
  - HIDDEN, 152
  - HTML, 57, 59
  - Hyper Text Markup Language, 57
  - i, 44
  - ICON, 1
  - if, 19
  - index, 68
  - int(n), 16
  - JAVA, 1
  - join, 92
  - keys, 103, 107
  - Kleene star, 49, 52
  - language typology, 25
  - lc, 31
  - le, 22
  - left-justification, 154
  - left-justified, 35, 175
  - lemma, 120
  - lemmatization, 120
  - lemmatization algorithm, 120
  - length, 32
  - letter frequency, 1
  - letter graphemic load, 1

- 
- letter phonemic load, 1
  - letter utility, 1
  - Lexa, 1
  - lexical bundle, 74
  - lexical comparisons, 122
  - Linux, 2
  - localize arrays and hashes in subroutines, 131
  - localize the variables, 131
  - log(n), 16
  - log(n)/log(10), 17
  - log(n)/log(2), 17
  - logical operator, 23
  - lt, 22
  - m//, 41
  - m/pattern/, 41
  - Macintosh, 2
  - main program, 126
  - make a reference to a hash, 136
  - making a reference to an array, 135
  - map, 93
  - Math::BigInt, 16
  - metacharacter, 61
  - mkdir, 150
  - module, 3, 95, 126, 131, 132, 134, 135, 140, 141, 142, 144, 145, 149, 152, 153, 171, 194
  - morpheme, 1
  - MS/DOS, 2
  - multi-dimensional array, 79, 82
  - my, 131
  - natural language processing, 3, 41, 57, 98, 118
  - ne, 22
  - nested bracketing, 73
  - N-gram, 98
  - not, 23
  - numbering the elements of an array, 79
  - OCP, 1
  - one-byte characters, 75
  - one-dimensional array, 79
  - open, 28
  - opendir, 150, 151
  - or, 23
  - ord, 33
  - OS/2, 2
  - package, 132
  - parameter, 127
  - part-of-speech, 120
  - per word entropy of English, 119
  - Perl, 1
  - phoneme, 1
  - pop, 87
  - Porter stemmer, 120
  - POS tag, 71
  - print, 8, 11
  - program editor, 7
  - push, 86
  - PYTHON, 1
  - q(string), 13
  - qq(string ), 13
  - quantifier, 49
  - quantitative linguistics, 1, 2, 41, 93, 94, 118
  - rand(n), 16
  - random sampling, 94
  - range, 164, 171
  - range operator, 21, 80
  - read, 30
  - readdir, 150
  - READONLY, 152
  - reference, 135
  - regular expression, 2, 41, 42, 43, 49, 50, 52, 54, 60, 61, 63, 72, 142, 184
  - regular expression operator, 61
  - rename, 151
  - return, 126, 129, 132, 134
  - reverse, 81
  - right-justification, 40, 154
  - rindex, 69
  - rmdir, 150
  - s, 48, 62
  - s///, 42, 48
  - scalar, 11, 132, 134, 135, 136, 141, 149, 192
  - script, 3, 9, 12, 52
  - select, 162

- 
- shift, 86
  - sin, 18
  - SNOBOL4, 1
  - sort, 89
  - SPITBOL, 1
  - splice, 88
  - split, 84
  - sprintf, 17
  - sqrt(n), 16
  - standard deviation, 176
  - STDIN, 27
  - structure of a reference, 135
  - sub, 126, 132
  - subroutine, 127, 130
  - substr, 69
  - syllabic word length, 56
  - syllable, 1, 25, 56
  - system(cls), 167
  - Tact, 1
  - time, 152
  - tokenize, 76
  - tr///, 46
  - trigram, 98
  - tris legomena, 175, 198
  - truncate, 153
  - TTR, 175, 198
  - uc, 31
  - ucfirst, 32
  - Unix, 2
  - unlink, 153
  - unshift, 86
  - use, 132
  - use Exporter, 134
  - use module, 134
  - use Text::Tabs, 95
  - utime, 152
  - values, 103, 110
  - variable, 11
  - VB, 1, 71, 72
  - VMS, 2
  - vocabulary growth, 164, 166
  - vocabulary overlap, 175
  - vocabulary richness, 1
  - vowel cluster, 56
  - w, 61
  - while, 28
  - wild card, 49
  - Win32::File::GetAttributes, 152
  - Win32::File::SetAttributes, 152
  - Windows (X86), 3
  - Windows Wordpad, 7, 11
  - word length distribution, 3, 187
  - word token, 1, 101, 109, 110, 121, 144, 188, 195
  - word type, 1, 78, 100, 101, 109, 110, 145, 195
  - WordSmith, 1
  - write filehandle, 155
  - x, 21, 45
  - XML, 60
  - Yule's K, 1, 125