Studies
in Quantitative Linguistics
9

Fan Fengxiang

Data Processing and Management for Quantitative Linguistics with Foxpro

RAM - Verlag

Data Processing and Management for Quantitative Linguistics with Foxpro

by

Fan Fengxiang

2010
RAM-Verlag

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

Preface

Imagine a researcher of Shakespearean plays is studying the Bard's stylistic characteristics with the quantitative approach. He has all the plays totalling about a million words stored in XML files. The immediate task before him is to remove all the XML codes from the files to get "pure" text. Next, he needs the following data: a wordlist with frequencies and word length both in letters and syllables, the vocabulary richness and frequency spectrum of each of the plays, lexical similarity and distance among the plays, the average word length in syllables and the average sentence length of each of the plays, collocations of certain words, number of rare words—hapax legomena, vocabulary growth rate, etc. However, life of a linguistic researcher is not as simple as that. To get a wordlist with word frequencies he'll need to lemmatize all the word tokens in those plays, and as the research progresses, some ad hoc research inspirations may pop up and new data are needed; he also has to constantly rearrange the data trying to find some patterns and retrieve some for a closer look, etc. These tasks would take ages to complete manually. The well known American scholar Ione Dodson Young used 25 years to make a concordance for the complete poetic works of Byron; she started the work in 1940 and didn't compete it until 1965!

With Foxpro, a powerful data processing and managing system, all the above can be done in a matter of a few minutes. This book, *Data Processing and Management for Quantitative Linguistics with Foxpro* gives detailed descriptions and instructions on how to gather, process and manage large amount of linguistic data with this data managing system. This book is aimed at literary and linguistic researchers, teachers and students at the undergraduate or postgraduate levels, EFL/ESL teachers and students, etc. It is also a very good book for corpus linguistics, text mining, information retrieval, and natural language processing. No previous computer programming experience is required of the reader except the ability to use the Windows Operating System.

All the examples for the commands and functions, as well as the demonstration programs in the book, are literary/linguistic oriented and of the author's own creation, 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. There is a CD-ROM attached to the book with all the Foxpro tables, examples, demonstration programs and non-copy-right textual materials for all the programs, exercises and model answers to these exercises.

There are different versions of Foxpro, and the latest version is Visual Foxpro 9. The Foxpro needed in this book is Foxpro 6 or higher. Foxpro can process any language in the world; however, in this book, it's used mainly to deal with English, occasionally Chinese. With some changes, the programs in the book can also be adapted to process other languages.

The following are some suggestions for tackling this book.

Firstly, this book is not for reading, but for careful reading plus repeated practice. That is, the reader should sit in front of the computer trying out each of the operators, commands, functions and examples many, many times while reading it. The operators, commands and functions in this book, totalling about 200, were carefully selected and are the most fundamental for linguistic computing. In some other computer languages there are fewer commands and functions; however, the users have to create commands and functions themselves when needed, and this makes these types of languages more difficult to learn and use for linguistic researchers and students. The reader of this book is not expected to remember all these operators, commands, functions, etc, by heart. He or she can always come back to this book to refresh his or her memory.

Secondly, as mentioned before, used as a course book, it'll take about a semester, roughly 36 lab hours to complete, and for each lab hour, the students need at least two more hours for home practice. For self-study, it'll take half a year. A person hurrying through the book in 10 days will probably learn nothing.

Thirdly, all the examples and exercises were carefully planned. The reader is not expected to solve all the problems in the exercises. One of the purposes of the exercises is for making the reader think about the possible applications of the operators, commands and functions etc learned; if the reader is unable to do the exercises, that's perfectly normal for a beginner; in such cases, go to the model answers, analyse them and then try them out. This is an important learning process.

Lastly, the author hopes that the above will not scare off potential readers. Please bear in mind that there are no magic books in the world from which a beginner can learn a computer language in 10 or 20 days. Learning a computer language from scratch is not like reading Shakespeare or Goethe for the first time; it's a long and sometimes painful process, and patience and perseverance are a must. But once learned, it'll be an open sesame for the learner to the wonderful linguistic and literary treasure trove that can last a life time.

The author is deeply indebted to Professor Gabriel Altmann for his insightful suggestions for this book and for his constant stimulating research ideas from which the author has benefited greatly; without his support this book wouldn't be possible. The author also wishes to thank Professor Reinhard Köhler for reading the manuscript and for his expert advice.

Fan Fengxiang

Table of Contents

1 Introduction

1.1 Scope and Methods of Quantitative Linguistics

Quantitative linguistics, as Köhler and Altmann define it, is the branch of linguistics that studies the multitude of quantitative properties which are essential for the description and understanding of the development and the functioning of linguistic systems and their components. The objects of QL research do, therefore, not differ from those of other linguistic and textological disciplines. In the Preface to *Quantitative Linguistics, an International Handbook*, Köhler, Altmann and Piotrowski list the following major areas of quantitative linguistics:

1. metricizing (scaling, quantifying, making measurable or quantitation, as M. Bunge calls it) of linguistic entities and qualities, and thereby providing the possibility of generating quantitative data from speech material based on operationalisation and measurement,
2. quantitative analysis and description of linguistic and textual objects,
3. numerical classification of linguistic and textual objects for the purpose of further investigation or for practical reasons,
4. development and application of statistical test procedures for diagnostic comparison of linguistic and textual objects and for trend detection,
5. modelling of linguistic structures, functions, and processes by means of quantitative models and mathematical methods,
6. theory construction by searching for universal laws of language and text and their embedding into an extensive nomological net,
7. explanation of linguistic phenomena (properties, structures, processes) by means of a theory,
8. embedding of linguistics into a general system of sciences, i.e. establishing resp. exploring interdisciplinary relations in the shape of generalization, analogy or specification,
9. elaborating a genuine linguistic methodology with regard to the particular characteristics of the linguistic subject,
10. practical applications to various areas such as those in contexts of learning and teaching, psychology/psycholinguistics/psychiatry, stylistics/forensics, computational linguistics and language technology, documentation science, content analysis, language planning, mass communication research and more.

Quantitative linguistics relies on quantification, measurement and ranking of components of a linguistic system and is generally data-intensive. However, the processing and management of large amount of linguistic data are extremely time consuming, tedious and error prone. Suppose we want to study the stylistic characteristics of Dickens's works, which total more than 5,000,000 words, such as vocabulary richness, word frequency distribution, etc. This can't be done manually, and a set of programs in a computer language are needed. But as the research progresses, some ad hoc tasks may arise, which may need the rearrangement of the data or the extraction of new data etc. In such cases more programs would be needed. If there is a computing tool that extracts data from a text or collection of texts and stores the data in an organized way for further processing or retrieval using a few simple natural-language-like commands instead of a set of complicated programs, the life of concerned researchers would be much simpler and more enjoyable since they themselves can use such a data processing and managing tool.

1.2 Visual Foxpro, an Overview

1.2.1 Advantage and capacity

Visual FoxPro (hereafter referred to as Foxpro) is a powerful and widely used computer database management system. It has a set of natural-language-like commands and functions for data processing and management. In addition, these simple commands and functions can be put together to form a program for continuous data manipulation. It's particularly suitable for linguistic computing because of the following:

1. It stores data in tables for processing and retrieval.
2. It has a set of easy-to-use, natural-language-like commands and functions for table handling.
3. It's very user-friendly; many of the commands and functions can be entered in its command window with instant results displayed on the screen.
4. It can perform complicated math operations and string manipulation. There is virtually no limit to what it can do in quantitative linguistic computing.

The only disadvantage of Foxpro is that the user has to pay for it, but considering the long term benefit it can bring to us, the money is well worth paying.

1.2.2 System requirement and installation

The latest version of Foxpro is Visual Foxpro 9.0. For linguistic computing, Visual Foxpro 6.0 is quite enough and is used throughout this book. Visual Foxpro 6.0 and higher requires an IBM-compatible computer with a Pentium class processor, with at least 128 MB RAM, and 500 MB free disc space. For operating systems, Visual Foxpro 6.0 and 7.0 are supported on Windows 98 or higher. While Visual Foxpro 8.0 is supported on Windows 2000 Service Pack 2 or higher, and Windows XP; Visual Foxpro 9.0 is supported on Windows 2000 Service Pack 3 or higher, and Windows XP. Commands, functions and operators of Visual Foxpro 6.0 and programs written in it can run in higher versions. Foxpro is very easy to install: shut down all the running application packages such as Microsoft WORD, insert the Foxpro CD-ROM, click *Prerequisites* and the setup will start until it's completed.

Start Foxpro and the main Foxpro window with its menu bar and command window appears, as shown in Figure 1.1. The command window can be hidden by clicking on the command window icon on the menu bar; click on it again the command window reappears. Its size can be adjusted by dragging one of its sides with the mouse.

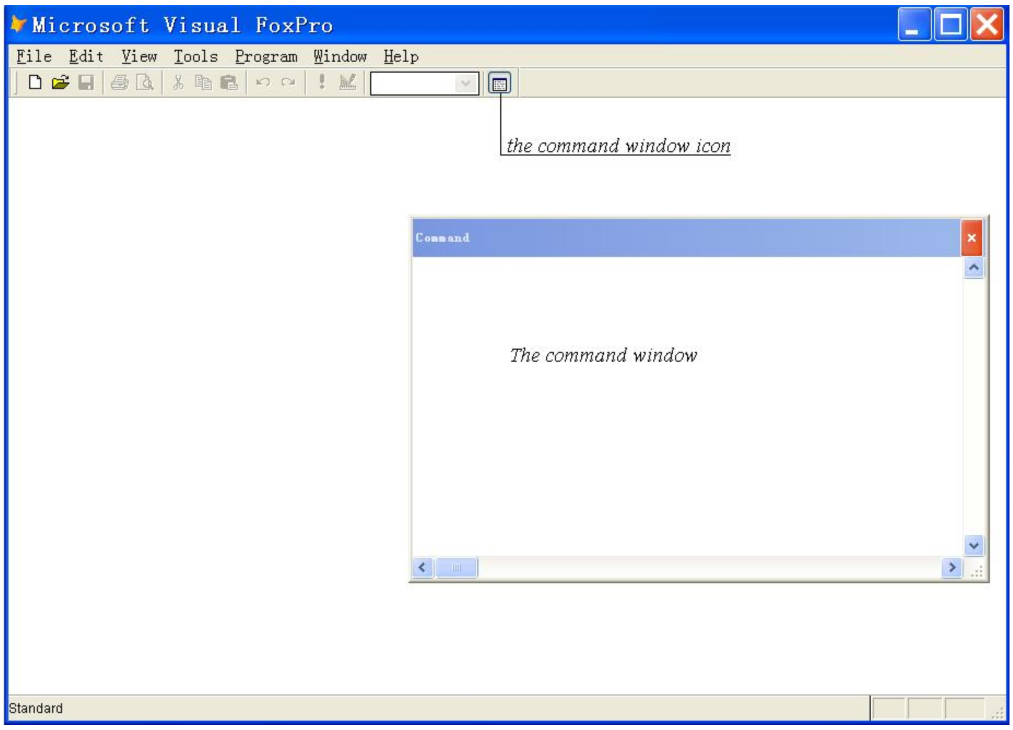


Figure 1.1 The Foxpro window with its menu bar and command window.

1.2.3 Foxpro variables

A Foxpro variable is a temporary storage that stores whatever it is given. They exist as long as Foxpro remains open after they are created. The name of a variable can be a single or a cluster of alphabetic characters and the underscore character "_". Arabic numerals can be used with these characters in variable names as long as they are not placed initially. The following are valid variable names:
a, counter, text_1, nloop, read_a, c_34, change_case, no_410, count_it, word, wordlist, length, etc.

Punctuation marks and characters such as *, -, +, =, (,), [,], {, }, %, @, &, ^, $, ~, \, /, |, >, <, %, #, etc, and Arabic numerals used alone or placed initially, are not allowed in naming variable. The following are invalid variable names:
*1, 284, $12, @, *w, b-13 &text, word-length, \get_text, word>, etc.*

Words used in Foxpro built-in functions and commands can't be used alone as variable names, either. For example, *do, if, count, list, display, delete, shared* and so on, but they can be used as variable names together with other legal characters, e.g. *count_word, list_texts* etc. Foxpro variable names, commands and functions are not case sensitive, so the variable name *read_text* is the same as *Read_Text*.

We can store data in a variable by using the command **store…to** or the equal sign =. This is called value assignment. For language processing, we mainly use two types of data in Foxpro: numeric data such as 34.56, 1003, and character data, such as *a*, *G*, *apples*, *words*, *that is easy* and so on. If we have a variable *v1*, we can assign any numeric or character value to it. When we assign character values to a variable, the characters must be enclosed between a pair of single quotes or double quotes. The result can be outputted to the screen by putting a question mark before the variable. Now start Foxpro and type the following in the command window. The sign ↵ stands for "press Enter":

```
number1=567 ↵
? number1 ↵
567

store 271 to number2 ↵
? number2 ↵
271

Phrase_a='Visual FoxPro' ↵
?phrase_a ↵
Visual FoxPro

store 'Quantitative Linguistics' to phrase_b ↵
?phrase_b ↵
Quantitative Linguistics
```

1.2.4 Foxpro operators

Foxpro has three major types of operator: character operators, numeric operators and relational operators.

1. Character operators: +, -, =, ==, $. + joins two strings together. - removes the trailing spaces of a string and then joins it with another string. = and == match a character or a string on the left with another character or string on the right. The character matching mode of = or == depends on the Foxpro commands **set exact on** and **set exact off**. If the command *set exact on* is issued, for the return value to be true, the two strings to be matched must be exactly the same; if *set exact off* (the default setting) is issued, for the return value to be true, the length of the two

strings doesn't have to be equal; the second string can be shorter than the first string, and as long as the characters of the two strings match one for one starting from the left of the strings until the end of the second string is reached, the return value is true. == checks whether a string exactly matches another string; the *set exact on* and *set exact off* commands have no affect on it. *$* checks whether a string is contained in another string. Unlike in Foxpro commands and functions, string literals (strings placed in single quotes or double quotes) are case sensitive. Now type the following in the command window and see the results. **&&** is used at the end of a statement to mark comments or explanations. It's non-executable and is ignored by the computer.

```
? ' Fox ' +'pro' ↵ &&there is a space after Fox
Fox pro

? 'Fox '- 'pro' ↵ &&there is a space after Fox
Foxpro

? 'Foxpro '= 'Fox' ↵
.T.

? 'Foxpro '= 'fox ' ↵
.F.

? 'Fox '= 'Foxpro ' ↵
.F.

? 'Foxpro '= 'Foxpro' ↵ &&there is a space after the first Foxpro
.T.

? 'Foxpro '==' Fox' ↵
.F.

? ' Foxpro'==' Foxpro' ↵
.T.

set exact on ↵
? 'Foxpro'='Fox' ↵
.F.

? 'Foxpro '='Foxpro' ↵ &&there is a space after the first Foxpro
.F.
```

```
? 'Fox’$’Foxpro' ↵
.T.

? 'fox'$'Foxpro' ↵
.F.
```

In Foxpro, the white space separating two words is also regarded as a character and can be assigned to a variable. One way of representing the white space is putting a white space between a pair of single quotes or double quotes. We can also assign nothing to a variable by using a pair of quotes without anything in between.

```
space=' ' ↵ &&there is a white space between the quotes
? 'Fox'+'pro' ↵
Foxpro

? 'Fox'+space+'pro' ↵
Fox pro

nothing='' ↵ &&no space between the quotes
? 'Fox'+nothing+'pro' ↵
Foxpro
```

2. Numeric operators: **+**, addition; **-**, subtraction; *****, multiplication; **/**, division; ****** or **^**, exponentiation; **%**, modulo (the remainder). Now type the following in the command window:

```
?5*6-30 ↵
0

?(45+67)/22**2 ↵
0.23

?11%2 ↵
1
```

3. Relational operators: <, less than; >, greater than; =, equal to; >=, greater than or equal to; <= less than or equal to; <>, #, **!=**, unequal to. Now type the following in the command window:

```
?23>56 ↵
.F.
```

```
?45>2 ↵
.T.

?5>=4 ↵
.T.

?6<=3 ↵
.F.

?7<>8 ↵
.T.

?9=10-1 ↵
.T.
```

Except <= and >=, these operators can also be used for string comparisons:

```
? 'm'> 'n' ↵
.F.

? 'b'> 'a' ↵
.T.

? 'What'<> 'what' ↵
.T.

? 'what'='what' ↵
.T.
```

1.2.5 Commands and functions for math operations

Apart from math operators, there are commands and functions for math operations. The following are some of them.

set decimal to *decimalplace* This command sets decimal places for math operations. The default decimal place of Foxpro is 2. Now type the following in the command window:

```
?10/3 ↵
3.33

set decimal to 5 ↵
```

```
?10/3 ↵
3.33333

set decimal to 0 ↵
?10/3 ↵
3
```

Foxpro allows the user to shorten the components of a command or function down to four letters if they are longer than four letters. For example, the above command can be written as *set decima to*, *set decim to* or *set deci to*, but not *set dec to*.

abs(*n*) This function returns the absolute value of *n*. Type:

```
?abs(-5) ↵
5
```

log(*n*), **log10**(*n*) The former returns the log of a number to the natural base *e* and the latter to the base 10. Type the following in the command window:

```
?log(20) ↵
2.9957

?log10(20) ↵
1.301
```

Foxpro doesn't have a built-in function for log to the base 2. We can convert *log(n)* and *log10(n)* to the base 2 using the following:

log(*n*)/**log**(**2**)

log10(*n*)/**log10**(**2**)

```
?log(100)/log(2) ↵
6.643856

?log10(100)/log10(2) ↵
6.643856
```

To check whether this result is correct, type

```
?2**6.643856 ↵
```

the result is 100.

pi() This function returns the constant π. Now type:

```
set decimal to 4 ↵
?pi()
3.1416
```

round(*n, decimalplace*) This function rounds off decimal numbers. Now type:

```
?round(3.1415926,3) ↵
3.142

?round(1.9,0) ↵
2
```

We can shorten *round* in this function to *roun*. Type:

```
?roun(3.1415926,3) ↵
3.142

?roun(1.9,0) ↵
2
```

int(*n*) This function discards decimals and keeps only the integer part of a number:

```
?int(10.96) ↵
10
```

floor(*n*) This function returns the nearest integer that is less than *n* (*n* is a decimal number):

```
?floor(-10.8) ↵
-11

?floor(10) ↵
10

?floor(10.8) ↵
10
```

```
?int(-10.8) ↵
-10

?int(10.8) ↵
10
```

ceiling(*n*) This function returns the next highest integer that is greater or equal to *n*:

```
?ceiling(-10.8) ↵
-10

?ceiling(10.8)
11
```

between(*n1*, *n2*, *n3*) This function tests whether *n2* is smaller than *n1* and *n3* (*n3*≥*n1*) in value:

```
?between(255,189.5,263.) ↵
.T.

?between(255,189.5,240) ↵
.F.

?between (5,7,12) ↵
.F.
```

This function can also be used for characters:

```
between('cat', 'ant', 'dog') ↵
.T.

?between('cat', 'ant', 'ax') ↵
.F.

?between(' cat', 'fox', 'horse') ↵
.F.
```

rand() This function generates random numbers between 0 and 1. Type:

```
set deci to 4 ↵
?rand() ↵
```

```
0.8723

?rand() ↵
0.0237

set deci to 15 ↵
?rand() ↵
0.851390329189599

?rand() ↵
0.213546234443784
```

To generate a series of random numbers with maximum randomness, use this function with a negative number in the brackets first and then use the function without any number in the brackets. Suppose we want to generate five random numbers, type:

```
rand(-391) ↵
?rand() ↵
0.45

?rand() ↵
0.13

?rand() ↵
0.12

?rand() ↵
0.89

?rand() ↵
0.36
```

mod(*n1***,** *n2*) This is the same as the % operator, which returns the remainder of *n1* divided by *n2*:

```
?mod(10,2) ↵
0

?mod(20,3) ↵
2
```

sqrt(*n*) This function returns the square root of a number. Type:

```
?sqrt(100) ↵
10

?sqrt(35.67) ↵
5.9724
```

exp(*n*) This function returns the n^{th} power of the natural base *e* 2.71828. Set decimal to 5 and then type:

```
?exp(1) ↵
2.71828

?exp(10) ↵
22026.4658
```

cos(*n*) This function returns the cosine of *n* in radians. Type:

```
?cos(1) ↵
0

?cos(pi()) ↵
-1
```

acos(*n*) This function returns in radians the arc cosine of *n*. The value of *n* ranges from -1 to +1. Type:

```
?acos(-1) ↵
3.1416

?acos(1) ↵
0
```

sin(*n*) This function returns the sine of *n*; *n* is expressed in radians. Type:

```
?sin(30) ↵
-0.988

?sin(pi()/2) ↵
1
```

asin(*n*) This function returns in radians the arc sine of *n*. Type:

```
?asin(1) ↵
1.5708

?asin(0.8) ↵
0.9273
```

tan(*n*) This function returns the tangent of *n*; *n* is expressed in radians. Type:

```
?tan(1) ↵
1.557

?tan(pi()/4) ↵
1
```

atan(*n*) This function returns in radians the arc tangent of *n*. Type:

```
?atan(1) ↵
0.7854

?atan(2) ↵
1.1071
```

dtor(*n*) This function converts *n* in degrees to radians. Type:

```
?dtor(90) ↵
1.5708

?dtor(180) ↵
3.1416
```

rtod(*n*) This function converts *n* in radians to degrees. Type:

```
?rtod(3.1416) ↵
180.004

?rtod(1.5708) ↵
90.002
```

To clean the main Foxpro window of the output produced by the commands and functions we have issued, use the command **clear**. Now type:

```
clear ↵
```

1.2.6 Foxpro programs

There are two modes of executing Foxpro commands and functions. One is what we have been doing now, that is, using the command window to execute commands and functions; the other is putting commands and/or functions in a file and let the computer carry out these commands and functions in the file one by one from the top to the bottom of the file. Such a file is the so called program. Foxpro programs have the file extension *prg*.

Before learning how to write a program, we'll look at the command needed for writing or revising programs or text files.

modify | [**command** [*programname*]] [**file** [*filename*]] |

If we type only *modify command* ↵ in the command window, an empty file called *program1* appears, in which we can write our program and then give it a name and save it. We can also type *modify command* followed by the program name, say, *prog*, and an empty file with the name of *prog.prg* appears, in which we can enter our program and then save it. If we want to revise the program after it's completed and closed, just type *modify command prog* ↵ in the command window and the program appears for our revision. If we change *command* into *file*, we can create or modify a text file. Now let's write our program. Type *modify command* ↵ in the command window, and then type the following into the empty file:

```
phrase1='This is the first '
phrase2='Foxpro program'
?phrase1+phrase2
? “We'll first perform math operations. 67*45+56.3/2**3=?”
a= 67*45+56.3/2**3
?a
? 'What is the remainder of 17 divided by 5?'
?17%5
```

After completing this little program, click *File* on the Foxpro menu bar and select *Save as* and then select a drive and folder and save it as *firstprog.prg* in one of your folders on your computer; the *prg* extension is automatically added. To run it, click the red exclamation mark on the menu bar or type *do firstprog* ↵ in the command window, and the result is shown on the main Foxpro window. To close the program, click the × sign on the upper right corner of the program. It can be opened again by either typing *modify command firstprog* in the command window, or click on *File* on the menu bar and select *open*, and select *program* in the *file type* box of the *open file* window, and locate the folder of the program and then get it.

We can also use another way to start writing a program. Click the new file icon 🗋 on the menu bar of the Foxpro window to get the *file type selection* box. Select *program* and then click on *New file*, an empty file appears. We can then write our program in it and then save it.

In Foxpro programs, a line consisting of commands, functions, operators etc and ending with a carriage return is called a statement. If a statement is too long, we can break it into two or more lines with a semicolon followed by a carriage return, as shown below:

```
create table lexinfo(texts c(25),tokens n(8), vocsize n(4),freq1 n(5),freq2;
n(5),freq3 n(5),freq4 n(5))
```

Although there are two lines, they form only one statement. In some programs in later chapters there are statements like the following:

```
create table lexinfo(tablename c(25),tokens n(8),textvoc n(6),vocgrowthn
(4),freq1 n(5), freq2 n(5),freq3 n(5),freq4 n(5),freq5 n(5),freq6 n(5),freq7
n(5),freq8 n(5))
```

These three lines are actually one long line wrapped around by the word processor because of the width of the page; there is no semicolon or carriage return at the end of the “first line” and “second line”. In cases like this the reader should enter the statement as one line in the program editor or break it into two or more lines with semicolons followed by a carriage return. Otherwise the program won’t run.

1.2.7 Commands for Foxpro settings

Foxpro commands, functions and programs can be executed under different Foxpro settings. There are commands for Foxpro settings, and we have learned some of them, e.g. *set exact on*, *set exact off*, *set decimal to*, etc. The following are some other commands for Foxpro settings.

set safety on This is the default setting. In this setting, when a file is going to be overwritten, deleted etc, the computer pauses to ask for the user’s confirmation. This setting is seldom used in programs because the user has to sit in front of the computer during the execution of a program to give instructions until the execution is completed, otherwise the computer would pause indefinitely.

set safety off This command allows the computer to overwrite a file or replace a file with another file that has the same filename without notifying the user, often used in programs.

set talk on This is the default setting. In this setting, when a program is running, real time information is displayed in the Foxpro window on the progress of the program. This setting slows down the computer considerably.

set talk off In this setting, real time information display is suppressed and the computer is much faster than in the *set talk on* setting.

set default to *path* This command tells the computer of the default drive and folder so that it can get files from or save files to that drive and folder.

cancel This command stops a program from where it is issued, often used for checking the results of a statement or for debugging. Now open *firstprog.prg* we’ve just written and put *cancel* after the third statement:

```
phrase1='This is the first '
phrase2='Foxpro program'
?phrase1+phrase2
cancel
?”We'll first perform math operations. 67*45+56.3/2**3=?”
a= 67*45+56.3/2**3
?a
? 'What is the remainder of 17 divided by 5?'
?17%5
```

Save the program and then run it. It stops after the third statement.

***** This command is always put at the leftmost position of a statement in a program for the computer to ignore this statement. This command is very useful for adding notes and comments in a program. Type the following in the command window:

```
This statement tests the function of * ↵
```

The above statement resulted in an error message. Now type:

```
* This statement tests the function of * ↵
```

The computer simply ignored the statement and no error message was given. We can use * to add notes to a program so that long after the program is written we can still understand it. It's also good practice to put a brief note stating the aim of the program at the top of it. Don't put a semicolon at the end of a note because the computer would take the next statement as part of the note and the program may crash. We can also use * to prevent a statement from being executed. Now open *firstprog.prg* again and revise it as follows:

```
*This is the first Foxpro program
phrase1='This is the first '
phrase2='Foxpro program'
?phrase1+phrase2
*?"We'll first perform math operations. 67*45+56.3/2**3=?"
*a= 67*45+56.3/2**3
*?a
? 'What is the remainder of 17 divided by 5?'
?17%5
```

Run it and see the result.

1.3 Conventions Used in This Book

In explaining Foxpro 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 **round**(*number, decimalplace*), which are for rounding off decimals, the function itself is **round**(); *number* and *decimalplace* should be replaced by the user with a decimal number and the desired decimal places.
2. In cases where a command contains several optional expressions, the options are in square brackets; if two or more optional expressions in square brackets of the same level are between a pair of vertical lines, only one expression can be selected. For example, in the command **append from** *filenames* | [**sdf**] [**delimited with** |[**tab**] [**blank**] [*character*]|] | [*fieldnames*] [**for** *condition*], **append from** *filename* should be followed by either **sdf** or **delimited with** plus one of the options **tab**, **blank** or *character*, and/or followed by *fieldnames* or **for** *condition* or both. If there are three dots in a command or function, the three dots represent more optional expressions. For example, in **create table** *tablename*(*fieldname1* **c**(*width*) [, *fieldname2* **n**(*digit*)] [, *fieldname3* **m**(*4*)]…), the three dots mean more such expressions can follow.
3. Foxpro tables have the file extension of *dbf*, e.g. *wordlist.dbf*, *vocgrowth.dbf*

etc; and Foxpro programs have the file extension of *prg*, e.g. *tokenizer.prg, binomial.prg* etc. In this book Foxpro tables are referred to without the file extension while other types of file are referred to with their file extensions. For example, *wordlist.dbf* is referred as *wordlist*, and *vocgrowth.dbf* as *vocgrowth*, etc.

The programs and data used in this book are placed at the RAM-Verlag on the Internet, as well as on a CD-ROM, and it has the following structure:

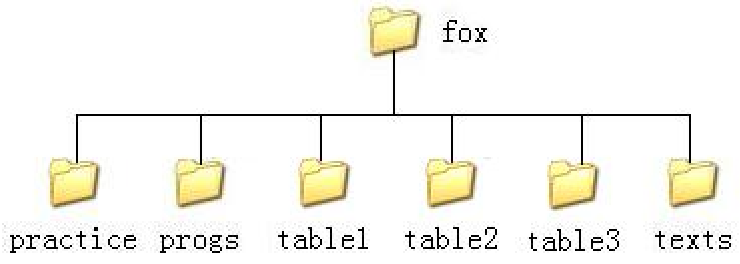


Figure 1.2 Folder structure

The contents of the folders are as follows:
practice: for holding programs, tables, texts etc created by the reader during practice, currently empty.
progs: containing all the Foxpro programs in this book, including the model programs for exercises at the end of each chapter. These programs were all written by the author and computer tested.
table1: containing three sets of wordlist tables. Each set has 50 wordlists made from 50 2000-word random samples from the BNC spoken text section. The difference among the sets is that one set is unlemmatized, one is lemmatized, and the third one has POS tags.
table2: containing three sets of wordlist tables. Each set has 50 wordlists made from 50 2000-word random samples from the BNC written text section. The difference among the sets is that one set is unlemmatized, one is lemmatized, and the third one has POS tags.
table3: holding tables such as *80vgrowth*, *postable*, *filename*, *wordlist* and so on.
texts: containing Lewis Carroll's *Alice's Adventures in Wonderland* (*alice.txt*), *Through the Looking-glass* (*lglass.txt*), 48 text chunks from *alice.txt* (*text1.txt* to *text48.txt*), a short passage in Chinese (*chinese.txt*), several supporting text files for some programs etc.

Copy or download the entire *fox* folder to a drive on your computer, say, *d* and make *d:\fox\practice* the default directory for your Foxpro practice by entering the following in the Foxpro command window:

```
set default to d:\fox\practice ↵
```

From now on we assume the default drive and folder on your computer for Foxpro practice is *d:\fox\practice*. At the end of each chapter there are exercises, some of which have model answers in *Model Answers to Selected Exercises* in *Appendices A*. Foxpro can process any type of language, but the language processed with Foxpro in this book is mainly English.

We have now had a glimpse of Foxpro. In the following chapters we'll have a detailed look at its commands, functions and utilities that can be used in linguistic computing. Apart from the commands, functions and utilities that are covered in this book, there are quite a number of other commands, functions and utilities that are of no immediate use for language processing. Interested readers can learn to use them through books on Foxpro that contain introductions to these commands, functions and utilities. Please be noted that all the textual data to be processed with Foxpro must be in pure text forms. If a text to be processed is a WORD document, convert it to a pure text file with the *txt* extension in WORD. For ease of explanation, from now on we'll put line numbers in our Foxpro programs. But the reader should never put line numbers in Foxpro programs intended for running because Foxpro does not allow line numbers in its programs.

Exercises

1. Assign the following values to properly named variables and output the values of the variables to the screen.
a. 34.56
b. Foxpro is a powerful data managing system

2. Assign each of the following words to different variables, join these variables together and output the result to the screen.
Foxpro
has
a
high
level
computer
language.

3. The Type/Token ratio (TTR) is obtained with $TTR = \frac{Types}{Tokens}$. However, Laufer & Nation and Biber et al use the following:

a. $TTR = 100\frac{types}{tokens}$.

While Köhler and Galle propose a method for calculating the type/token ratio of a section of a text, *TTRx*:

b. $TTR_x = \dfrac{t_x + T - \dfrac{xT}{N}}{N}$,

where x is the length of the section of the text, t_x the number of types of a section of a text, T the total number of types of the text, N the length of the entire text. If $t_x = 400$, $x = 1000$, $T = 1200$, $N = 2000$, calculate TTR and TTR_x using a and b.

4. Fan and Altmann tested the following hypothesis: the shorter a word (the number of syllables it has) the more compounds it can form. This relationship can be expressed with the following:

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

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
4	2.71
5	0.69

5. Do the following.
a. One of the methods for *N*-gram smoothing is the add-one smoothing. The smoothed probability of a *N*-gram is obtained with

$$P = \frac{c_i + 1}{N + V},$$

where c_i is the observed frequency of a *N*-gram in a corpus, N is the frequency of the first word of the *N*-gram, and V the size of vocabulary. Calculate the smoothed probability of the bigram *inside out* and *happy time* from a corpus whose vocabulary size is 13,500. The frequency of *inside out* and *happy time* is respectively 3 and 2; the frequency of *inside* and *happy* is respectively 23 and 45.
b. Another *N*-gram smoothing method is the Good-Turing estimation. The smoothed probability of a *N*-gram is obtained with

$$P = (c+1)\frac{N_{c+1}}{\frac{N_c}{N}},$$

where c is the count of N-grams of certain frequency (frequency of frequencies), N_c is the number of N-grams with count c, and N is the frequency of the first word of a N-gram. The following is part of the frequency distribution of the bigrams of a corpus.

C	Number of bigrams
1	10043
2	2331
3	1125
4	532

If the bigrams *run rampant* and *strong tea* respectively occur once and 3 times in this corpus, and *run* and *strong* respectively occur 145 times and 76 times, calculate the smoothed probability of the two bigrams using the Good-Turing estimation.

6. In performing the ANOVA test on sets of data consisting of percentages such as 23%, 36%, 67% etc, to normalize the data and stabilize the variances, we can use the arc sine square root transformation to transform the data and then convert the result of the transformation into angle degrees. The arc sine square root transformation procedure is as follows:

a. get the square root of each of the values of the data sets,
b. get the arc sine of the square root,
c. convert the radians into degrees.

Now do the arc sine square root transformation to the following set of data:
12%, 15%, 17%, 20%, 22%, 27%, 30%, 34%, 35%, 39%, 40%, 44%

7. Tuldava proposes that the relationship between vocabulary size V and the length of text is $V = Ne^{-\alpha(\ln N)^{\beta}}$, while Guiraud and Sánchez & Cantos describe such relationship with $V = a\sqrt{N}$. If for Tuldava's model α = 0.009152, β = 2.3057, N = 1000000; for Guiraud, Sánchez & Cantos's model α = 65.7365677, and N = 1000000; calculate V of both models.

8. Honoré proposed the following relationship:

$$H = 100\frac{\ln N}{1 - \frac{v(1,N)}{v(N)}},$$

N is the length of a text, *v(1,N)* the number of hapax legomena. H is more or less constant. Now calculate H for:

a. $N = 98000$, $v(1,N) = 3473$,

b. $N = 182000$, $v(1,N) = 4536$.

9. 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 ($V = 20$):

Rank	Frequency
1	1635
2	872
3	825
4	730
5	687
6	540
7	531
8	528
9	513
10	410
11	398
12	367
13	364
14	315
15	274
16	263
17	247
18	211
19	194
20	182

10. Compute the following:

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

2 Foxpro Tables

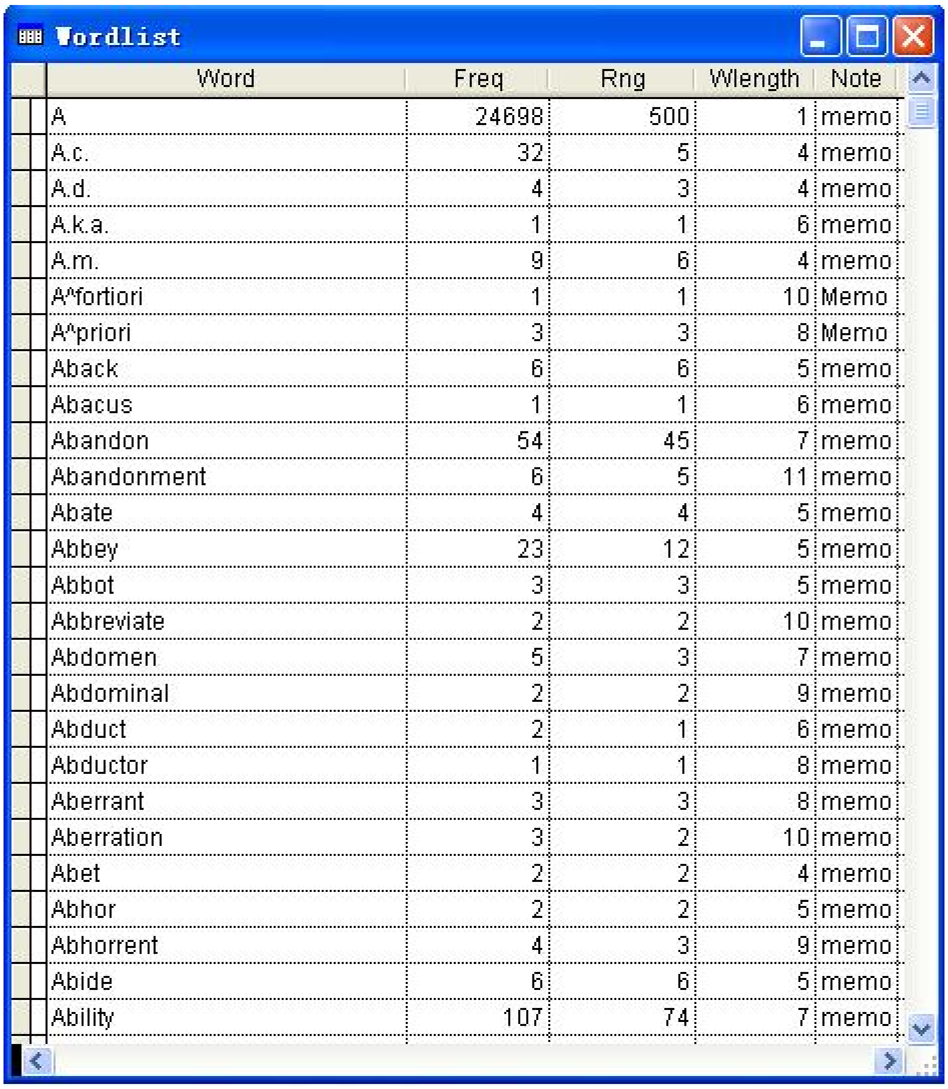
2.1 Introduction

In Foxpro data are mainly stored and handled in Foxpro tables. Like ordinary tables, Foxpro tables consist of rows and columns. A row in a Foxpro table is called a record, and a column a field. A Foxpro table can have as many as 1,000 million records and 255 fields, forming 1,000 million × 255 cells, and each can hold an item of data. This item of data can be a word, a number or any other types of alpha-numeric data. These cells can be used to store linguistic data such as words or phrases, word frequency, word length, sentence length and so on. The maximum width of a field for non-numeric data is 254 characters, wide enough for any word or phrase. There is a special type of field, the memo field, which can store data of unlimited length. This is particularly useful for storing sentences or texts. For numeric data, the width of a field is 20 digits. Apart from alpha-numeric data, a field can also store graphic data such as charts, pictures and so on. Data of this sort is called general data. All Foxpro tables have the file extension *dbf*.

Figure 2.1 is part of a Foxpro table called *wordlist* (in *d:\fox\table3*). It contains the vocabulary of 500 2000-word samples randomly drawn from the written text section of the BNC. The table has five fields: *word*, *freq*, *rng*, *wlength* and *note*. The first four fields hold respectively words, frequency, range (the number of samples a word occurs in) and word length in letters. The fifth field is a memo field, which has the cell marker *memo* in all the cells. Double click on the cell marker *memo* and the contents are displayed on the screen. If a cell in a memo field contains data, its cell marker is *Memo*, instead of *memo*. We can see now only the sixth and seventh cells of the *note* field contain data but the rest are empty. The width of the *word* field is 25 characters, wide enough for holding words; the width of the *freq* field, *rng* field and *wlength* field are respectively 10 digits, 10 digits and 6 digits. The naming of a table, as well as its fields, is the same as that of Foxpro variables, but it's better for the names to be suggestive. If we want to create a table to hold two sets of wordlists and their respective frequency, we can name the table *wordlist*, and the fields *word1*, *word2*, *freq1*, *freq2*.

2.2 Table Creation and Modification

Foxpro tables can be classified into simple tables and multi-field tables. The former has under 11 fields and the latter 11—255 fields.



Wordlist

Word	Freq	Rng	Wlength	Note
A	24698	500	1	memo
A.c.	32	5	4	memo
A.d.	4	3	4	memo
A.k.a.	1	1	6	memo
A.m.	9	6	4	memo
A^fortiori	1	1	10	Memo
A^priori	3	3	8	Memo
Aback	6	6	5	memo
Abacus	1	1	6	memo
Abandon	54	45	7	memo
Abandonment	6	5	11	memo
Abate	4	4	5	memo
Abbey	23	12	5	memo
Abbot	3	3	5	memo
Abbreviate	2	2	10	memo
Abdomen	5	3	7	memo
Abdominal	2	2	9	memo
Abduct	2	1	6	memo
Abductor	1	1	8	memo
Aberrant	3	3	8	memo
Aberration	3	2	10	memo
Abet	2	2	4	memo
Abhor	2	2	5	memo
Abhorrent	4	3	9	memo
Abide	6	6	5	memo
Ability	107	74	7	memo

Figure 2.1 Table *wordlist* and its fields

2.2.1 Creating simple tables

The command for creating a table is as follows:

> **create table** *tablename*(*fieldname1* **c**(*width*) [, *fieldname2* **n**(*digit*)] [, *fieldname3* **m**(*4*)]…)

In the table creation command, the letters *c*, *n* and *m* before *(width)*, *(digit)* and *(4)* respectively stand for character field, numeric field and memo field. *width* and *digit* stand for the width of the field, i.e. how many characters or digits a cell of a field can hold. They should be replaced by the user with numbers in actual table creation. If we want the table to hold strings and numbers up to 25 characters and

10 digits in width, put 25 and 10 in the fields respectively. As mentioned before, for character fields the maximum width is 254, for numeric fields it's 20 digits, including the decimal point. By convention the width of the memo field is always 4, but it can contain data of almost any length. If we wish to hold decimal numbers in a numeric field such as 390913.1416, replace *n(digit)* with *n(11,4)* because the total length of 390913.1416 is 11 digits (including the decimal point), and has 4 decimal places.

Now let's create a table called *table1* with the same field names as *wordlist*, but with different field width. The width of the character field is 45 while those of the numeric fields are respectively 12, 7 and 6. The *freq* field hold decimals with 6 decimal places. Now type in the command window:

```
set default to d:\fox\practice ↵
```

then type:

```
create table table1(word c(45),freq n(12,6),rng n(7),wlength n(6),note m(4))
↵
```

table1 is now created and it's in *d:\fox\practice*. To view it, type **browse** in the command window. To hide it from the screen, either press Esc or click on the × sign on the upper right corner of the table. The commands to close a table is **use**, which physically closes the table; or

close | [**databases**] [**all**] |

close databases closes all open tables while *close all* closes all tables, programs and text files that are open in Foxpro. Now type:

```
use ↵
```

table1 is physically closed. The command to open a table is

use *tablename*

To open *table1* and view it again, first make sure you are now in its drive and directory, and type:

```
use table1 ↵
browse ↵
```

table1 appears again. If you are not in its drive and directory, type:

```
use d:\fox\practice\table1 ↵
browse ↵
```

To close it again type:

```
close databases ↵
```

Once a table is created using the *create table* command, the table is stored on the hard disc and can be closed and opened as we wish. The following table creation command creates a temporary table. Once it's closed, it's erased and can't be opened again.

create cursor *tablename*(*fieldname1* **c**(*width*) [, *fieldname2* **n**(*digit*)] [, *fieldname3* **m**(*4*)]…)

Now enter in the command window the following statement:

```
create cursor temp(word c(25),freq n(5),rng n(4),wlength n(4),note m(4)) ↵
browse ↵
```

A temporary table called *temp* with five fields is created and we can input data to it or output data from it. But once it's physically closed it's automatically removed and can't be accessed again. Tables thus created are often used in programs that need temporary tables that are of no use after program execution so that we don't have to delete them manually.

2.2.2 Table modification

There are commands for modifying the structure of a table, e.g. changing its field width, data types, decimal places, dropping a field, adding a field or renaming a field. We can modify the structure of a table either manually or automatically. The command for manual modification is as follows:

modify structure

Unlike other commands we have learned, this one is used only in the command window. Now open *table1* again and type in the command window:

```
modify structure ↵
```

the table designer appears as shown in Figure 2.2. If we want to reduce the width of *word* to 25 characters, either type 25 in the width box, or scroll the down arrow until the width is 25. To change the data type of *rng* from numeric to character with a width of 10, click on the down arrow and select *character* and then select 10 in *width*; to drop the field *note*, click on it to mark it and then click *delete*; to change the decimal places of *freq* from 6 to 0, delete 6 in the *decimal* box; to add a numeric field called *random* with width 18 and 15 decimal places, click on the empty box under *field* and enter *random* in it, then select numeric in the data type box and scroll the down arrow of the width box and decimal box respectively to get the desired width. Click on *OK* and the modification is completed.

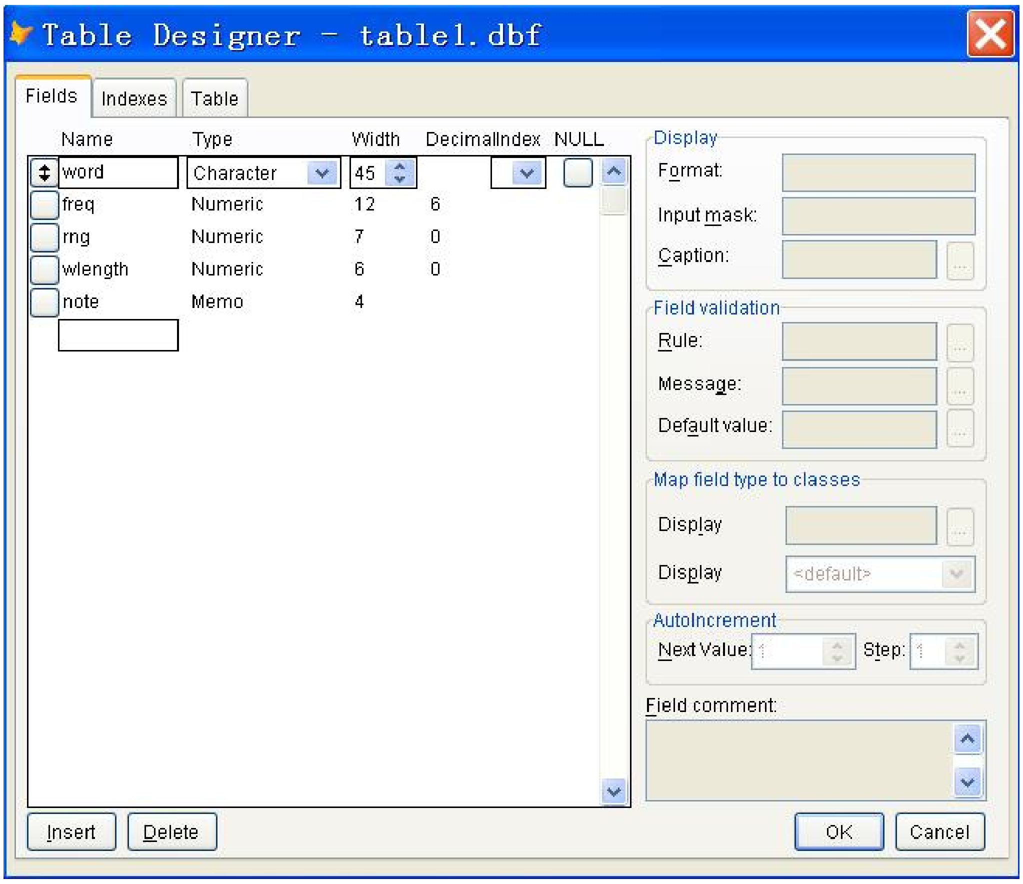


Table Designer - table1.dbf

Name	Type	Width	Decimal	Index	NULL
word	Character	45			
freq	Numeric	12	6		
rng	Numeric	7	0		
wlength	Numeric	6	0		
note	Memo	4			

Figure 2.2 Table Designer

The following command is for automatic table modification:

alter table *tablename* [**alter column** *fieldname datatype*(*width*)] [**rename column** *oldfieldname* **to** *newfieldname*] [**drop** *fieldname*] [**add** *fieldname datatype*(*width*)]   This command can be used either in the command window or in programs. Now open *table1* again. Suppose we want to change the width of *word* to 30, width of *freq* to 10 with 4 decimal places, the data type of *rng* back to numeric with a width of 6 digits, rename *wlength* to *length*, discard *random* and add a character field *context* with a width of 100, type in the command window:

```
alter table table1 alter column word c(30) alter column freq n(10,4) alter
column rng n(6) rename column wlength to length drop random add context
c(100) ↵
browse ↵
```

2.2.3 Creating multiple field tables

There are occasions when we need a table with multiple fields. For example, a table with 85 fields will be needed if we want to hold the vocabulary growth data of 80 sets of samples from a mega-corpus, as well as token number, mean vocabulary growth of the 80 sets, the standard deviation of the individual vocabulary growth of the 80 sets, and the 95% confidence intervals of the vocabulary growth. Such a table is almost impossible to create by entering commands in the command window. We can write a program to do it automatically. Before writing the program, we'll first look at the following commands and functions.

& *&* is Foxpro's macro operator, which can turn a string literal into a Foxpro command. Now hide *table1* if it's still visible on screen, and type:

```
word='browse' ↵
? word ↵
browse
```

the word *browse* is outputted to the screen. Now enter the following:

```
word='browse' ↵
&word ↵
```

Instead of the word *browse, table1* appears because the macro operator *&* regards the string literal *browse* stored in the variable as the command **browse**. Enter the following in the command window:

```
maketable='create table table2(v1 n(8), v2 n(8), v3 n(8), v4 n(8)) ' ↵
?maketable ↵
create table table2(v1 n(8), v2 n(8), v3 n(8), v4 n(8))
```

the value of *maketable* is outputted to the screen.

```
&maketable ↵
```

a table called *table2* with four fields is created.

at(*character,string*) This function measures the position of the first occurrence of a character or characters in a string. Now type:

```
?at('r', 'tomorrow') ↵
5
```

The above statement measures the position of the first occurrence of *r* in *tomorrow,* which is 5.

rat(*character,string*) This function measures the position of the last occurrence of a character in a string. Now type:

```
?rat('r', 'tomorrow') ↵
6
```

left(*string,n*) This function gets *n* characters from a string from the left side of the string. Now type:

```
?left('Foxpro',3) ↵
Fox
```

right(*string,n*) This function gets *n* characters from a string from the right. Type:

```
?right('Foxpro',3) ↵
pro
```

alltrim(*string*) This function removes spaces on either side of *string*. Type:

```
string1='anti' ↵
string2=' clock ' ↵ &&note the white space on either side of clock
string3='wise' ↵
string=string1+string2+string3 ↵
?string ↵
anti clock wise

string=string1+alltrim(string2)+string3 ↵
?string ↵
Anticlockwise
```

str(*number*) This function converts a number into a string. Type:

```
a=10+10+10+10 ↵
?a ↵
40

a='10'+'10'+'10'+'10' ↵
?a ↵
10101010
```

In *a=10+10+10+10*, the four 10's are real numbers but those in *a ='10'+'10'+'10'+'10'* are strings. Although they look exactly the same in appearance, they are represented with different machine codes in the computer. Now type:

```
word='text' ↵
number=1 ↵
?word+number ↵
```

A warning message appears which reads: "Operator/operand mismatch." Now type:

```
?word+str(number) ↵
text          1
```

This time *word* and *number* can be joined together but there are 9 spaces between them. To eliminate these spaces, put *str(number)* inside *alltrim(string)*:

```
?word+alltrim(str(number)) ↵
text1
```

for *variable* **=** *n* **to** *x*…**endfor** This command creates a loop between *for variable = n to x* and *endfor*. The initial value of the variable is *n* and is increased by 1 until its value is *x*. Statements in between can be carried out *x – n+ 1* times. Suppose we want to calculate 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10, enter the following in the command window, be sure not to press Enter at the end of each statement but use the down key ↓ on the keyboard to move to a new line:

```
a=0
for i=1 to 10
a=a+i
?a
```

```
endfor
```

then press the left button of the mouse and drag the mouse from *a = 0* to *endfor* to completely highlight the five lines of statements and then press Enter, the result is displayed on the main screen. Initially *a* is assigned the value of 0, but in the third statement it becomes 1 because the initial value of *i* is 1 and it's added to *a* in the third statement. After the fourth statement is carried out, which outputs the current value of *a* to the screen, the computer goes back to the second statement and increases the value of *i* by 1, which is now 2, and then goes to the third statement, after which the value of *a* becomes 3. The loop continues until *i* becomes 10. The result is shown below:

```
1
3
6
10
15
21
28
36
45
55
```

Now we'll write a program *multifield.prg* to create a multiple field table called *multifield*. The table has 85 fields. The first field is called *tokens* holding number of word tokens with a width of 8 digits. The next four fields are *mv, sdv, intervl* and *intervu*; they are all numeric fields holding 12 digits with 4 decimal places. These fields are respectively for mean vocabulary growth, the standard deviation of the 80 sets of vocabulary growth, the lower bound of the 95% confidence interval of the vocabulary growth and the upper bound of the 95% confidence interval of the vocabulary growth. The rest are 80 numeric fields for the vocabulary growth of each of the 80 sets of samples; their width is 6, holding only integers. We name them *v1, v2, v3…v80*. Now enter the following in the command window:

```
set default to d:\fox\practice ↵
modify command multifield ↵
```

The program editor opens. Enter the following in it (be sure not to enter the line numbers):

```
1.  fields1="&&there is no space between the two single quotes
2.  for i=1 to 80
3.  fields1=fields1+'v'+alltrim(str(i))+' n(6),' &&note the comma
4.  endfor
```

```
5.  fields1=left(fields1,rat(',',fields1)-1)
6.  fields2='(tokens n(8),mv n(12,4),'+'sdv n(12,4),'+'intervl
    n(12,4),'+'intervu n(12,4),'+fields1+')'
7.  create table multifield &fields2
8.  browse
```

Statement 1 initializes the variable *fields1*. Statements 2—4 create a loop, in which statement 3 is carried out 80 times. When *i* = 1, *fields1* is given the string literal *v1 n(6),*, when *i* = 2, *fields1* becomes *v1 n(6),v2 n(6),*, and when *i* = 80, *fields1* contains *v1 n(6), v2 n(6), v3 n(6), v4 n(6)…v80 n(6),*. Statement 5 cuts the comma after *v80 n(6)*. In statement 6, the variable *fields2* is assigned *(mv n(12,4),sdv n(12,4),intervl n(12,4),intervu n(12,4), v1 n(6)…v80(6))*. Statement 7 creates the table *multifield* by using the macro operator *&*. Now save the program in *d:\fox\practice* as *multifield.prg* and run it by clicking on the red exclamation mark on the menu bar. *multifield* with 85 fields is created instantly.

2.3 Foxpro Table Work Areas

Foxpro has 32,767 work areas in which to open tables. When we first create a table or use a table without specifying its work area, Foxpro puts it in work area 1 by default. If we create five tables, Foxpro automatically assigns them to work areas 1, 2, 3, 4 and 5 in order of the creation sequence. We can’t keep five tables open at the same time in one work area. When a table is opened in one work area, a previously opened table in this area is automatically closed. So to open a new table but keep a previously opened table open, we can select a new work area for the new table. The command to select a work area for a table is:

select *workareanumber*

The following function is for checking in which area a table is open:

select()

Now we’ll create five tables and practice selecting work areas for them. Type the following in the command window:

```
creat table test1(word c(4)) ↵
creat table test2(word c(4)) ↵
creat table test3(word c(4)) ↵
creat table test4(word c(4)) ↵
creat table test5(word c(4)) ↵
```

Then type:

```
browse ↵
?select() ↵
5
```

test5 is the fifth table created so it’s given work area 5.

Now enter the following:

```
select 1 ↵
browse ↵
select 2 ↵
browse ↵
select 3↵
browse ↵
select 4 ↵
browse ↵
```

test1, *test2*, *test3*, *test4* appear one after another. This means they are open in work areas 1, 2, 3, and 4 assigned automatically by Foxpro, and we can access them by selecting the work area they are in. Now let’s assign different work areas to these tables and open them in these areas. Type:

```
close data ↵
select 15 ↵
use test1 ↵
select 16 ↵
use test2 ↵
select 17 ↵
use test3 ↵
select 18 ↵
use test4 ↵
select 19 ↵
use test5 ↵
```

The five tables are now all open in the work areas just assigned to them. Type:

```
select 18 ↵
browse ↵
```

test4 appears on the screen.

2.4 Data Input and Output in Tables

2.4.1 Data input

There are three ways to input data to a Foxpro table. They are manual input, input from another table and input from a text file.
1. Manual input. For new tables we can use either the commands **append** or **insert** to input data manually. Now open *table1* you have created (in *d:\fox\practice*) and enter either *append* ↵ or *insert* ↵ in the command window; a data input box appears, as shown in Figure 2.3. The field names are on the left of the first row. Click on the highlighted part of the *word* field and start inputting data. After completing entering data in a field, press Enter and the cursor automatically moves to the next field. Now enter the following in the highlighted area: *study* ↵, *24* ↵, *3* ↵, *5* ↵, the cursor moves all the way down to the field *context*. Type *He is in a brown study*. After completing data input, either press Esc or click on the × sign on the upper right corner of the data input box. To view the table, type *browse* ↵.
2. Input from a table. In actual practice, the manual input mode is rarely used because it's too slow. Data are either automatically appended from an existing table or from a text file. The command for appending data from another table is:

append from *tablename* [*fieldnames*] [**for** *condition*]

To append the contents of a table to another table, the two tables must have fields of the same name for the same data type. For example, suppose *tablea* has three fields *word*, *freq*, *wlength*, with the first field holding character data and the rest numeric data, and *tableb* has four fields *word*, *frequency*, *range*, *wlength*, with the first field holding character data and the rest numeric data, only the contents of the fields *word*, *wlength* can be appended to *tablea* from *tableb*. If we want to append all the contents of *wordlist* (in *d:\fox\table3*) to *table1*, open *table1*, modify the structure of *table1* and change *length* back to *wlength*, and then type in the command window:

```
append from d:\fox\table3\wordlist ↵
browse ↵
```

table1 appears fully loaded with all the contents of *wordlist* except the memo field, which *table1* doesn't have. The command to physically remove all the contents of a table is:

zap The command *zap* physically deletes everything in a table, and should be used with great care. Now empty *table1* for further use by typing:

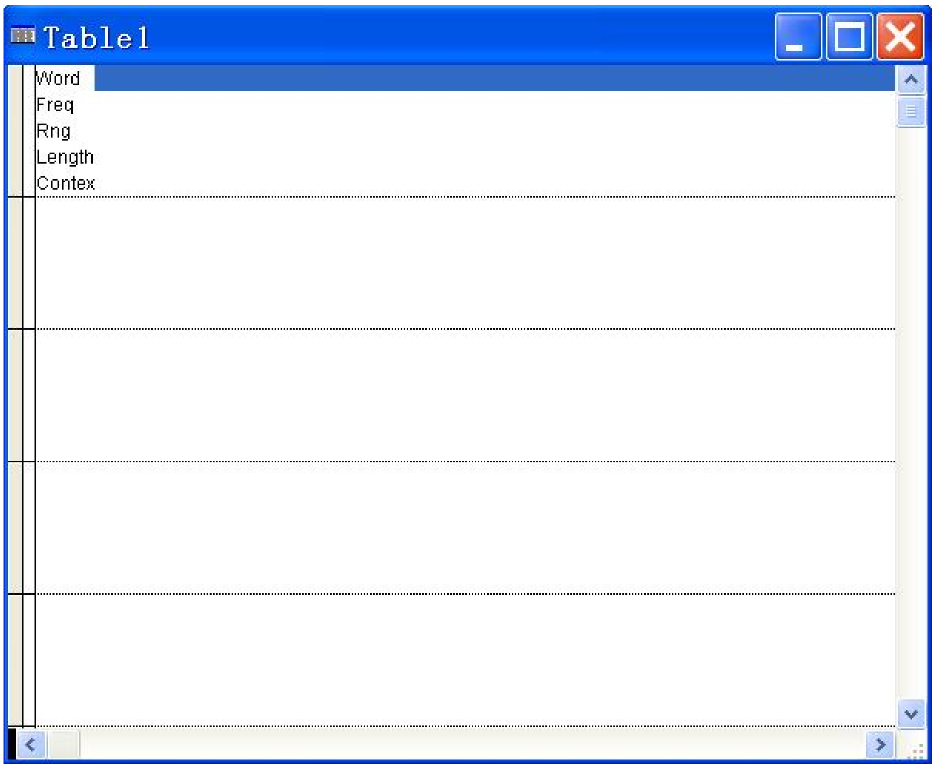


Figure 2.3 Data input box for manual input

```
zap ↵
```

If we want to append only the data in the *word* and *freq* fields of *wordlist*, type:

```
append from d:\fox\table3\wordlist fields word,freq ↵
browse ↵
```

only the contents in the *word* and *freq* fields of *wordlist* are appended to *table1*.

We can specify what data to append. Suppose we want to append only words with length between 3 and 7 letters with frequency higher than 20, zap *table1* again and type:

```
append from d:\fox\table3\wordlist for wlength>=3 and wlength<=7 and
freq>20 ↵
browse ↵
```

To append only words whose first letter is *B*, type:

```
append from d:\fox\table3\wordlist for word='B' ↵
browse ↵
```

To append words with the letter clusters *scl*, type:

```
append from d:\fox\table3\wordlist for 'scl'$word ↵
browse ↵
```

recno() This function measures the position of a record in a table. Move the record pointer (the little black arrow on the left edge of a record) to the second row in *table1* then type:

```
?recno() ↵
```

2 appears on the screen. To append words between the 50th record and 150th record (inclusive) in *wordlist*, type:

```
append from d:\fox\table3\wordlist for recno()>=50 and recno()<=150 ↵
browse ↵
```

101 words are appended from *wordlist* between the 50th record and 150th record.

reccount() This function measures the number of records of a table. Type:

```
use d:\fox\table3\wordlist ↵
?reccount() ↵
23926
```

3. Input from text files. The following is the command for appending data from a text file to a table:

append from *filename* | [**sdf**] [**delimited with** |[**tab**] [**blank**] [*character*]|] | [*fieldnames*] [**for** *condition*]

To use this command, data to be inputted must be arranged in columns. The *sdf* option is used when the columns are separated with spaces, and the width of the columns is the same as those in the table. Look at the following data from *appendsdf.txt* in *d:\fox\texts*. The first column contains words, the second frequency, the third range, and the last word length:

```
A                          25897  500   1
A.m.                       9      7     4
Aback                      1      1     5
Abandon                    62     51    7
Abandonment                9      9     11
Abate                      2      2     5
```

Abbey	*24*	*13*	*5*
Abbot	*10*	*3*	*5*
Abbreviate	*2*	*2*	*10*
Abbreviation	*6*	*3*	*12*
Abdicate	*2*	*2*	*8*
Abdomen	*1*	*1*	*7*
Abdominal	*1*	*1*	*9*
Aberrant	*2*	*2*	*8*
Aberration	*3*	*3*	*10*
Abet	*1*	*1*	*4*
Abeyance	*2*	*2*	*8*
Abhorrence	*1*	*1*	*10*
Abhorrent	*2*	*2*	*9*
Abhor	*1*	*1*	*6*
Abide	*12*	*12*	*5*
Ability	*116*	*87*	*7*
Abject	*4*	*4*	*6*
Abjure	*1*	*1*	*6*
Ablaze	*1*	*1*	*6*
Able	*258*	*181*	*4*
Ablest	*1*	*1*	*6*
Ably	*1*	*1*	*4*
Abnormal	*2*	*2*	*8*
Abnormality	*2*	*1*	*11*
Abnormally	*2*	*2*	*10*
Aboard	4	2	6

The width of the four columns is respectively 25, 10, 10, and 6. Now create a four-field table with field width of 25, 10, 10 and 6 each and name the fields *word*, *freq*, *rng* and *wlength* and then type:

```
append from d:\fox\texts appendsdf.txt sdf ↵
```

The data is appended without a hitch. We can also specify the appending conditions. For example, if we want to append words whose length is greater than 10, type:

```
append from d:\fox\texts\appendsdf.txt sdf for wlength>10 ↵
```

Only the words longer than ten letters are appended.

Data can also be arranged in columns separated with a single space, a comma or a tab. Look at the file *appedblan.txt* in *d:\fox\texts*:

```
A 25897 500 1
A.m. 9 7 4
```

```
Aback 1 1 5
Abandon 62 51 7
Abandonment 9 9 11
Abate 2 2 5
Abbey 24 13 5
Abbot 10 3 5
Abbreviate 2 2 10
Abbreviation 6 3 12
Abdicate 2 2 8
Abdomen 1 1 7
Abdominal 1 1 9
Aberrant 2 2 8
Aberration 3 3 10
Abet 1 1 4
Abeyance 2 2 8
Abhorrence 1 1 10
Abhorrent 2 2 9
Abhor 1 1 6
Abide 12 12 5
Ability 116 87 7
Abject 4 4 6
Abjure 1 1 6
Ablaze 1 1 6
Able 258 181 4
Ablest 1 1 6
Ably 1 1 4
Abnormal 2 2 8
Abnormality 2 1 11
Abnormally 2 2 10
Aboard 4 2 6
```

Each row contains a word, its frequency, range and length, separated by a space. For such data, we can use the *delimited with blank* option. Now zap the table you just created for appending data from *appendsdf.txt,* and type:

```
append from d:\fox\texts\appendblan.txt delimited with blank ↵
```

The contents of *appendblan.txt* are appended to the table. We can specify the fields to which data are appended and under what conditions. If we wish to append data to the word field and the range field under the condition that the range is between 4 and 10, type:

```
append from d:\fox\texts\appendblan.txt delimited with blank fields
word,rng for rng>4 and rng<10 ↵
```

Only three records satisfying the condition are appended to the word field and range field. If instead of a single space, the data in each row are separated by a single comma or a tab, the *delimited with blank* part of the above command should be changed to *delimited with ‘,’* or *delimited with tab*.

append memo *fieldname* **from** *filename* [**overwrite**] This command appends the contents of a text file to a memo field cell of the current record. The *overwrite* option replaces the old contents with the new ones. Without *overwrite*, this command adds new contents to the old contents. Now use *d:\fox\table3\wordlist*, move the record pointer to the second record and type:

```
append memo note from d:\fox\texts\appendsdf.txt ↵
```

The cell marker *memo* now is turned to *Memo*, suggesting this cell is no longer empty. Click on the cell and the contents just appended are displayed.

In actual language processing we rarely have such ready data for analysis. Most probably we have only raw texts with millions of words from which to get useful data. To do this, we must first tokenize the raw text, breaking it apart into words arranged in a single column before putting them in Foxpro tables for further processing. To do this, we need a new set of commands and functions.

insert blank This command inserts a blank record right after the current record.

append blank This command appends a blank record at the bottom of a table.

filetostr('*filename*') This function puts the contents of a text file to a string variable. For example, if we want to put the contents of the file *shorttext.txt* in *d:\fox\texts* to a variable called *textinput*, type:

```
textinput=filetostr('d:\fox\texts\shorttext.txt') ↵
```

To check whether *textinput* is loaded with the contents of *shorttext.txt*, type:

```
?textinput ↵
```

The contents are displayed on the screen. Don’t try to do this to long text files because it would take the computer quite a while to display it. But if such does happen, press Esc to halt it.

strtofile(*stringname*, '*filename*') This function does the reverse. It puts the contents stored in a string variable to a text file. If we want to put the contents of *textinput* to a file called *temp.txt* in d:\fox\practice, type:

```
strtofile(textinput, 'd:\fox\practice\temp.txt') ↵
```

To check whether there is such a file, type:

```
modify file d:\fox\practice\temp.txt ↵
```

strtofile(stringname, 'filename') is not additive. That is, the contents, if there is any, of the target text file are replaced by the contents stored in the variable. To make it additive, add *,.t.* after *filename*, that is: *strtofile (stringname, 'filename',.t.)*. Now type:

```
strtofile(textinput, 'd:\fox\practice\temp.txt',.t.) ↵
modify file d:\fox\practice\temp.txt ↵
```

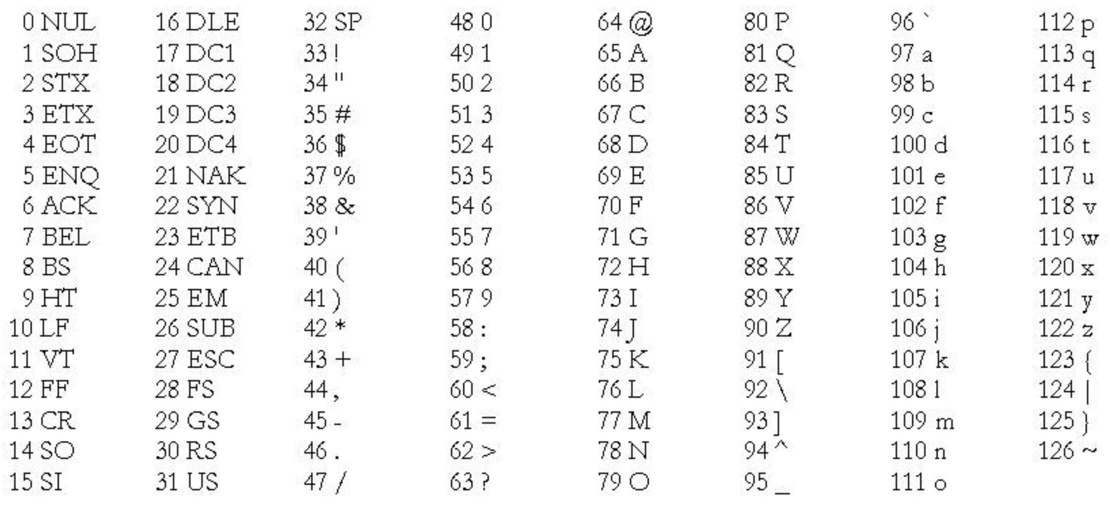
The contents of *textinput* are added to the contents of *temp.txt*, instead of replacing it.

chr(*n*) This function is very useful in string manipulation in Foxpro. It returns one of the 256 ASCII characters depending on the value of *n*, which ranges from 0 to 255. Figure 2.4 lists 126 ASCII characters and their corresponding decimal numbers. Some of the characters are invisible. For example, 7 represents the bell sound, 8 back space, 9 the tab key, 13 carriage return, 32 space etc. *chr(13)* is the most important in tokenizing a text and therefore must be learned by heart. Now type:

```
?chr(65) ↵
A

?chr(97) ↵
A

?chr(35) ↵
#

?chr(56) ↵
8
```

If we type *chr(7)* ↵ we can hear a beep if the computer has a speaker. Now enter the following:

```
? 'The following are not letters: ' + chr(33) +chr(34)+ chr(35)+ chr(36) +
chr(37)+chr(38)+chr(39)+chr(40)+chr(41)+chr(42)+chr(43)+chr(34)+chr(45
)+chr(46)+chr(47) ↵
The following are not letters: !”#$%&'()*+"-./
```



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	

Figure 2.4 The ASCII character table with their decimal numeric values

strtran(*string,characters1,characters2*) This function replaces *characters1* with *characters2* within *string*. Type the following:

```
phrase='potato:chips' ↵
letter1='i' ↵
letter2='a' ↵
punctuation=': ' ↵
?strtran(phrase,letter1,letter2) ↵
potato:chaps

?strtran(phrase,punctuation, '') ↵&&no space between the quotes
Potatochips

letter1='ta' ↵
nothing=’’ ↵ &&no space between the quotes
?strtran(phrase,letter1,nothing) ↵
poto:chips
```

Now we’ll use the functions we’ve just learned to break *shorttext.txt* into individual words and remove the punctuation marks and tabs. Type the following statements one by one in the command window:

```
textinput=filetostr('d:\fox\texts\shorttext.txt') ↵
textinput=strtran(textinput,chr(9),'') ↵ && no space between the two quotes
textinput=strtran(textinput,chr(44),'') ↵
textinput=strtran(textinput,chr(46),'') ↵
textinput=strtran(textinput,chr(32),chr(13)) ↵
strtofile(textinput,'d:\fox\practice\temp.txt') ↵
```

The first statement put the contents of *shorttext.txt* to *textinput*, while the second, third, fourth statements replace *chr(9)*, *chr(44)* and *chr(46)* respectively representing tabs, commas and full stops, with nothing (represented by two single quotes without any space between them). The fifth statement replaces *chr(32)* representing white space with *chr(13),* the carriage return. We can also use ‘ ‘ (two single quotes with a white space in between) instead of *chr(32). chr(13)* serves as a tokenizer, breaking the text into individual words arranged in one column. The last statement puts the tokenized contents of *textinput* to *temp.txt.* The contents of *temp.txt* are as follows:

```
Cat
cat
on
the
mat
A
word
is
characterized
by
the
company
it
keeps
```

chrtran(*string*,*characters1*,*characters2*) This function replaces *characters-1* in *string* with *characters2*. Unlike the *strtran()* function, characters in *characters1* don’t have to be contiguous in *string*. The first character of *character1* in *string* is replaced by the first character of *character2*, the second character of *character1* is replaced by the second character in *character2* etc. If *character2* has more characters than *character1*, then those characters of *characters2* whose position exceeds that of the last character of *characters1* are ignored. And if *characters1* has more characters than *characters2*, those characters of *characters1* in *string* whose position exceeds that of the last character in *characters2* are replaced with nothing. Type:

?chrtran(‘abaout‘,'a','what') ↵
wbwout

?strtran('abaout', 'a', 'what') ↵
whatbwhatout

?chrtran('abaout', 'au', 'what') ↵
wbwoht

?strtran('abaout','au','what') ↵
abaout

?chrtran('abaout', 'au', 'w') ↵
wbwot

?strtran('abaout','au','w') ↵
abaout

nothing=''↵ &&no space between the quotes
?chrtran('abaout', “a123*&%.(/$?!'”, nothing) ↵
bout

?strtran('abaout', 'a123*&%.(/$?!',nothing) ↵
abaout

?chrtran('a12b3a*o&u%t.(/$?!','123*&%.(/$?!',nothing) ↵
abaout

?strtr('a12b3a*o&u%t.(/$?!','123*&%.(/$?!',nothing) ↵
*a12b3a*o&u%t.(/$?!*

len(*string*) This function measures the length of a string in number of characters. Type:

?len('linguistics') ↵
11

Blanks within a string or on either side of it are also counted by the function. Now type:

?len(' Foxpro for quantitative linguistics ') ↵ &&there is a space on either side of the string

```
*37*

nothing='' ↵ &&there is no space between the quotes
?len(nothing) ↵
*0*
```

To measure the length between the first *F* and the last *s* inclusive, use *alltrim()* nested inside l*en()*:

```
?len(alltrim(' Foxpro for quantitative linguistics ')) ↵ &&there is a space on
either side
*35*
```

Now let’s tokenize *alice.txt* in *d:\fox\texts* with the commands and functions we’ve just learned. First we’ll create a table called *alicetoken* in *d:\fox\practice*. It has three fields, the field *word* for storing words, *freq* for word frequency and *wlength* for word length:

```
create table d:\fox\practice\alicetoken (word c(30),freq n(4),wlength n(4)) ↵
```

Then enter the following statements in the command window:

```
set default to d:\fox\practice ↵
carriage=chr(13) ↵
spaces=chr(32) ↵
textinput=filetostr('d:\fox\texts\alice.txt') ↵
textinput=strtran(textinput,spaces,carriage) ↵
strtofile(textinput, 'temp.txt') ↵
append from temp.txt sdf ↵
browse ↵
```

In statements 3 and 4, *chr(13)* and *chr(32)* are respectively assigned the more self-explanatory variable *carriage* and *spaces*. The text is tokenized by statement 5 and appended to *alicetoken*, but the entire frequency field, word length field and part of the word field are empty, and some words have punctuation marks. The table needs to be further processed.

delete [**all**] [**for** *condition*] This command is used to delete records from a table. *delete* used alone deletes only the current record. Now move the record pointer to a record in the table you just created and type

```
delete ↵
```

the left side of the record is marked by a dark square. Now type:

```
delete all ↵
```

The left edge of the entire table is darkened. The *delete* command doesn't physically remove the deleted records. We can retrieve the deleted records by issuing the following command:

recall [**all**] *recall* without *all* retrieves the deleted record where the record pointer is; with *all*, all the deleted records are retrieved. Now type:

```
recall all ↵
```

the dark mark on the left edge of the table disappears, and the deleted records are all retrieved.

The following command physically removes the records deleted by the *delete* command:

pack

Now type

```
delete for recno()<10 ↵
delete for word='' ↵ &there is a space between the quotes
pack ↵
```

The first statement deletes all the records whose record number is smaller than 10. The second statement deletes records whose word field is empty. However, statement like this should be used with care. If some words in the word field have blanks preceding them, they will be deleted. The last statement physically removes these deleted records from the table and can't be recalled. So *pack* should be used with great care.

blank [**all**] [**for** *condition*] This command, used without *all*, replaces the current record with a blank. Used with *all*, all the records of a table are replaced with blanks. We can also specify which record to be replaced with blanks. Now type:

```
blank ↵
browse ↵
```

Only one record is blanked. Type:

```
blank for recno()<20 ↵
browse ↵
```

All the records whose record number is smaller than 20 are blanked.

proper(*string*) This function capitalizes the first letter of *string*. Type:

```
?proper('foxpro') ↵
Foxpro
```

lower(*string*) This function turns the characters of *string* into lower cases. Type:

```
?lower('FOXPRO') ↵
foxpro
```

upper(*string*) This function turns the characters of *string* into upper cases. Type:

```
?upper('foxpro') ↵
FOXPRO
```

replace [**all**] *fieldname* **with** | [*string*] [*number*] | [**for** *condition*] This command replaces the field of the specified records with a string or a number. To replace a single record of a field with a string or a number, move the record pointer to the record and then issue the command. If we want to replace *the* in *alicetoken* with *THE*, move the record pointer to the record where *the* is and type in the command window:

```
replace word with 'THE' ↵
```

To replace the entire frequency field with 1, type:

```
replace all freq with 1 ↵
```

sort to *tablename* **on** *fieldname* [**descending**] This command sorts a field *fieldname* of an open table and puts the result to another table *tablename*. Without *descending*, the field is sorted in ascending order. Now remove the contents of *alicetoken* using *zap*, and reload it with all the words of *alice.txt*, remove all the empty records, sort the word field of *alicetoken* in descending order and output the result to a table called *alicedesc* by typing:

```
sort to alicedesc on word descending ↵
use alicedesc ↵
browse ↵
```

index on *fieldname* **tag** *fieldname* [**descending**] This command is used to sort a field of an open table either in ascending order (without *descending*) or descending order (with *descending*). However, unlike the *sort* command, this command doesn't change the original record number of a record. Now move the record pointer to the second record of *alicedesc* and type in the command window:

```
?recno() ↵
2
```

Now open the table *alicetoken* and type:

```
index on word tag word desc ↵
brow ↵
```

The words are sorted in descending order. Now move the record pointer to the second record of the table and type in the command window:

```
? recno() ↵
719
```

We can use this command to sort a field not only at the leftmost position of the field, but also anywhere else. In *d:\fox\table3* there is a table called *filename* holding imagined file names in the field *fname,* sorted in the following order: *BOA1.TXT*, *BOA10.TXT*, *BOA11.TXT*,…*BOA2.TXT*, *BOA20.TXT*, *BOA21* etc. This is because the numbers here 1, 10, 11, 2, 20, 21and so on after *BOA* are actually characters, so *BOA1.TXT* is followed by *BOA10.TXT* instead of *BOA2.TXT*. Now we'll rearrange them so that they will be arranged as *BOA1.TXT, BOA2.TXT, BOA3.TXT…BOA10.TXT* and so on. Open the table and type in the command window:

```
index on left(alltr(fname),len(alltr(fname))-6) tag fname ↵
```

The file names are now sorted in the way desired, i.e., *BOA1.TXT*, *BOA2.TXT*, *BOA3.TXT* etc. This is because file names like *BOA1.TXT* are eight characters in length, while that of *BOA10.TXT*, *BOA11.TXT* etc are nine characters in length. *left(alltr(fname),len(alltr(fname))-6)* gets *BO* from *BOA1.TXT* to *BOA9.TXT*, but *BOA* from *BOA10.TXT* upwards. When sorting in ascending order, *BO* precedes

BOA. Therefore *BOA1.TXT…BOA9.TXT* precede *BOA10.TXT*, *BOA11.TXT* and so on.

delete tag all This command makes an indexed table un-indexed.

total to *tablename* **on** *fieldname* This command combines identical character records and is often used to calculate word frequency. For this command to work, the *index* command must be issued first.

Now we'll write a program for making a wordlist of *alice.txt* in *d:\fox\texts*. We'll call the program *alice.prg*. Type:

```
set default to d:\fox\practice ↵
modify command alice ↵
```

and then enter the following statements in the now empty *alice.prg* without the line number.

```
1.  close data
2.  create table aliceword (word c(30), freq n(8), wlength n(3))
3.  nothing='' &&there is no space between the quotes
4.  carriage=chr(13)
5.  spaces=chr(32)
6.  textinput=filetostr('d:\fox\texts\alice.txt')
7.  textinput=strtran(textinput, '-',spaces)
8.  textinput=strtran(textinput,spaces,carriage)
9.  strtofile(textinput, 'temp.txt')
10. append from temp.txt sdf
11. replace all word with chrtran(word, ',.`[?]_ ”!:;()*',nothing)
12. replace all word with strtran(word, “'”,nothing)&&there is a single
    quote between the double quotes
13. replace all word with proper(word)
14. dele all for word=spaces
15. pack
16. replace all freq with 1
17. index on word tag word
18. total to temp on word
19. zap
20. append from temp
21. replace all wlength with len(alltrim(word))
22. browse
```

In this program, statement 1 closes any open tables. Statement 6 puts the contents of *alice.txt* to the variable *textinput*. Statement 7 replaces "-" with a white space in words like *what-do-you call-it*. Statement 8 performs tokenization, replacing all white spaces in the text with carriage returns so that all the words of the text are arranged in a column as shown below:

Alice
was
beginning
to
get
very
tired
of
sitting
by
her
sister
on
the
bank,
and
of
having
nothing
to
do:

Statement 9 put the tokenized text stored in *textinput* to a temporary file *temp.txt*, which is then appended to the table *aliceword*. Statements 11, 12 and 13 respectively remove punctuation marks and non-letter symbols and capitalize the first letters of the words in the word field for calculating the frequency of words such as *cat* and *Cat*. Statements 14—15 remove records whose word fields are empty, and statement 17 sorts the word field. Statement 18 combines identical records and calculates word frequency, and the result is stored in a temporary table called *temp*. Statement 19 empties the old contents of *aliceword*, and statement 20 appends the contents of *temp* to *aliceword*. Statement 21 measures word length; *alltrim(word)* is nested inside the *len()* function to remove the trailing blanks after them.

In the table there are words like *Dont, Doesnt, Youve, Youll, Theres* etc; this is caused by the removal of the apostrophe ‘ by statement 12 in the program. Their original forms are *Don't, Doesn't, You've, You'll, There's*, etc. If we want to separate each of the contracted forms into two words, i.e. *Don't* into *Do* and *not*, the program should be modified. We leave this task to the reader as an exercise.

2.4.2 Data output

The contents of a table can be outputted to another table or a text file. There are several commands for data output. Before we look at these commands and their use, open the table *aliceword* if it's closed.

copy to *tablename* [*fieldnames*] [**for** *condition*] [**foxplus**] This command copies the contents of an open table to another table. If we want to copy the contents of *aliceword* to another table called *temp*, type:

```
copy to temp ↵
```

We can specify the records or fields to copy. Suppose we want to copy to *temp* only words beginning with *C* with frequency between 5 and 10 and word length between 3 and 7, type:

```
copy to temp for word='C' and freq>=5 and freq<=10 and wlength >=3 and
 wlength<=7 ↵
```

If the *foxplus* option is used, the new table can be read by the statistical package *SPSS for WINDOWS.* Type:

```
copy to temp field freq,wlength foxplus ↵
```

The new table *temp* can be opened by *SPSS* for statistical analysis.

copy to *filename* [*fieldnames*] | [**sdf**] [**delimited with** |[**blank**] [**tab**] [*character*]|] | [**for** *condition*] This command copies the contents of a table to a text file. If we want to output the contents of *aliceword* to a text file called *aliceword.txt* and keep the field width unchanged in the text file, type:

```
copy to aliceword.txt sdf ↵
```

The contents of the entire table are outputted to *aliceword.txt*. To view the file, type:

```
modify file aliceword.txt ↵
```

Be sure to close the file after viewing. If we want to output words whose frequency is greater than 20, type:

```
copy to aliceword.txt field word,freq for freq>20 sdf ↵
modify file aliceword.txt ↵
```

Now try the following:

```
copy to aliceword.txt delimited with tab ↵
modify file aliceword.txt ↵
```

character in the *character* option in this command can be any single printable character on the keyboard. Now try the following:

```
copy to aliceword.txt delimited with '*' ↵
modify file aliceword.txt ↵ &&be sure to close the text after viewing
copy to aliceword.txt delimited with “'”' ↵
modify file aliceword.txt ↵
copy to aliceword.txt delimited with '/' ↵
modify file aliceword.txt ↵
```

list [*fieldname*] [**for** *condition*] [**to** *filename*] [**noconsole**] [**off**] This command sends the contents of specified fields to the screen or a file. If we type:

```
list ↵
```

the contents of the entire table are sent to the screen with record numbers. To stop the listing, press Esc. If we want to send the contents of *aliceword* to a text file called *temp.txt* without record numbers, type:

```
list to temp.txt noconsole off ↵
```

noconsole prevents outputting the contents to the screen, and *off* tells the computer not to output the record numbers. The above statement keeps the field names, as well as the field width, in the text file. We can specify conditions on data outputting. If we want to output only words whose record number is smaller than 20, type:

```
list to temp.txt field word for recno()<20 noconsole off ↵
```

display [**all**] [*fieldnames*] [**for** *condition*] [**to** *filename*] [**noconsole**] [**off**] This command is the same as the *list* command except that to output the contents of a table to a text file or to the screen, *all* has to be used, otherwise only one record is outputted. In addition, when outputting data to the screen or to a text, data is displayed one screen at a time and the display pauses between screens. A key has to be pressed for the display to continue. But used with *noconsole*, all the contents are displayed without pause. Now type:

```
display all to temp.txt for freq>=100 noconsole off ↵
```

copy memo *fieldname* **to** *filename* [**additive**] This command copies the contents of a memo field of the current record to a text file. The default action of this command is *overwrite*, that is, the new contents replace the existing contents of the text file. To add new contents to the text without removing the old ones, the *additive* option should be used. Now open *wordlist* in *d:\fox\table3*, move the record pointer to a record with a loaded memo field and then type in the command window:

```
copy memo note to temp.txt ↵
```

The contents of the memo field of the current record are outputted to a text file called *temp.txt*.

2.5 Application

2.5.1 Lexical comparison

Now let's use some of the commands and functions we have learned to write practical language processing programs. We'll make a lexical study on *Alice's Adventures in Wonderland* and *Through the Looking-glass* (*alice.txt* and *lglass.txt* in *d:\fox\texts*). We'll calculate their respective vocabulary size, word frequency, word length, lexical similarity and difference and then store the results both in tables and text files. We'll write three programs: *awordlist.prg*, *lwordlist.prg* and *compare.prg*. *awordlist.prg* and *lwordlist.prg* make a frequencied wordlist respectively for *alice.txt* and *lglass.txt*, while *compare.prg* compares the two wordlists for lexical overlap between them.

awordlist.prg

```
set default to d:\fox\practice
set safety off
close data
create table awordlist (word c(25),freq n(10),wlength n(4))
nothing=''
spaces=chr(32)
carriage=chr(13)
textinput=filetostr('d:\fox\texts\alice.txt')
textinput=strtran(textinput,'-',spaces)
textinput=strtran(textinput,spaces,carriage)
strtofile(textinput,'temp.txt')
append from temp.txt sdf
```

```
13. replace all word with chrtran(word, ',.`[?]_”!:;()*',nothing)
14. replace all word with strtran(word,"'",nothing)&&there is a single quote
    between the double quotes
15. replace all word with prop(word)
16. replace all freq with 1
17. index on word tag word
18. total to temp on word
19. zap
20. append from temp
21. replace all wlength with len(alltrim(word))
22. delete all for len(alltrim(word))=0    &&this removes empty records
23. pack
24. copy to awordlist.txt sdf
```

lwordlist.prg

```
1.  set default to d:\fox\practice
2.  set safety off
3.  close data
4.  create table lwordlist (word c(25),freq n(10),wlength n(4))
5.  nothing=’’
6.  spaces=chr(32)
7.  carriage=chr(13)
8.  textinput=filetostr('d:\fox\texts\lglass.txt')
9.  textinput=strtran(textinput,'-',spaces)
10. textinput=strtran(textinput,spaces,carriage)
11. strtofile(textinput,'temp.txt')
12. append from temp.txt sdf
13. replace all word with chrtran(word, ',.`[?]_"!:;()*',nothing)
14. replace all word with strtran(word,”’”,nothing)&&there is a single quote
    between the double quotes
15. replace all word with prop(word)
16. replace all freq with 1
17. index on word tag word
18. total to temp on word
19. zap
20. append from temp
21. replace all wlength with len(alltrim(word))
22. delete all for len(alltrim(word))=0 &&this removes empty records and -
23. pack
24. copy to lwordlist.txt sdf
```

compare.prg

```
1.  set default to d:\fox\practice
```

```
 2. set safety off
 3. close data
 4. create table aliceglass (word c(25),freq n(12,5))
 5. append from awordlist
 6. replace all freq with freq*100000
 7. append from lwordlist
 8. index on word tag word
 9. total to temp on word
10. zap
11. append from temp
12. copy to shareword for mod(freq,100000)>0 and freq>100000
13. copy to aliceonly for mod(freq,100000)=0
14. copy to lglassonly for freq<100000
15. copy to lglassonly.txt for freq<100000
16. use aliceonly
17. replace all freq with freq/100000
18. copy to aliceonly.txt sdf
19. use shareword
20. replace all freq with freq/100000
21. copy to shareword.txt sdf
```

awordlist.prg and *lwordlist.prg* are very similar to *alice.prg*. They tokenize *alice.txt* and *lglass.txt*, put the tokenized text into a table, remove punctuation marks and non-alphabetic characters and calculate word frequency and word length. *compare.prg* picks out words that are unique to *alice.txt* and *lglass.txt* and those shared between them. In *compare.prg*, statement 4 creates a table called *aliceglass* for holding the contents of *awordlist* and *lwordlist*. Statement 5 appends words and their frequency from *awordlist*. The word frequency from *awordlist* is subsequently multiplied with 100,000 in statement 6 for lexical comparison; the multiplicand must be multiples of ten and should be at least 10 times larger than the highest frequency of the two wordlists to be compared. Statements 7 to 11 append the contents of *lwordlist*, combine identical words, i.e. words shared between the two wordlists, and then re-calculate their frequency. For example, *a* now has a frequency of 620,000,759 ($620 \times 100000+795=$ 620000759). Statements 12 picks out words shared between *awordlist* and *lwordlist* and put them to *shareword*. The logic behind this is that the frequency of words occurring both in *awordlist* and *lwordlist* now are larger than 100,000 and have remainders if divided by 100,000. Statements 13 and 14 put words unique to *awordlist* and *lwordlist* to *aliceonly* and *lglassonly* respectively. Statement 17 returns word frequencies in *aliceonly* to their original values, while statement 20 separates the frequency of the shared words between *awordlist* and *lwordlist* with a decimal point. Those on the left of the decimal point are the word frequency of *awordlist*; those on the right are the word frequency of *lwordlist*.

2.5.2 Processing multiple texts in a table

Next we’ll write a program called *multitext.prg* for extracting lexical information from multiple texts in a multi-field Foxpro table, such as word frequency, word length (in letters), and word range (in how many text chunks a word occurs). Word range is a very important concept in quantitative linguistics. For example, in selecting words to be taught in a language course, word range, together with word frequency, must be taken into consideration; word range is also referred to as word cotextuality, and is related to Köhler´s self-regulating cycle hypothesis that high cotextuality results in high frequency.

In *d:\fox\texts* there are 48 text chunks from *alice.txt*. We’ll create a table called *multitext* holding the vocabulary of all the 48 text chunks, the range of these words and their frequencies in the individual text chunks and the length of these words.

multitext.prg

```
1.  set default to d:\fox\practice
2.  set safe off
3.  close data
4.  nothing=''
5.  tabs=chr(9)
6.  carriage=chr(13)
7.  spaces=chr(32)
8.  fields1=nothing &&initialize the variable fieldnaname or an error message will result in statement 10
9.  for i=1 to 48
10. fields1=fields1+'freq'+alltrim(str(i))+' n(6),'
11. endfor
12. fields2='(word c(25),'+'totalfreq n(6),'+'rng n(6),'+ fields1+'wlength n(4))'
13. create table multitext &fields2
14. recordnumber=0
15. for i=1 to 48
16. texts='d:\fox\texts\text'+alltr(str(i))+'.txt'
17. frequency='freq'+alltrim(str(i))
18. textinput=filetostr('&texts')
19. textinput=strtran(textinput,'-',spaces)
20. textinput=strtran(textinput,tabs,nothing)
21. textinput=strtran(textinput,spaces,carriage)
22. strtofile(textinput,'temp.txt')
23. append from temp.txt sdf
24. replace all &frequency with 1 for recno()>recordnumber
25. recordnumber =reccount()
```

```
26. endfor
27. replace all word with chrtran(word, ',.`[?]_”!:;()*',nothing)
28. replace all word with strtran(word,”'”,nothing)&&there is a single quote
    between the double quotes
29. delete all for word=spaces
30. pack
31. replace all word with prop(word)
32. replace all totalfreq with 1
33. index on word tag word
34. total to temp on word
35. zap
36. append from temp
37. replace all wlength with len(alltrim(word))
38. fields1=nothing
39. for i=1 to 48
40. fields1=
    fields1+'round('+'freq'+alltr(str(i))+'/('+'freq'+alltr(str(i))+'+1),0)+'
41. endfor
42. fields1=left(fields1,len(fields1)-1)
43. replace all rng with &fields1
```

In this program, statements 9 to 13 create a 52-field table *multitext*. Statement 14 creates a position marker *recordnumber* whose initial value is set to 0. Statements 15 to 26 create a loop, in which the 48 text chunks are tokenized and loaded one by one into the table *multitext*. In the loop, when *i* = 1, the variable *texts* is assigned the string literal *d:\fox\texts\text1.txt* and the variable *frequency* the string literal *freq1*, which is a field name holding the word frequency of *text1.txt*. Statement 18 then puts the contents of *text1.txt* to *textinput*. Statements 19—21 remove hyphens, tabs and tokenize the contents of *textinput*, which are outputted to *temp.txt* in statement 22. Statement 23 appends the contents of *temp.txt* to the table *multitext*. Statement 24 replaces the frequency field *freq1* with 1. Statement 25 assigns the position marker *recordnumber* the number of records in *multitext* after loading the tokenized words of *text1.txt*. If *text1.txt* has 553 words, then the value of *recordnumber* is 553. When *i* = 2 *texts* is assigned the string literal *d:\fox\texts\text2.txt*, and *frequency* the string literal *freq2*, and statement 18 assigns the contents of *text2.txt* to *textinput*. Statements 19 to 21 remove hyphens, tabs and tokenize the contents of *textinput*, which again are outputted to *temp.txt*, with its old contents overwritten. The new contents of *temp.txt* are subsequently appended to *multitext* too. Statement 24 replaces the frequency field *freq2* with 1 starting from record number 554 upwards. Statement 25 assigns the position marker *recordnumber* the number of records in *multitext* after loading the tokenized words of *text2.txt*. When *i* reaches 48, all the words of the 48 text chunks are loaded into *multitext*. Statements 27—30 remove punctuation marks, non-alphabetic characters and empty records. Statements 31—37 turn the first letters of the words into upper case, combine identical words, calculate their frequency and measure word length. Statements 38—43 calculate the range of the words. That is, in how many of the 48 text chunks these words occur. Statement 38 empties the variable *fields1* for new contents. In calculating range, all non-zero frequencies in *freq1* to *freq48* are regarded as 1. If a word occurs only in two of the texts, say 45 times in *text1.txt* and once in *text2.txt,* then its range can be obtained with *round(45/(45+1),0)+ round(1/(1+1),0) + round(0/(0+1),0) +round(0/(0+1),0) +…=* 2 , because *round(45/(45+1),0)* equals 1, so does *round(1/(1+1),0),* while *round (0/(0+1),0)* equals 0. Statements 39—43 create a loop, at the end of which *fieldname* holds the string literal *"round(freq1/(freq1+1),0)+ round(freq2 / (freq2+1),0) + round(freq3/(freq3+1),0) +… round (freq44/(freq48+1),0) +"*. Statement 42 removes the trailing plus sign "+" in *fields1* and the last statement replaces the range field with the range of all the words using the macro operator *&*. When the program has run, put the range field in descending order by typing:

```
index on rng tag rng descending ↵
```

To view the results, type:

```
browse ↵
```

Figure 2.5 is part of *multitext*. With this table, we can extract a lot of useful lexical information on all the 48 text chunks. For example, if we want to output the words shared between *text23.txt* and *text26.txt* to a table called *text23_26*, type:

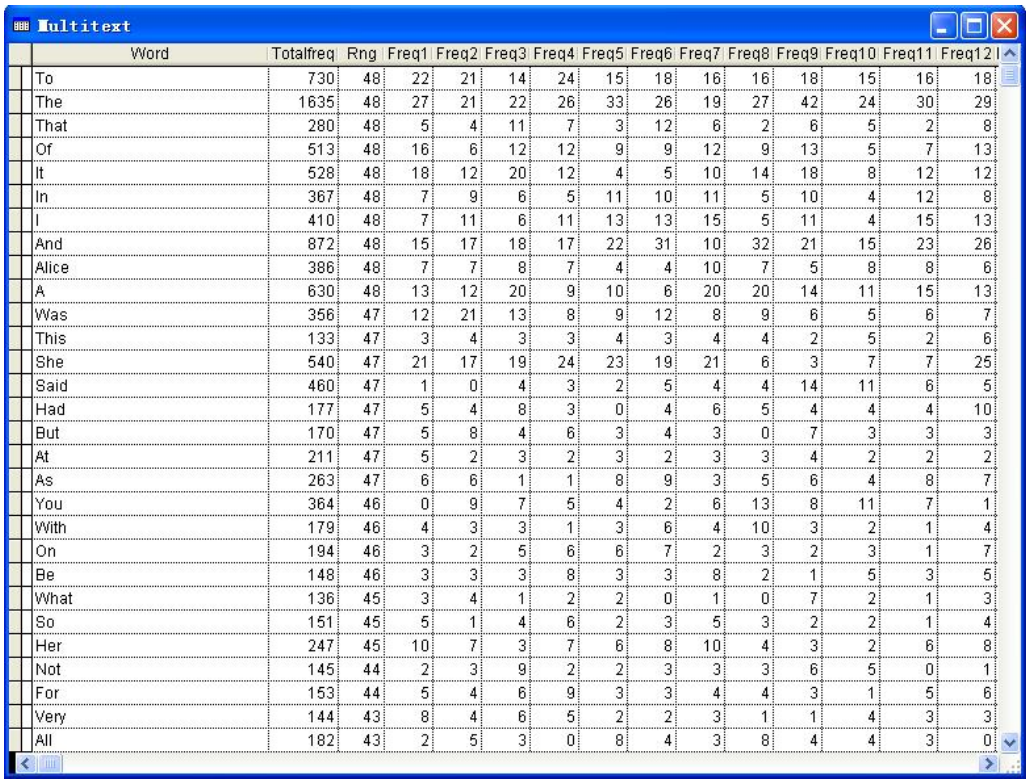
```
copy to text23_26 for freq23>0 and freq26>0 fields word, freq23,freq26 ↵
```

To output to a table called *text1_2* words that *text1.txt* does not have but which occur in *text2.txt,* type:

```
copy to text1_2 for freq1=0 and freq2>0 fields word, freq1,freq2 ↵
```

Statements like the above are particularly useful in compiling word lists for individual lessons of a language course book. To output words that appear in all the 48 text chunks without their record numbers, type:

```
list off word,rng for rng=48 ↵
```



Word	Totalfreq	Rng	Freq1	Freq2	Freq3	Freq4	Freq5	Freq6	Freq7	Freq8	Freq9	Freq10	Freq11	Freq12
To	730	48	22	21	14	24	15	18	16	16	18	15	16	18
The	1635	48	27	21	22	26	33	26	19	27	42	24	30	29
That	280	48	5	4	11	7	3	12	6	2	6	5	2	8
Of	513	48	16	6	12	12	9	9	12	9	13	5	7	13
It	528	48	18	12	20	12	4	5	10	14	18	8	12	12
In	367	48	7	9	6	5	11	10	11	5	10	4	12	8
I	410	48	7	11	6	11	13	13	15	5	11	4	15	13
And	872	48	15	17	18	17	22	31	10	32	21	15	23	26
Alice	386	48	7	7	8	7	4	4	10	7	5	8	8	6
A	630	48	13	12	20	9	10	6	20	20	14	11	15	13
Was	356	47	12	21	13	8	9	12	8	9	6	5	6	7
This	133	47	3	4	3	3	4	3	4	4	2	5	2	6
She	540	47	21	17	19	24	23	19	21	6	3	7	7	25
Said	460	47	1	0	4	3	2	5	4	4	14	11	6	5
Had	177	47	5	4	8	3	0	4	6	5	4	4	4	10
But	170	47	5	8	4	6	3	4	3	0	7	3	3	3
At	211	47	5	2	3	2	3	2	3	3	4	2	2	2
As	263	47	6	6	1	1	8	9	3	5	6	4	8	7
You	364	46	0	9	7	5	4	2	6	13	8	11	7	1
With	179	46	4	3	3	1	3	6	4	10	3	2	1	4
On	194	46	3	2	5	6	6	7	2	3	2	3	1	7
Be	148	46	3	3	3	8	3	3	8	2	1	5	3	5
What	136	45	3	4	1	2	2	0	1	0	7	2	1	3
So	151	45	5	1	4	6	2	3	5	3	2	2	1	4
Her	247	45	10	7	3	7	6	8	10	4	3	2	6	8
Not	145	44	2	3	9	2	2	3	3	3	6	5	0	1
For	153	44	5	4	6	9	3	3	4	4	3	1	5	6
Very	144	43	8	4	6	5	2	2	3	1	1	4	3	3
All	182	43	2	5	3	0	8	4	3	8	4	4	3	0

Figure 2.5 Wordlist of the 48 text chunks

2.5.3 Vocabulary growth

Vocabulary size is a function of text length. As the latter increases, so does the former, but the relationship is non-linear. The slope of vocabulary growth curve gradually decreases as text length increases but will never flattens out. According to Baayen, after sampling 90,000,000 words from the BNC, the vocabulary growth curve was still in the LNRE (large number of rare events) zone. The relationship between vocabulary size and text length is extremely important in the study of vocabulary richness. Wimmer and Altmann list the main approaches to the study of vocabulary richness by Yule, Guiraud, Muller, Dugast, Brunet, Herdan, Kuraszkiewicz, Ejiri and Smith, Tuldava, Köhler, Galle etc, all of which depend on the relationship between vocabulary size and text length. Now we'll write a program to compute vocabulary growth of 100,000 words of texts at an interval of 2,000 words. To do this, 50 texts chunks were randomly sampled from the written text section of the BNC. The lemmatized wordlists of each of the 50 texts are in *d:\fox\table2* in the form of Foxpro tables with names from *bncwlem1* to *bncwlem50*. Apart from examining how vocabulary size changes as the number of word tokens increases, we'll also look at the range of these words, i.e. in how many texts these words occur. We want the program to do two things: a. making a wordlist of the 50 tables put together, with word frequency, word range and word length; b. computing vocabulary growth as the tables are put together one by one and the number of new words a table contributes to the vocabulary growth. The program is as follows:

vocgrowth.prg

```
1.  set default to d:\fox\practice
2.  close data
3.  set safe off
4.  set talk off
5.  clear
6.  creat table wordlist(word c(25),freq n(8),rng n(5),wlength n(4))
7.  creat table vocincrease(tokens n(10),textvoc n(8),vocgrowth n(8),
    newvoc n(5))
8.  vocnumber1=0
9.  vocnumber2=0
10. tokennumber=0
11. select 1
12. for i=1 to 50
13. tablename='d:\fox\table2\bncwlem'+alltrim(str(i))
14. append from &tablename
15. tokennumber=tokennumber+2000
16. vocnumber2=reccount()
17. textvocsize=vocnumber2-vocnumber1
18. replace all rng with 1 for rng=0
19. index on word tag word
20. total to temp on word
21. zap
22. append from temp
23. vocnumber2=reccount()
24. vocincrease=vocnumber2-vocnumber1
25. vocnumber1=vocnumber2
26. select 2
27. append blank
28. replace tokens with tokennumber
29. replace textvoc with textvocsize
30. replace vocgrowth with vocnumber2
31. replace newvoc with vocincrease
32. select 1
33. endfor
34. sele 1
35. replace all wlength with len(alltr(word))
```

In this program, statements 6—7 create two tables: *wordlist* and *vocgrowth*. The former holds words from the 50 lemmatized BNC wordlist tables and word frequency, range and length in fields *word*, *freq*, *rng* and *wlength* respectively; the latter is for cumulative number of tokens, vocabulary sizes of individual tables, vocabulary growth at 2,000-word intervals, and number of new words a table contributes to the vocabulary growth in the fields *tokens*, *textvoc*, *vocgrowth* and *newvoc* respectively. The two tables are respectively in work area 1 and work area 2. Statements 8—10 assign zero to *vocnumber1*, *vocnumber2* and *tokennumber*, which measure the vocabulary size before a new table is added to *wordlist*, the vocabulary size after a new table is appended, and the cumulative number of tokens, which increases by 2,000. Statement 11 accesses *wordlist*, and statements 12 to 33 create a loop, in which the 50 tables are loaded one by one into *wordlist* and processed. *textvocsize* stores the vocabulary size of a table loaded into *vocgrowth*, and *vocincrease* stores the number of new words a table produces for the vocabulary growth. When *i* = 1, *bncwlem1* is loaded; *tokennumber* is now 2,000, and *vocnumber2*, *textvocsize*, and *vocincrease* are all the same at this stage. Statement 18 assigns 1 to *rng* for all the words appended from *bncwlem1*. In statement 25 *vocnumber1* is given the value of *vocnumber2*, which is the current number of records of *wordlist* after *bncwlem1* is loaded. Statement 26 selects work area 2 and accesses *vocgrowth*. Statement 27 creates a blank record in *vocgrowth* for storing cumulative number of tokens, vocabulary sizes of individual tables, vocabulary growth and the new vocabulary a table contributes to the vocabulary growth, which is done by statements 28—31. Statement 34 accesses *wordlist* for the next round of processing. When *i* = 2 *bncwlem2* is loaded, and statement 15 increases *tokennumber* by 2,000, which is now 4,000. Statement 16 assigns *vocnumber2* the current number of records of *wordlist*. Statement 17 measures the vocabulary size of *bncwlem2* by subtracting *vocnumber1* from *vocnumber2*; *vocnumber1* now holds the number of records of *wordlist* before *bncwlem2* is loaded (assigned in statement 25 in the previous round). Statement 18 assigns 1 to *rng* for all the newly appended words, and statements 19—22 combine identical words of *bncwlem1* and *bncwlem2* together, measuring their range at the same time. Statement 23 gets the number of records in *wordlist* after identical words are combined. Statement 24 calculates the number of new words *bncwlem2* produces by subtracting *vocnumber1*, the number of records of *wordlist* before *bncwlem2* is loaded, from *vocnumber2*. Statement 25 gives the current value of *vocnumber2* to *vocnumber1*, and statements 27 to 31 input the newly obtained data to *vocgrowth* in work area 2, after which the program switches to work area 1 and ready for the next round. The above processes are repeated until all the 50 tables are appended. In statement 35 the program switches to work area 1 and measures word length of all the words in *wordlist*, thus ending the program.

Exercises

1. Create a two-field table and append all the words and their frequency from *words.txt* in *d:\fox\texts*.

2. Use the command for automatic table modification to modify the table you have just created, renaming the two fields, changing their width and adding a new numeric field with a width of 12 and 4 decimal places.

3. Open the table *wordlist* in *d:\fox\table3* and output every other word to a new table.

4. Use *wordlist* and output words beginning with *Ex* to a table, and then output word ending in *ed* to another table.

5. The Zipf rank of a wordlist is obtained by assigning ranks to words arranged in decreasing frequency. The word with the highest frequency is given a Zipf rank of 1, the second highest frequency a Zipf rank of 2 and so on until the end of the wordlist is reached. The *h*-point is the point in a wordlist where the Zipf rank equals the corresponding word frequency. For example, if a word has a Zipf rank of 10 and its frequency is also 10, then the *h*-point is 10. The *h*-point is useful for studying vocabulary richness and text themes. Create a three-field table, one for words, another for word frequency and the third for Zipf ranks of the words. Append words and frequency from *wordlist* in *d:\fox\table3* and assign Zipf ranks to the words and locate the *h*-point.

6. Write a program to create a two-field table, one field for the names of all the 48 texts (*text1.txt—text48.txt*) in *d:\fox\texts*, the other for the contents of these text chunks, then input the text names and their contents in their respective field.

7. In quantitative linguistics and natural language processing, we often use the concept of binomial distribution. The equation for calculating the binomial distribution is:

$$b(r;n,p)=\binom{n}{r}p^r(1-p)^{(n-r)},$$

where $\binom{n}{r}=\frac{n!}{(n-r)!r!}$. n is the number of trials, and r the number of successes out of n trials; p is the probability of success in any trial. If n = 6, r = 3, p = 0.5, write a program to calculate $b(r;\, n, p)$.

8. Check the fit of the following models to the vocabulary growth in the table *vocincrease* created with *vocgrowth.prg*. *V*: vocabulary size, *N*: text length.

(1) $V = \alpha(\ln N)^{\beta}$ (Brunet), *a* = 0.003315956, β = 6.017229305.

(2) $V = \alpha N^{\beta}$ (Herdan, Heaps), *a* = 65.73656, β = 0.4291.

(3) $V = a\sqrt{N}$ (Guiraud, Sánchez & Cantos), *a* = 24.706408821.

(4) $V = \dfrac{Z(\ln Z - \ln N)N}{(\ln Z + \alpha)(Z - N)}$ (Orlov). *Z* = 132000， *a* =1.48369912.

9. Modify *vocgrowth.prg* so that it can calculate the word frequency, word range and word length of the words of all of the 48 text chunks of *alice.txt* in *d:\fox\texts*, as well as the word frequency of the individual texts. Output the words that are unique to *text44.txt* to a new table.

10. Use *wordlist* in *d:\fox\table3*, copy it to *d:\fox\practice\test* and do the following using the command window:

a. sort the word field on the second letter from the left;

b. sort the word field on the last letter of the words.

3 Number Crunching and Pattern Matching in Foxpro Tables

3.1 More Functions and Commands for Math Operation in Tables

In 1.2.5 we looked at some functions for math operations. Now we'll learn some other functions and commands for math operations.

count to *variable* **for** *condition* This command counts the number of records of a table satisfying the specified conditions and stores the result to *variable*. If we want to count the number of words with length 3 in *wordlist* in *d:\fox\table3* and store the result in a variable called *length3*, type:

```
count to wordnumber for wlength=3 ↵
?wordnumber↵
826
```

To count the number of words longer than 7 letters occurring more than 5 times and store the result in *wordnumber,* type:

```
count to wordnumber for freq>5 and wlength>7↵
? wordnumber ↵
2959
```

sum [*fieldname* **to** *variable*] [**for** *condition*] This command sums numeric data of a field to a variable, with an optional condition. For example, to get the number of word tokens whose corresponding word types occur 15 times and store the result in *wordnumber*, type:

```
sum freq to wordnumber for freq=15 ↵
?wordnumber ↵
2430
```

If only *sum* is used, all the numeric fields are summed and the result displayed on the screen.

```
 sum ↵
  freq          rng         wlength
903441.00    292256.0    184551.00
```

average [*fieldname* **to** *variable*] [**for** *condition*] This command calculates the arithmetic mean of a numeric field and stores the result to *variable*, with an optional condition. To calculate the mean word length of words that occur less than 3 times and store the result in *mlength*, type:

```
average wlength to mlength for freq<3 ↵
?mlength ↵
8.17
```

If only *average* is used, all the numeric fields are averaged and the result displayed on the screen.

```
average ↵
freq          rng       wlength
37.76         12.22     7.71
```

min(*n1,n2,…nx*) This function picks out the smallest number among the numbers within the brackets. The number of values to be compared can not exceed 26. Now type:

```
? min(1000034,5612992) ↵
1000034

?min(34,56,12,99,0.1,-2) ↵
-2
```

The following results in an error message because the number of values to be compared exceeds 26:

```
?min(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27) ↵
Too many arguments.
```

max(*n1,n2,…nx*) This function selects the largest number among the numbers within the brackets. The number of values within the brackets can't exceed 26, too. Type:

```
?max(90078,436779000) ↵
436779000

?max(0.0067,0.000089714) ↵
0.0067
```

calculate [**avg**(*fieldname*)] [, **min**(*fieldname*)] [, **max**(*fieldname*)] [,

std(*fieldname*)] [, **var**(*fieldname*)] [, **cnt**(*fieldname*)] [, **sum**(*fieldname*)] [**to** *variable1*, *variable2…*] [**for** *condition*] This command calculates the average, minimal value, maximal value, standard deviation, variance, number of records, and sum of a numeric field and stores the results to their corresponding variables. Except for *std(fieldnam)* and *var(fieldname)***,** this command combines some of the math functions we have covered in the previous chapter. *min(fieldname)* and *max(fieldname)* respectively pick out the smallest and the largest value of a numeric field, no matter how many cells there are in the field. Now type:

```
calculate std(wlength),var(wlength),cnt(wlength) to s1,s2,s3 for wlength>2
and wlength<7 ↵
STD(wlength)          VAR(wlength)          CNT(wlength)
1.00                  0.99                  8365

?s1 ↵
1

?s2 ↵
0.99

?s3 ↵
8365
```

To get the minimal and maximal word length of words with frequency between 400 and 700, type:

```
calculate min(wlength),max(wlength) for freq>400 and freq<700 ↵
MIN(wlength)          MAX(wlength)
2                     10
```

3.2 Moving the Record Pointer and Creating Conditional Statements

During data processing in a table, we often need to move the record pointer to a specified position; sometimes we need to set conditions for the execution or non-execution of a statement or a series of statements in a program. The following commands are for such purposes.

go | [**top**] [**bottom**] [*n*] | *go top* moves the record pointer to the top of a table, *go bottom* moves the record pointer to the bottom of a table, while *go n* moves the record pointer to the *n*th record of the table. For example, go 178 moves the record pointer to the 178th record of a table. Now open *wordlist* in *d:\fox\table3* and type the following in the command window:

```
go 24 ↵
brow ↵
go bottom ↵
brow ↵
go top ↵
brow ↵
```

skip | [*n*] [-*n*] | This command moves the record pointer *n* steps forward or backward in a table. *skip -5* moves the record pointer 5 steps backward from its current position, while *skip 10* moves the record pointer 10 steps forward. *skip* used alone moves the record pointer one step forward. Now type in the command window:

```
skip 23 ↵
brow ↵
skip -10 ↵
brow ↵
skip ↵
brow ↵
```

bof() This function tests whether the record pointer is over the top of a table. Type:

```
go 11 ↵
?bof()↵
.F.

go top ↵
?bof() ↵
.F.

skip -1 ↵
?bof() ↵
.T.
```

eof() This function tests whether the record pointer has passed the bottom of a table. Type:

```
go 58 ↵
?eof() ↵
.F.
```

```
go bottom ↵
?eof() ↵
.F. ↵

skip ↵
?eof() ↵
.T.
```

do while *condition*…**enddo** This command creates a loop, and as long as the specified condition is met, statements between *do while condition* and *enddo* are executed repeatedly. In processing data within a table, this command is often used with *eof()*.

exit This command is used for ending a loop.

if *condition*…[**else** *condition*]…**endif** This command is used to execute a statement if a condition is met.

In *d:\fox\table3* there is a table *spwordlist* containing the vocabulary of 500 2000-word samples from the spoken text section of the BNC. Suppose we want to output the words in *spwordlist* whose range is 500 to the screen, we can use the following little program to do it:

doif.prg

```
1. set default to d:\fox\practice
2. use d:\fox\table3\spwordlist
3. do while not eof()
4. if rng=500
5. ?word
6. endif
7. skip
8. enddo
```

Statement 3 and statement 8 create a loop, in which, as long as the record pointer is still within the table, the statements between them are executed again and again, and when the record pointer comes to a record whose range field is 500, statement 5 is executed and the result is shown on the screen as follows:

```
A
And
Be
Can
Do
Get
```

Go
Have
In
It
Know
No
Not
Of
Oh
On
That
The
Then
There
To
Well
What
Will
With
Yeah
You

Next, we'll see how to use *exit* and *if condition…else condition…endif* in programs. In natural language processing, we often need to separate a text or a corpus into *N*-grams, i.e., bigrams, trigrams and so on. *bigram.prg* separates *alice.txt* in *d:\fox\texts* into bigrams.

bigram.prg

```
set defa to d:\fox\practice
set safe off
set talk off
clear
close data
creat cursor wordtable (word c(25))
creat table bigram (bgram c(40),freq n(6))
nothing=''
carriage=chr(13)
spaces=chr(32)
twowords=nothing
textinput=fileto('d:\fox\texts\alice.txt')
textinput=chrtran(textinput,'`*()_',spaces)
textinput =alltr(strtr(textinput,spaces,carriage))
strtof(textinput,'temp.txt')
sele 1
```

```
17. appe from temp.txt sdf for word<>spaces
18. go top
19. do while not eof()
20. for i=1 to 2
21. position=recn()
22. twowords=twowords+alltr(word)+spaces
23. skip
24. endfo
25. twowords=twowords+carriage
26. if position<recc()
27. go position
28. else
29. exit
30. endif
31. enddo
32. strtof(twowords,'temp.txt')
33. sele 2
34. appe from temp.txt sdf
35. replace all freq with 1
36. inde on bgram tag bgram
37. total to temp on bgram
38. zap
39. appe from temp
40. copy to bigram.txt sdf
41. modi file bigram.txt
```

In this program, statement 6 creates a temporary table called *wordtable* in work area 1. Tables of this kind will be automatically deleted after the program has run. It holds the tokenized *alice.txt*. Statement 11 initializes the variable *twowords* by assigning it *nothing*; this variable is for storing bigrams. Statements 12—15 respectively put the contents of *alice.txt* to *textinput*, replace `*,()_ with a white space, tokenize *alice.txt* and output the tokenized *alice.txt* to *temp.txt*. Statement 16 accesses the temporary table *wordtable*, which is now open in work area 1, and appends the tokenized *alice.txt* from *temp.txt* minus the white spaces. Statement 18 moves the record pointer to the top of the temporary table *wordtable*. Statements 19—31 create a loop that ends when the end of *wordtable* is reached. In this loop, words are taken one by one from *wordtable* to form bigrams, which are stored in the variable *twowords*. Let's see how this is done in the program. The following are the first 24 records of *wordtable*.

ALICE'S
ADVENTURES
IN
WONDERLAND

```
CHAPTER
I
Down
the
Rabbit-Hole
Alice
was
beginning
to
get
very
tired
of
sitting
by
her
sister
on
the
bank,
...
```

Statements 20—24 create a two-round loop within the *do…enddo loop*. Initially, when *i* = 1 the record pointer is at the first record, and statement 21 assigns 1 to *position*. In statement 22 *twowords* is given the first record *ALICE'S* plus a white space, and statement 23 moves the record pointer to the next record and the program goes back to statement 20 to increase *i* by one, which is now 2. *position* now becomes 2 in statement 21. In statement 22 *twowords* is given the second record *ADVENTURES* plus a white space. Now it contains the first bigram *ALICE'S ADVENTURES.* Statement 23 moves the record pointer to the third record. Since *i* is now 2, the programs proceeds to statement 25, which puts a carriage return to the end of *twowords* so that the second bigram will start on a new line. The second bigram should be *ADVENTURES IN*, which is to be formed by the second record and the third record of *wordtable*; but the record pointer is now at the third record, so statement 27 moves the record pointer back to the second record, under the condition set by statement 26 that *position* is smaller than the total number of records in *wordtable*. After statement 27 is carried out, the program returns to statement 20 and repeats the above process until *position* equals the number of records in *wordtable*, i.e. all the words in *wordtable* have been turned into bigrams. Statement 32 puts the bigrams stored in *twowords* to *temp.txt*. Statement 33 accesses the table *bigram* now open in work area2, and statements 34—40 append the bigrams to it, compute their frequency and copy the frequencied bigrams to a text file *bigram.txt*.

3.3 Math Operation in Foxpro Tables

In quantitative linguistics we often have to do very complicated computation to get certain linguistic measurements. These tasks can be easily and efficiently done in Foxpro tables. We'll look at several such examples in this section.

3.3.1 Creation of frequency spectrum

Frequency spectrum is a table listing word frequency classes of a text and the number of words belonging to each of these classes. For example, in *alice.txt*, the frequency spectrum for words occurring once to 5 times is:

m	V(m,N)
1	1133
2	401
3	233
4	151
5	95

(see *aliceword* created in 2.4.1). Here *m* is the frequency class and *V*(*m,N*) the number of words belonging to a class. In *alice.txt,* there are 1,133 words occurring once, 401 words occurring twice, 233 word occurring three times, 151 words occurring 4 times, and 95 words occurring 5 times.

Now we'll write a program making a frequency spectrum for *lwordlist* (in *d:\fox\practice* created in 2.5.1), which is a wordlist for *Through the Looking-glass* (*lglass.txt*, in *d:\fox\texts*), and put the result in a table called *spectrum* and in a text file *spectrum.txt* as well.

spectrum.prg

```
set defa to d:\fox\practice
close data
set safe off
create table spectrum(m n(8),vmn n(8))
for i=1 to 1589 && the highest frequency in lwordlist
use lwordlist
count to freqclass for freq=i
if freqclass>0
use spectrum
append blank
replace m with i
replace vmn with freqclass
endif
endfor
use spectrum
copy to spectrum.txt sdf
```

Statement 4 creates a two-field table *spectrum*, *m* holding frequency, and *vmn* the number of words with frequency of *m*. Statements 5—14 create a loop, within which the number of words in *lwordlist* with frequency *m* is counted and the result stored in *freqclass*. In *lwordlist*, the highest frequency is 1,589, but the second highest frequency is 907, not 1,588, and the third highest is 765. Statements 8—13 avoid 0 to be put to *spectrum*. When *freqlass* is 0, the program goes back to statement 5 and *i* is increased by 1. Statements 9—12 are carried out if *i* corresponds to a frequency in *freq* field of *lwordlist*, in which case *freqlass* is greater than 0. The following is the frequency spectrum of *Through the Looking-glass*.

Table 3.1 Frequency spectrum of *Through the Looking-glass*

m	*V(m,N)*	*m*	*V(m,N)*	*m*	*V(m,N)*	*m*	*V(m,N)*
1	*1168*	*30*	*3*	*63*	*3*	*123*	*1*
2	*458*	*31*	*3*	*64*	*2*	*124*	*1*
3	*262*	*32*	*3*	*65*	*2*	*125*	*1*
4	*158*	*33*	*5*	*66*	*2*	*126*	*3*
5	*123*	*34*	*3*	*67*	*2*	*135*	*1*
6	*78*	*35*	*3*	*68*	*2*	*137*	*1*
7	*65*	*36*	*7*	*69*	*1*	*142*	*1*
8	*47*	*37*	*4*	*70*	*2*	*146*	*1*
9	*31*	*38*	*3*	*71*	*3*	*147*	*1*
10	*25*	*39*	*1*	*74*	*2*	*148*	*1*
11	*23*	*40*	*3*	*75*	*2*	*153*	*1*
12	*36*	*41*	*3*	*76*	*1*	*157*	*1*
13	*20*	*43*	*2*	*78*	*1*	*179*	*1*
14	*19*	*45*	*1*	*79*	*1*	*194*	*1*
15	*26*	*46*	*2*	*80*	*1*	*199*	*1*
16	*13*	*48*	*2*	*81*	*1*	*203*	*1*
17	*16*	*49*	*2*	*85*	*1*	*204*	*1*
18	*7*	*50*	*3*	*86*	*1*	*216*	*1*
19	*12*	*51*	*1*	*87*	*1*	*221*	*1*
20	*11*	*52*	*1*	*88*	*1*	*229*	*1*
21	*9*	*53*	*4*	*90*	*2*	*249*	*1*
22	*10*	*54*	*4*	*94*	*1*	*270*	*1*
23	*6*	*55*	*1*	*95*	*1*	*314*	*2*
24	*3*	*56*	*4*	*97*	*1*	*354*	*1*
25	*3*	*57*	*3*	*98*	*1*	*409*	*1*
26	*6*	*58*	*1*	*103*	*1*	*434*	*1*
27	*4*	*59*	*2*	*111*	*1*	*472*	*1*
28	*6*	*60*	*2*	*116*	*1*	*489*	*1*
29	*3*	*62*	*2*	*117*	*2*	*506*	*1*

521	*2*	*732*	*1*	*907*	*1*
562	*1*	*765*	*1*	*1589*	*1*

3.3.2 The distribution of hapax legomena

Hapaxes are a very important word category and several linguistic measures rely on the number of hapaxes. For example, the vocabulary growth rate $P(N)$, the number of hapaxes $V(1,N)$ and the size of a text have the following relationship:

$$P(N) = \frac{V(1,N)}{N}$$

Generally, hapaxes account for about 40% of the vocabulary of a text. Now we'll write a program called *hapax.prg* to compute *P(N)* of the 48 text chunks from *alice.txt* in *d:\fox\texts*, the ratio between the number of hapaxes and vocabulary size in each of the texts, the mean word length of hapaxes, and the standard deviation of the vocabulary sizes and the number of hapaxes of the individual texts. Information on the word frequencies of the 48 text chunks is in *multitext* (*d:\fox\practice*) created in 2.5.2.

hapax.prg

```
set default to d:\fox\practice
set safe off
set talk off
close data
clear
use multitext
create table texthapax (texts c(10),vocsize n(6,2),hapsize n(6,2), hvratio
n(6,2),pn n(6,2),mhlength n(6,2))
for i=1 to 48
select 1 &&access multitex open in work area 1
wordfield='text'+alltr(str(i))
freqfield='freq'+alltr(str(i))
count to hapaxnumber for &freqfield=1
count to vocnumber for &freqfield>0
sum &freqfield to tokennumber
ratio=hapaxnumber/vocnumber
average wlength to meanhaplength for &freqfield=1
sele 2&&access texthapax open in work area 2
append blank
replace texts with wordfield
replace vocsize with vocnumber
```

```
21. replace hapsize with hapaxnumber
22. replace hvratio with ratio
23. replace pn with hapaxnumber/tokennumber
24. repl mhlength with meanhaplength
25. endfor
26. set talk on
27. calculate avg(vocsize),avg(hapsize),avg(hvratio),avg(pn),
    avg(mhlength)
28. calculate min(vocsize), min(hapsize), min(hvratio), min(pn),
    min(mhlength)
29. calculate max(vocsize),max(hapsize), max(hvratio), max(pn),
    max(mhlength)
30. calculate std(vocsize),std(hapsize),std(mhlength)
```

Although this program has 30 statements, it's very easy to understand. Statement 3 sets talk off to suppress screen display. Statement 7 creates a 6-field table *texthapax*, holding text names, vocabulary sizes, number of hapaxes, hapax/vocabulary ratio, vocabulary growth rates and average length of hapaxes of the individual texts. Statements 8—25 create a loop, in which the distribution of vocabulary sizes and the number of hapaxes, hapax/vocabulary ratio, vocabulary growth rates and average hapax length of the individual texts are computed and appended to *texthapax*. Statements 26 sets talk on for displaying related information on the screen. Statements 27—30 compute related averages, standard deviation, minimum values, maximum values, etc and output the results to the screen.

3.3.3 Yule's *K*

Yule's *K* is a lexical constant proposed by Yule, who claims it to be independent of text length. Yule's *K* can be used as a measure for vocabulary richness and for author identification. To compute *K* of a text, we should turn the text into a wordlist with word frequency, make a frequency spectrum, and then compute *K*. The formula for computing *K* is:

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

where *m* is the frequency classes, and *V*(*m*,*N*) is the number of words whose frequency is *m*, and *N* the number of words a text has. Now we'll write a program called *yulek.prg* to compute *K* of *Through the Looking-glass* (*lglass.txt* in *d:\fox\texts*) using its frequency spectrum we just made in 3.3.1. There are 29,633 words in *lglass.txt,* and the number of records in *spectrum* is 122, which means there are 122 different frequency classes in *lglass.txt*.

yulek.prg

```
set default to d:\fox\practice
set safe off
close data
cumu=0
use spectrum
do while not eof()
cumu=cumu+m**2*vmn
skip
enddo
?10000*((cumu-29633)/29633**2)
```

Yule's *K* is 90.7566.

3.3.4 Per word entropy of English

Entropy measures the average uncertainty of a single random variable and is expressed as the following:

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

It was first proposed by Shannon, who computed the per-letter entropy of English to be 1.3 bits. Here $p(x)$ is the probability of the occurrence of x. Another concept is perplexity, which is obtained with:

$$Perplexity = 2^H.$$

Both entropy and perplexity are widely used in natural language processing. Now we'll write a program called *entropy* to compute the per-word entropy and perplexity of *wordlist* in *d:\fox\table3*

entropy.prg

```
set defa to d:\fox\practice
close data
set safe off
set decimal to 16
create table entropytable(word c(25),freq n(8),prob n(18,16),logfreq
   n(20,16),entropy n(18,16))
append from d:\fox\table3\wordlist field word,freq
```

```
7. sum freq to tokennumber
8. replace all prob with freq/tokennumber
9. replace all logfreq with 1/(log10(2)/log10(prob))
10. replace all entropy with prob*logfreq
11. sum entropy to entropysum
12. append blank
13. replace word with 'ENTROPY:'
14. replace entropy with -entropysum
15. append blank
16. replace word with 'PERPLEXITY:'
17. replace entropy with 2**-entropysum
18. brow
```

Statement 5 creates a five-field table holding words, their frequency, the probabilities of these words, the log probabilities to the base 2 and $p(x)\log_2 p(x)$. Statement 7 gets the total number of tokens. Statement 9 computes $\log_2 p(x)$, statement 10 $p(x)\log_2 p(x)$, and statement 11 $\sum_{x\in X} p(x)\log_2 p(x)$. Statements 14 and 17 respectively give *H* and *perplexity*, which are respectively 9.6449 and 800.5864.

3.3.5 Word length in syllables

Altmann proposes that the longer a language construct, the shorter its components. Mathematically,

$$y = Ax^{-b},$$

where *x* is a language construct, *y* its components, and *A* and *b* are parameters. We'll check whether this relationship holds between word length *x* measured in number of syllables and mean syllable length *y* in number of letters in *wordlist* (in *d:\fox\table3*), for which $A = 4.1484$ and $b = 0.30896$. We need a program that first calculates word length in syllables and then checks the fit of $4.1484x^{-0.30896}$ to the observed mean syllable length of *wordlist*.

A word has the following syllabic structure: (nV)nCnV[nC(nV)]. nV is a vowel or a vowel cluster and nC is a consonant or a consonant cluster. The elements within the round brackets are optional and those in the square brackets can be reduplicated. 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:

1. 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 ;

2. *ble*, *ple*, *sm* at the end of a word constitute a syllable, e.g., *people, syllable*, *isolationism*, etc;
3. 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.

For the sake of simplicity, *ia* will be regarded as forming two syllables while *ea*, *io*, *uo* and other vowel clusters as forming one syllable.

Before writing the program, we'll look at a function for measuring the occurrences of a character or characters in a string.

occurs(*string1,string2*) This function measures the occurrence of *string1* in *string2*. Now type:

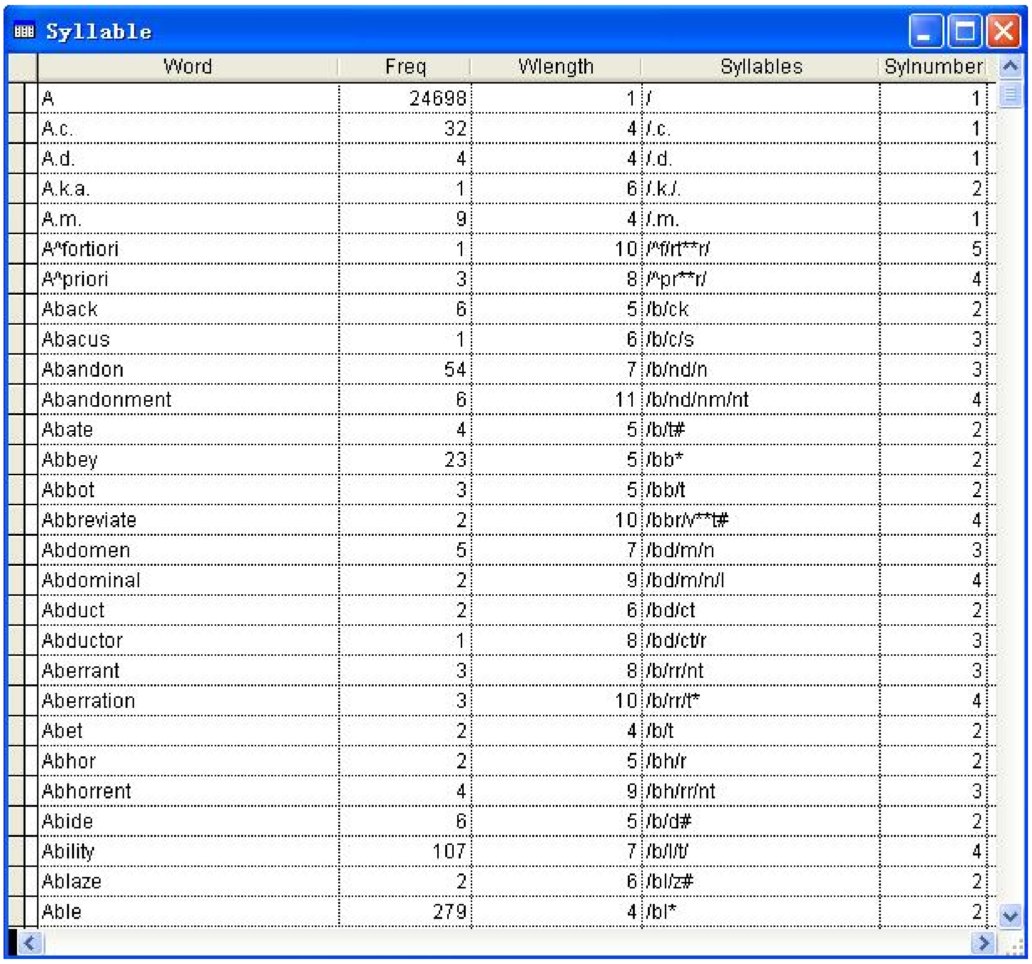
```
?occurs(wh','what is that?') ↵
1

?occurs('t', 'what is that? ') ↵
3
```

syllable.prg

```
set defa to d:\fox\practice
set safe off
set talk off
clear
close data
create table sylength(sylnumber n(6),wordnum n(5),avsylength n(6,4),
prediclen n(6,4))
create table syllable(word c(25),freq n(8),wlength n(6),syllables c(25),
sylnumber n(5))
select 2&&access syllable
append from d:\fox\table3\wordlist fiel word,freq
replace all syllables with word
replace all wlength with len(alltr(word))
replace all syllables with strtr(syllables,'ee ','*')
replace all syllables with strtr(syllables,'sm ','*')
replace all syllables with strtr(syllables,'ple ','pl*')
replace all syllables with strtr(syllables,'ble ','bl*')
replace all syllables with strtr(syllables,'iu','**')
replace all syllables with strtr(syllables,'ia','**')
replace all syllables with strtr(syllables,'ion','*')
replace all syllables with strtr(syllables,'io','**')
replace all syllables with strtr(syllables,'ey ','*')
replace all syllables with strtr(syllables,'ay ','*')
```

```
22. replace all syllables with strtr(syllables,'gue ','g/')
23. replace all syllables with strtr(syllables,'e ','#')&&note the white space
    after e
24. replace all syllables with strtr(syllables,'y','/')
25. replace all syllables with strtr(syllables,'ere','/')
26. replace all syllables with strtr(syllables,'ively ','/vl/')
27. replace all syllables with strtr(lower(syllables),'a','/')
28. replace all syllables with strtr(lower(syllables),'e','/')
29. replace all syllables with strtr(lower(syllables),'i','/')
30. replace all syllables with strtr(lower(syllables),'o','/')
31. replace all syllables with strtr(lower(syllables),'u','/')
32. replace all syllables with strtr(syllables,'//','/')
33. replace all syllables with strtr(syllables,'//','/')
34. replace all sylnumber with occurs('/', syllables)+occurs('*', syllables)
35. replace all sylnumber with 1 for sylnumber =0
36. for i=1 to 10
37. calcul sum(wlength/sylnumber), cnt(wlength) to sumlength,
    wordnumber for sylnumber=i
38. if wordnumber>0
39. select 1
40. append blank
41. replace sylnumber with i
42. replace wordnum with wordnumber
43. replace avsylength with sumlength/wordnumber
44. endif
45. selec 2
46. endfor
47. select 1
48. replace all prediclen with 4.1484*sylnumber**-0.30896
49. brow
```

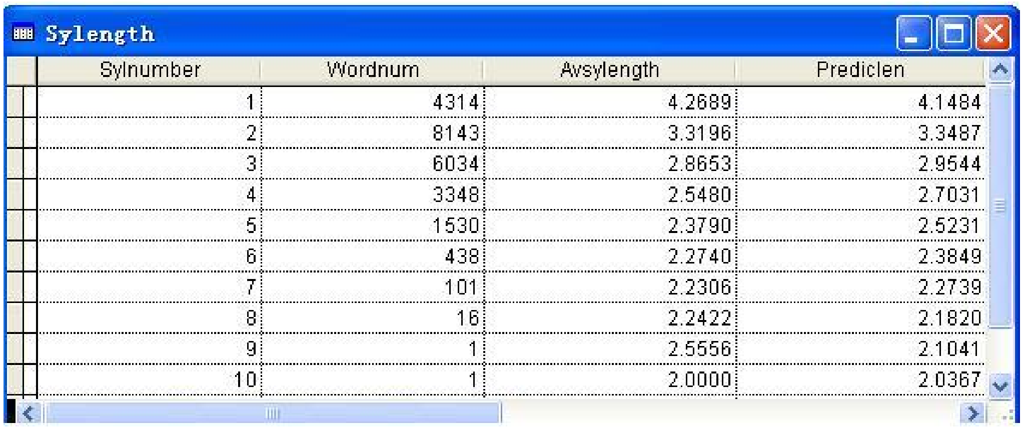
In this program, statement 6 creates *sylength* in work area 1, whose fields *sylnumber*, *wordnum*, *avslength* and *prediclen* respectively hold number of syllables in words, number of words that have 1 to 10 syllables, average syllable length (in letters) and predicted average syllable length. Statement 7 creates *syllable* in work area 2 with fields *word*, *freq*, *wlength*, *syllables*, *sylnumber* respectively holding words, word frequency, word length in letters, word separated into syllables and number of syllables in words. Statement 9 appends words and word frequency from *wordlist*. Statement 10 puts all the words in the *word* field into the now empty field *syllables* for syllable separation. Statements 12—21 replace certain graphemes with *, which stands for a syllable (in this program, both * and / are used as a syllable marker). For example, *ion* normally forms only one syllable, so it's replaced with a single *, while *ia* often forms two syllable so it's replaced with **. Statement 23 replaces *e* at the end of a word with # meaning it should not be counted as a syllable formed with the preceding consonant. Statements 27—33 mark vowels for syllable separation. Statements 32—33 combine contiguous vowel markers into one. The two identical statements are for words such as *beauty*, which becomes *b///t/* after statements 24, 27, 28 and 31. But *beauty* has only two syllables, so statement 32 turns *b///t/* to *b//t/*, and statement 33 turns *b//t/* to *b/t/*, meaning it has two syllables. Statement 34 gets the number of syllables in a word by counting the number of / and * in it. Statements 36 to 46 create a loop. Statement 36 sets the initial value of *i* to 1 and its maximum value to 10 since the longest word length in syllables in *wordlist* is 10. Statement 37 sums syllable length (in letters) of words that have *i* syllables and count the number of words with *i* syllables. In case certain word syllabic length can't be measured, such as *Dr.*, statement 35 ensures that such words have 1 syllable. Statement 48 computes the fit of $4.1484x^{-0.30896}$ to the observed average syllable length (in letters). Figure 3.1 is part of *syllable*. The result is fairly accurate. However, in linguistic computing, especially in tasks such as syllable counting, parts of speech tagging etc, it's almost impossible to achieve a 100% accuracy, so quite often manual checking is needed to weed out possible errors. The reader can check the entire table for errors and see if it's possible to improve the program to avoid such errors. Figure 3.2 is *syllable*; the table respectively stores in *sylnumber, wordnum*, *avsylength* and *prediclen* the number of syllables from 1 to 10, the observed average syllable length in letters and the predicted values. The fit is good. However, the average length of syllables for the word with 9 syllables is suspicious because it's 2.5556. Checking *syllable* reveals that the word is *pancreaticoduodenectomy*, which has 23 letters and 11 syllables instead of 9. This is caused by taking *ea* and *uo* as forming one syllable each instead of two. Its actual length in syllables is 2.09. Correcting the mistake, the fit is much better.



Syllable

Word	Freq	Wlength	Syllables	Sylnumber
A	24698	1	/	1
A.c.	32	4	/.c.	1
A.d.	4	4	/.d.	1
A.k.a.	1	6	/.k./.	2
A.m.	9	4	/.m.	1
A^fortiori	1	10	/^f/rt**r/	5
A^priori	3	8	/^pr**r/	4
Aback	6	5	/b/ck	2
Abacus	1	6	/b/c/s	3
Abandon	54	7	/b/nd/n	3
Abandonment	6	11	/b/nd/nm/nt	4
Abate	4	5	/b/t#	2
Abbey	23	5	/bb*	2
Abbot	3	5	/bb/t	2
Abbreviate	2	10	/bbr/v**t#	4
Abdomen	5	7	/bd/m/n	3
Abdominal	2	9	/bd/m/n/l	4
Abduct	2	6	/bd/ct	2
Abductor	1	8	/bd/ct/r	3
Aberrant	3	8	/b/rr/nt	3
Aberration	3	10	/b/rr/t*	4
Abet	2	4	/b/t	2
Abhor	2	5	/bh/r	2
Abhorrent	4	9	/bh/rr/nt	3
Abide	6	5	/b/d#	2
Ability	107	7	/b/l/t/	4
Ablaze	2	6	/bl/z#	2
Able	279	4	/bl*	2

Figure 3.1 Part of the table *syllable*



Sylength

Sylnumber	Wordnum	Avsylength	Prediclen
1	4314	4.2689	4.1484
2	8143	3.3196	3.3487
3	6034	2.8653	2.9544
4	3348	2.5480	2.7031
5	1530	2.3790	2.5231
6	438	2.2740	2.3849
7	101	2.2306	2.2739
8	16	2.2422	2.1820
9	1	2.5556	2.1041
10	1	2.0000	2.0367

Figure 3.2 The uncorrected *sylength*

3.4 Commands and Functions for Pattern Matching

locate for *condition* This command locates a record satisfying the specified condition in a table. Now open *wordlist* in *d:\fox\table3*. To locate a word with length 16, type:

```
locate for wlength=16 ↵
?word ↵
Aquaintanceship
```

Type *brow* ↵ and we can see the record pointer is at the record containing *Aquaintaceship*.

continue This command continues the action of the *locate* command. Now type:

```
continue ↵
?word ↵
Administratively

brow ↵
```

The record pointer is at the record containing *Administratively*.

like(*string1*, *string2*) This function checks whether *string1* and *string2* are identical. Type:

```
?like('Fox', 'Foxpro') ↵
.F.

?like('Fox', 'fox') ↵
.F.

?like('Fox', 'Fox') ↵
.T.
```

We can use wild cards in *string1*. *?* stands for any single character and * for any number of characters. Now type:

```
?like('Fox*', 'Foxpro') ↵
.T.

?like('Foxp?? ', ''Foxpro') ↵
.T.
```

```
?like('F?x*', 'Foxpro') ↵
.T.
```

We can use this function with the *locate* command for pattern matching. If we want to know whether there are words in *wordlist* that have the letter cluster *scl*, type:

```
locate for like('*scl*',lower(word)) ↵
?word ↵
Disclaim

locate for like('Scl*', word) ↵
?word ↵
Sclerosant
```

The *like()* function can be used with other commands, too. The following lists all the words in *wordlist* that have the letter cluster *scl*:

```
list all for like('*scl*',lower(word)) ↵
```

All words that have the letter cluster are displayed on the screen.

scan for *condition*...**endscan** This command searches for records satisfying the specified condition. Unlike the *locate* command, this command is much faster and searches for all the records meeting the specified conditions. Type the following in the command window. Press the down key to start a new line. After completing entering all the lines, drag the mouse from the first line down to the last line to highlight them, then press Enter:

```
scan for like('A?b*c?',alltr(word))
?word
endscan ↵
```

The result is as follows:

```
Ambiance
Ambience
Ambivalence
Ambulence
```

If we want to search for words whose length is between 15 and 20 letters (inclusive) containing *ou* and ending in *ly*, and output the words meeting these conditions to the screen with their length, enter the following in the command window:

```
scan for wlength>=15 and wlength<=20 and like('*ou*ly',alltr(word))
?word+alltr(str(wlength))
endscan
```

Highlight the three statements and then press Enter, the following are shown on the screen:

```
Conscientiously       15
Contemporaneously     17
Inconspicuously       15
Instantaneously       15
Surreptitiously       15
Unceremoniously       15
```

seek(*string*) This function searches a table for the specified *string*. For this function to work, the table must be indexed. Now copy *wordlist* to *test*. Open *test* and type the following in the command window:

```
seek('Abandon') ↲
```

A warning message pops up saying the table has no index order set. Now type the following:

```
index on word tag word ↲
seek('Abandon') ↲
?recno() ↲
10
```

The *seek* function has found the word *Abandon*, which is in record 10.

Now we'll write a program separating *text1.txt* in *d:\fox\texts* into sentences and measure the length of these sentences. Before processing a piece of text, we should first examine it carefully to determine its general linguistic patterns and exceptions to these patterns. This is called language modelling in natural language processing. In *text1.txt*, punctuation marks ".", "*?*" and "*!*" generally end a sentence except in a few cases where the end of a sentence is marked by "*.)*", "*!'*" or "*?'*". In addition sentences are often broken by carriage returns. So we should take these characteristics into consideration during programming.

sentlength.prg

```
1. set default to d:\fox\practice
2. set safety off
3. close data
4. create table sentlen(sent1 c(250),sent2 c(250),sent3 c(250),sent
   m(4),slength n(4))
```

```
5.  linebreak=chr(10)
6.  carriage=chr(13)
7.  spaces=chr(32)
8.  textinput=filetostr('d:\fox\texts\text1.txt')
9.  textinput=strtr(textinput,carriage+linebreak,spaces)
10. textinput=strtr(textinput,spaces+spaces,carriage)
11. textinput=strtr(textinput, '.)', ').'+carriage)
12. textinput=strtr(textinput,”!'”,” '!”+carriage)
13. textinput=strtr(textinput,”?'”,”'?”+carriage)
14. textinput=strtr(textinput,'.','.'+carriage)
15. textinput=strtr(textinput,'?','?'+carriage)
16. textinput=strtr(textinput,'!','!'+carriage)
17. strtofil(textinput,'temp.txt')
18. append from temp.txt sdf
19. replace all sent1 with alltrim(sent1)
20. delete for sent1=spaces
21. pack
22. replace all sent with alltr(sent1+sent2+sent3)
23. replace all slength with occurs(' ',sent)+1
24. brow
```

In this program, statement 4 creates a table with 5 fields. The first three fields, *sent1*, *sent2* and *sent3* are for holding sentence fragments. Three fields together can hold 750 characters, about 130 words, enough for the longest sentence in *text1*.txt. *sent* is a memo field for holding complete sentences and for measuring sentence length. *slength* is for sentence length measured in number of words. Statement 5 assigns *chr(10)*, a line breaking character, to *linebreak*. This character is used in combination with *chr(13)* in *text1.txt* at the end of each line. Statements 9 replaces the character combination carriage return plus line breaker with a white space so that no sentences are broken in the middle by these characters. Statement 10 converts two or more contiguous spaces into carriage returns so that chapter and section titles, such as *Chapter 1*, *Down the Rabbit-Hole*, which have more than two white spaces preceding them, are each placed in a new line again (Statement 9 puts them in one line.). Statements 11—16 separate *text1.txt* into individual sentences while keeping the punctuation marks at the end of the sentences. Statements 11—13 replace “*.)*”, “*!* ’” and “*?* ’” with “*).*”, “’*!*” and “’*?*” plus a carriage return so that “*)*” and ” ’” won’t be placed in a new line by statements 13—16. Statement 22 combines *sent1*, *sent2* and *sent3* together. The longest sentence in *text1.txt* has 107 words, 565 characters. *sent1* holds the first 250 characters (including white spaces) of the long sentence:

(when she thought it over afterwards, it occurred to her that she ought to have wondered at this, but at the time it all seemed quite natural); but when the Rabbit actually TOOK A WATCH OUT OF ITS WAISTCOAT-

POCKET, and looked at it, and then hurried

sent2 holds the second part of the sentence:

on, Alice started to her feet, for it flashed across her mind that she had never before see a rabbit with either a waistcoat-pocket, or a watch to take out of it, and burning with curiosity, she ran across the field after it, and fortunately was jus

sent3 holds the remaining:

t in time to see it pop down a large rabbit-hole under the hedge.

Statement 22 combines *sent1*, *sent2* and *sent3* together and put the contents to the memo field *sent*, which can practically store texts of any length. Statement 23 calculates the length of all the sentences in number of words by counting the number of spaces within each sentence.

found() This function checks whether a search is successful or not. Now open *wordlist* in *d:\fox\table3* and copy it to *d:\fox\practice\wordlist.* Enter the following in the command window. Press the down key to begin a new line. Highlight the two statements by dragging the mouse from the start of the first statement to the end of the second, and then press Enter:

```
locate for word='Abandon'
?found()
.T.
```

Then try locating a non-word string *Axxx*, the result is *.F.*

select | [*] [*fieldnames*] | **from** *tablename* **where** *condition* [**order by** *fieldname* [**descending**]] **having** [*condition*] [| [**into table** *tablename*] [**to** *filename*] |] [**additive**] [**noconsole**] This command searches for records meeting the condition in specified fields and outputs the selected records in specified fields or all the fields, represented by *, to a table , to the screen or to a text file. If we want to search in *wordlist* for words whose second letter is *d* and which ends in *t*, with length>6, and output the result in order of descending frequency to a table called *temp*, enter in the command window the following statements:

```
select word,freq,wlength from wordlist where like('?d*t',alltr(word)) order
by freq descending having wlength>6 into table temp ↵
brow ↵
```

temp is open with the selected records. To output the above result to a text file called *temp.txt*, type:

```
select word,freq,wlength from wordlist where like('?d*t',alltr(word)) order
```

```
by freq descending having wlength>6 to temp ↵
modi file temp.txt ↵
```

The result is outputted to *temp.txt*. The file extension *txt* is automatically added. At the same time it's also sent to the screen. To suppress the screen display, type:

```
select word,freq,wlength from wordlist where like('?d*t',alltr(word)) having
wlength>6 to temp noconsole ↵
modi file temp.txt ↵
```

If *additive* is used (it must be put before *noconsole*), then the result is added to the old contents of *temp.txt*, instead of overwriting it:

```
select word,freq,wlength from wordlist where like('?d*t',alltr(word)) having
wlength>6 to temp additive noconsole ↵
modi file temp.txt ↵
```

To select words with frequency of 20, 30, 40 and output all the fields of the records satisfying the condition in ascending order, type:

```
select *from wordlist where freq in (20,30,40) order by freq ↵
```

replicate(*character*, *n*) This function replicates *character n* times. If *n* is not an integer, it's rounded down. Type:

```
?replicate('*',5) ↵
*****

?replicate('*',5.7) ↵
*****
?replicate(' ',25)+'Foxpro' ↵
                         Foxpro
```

The first statement replicates * five times, while the second does the same. The third replicates a space 25 times and add them to the left of the string *Foxpro*. The *replicate* function is very useful in text formatting. In *d:\fox\texts\poem.txt* there is a poem by John Keats arranged in left justification:

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 mosss'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--

We'll write a program to re-arrange it with centre justification. The program is as follows:

cjust.prg

```
1. set defa to d:\fox\practice
2. close data
3. set safe off
4. create table poem(lines c(80))
5. spaces=' '
6. append from d:\fox\texts\poem.txt sdf
7. replace all lines with replicate(spaces,80/2-len(alltrim(lines))/2)+ alltrim
   (lines)
8. copy to cjustify.txt sdf
9. modify file cjustify.txt
```

In this program, the centre justification is done by statement 7, which does centre justification by adding spaces to the left of each line of the poem so that the mid point of every line is placed at the centre of a line 80 characters long. 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 mosss'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—

3.5 Pattern Matching in Tables

3.5.1 Extraction of lexical bundles

In conversation and written discourse we often see word sequences such as at the same time, it used to be, for a long time and so on. Biber calls them lexical bundles and defines lexical bundles as recurring sequences of word forms in natural discourse. The following are some of the lexical bundles commonly used in conversation, taken from Longman Grammar of Spoken and Written English:

it's going to be	the end of the	and the other one
it's got to be	the back of the	the other day and
it must have been	the middle of the	the one with the
it used to be	the other side of	the last time I
it's a bit of	other side of the	o'clock in the
it's a lot of	the side of the	at the end of
and it was a	the top of the	in the middle of
it was a bit	the bottom of the	at the back of
it was in the	end of the day	on top of the
it's not too bad	the end of it	for a couple of
that's going to be	the rest of the	for the rest of
i was in the	the rest of it	at the same time
but the thing is	that sort of thing	for a long time
the only thing is	the name of the	by the time I
some of them are	most of the time	in the morning and
it's a bit of a	quite a lot of	up in the morning
it's a lot of money	or something like that	on the other side
it's nothing to do with	and things like that	in the first place
that's what I said to	nothing to do with	

These bundles are in bundle.txt in *d:\fox\texts*. We'll write a program searching for these lexical bundles in lglass.txt in *d:\fox\texts*; if a sentence containing one of the bundles listed above is found, it's extracted from the text and put in a text file. The program is as follows.

bundle.prg

```
1. set default to d:\fox\practice
2. set safety off
3. close data
4. create table lexbundle(bundlfield c(30),freq n(4))
5. create table sentence(sent1 c(250),sent2 c(250),sent3 c(250),sent m(4))
6. textinput=filetostr('d:\fox\texts\lglass.txt')
7. tabs=chr(9)
8. linebreak=chr(10)
9. carriage=chr(13)
```

```
spaces=chr(32)
bundletext=''
bundlenumber=0
textinput=strtr(textinput,carriage+linebreak,spaces)
textinput=strtr(textinput,spaces+spaces,carriage)
textinput=strtr(textinput,'.','.'+carriage)
textinput=strtr(textinput,'.”','.”'+carriage)
textinput=strtr(textinput,'?','?'+carriage)
textinput=strtr(textinput,'!','!'+carriage)
textinput=strtr(textinput,'!”','!”'+carriage)
strtofil(textinput,'temp.txt')
select 2
append from temp.txt sdf
replace all sent1 with alltrim(sent1)
delete all for sent1=spaces
pack
replace all sent with alltr(sent1+sent2+sent3)
select 1
append from d:\fox\texts\bundle.txt sdf
go top
do while not eof()
bundle= alltr(bundlfield)
select sent from sentence where like('*'+bundle+'*',lower(sent)) into table temp
counter=reccount()
if counter>0
bundlenumber=bundlenumber+1
bundletext=bundletext+'('+alltr(str(bundlenumber))+'). '+upper(bundle)+carriage
replace all sent with strtr(lower(sent), bundle,'** '+upper(bundle)+' **') &&the sent field in the table temp
go top &&the top of temp
do while not eof()
bundlesent=alltr(sent)
bundletext=bundletext+tabs+alltr(str(recn()))+'. '+bundlesent+carriage
skip
enddo
bundletext=bundletext+carriage
endif
select 1 &&access the table lexbundle
replace freq with counter
skip
enddo
```

```
50. strtofile(bundletext,'bundleresult.txt')
51. modi file bundleresult.txt
```

This program creates two tables, *lexbundle* and *sentence*, the first for holding lexical bundles from *bundle.txt*, the second for storing the sentences of *lglass.txt*. There are four fields in sentence. *sent1*, *sent2*, *sent3* are for holding sentence fragments, while *sent* for holding complete sentences. Statements 10—11 initialize two variables *bundletext* and *bundlenumber* for holding lexical bundles and their sequence number. Statements 13—19 divide *lglass.txt* into sentences. Statement 28 loads the lexical bundles into *lexbundle*. Statements 30—49 create a loop, in which one by one the lexical bundles in *lexbundle* are assigned to the variable *bundle* by statement 31, and then searched for in the sent field of *sentence* and outputted to *temp* if found, done by statement 32. Note that the contents stored in the variable *bundle* and the *sent* field are converted to lower cases to make the search case-insensitive. The wild card * on either side of *bundle* is used to ensure the successful extraction of sentences containing one of the lexical bundles. Suppose *bundle* contains *it must have been*, and one of the sentences in the *sent* field of sentence is *You see, Kitty, it MUST have been either me or the red king*., a hit results because the sentence is turned into lower cases, and the wild card * on either side of bundle matches respectively *You see, Kitty, and either me or the red king*. All the sentences containing *it must have been* are put to the table *temp*. Statement 33 assigns the number of such sentences to *counter* by counting the number of records in *temp*. Statement 34 sets a condition: if the search is successful, i.e. *counter* > 0, which means *temp* is not empty, statements 35—44 are executed. Statement 35 counts the number of a lexical bundles found in the *sent* field in *sentence*. Statement 36 converts the number into a string followed by the lexical bundle in upper case plus a carriage return, all of which are assigned to *bundletext*. *carriage* ensures that each lexical bundle stored in *bundletext* is placed on a new line. At this stage *bundletext* serves as a heading, under which the sentences containing the lexical bundle are listed. Statement 37 changes the lexical bundle in the extracted sentences stored in the field *sent* of *temp* into upper cases and mark them on either side of the bundle with ** for easy viewing. Statements 38—43 are a loop, in which the sentences in *sent* of *temp* containing the extracted lexical bundle are assigned to *bundlesent* one by one, indented on the left with a tab. Statement 46 accesses the table *lexbundle*, and statement 47 inserts the number of occurrences of the lexical bundle in the field *freq*. Statement 48 moves the record pointer to the next lexical bundle and the program moves back to statement 30 and repeats the above process, until the end of *lexbundle* is reached. However, if the condition set by statement 34 is not met, i.e. *counter* = 0, which means no sentences containing the lexical bundle are found, the program goes directly to statement 46, accessing *lexbundle*, moving the record pointer one step forward, going back to statement 31 and getting the next lexical bundle and starting a new round of searching.

Statement 50 outputs the contents of *bundletext* to *bundleresult.txt*, which looks as follows:

```
  (1). IT MUST HAVE BEEN
    1. alice looked up at the rocking-horse-fly with great interest, and
made up her mind that ** IT MUST HAVE BEEN ** just repainted, it
looked so bright and sticky; and then she went on.
    2. but it looked a little ashamed of itself, so i think ** IT MUST
HAVE BEEN ** the red queen.
    3. you see, kitty, ** IT MUST HAVE BEEN ** either me or the red
king.
  (2). I WAS IN THE
    1. `so i shall be as warm here as ** I WAS IN THE ** old room,'
thought alice: `warmer, in fact, because there'll be no one here to scold
me away from the fire.
  (3). THE END OF THE
    1. a sudden thought struck her, and she took hold of ** THE END
OF THE ** pencil, which came some way over his shoulder, and began
writing for him.
    2. `i'll see you safe to ** THE END OF THE ** wood -- and then i
must go back, you know.
    3. i'll go with you to ** THE END OF THE ** wood --
    4. they had just come to ** THE END OF THE ** wood.
  (4). THE BACK OF THE
    1. `i suppose they've each got "tweedle" round at ** THE BACK OF
THE ** collar,' she said to herself.
  (5). THE MIDDLE OF THE
    1. ** THE MIDDLE OF THE ** night.
  (6). THE OTHER SIDE OF
    1. she very soon came to an open field, with a wood on ** THE
OTHER SIDE OF ** it:
    2. and was that really - was it really a sheep that was sitting on **
THE OTHER SIDE OF ** the counter?
  …
```

3.5.2 Collocational association of *run*

Collocation is very important in natural language processing, corpus linguistics and language teaching and research. There are statistical tests for collocational associations, the *t* test and the chi-square test, and measures such as the likelihood ratio and mutual information. We'll write a program to get a complete concordance of the word *run* in *alice.txt*. The concordance will be arranged in the KWIC (Key Word In Context) format with a four-word context on either side of the key word, like the following:

hat she had to	RUN back into the wood
you doing out here?	RUN home this moment, and
dear, certainly: but now	RUN in to your tea;
keep herself from being	RUN over; and the moment

The likelihood ratios between *run* and its first right collocates are then computed to check for significant collocational associations. To compute the likelihood ratio, the following data are needed:
c_1: the frequency of the key word
c_2: the frequency of the first right collocate of the key word
c_{12}: the frequency of the key word occurring with its first right collocate
n: size of text or corpus
p: c_2/n
p_1: c_{12}/c_1
p_2: $(c_2-c_{12})/(n-c_1)$
The likelihood ratio $\log\lambda$ is obtained with

$$\log\lambda = \log\frac{b(c_{12},c_1,p)b(c_2-c_{12},n-c_1,p)}{b(c_{12,},c_1,p_1)b(c_2-c_{12},n-c_1,p_2)}$$
$$= \log b(c_{12}, c_1, p) + \log b(c_2\text{-}c_{12}, n-c_1, p)$$
$$-\log b(c_{12}, c_1, p_1) - \log b(c_2-c_{12}, n-c_1, p_2)$$

where b stands for binomial distribution, $b(k, n, x) = x^k(1\text{-}x)^{n\text{-}k}$. Therefore,

$$\log\lambda = \log(p^{c_{12}}(1-p)^{(c_1-c_{12})}) + \log(p^{(c_2-c_{12})}(1-p)^{(n-c_1)-(c_2-c_{12})})$$
$$-\log(p_1{}^{c_{12}}(1-p_1)^{(c_1-c_{12})}) - \log(p_2{}^{(c_2-c_{12})}(1-p_2)^{(n-c_1)-(c_2-c_{12})})$$

which can be rewritten as

$$\log\lambda = \log(p)c_{12} + \log(1-p)(c_1-c_{12}) + \log(p)(c_2-c_{12})$$
$$+\log(1-p)((n-c_1)-(c_2-c_{12})) - \log(p_1)c_{12} - \log(1-p_1)(c_1-c_{12})$$
$$-\log(p2)(c_2-c_{12}) - \log(1-p_2)((n-c_1)-(c_2-c_{12})).$$

$\log\lambda$ is then multiplied with -2 since $-2\log\lambda$ is χ^2 distributed. In the χ^2 distribution table, the significance level of $\alpha = 0.05$ is 3.84 for one degree of freedom, so for a collocational association to be significant, $-2\log\lambda$ should be$\geq$3.84.

The program is as follows:

likelihood.prg

```
1.  set default to d:\fox\practice
2.  set safe off
```

```
set talk off
set decimal to 8
clear
create table wordtoken(word c(25),freq n(8))
create table kwictable (context c(120),freq n(5))
create table likehood (context c(25),freq1 n(4),freq2 n(4),lkhratio n(14,8))
close data
nothing=''
kwic=nothing
carriage=chr(13)
spaces=chr(32)
textinput=filetostr('d:\fox\texts\alice.txt')
textinput=strtran(textinput, '-',spaces)
textinput =strtran(textinput,spaces,carriage)
strtofile(textinput,'temp.txt')
select 1
use wordtoken
append from temp.txt sdf for word<>spaces
n=reccount()&&the total number of word tokens, needed in likelihood ratio
go top
scan for lower(alltr(word))=='run' or lower(alltr(word))=='runs' or lower(alltr(word))=='running' or lower(alltr(word))=='ran'
replace word with upper(word)
keyword=alltrim(word)
skip -4
for i=1 to 9
kwic=kwic+alltrim(word)+spaces
skip
endfor
sele 2
use kwictable
append blank
keywordposition=at(keyword, kwic)
replace context with replicate(spaces,40-keywordposition)+kwic
kwic=nothing
sele 1
endscan
sele 2
inde on righ(context,80) tag context
copy to run.txt sdf field context
copy to temp
```

```
select 3
use temp
replace all context with strtr(context,left(context,40),nothing)
replace all context with strtran(context,left (context, at(spaces,
 context)),nothing)
replace all context with left(context,at(spaces,context))
select 3
use likehood
append from temp
replace all context with chrtr(context,'.,:;”()-`*[?]_!',nothing)
replace all context with strtr(context,”"”,nothing)
replace all context with proper(context)
replace all freq2 with 1
index on context tag context
total to temp on context
zap
append from temp
select 1&&access the table wordtoken
replace all freq with 1
index on word tag word
total to temp on word
zap
append from temp
replace all word with chrtr(word,'.,:;()-[?]_`*”!',nothing)
replace all word with strtr(word,”"”,nothing)
replace all word with prop(word)
inde on word tag word
total to temp on word
zap
append from temp for word<>spaces
sum freq to c1 for alltr(word)=='Run' or alltr(word)=='Runs' or
 alltr(word)=='Running' or alltr(word)=='Ran'
select 3 &&access the table likehood
go top
do while not eof()
getword=alltr(context)
select 1
locate for alltr(word)==getword
collocatefreq=freq
select 3
replace freq1 with collocatefreq
skip
enddo
```

```
84. select 3
85. dele all for freq1=0
86. pack
87. go top
88. do while not eof()
89. c12=freq2
90. c2=freq1
91. p=c2/n
92. p1=c12/c1
93. if c2-c12=0
94. p2=(c2+0.01-c12)/(n-c1)&&0.01 is added in cases c1=c2, p2 will be 0
     and the program will crash!
95. else
96. p2=(c2-c12)/(n-c1)
97. endif
98. lkhvalue=log(p)*c12+log(1-p)*(c1-c12)+log(p)*(c2-c12)+log(1-p)*((n-
     c1)-(c2-c12))-log(p1)*c12-log(1-p1)*(c1-c12)-log(p2)*(c2-c12)-log(1-
     p2)*((n-c1)-(c2-c12))
99. repl lkhratio with lkhvalue*-2
100.   skip
101.   enddo
102.   index on lkhratio tag lkhratio descending
103.   brow
```

The program is a bit too long, but its structure is fairly simple. It can be divided into five sections, and we can use *browse* plus *cancel* to check the result of each section. If the intended result of the section is achieved, we can then remove *browse* and *cancel* and put them to other sections. The first section is between statements 1—22, for table creation and data input; the second section is between statements 23—42 for extracting *run* and its variants *runs*, *ran*, and *running* and putting them in the KWIC format, with the keyword in centre and a four-word context on either side. The third section is between statements 43—58 for getting the first right collocate of the target word and putting it into the table *likehood*. The fourth section is between statements 59—72 for calculating word frequencies. The last section is between statements 73—103, for computing the likelihood ratios of the keyword *run* with its first right collocates.

In the first section, statements 6—8 create three tables *wordtoken*, *kwictable* and *likehood*. The fields *word* and *freq* in *wordtoken* are for words from *alice.txt* and their frequencies. The fields *context* and *freq* in *kwictalbe* are for the key words with their four-word contexts on either side, and their frequencies. In the second section, statements 23—38 create a loop using the command *scan…endscan*, in which *run* and its variants are searched for in the *word* field of *wordtoken*. Within the loop, statement 24 turns the located key word into upper case, which is then assigned to the variable *keyword* in statement 25. Statements 26—30 produce the left four-word context and the right four-word context, with the key word placed in the centre. The key word and its contexts are assigned to *kwic*. Statement 34 determines how many characters away is the key word from the left of *kwic.* Statement 35 ensures that all the key words are placed 40 characters away from the leftmost of the field *context* in *kwictable*. Statements 41 and 42 respectively copy the contents of *kwictable* to a text file *run.txt* and a temporary table *temp* for further processing.

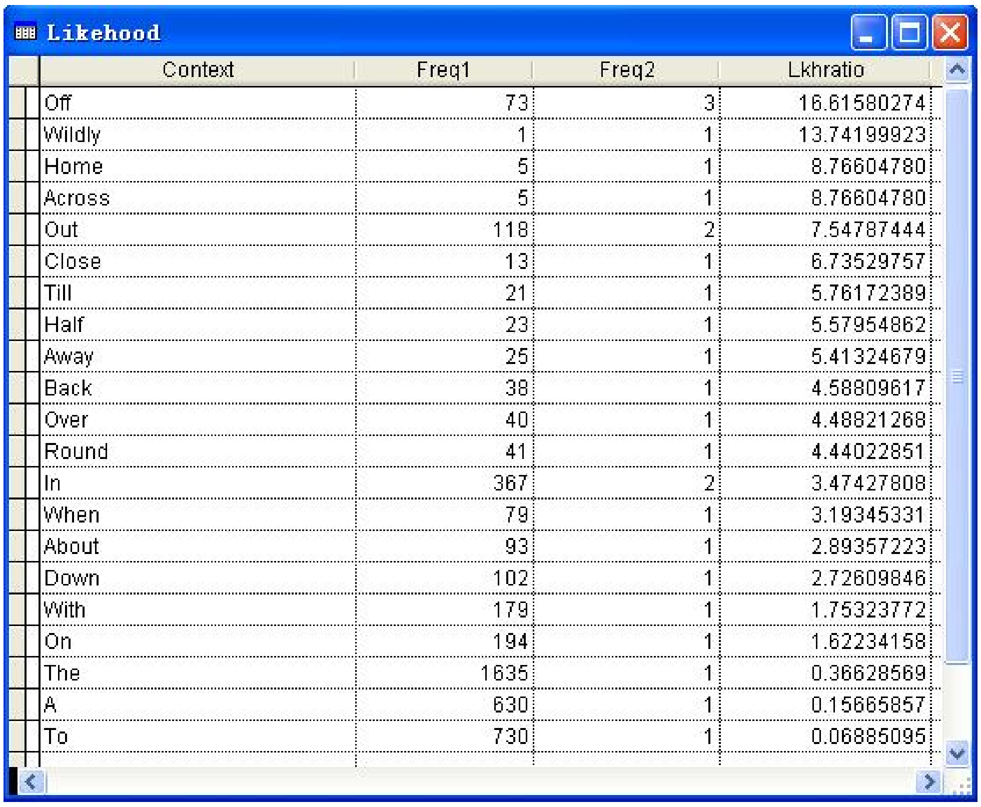
In the third section, statements 45, 46 and 47 respectively remove the left context, the key word, and get the first right collocates of the key word. Statements 50—58 append the first right collocates of the key words to the table *likehood*, remove the punctuation marks etc, and calculate their frequencies, which are actually the frequencies of the key word with its first right collocates.

The fourth section (statements 59—72) calculate word frequencies and get the total occurrences of the key words.

In the last section starting from statement 73, the first right collocates in the context field of *likehood* are taken one by one and searched for in the table *wordtoken* for their frequencies, which is then assigned to the variable *collocatefreq*. Statement 80 appends *collocatefreq* to the field *freq1* in *likehood.* Statements 87—101 compute the likelihood ratios.
The following is part of the result stored in *run.txt.*

```
     burning with curiosity , she RAN  across the field after
                just now, only it RAN  away when it saw
              Rabbit with pink eyes RAN  close by her. There
           she appeared; but she RAN  off as hard as
        much frightened that she RAN  off at once in
                 Alice got up and RAN  off, thinking while she
            through the door, she RAN  out of the house,
         trampled under its feet, RAN  round the thistle again;
        answered `Come on!' and RAN  the faster, while more
                off at once, and RAN  till she was quite
  the unfortunate gardeners, who RAN  to Alice for protection.
      King and the executioner RAN  wildly up and down
           the garden!' and she RAN  with all speed back
                that she had to  RUN  back into the wood
```

Figure 3.3 is *likehood*. From *in* upwards all the values in *lkhratio* are greater than 3.84. These words can be regarded as having significant collocational associations with *run* in *alice.txt*. Likelihood ratios are very sensitive in capturing collocational associations for collocates with low frequency. For example, *wildly* occurs only once in *alice.txt*, and it occurs with *run*; the likelihood ratio between *run* and *wildly* is 13.742, which can be regarded as highly significant.



Context	Freq1	Freq2	Lkhratio
Off	73	3	16.61580274
Wildly	1	1	13.74199923
Home	5	1	8.76604780
Across	5	1	8.76604780
Out	118	2	7.54787444
Close	13	1	6.73529757
Till	21	1	5.76172389
Half	23	1	5.57954862
Away	25	1	5.41324679
Back	38	1	4.58809617
Over	40	1	4.48821268
Round	41	1	4.44022851
In	367	2	3.47427808
When	79	1	3.19345331
About	93	1	2.89357223
Down	102	1	2.72609846
With	179	1	1.75323772
On	194	1	1.62234158
The	1635	1	0.36628569
A	630	1	0.15665857
To	730	1	0.06885095

Figure 3.3 Likelihood ratios of the collocates of *run* and its variants

3.5.3 Computing mean letter utility

According to Altmann, a letter has a set of properties, such as graphemic load, phonemic load, frequency, letter utility, etc. Letter utility refers to the occurrences of a letter in different positions of graphemes formed with it. It's computed with the following:

$$PP_{<x>} = \sum_{x \in n_{<X>}} w_x \,,$$

where $PP_{<x>}$ is the letter utility of a letter, and W_x is its occurrences in different positions in graphemes. The mean letter utility is computed with the following:

$$\overline{PP}_{<x>} = \frac{1}{|n_{<x>}|} \sum_{x \in n_{<x>}} w_x \,,$$

where $n_{<x>}$ is the number of different graphemes a letter occurs in. For example, the letter *q* occurs in the following five graphemes representing the English phoneme /*k*/:

cq, cqu, q, qu, que. $PP_{<q>} = 2 + 2 + 1 + 1 + 1 = 7$; $\overline{PP}_{<q>} = 7/5 = 1.4$.

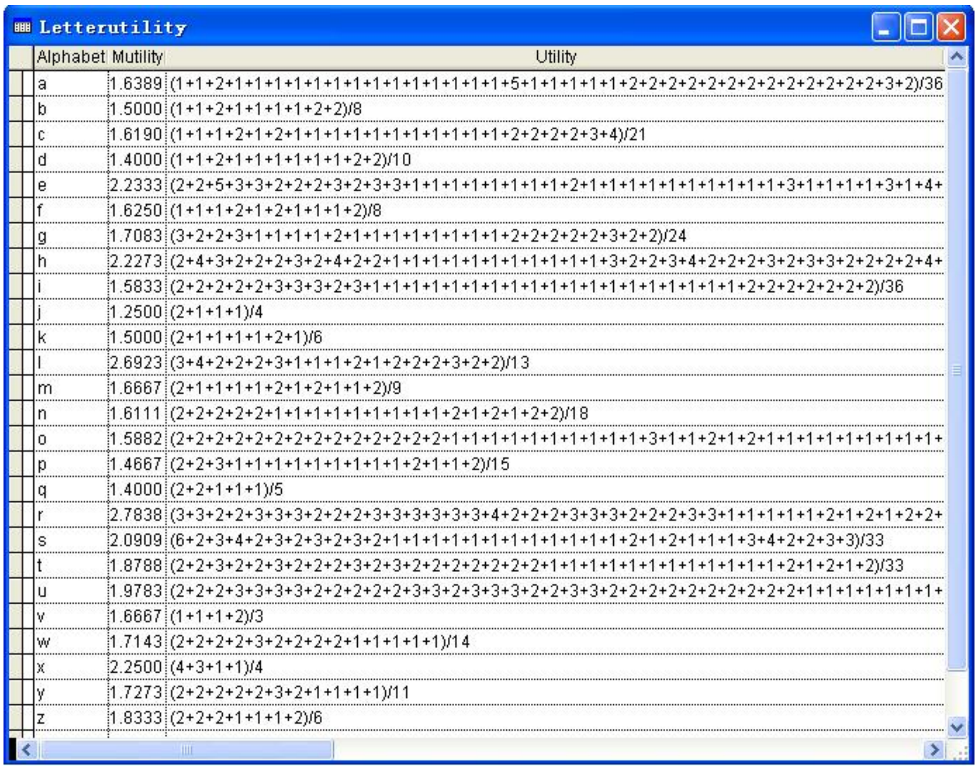
In the English language, mean letter utility can measure the relevance of a letter in graphemes since the earlier a letter appears in a grapheme the more it contributes to its phonetic value. In *d:\fox\table3* there is a table *graphemetable* containing 271 different graphemes extracted from the one million word Brown Corpus. The following is a program for computing the mean letter utility of the 26 English letters.

letterutility.prg

```
set defa to d:\fox\practice
clos data
set safe off
nothing=''
addpositions=nothing
countgrapheme=0
create table letterutility(alphabet c(2),mutility n(6,4),utility c(250))
for i=97 to 122
append blank
replace alphabet with chr(i)
endfor
select 2
use d:\fox\table3\graphemetable
sele 1
go top
do while not eof()
letter=alltrim(alphabet)
sele 2
scan for letter$grapheme
countgrapheme=countgrapheme+1
if occurs(letter,grapheme)>1
positions=alltrim(str(at(letter,alltrim(grapheme))))+'+'+alltrim(str(rat(letter,alltrim(grapheme)))) &&measure the two positions of letter in grapheme
else
positions=alltrim(str(at(letter,alltrim(grapheme))))
endif
addpositions=addpositions+positions+'+'
endscan
addpositions=left(addpositions,rat('+',addpositions)-1) &&remove the trailing +
meanutility='('+addpositions+')'+'/'+alltrim(str(countgrapheme))
```

```
30. select 1
31. replace utility with meanutility
32. replace mutility with &meanutility
33. addpositions=nothing&&empty it for the next round
34. countgrapheme=0
35. skip
36. enddo
37. brow
```

In this program statements 5—6 initialize *addpositions* and *countgrapheme*. The former holds the positions of the target letter in graphemes formed with it, with a + sign after each position number; the latter the number of graphemes the letter occurs in. Statement 7 creates *letterutility* with three fields: *alphabet* for the 26 letters, *mutility* for mean letter utility, and *utility* for the contents stored in *addpositions*. Statements 8—11 load the 26 letters. Statements 19—27 search in the grapheme field of *graphemetable* for the target letter, measure its positions in graphemes containing it, and store its position numbers separated by a plus sign in *addpositions*. Statement 29 puts *addpositions* in brackets followed by a division sign /. Statement 32 converts the contents of *addpositions* into math operation using the macro operator & and puts the result in *mutility*. Figure 3.4 is part of *letterutility*.


Letterutility

Alphabet	Mutility	Utility
a	1.6389	(1+1+2+1+1+1+1+1+1+1+1+1+1+1+1+1+5+1+1+1+1+1+2+2+2+2+2+2+2+2+2+2+2+2+3+2)/36
b	1.5000	(1+1+2+1+1+1+1+2+2)/8
c	1.6190	(1+1+1+2+1+2+1+1+1+1+1+1+1+1+1+1+2+2+2+2+3+4)/21
d	1.4000	(1+1+2+1+1+1+1+1+1+2+2)/10
e	2.2333	(2+2+5+3+3+2+2+2+3+2+3+3+1+1+1+1+1+1+1+1+2+1+1+1+1+1+1+1+1+1+3+1+1+1+1+3+1+4+
f	1.6250	(1+1+1+2+1+2+1+1+1+2)/8
g	1.7083	(3+2+2+3+1+1+1+1+2+1+1+1+1+1+1+1+1+2+2+2+2+2+3+2+2)/24
h	2.2273	(2+4+3+2+2+2+3+2+4+2+2+1+1+1+1+1+1+1+1+1+1+1+3+2+2+3+4+2+2+2+3+2+3+3+2+2+2+2+4+
i	1.5833	(2+2+2+2+2+3+3+3+2+3+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+2+2+2+2+2+2+2)/36
j	1.2500	(2+1+1+1)/4
k	1.5000	(2+1+1+1+1+2+1)/6
l	2.6923	(3+4+2+2+2+3+1+1+1+2+1+2+2+2+3+2+2)/13
m	1.6667	(2+1+1+1+1+2+1+2+1+1+2)/9
n	1.6111	(2+2+2+2+2+1+1+1+1+1+1+1+1+1+2+1+2+1+2+2)/18
o	1.5882	(2+2+2+2+2+2+2+2+2+2+2+2+2+2+1+1+1+1+1+1+1+1+1+3+1+1+2+1+2+1+1+1+1+1+1+1+1+1+
p	1.4667	(2+2+3+1+1+1+1+1+1+1+1+1+2+1+1+2)/15
q	1.4000	(2+2+1+1+1)/5
r	2.7838	(3+3+2+2+3+3+3+2+2+2+3+3+3+3+3+3+4+2+2+2+3+3+3+2+2+2+3+3+1+1+1+1+1+2+1+2+1+2+2+
s	2.0909	(6+2+3+4+2+3+2+3+2+3+2+1+1+1+1+1+1+1+1+1+1+1+1+2+1+2+1+1+1+3+4+2+2+3+3)/33
t	1.8788	(2+2+3+2+2+3+2+2+2+3+2+3+2+2+2+2+2+2+2+1+1+1+1+1+1+1+1+1+1+2+1+2+1+2)/33
u	1.9783	(2+2+2+3+3+3+3+2+2+2+2+2+3+3+2+3+3+3+2+2+3+3+2+2+2+2+2+2+2+2+2+2+1+1+1+1+1+1+1+
v	1.6667	(1+1+1+2)/3
w	1.7143	(2+2+2+2+3+2+2+2+2+1+1+1+1+1)/14
x	2.2500	(4+3+1+1)/4
y	1.7273	(2+2+2+2+2+3+2+1+1+1+1)/11
z	1.8333	(2+2+2+1+1+1+2)/6

Figure 3.4 Part of *letterutility*

Exercises

1. The table *80vgrowth* in *d:\fox\table3* contains the vocabulary growth data of 80 sets of samples from the BNC written text section, computed at a 2000-word interval. Each set has 500 2000-word samples totalling 1,000,000 words, randomly drawn without replacement. Copy the table to *d:\fox\practice\80vgrowth* and write a program to compute the mean vocabulary growth of the 80 sets as the number of samples increases, the standard deviations of the vocabulary growth of the 80 sets, and the 95% confidence intervals of the vocabulary growth for the 80 sets. The standard deviation is obtained with:

$$sdv = \sqrt{\frac{\Sigma(x-\bar{x})^2}{N}}$$

and the 95% confidence interval is obtained with $\bar{x} \pm 1.96 \cdot sdv$. You should add four more fields to the newly copied table to hold the mean vocabulary growth, the standard deviation, and the upper and lower bounds of the 95% confidence interval.

2. In Exercise 9 of Chapter 1 we wrote a program called *arclength.prg*, in which we computed the arch length of a set of 20 imagined word rank-frequencies. That way of computing is very time-consuming, error prone and almost impossible for larger set of data. Now rewrite the program and compute the arch length within a table.

3. According to Quirk et al, semi-auxiliaries are verb idioms which express modal or aspectual meaning and which are introduced by one of the primary verbs HAVE and BE. The following verb idioms are semi-auxiliaries: *be able to, be about to, be apt to, be bound to, be due to, be going to, be likely to, be meant to, be obliged to, be supposed to, be willing to, have to*. These semi-auxiliaries are in *d:\fox\texts\semiaux.txt*. Write a program to extract the sentences that contain one of the semi-auxiliaries in *alice.txt* and *lglass.txt*. Mark the modal auxiliaries with two asterisks on either side and capitalize all the letters of the modal auxiliaries.

4. Write a program to pick out the sentences in *alice.txt* and *lglass.txt* that contain the phrases *more…than* or *more than*.

5. Copy *wordlist* in *d:\fox\table3* to *d:\fox\practice\test* and use *test* to do the following by entering statements in the command window:
copy all the words ending in *ship*, *hood*, *dom* and *craft* to a new table;
centre-justify all the words in the word field of *test*, which is 25 characters in length;

right-justify all the words;
left-justify all the words.

6. Modify *likelihood.prg* so that it can produce a concordance for *get* and its variants in *lglass.txt* with a 5-word context on either side and compute the likelihood ratios between *get* and its variants and their first right collocates.

7. The *t*-test can be used for testing collocational associations between two words. *t* is obtained with the following:

$$t = \frac{\bar{x} - \mu}{\sqrt{\frac{s^2}{N}}}.$$

N: size of text or corpus
$\bar{x}$: frequency of the key word with its first right collocate divided by *N*
μ: (frequency of the key word/N)×(frequency of the first right collocate)/N)
s^2: frequency of the key word with its first right collocates/N; $s^2 = \bar{x}$
N: size of corpus

The *t*-test is often used for ranking collocations rather than checking for level of significance. Now modify *likelihood.prg* so that it can get a concordance of *make* and its variants in *lglass.txt* and compute the *t* values between the key word and its first right collocates and then rank them in descending order.

8. Modify *bigram.prg* so that it can produce trigrams from *alice.txt*.

9. Modify *hapax.prg* so that it can do the following using *multitext* (in *d:\fox\practice*) created in 2.5.2.:
get the vocabulary size, number of hapaxes and dis legomena in each of the 48 text chunks and compute their standard deviation, mean and their maximum and minimum number;
compute the ratio between the number of dis legomena and hapaxes in each of the texts, the mean ratio, maximum ratio and minimum ratio;
compute the mean word length of the hapaxes and dis legomena of each text and their average mean length, the maximum mean length and minimum mean length.

10. In EFL (English as a Foreign Language) teaching, EFL course designers often have to estimate the lexical coverage of the list of words to be taught to the learner. That is, what percentage of the word tokens of texts the learner is supposed to read after completing the course the intended set of vocabulary can cover. Assuming the word types in *wordlistb* (in *d:\fox\table3*) are the set of

vocabulary a student of English should acquire, write a program to compute the average number of word tokens of the 48 texts in *d:\fox\texts*, *wordlistb*'s lexical coverage over each of the 48 texts, its average coverage, maximum coverage and minimum coverage, and the standard deviation.

4 String Manipulation in Tables and Texts

One of the main tasks in linguistic and literary computing is string handling. For example, if we want to study the distribution of parts of speech in a tagged corpus, we must first remove the words attached to their POS tags before dealing with the POS tags. To make a wordlist for a tagged corpus, it's just the other way round. Foxpro is equipped with many commands and functions for string manipulation, some of which we have already learned in previous chapters. In this chapter we'll look at some more commands and functions that can be used in string manipulation.

4.1 Commands and Functions

asc(*string*) This function gets the ASCII code of the first character of a string. Type:

```
?asc('A') ↵
65

?asc('a') ↵
97

?asc('apple') ↵
97
```

val(*string*) This function turns a number character in a string such as *45A*, *6.78C* etc into a number, discarding the following letters. If the first character is not a number character, then this function returns zero, but if the first character is a minus or plus sign followed by number characters and letters, the function returns the sign as well as the number. Type:

```
?val('678.66A') ↵
678.66

?val('-34c') ↵
-34

?val('CD35') ↵
0.00
```

In *d:\fox\table3* there is a table called *files* containing 406 file names from *1.txt* to *406.txt*. Since these numbers are actually characters so they are arranged by the computer in the following order: *1.txt*, *10.txt*, *100.txt*, *101.txt*, *102.txt* and so on. We can use the *val(string)* function to rearrange them in the order of *1.txt*, *2.txt*, *3.txt* and so on. Type:

```
use d:\fox\table3\files ↵
index on val(filename) tag filename ↵
brow ↵
```

The order of the file names are re-arranged as desired.

isblank(*string*) This function checks whether *string* is a blank. Type:

```
?isblank('     a') ↵
.F.

isblank('         ') ↵
.T.
```

empty(*string*) This function is the same as *isblank(string)* except in evaluating formatting characters such as *chr(9)*, *chr(10)* and *chr(13)*. Type:

```
a = ' ' ↵
?empty(a) ↵
.T.

isblank(a) ↵
.T.

a = chr(13) ↵
?empty(a) ↵
.T.

?isblank(a) ↵
.F.
```

isdigit(*string*) This function checks whether the first character of a string is a number. Type:

```
a='45bc' ↵
?isdigit(a) ↵
.T.
```

```
a='apple' ↵
?isdigit(a) ↵
.F.
```

isalpha(*string*) This function checks whether the first character of *string* is an alphabetic character. If the string checked is alphabetic, the return value is *.T.*, otherwise it's *.F.* The following program checks which of the 256 ASCII codes are alphabetic characters. For some computers the characters whose ASCII code is larger than 126 are unprintable.

isalpha.prg

```
set defa to d:\fox\practice
close data
set safe off
clear
create table alphabet(chrcode c(10),characters c(5),ischar c(5))
for i=0 to 255
append blank
codes='chr('+alltr(str(i))+')'
replace chrcode with codes
replace characters with chr(i)
if isalph(chr(i))=.f.
replace ischar with '.F.'
else
replace ischar with '.T.'
endif
endfor
brow
```

isupper(*string*) This function checks whether the first character of *string* is in upper case. Type:

```
?isupper('Foxpro') ↵
.T.

?isupper('foxpro') ↵
.F.
```

islower(*string*) This function checks whether the first character of *string* is in lower case. Type:

```
islower('Apple') ↵
.F.
```

```
islower('apple') ↵
.T.
```

space(*n*) This function produces *n* number of spaces. Type:

```
? 'Fox'+space(16)+ 'pro' ↵
Fox                pro
```

padl(*string*,*n*,*character*) This function pads *string* on the left with a number of *character*. The number is determined by *n* minus the length of *string*. This function is often used for right justification. Type:

```
?padl('word',18, '*') ↵
**************word

?padl('word',18, ' ') ↵
              word

?padl('frequency',18, '*') ↵
*********frequency

?padl('frequency',18, ' ')
         frequency
```

padr(*stirng*,*n*,*character*) This function pads *string* on the right with a number of *character*. The number is determined by *n* minus the length of *string*. This function is often used for left justification.

```
?padr('word',18, '*') ↵
word**************

?padr('frequency',18, '*') ↵
frequency*********
```

padc(*string*,*n*,*character*) This function centre-justifies *string* by putting a number of *character* on either side of *string*, and the total number of characters is *n* minus the length of *string*.

```
?padc('word',20, '*') ↵
********word********

?padc('wordlist',20, '*') ↵
```

```
******wordlist******
```

rtrim(*string*) This function removes the trailing blanks of *string*.

```
w1='Fox        '↵
w2='pro' ↵
?w1+w2 ↵
Fox        pro

?rtrim(w1)+w2 ↵
Foxpro
```

ltrim(*string*) This function removes the preceding blanks of *string*.

```
w1='fox' ↵
w2='        pro' ↵
?w1+w2 ↵
Fox        pro

?w1+ltrim(w2) ↵
Foxpro
```

evaluate(*string*) This function turns a numeric character expression into numeric expressions and returns the result of the numeric expression.

```
a='2**3+4/3' ↵
?a ↵
2**3+4/3

?evaluate(a) ↵
9.33
```

text…endtext This command outputs to the screen lines of text between *text* and *endtext*. Type the following in the command window. Use the down key to move to a new line. Highlight the newly entered statements by dragging the mouse from the left of *text* to the end of *endtext* and then press Enter.

```
text
This is a demonstration of the use of text…endtext
endtext
```

The text *This is a demonstration of the use of text…endtext* is outputted to the screen.

substr(*string*,*n1*,*n2*) This function cuts a chunk from *string* from position *n1* to position *n2*.

```
?substr('handsome',1,4) ↵
hand

?substr('handsome',5,4) ↵
some
```

stuff(*string*,*n1*,*n2*,*character*) This function replaces *string* with *character* from position *n1* to position *n2*.

```
?stuff('handsome',1,4, 'two') ↵
twosome

?stuff('handsome',1,4, '') ↵
some
```

isleadbyte(*string*) In the computer machine codes, a single character such as *a*, *c*, *f* etc in languages such as English consists of a single byte, while that of some other languages such as Chinese consists of two bytes. This function checks whether the first character of *string* is a double-bye character. If it is, the return value is *.T.*, otherwise it is *.F*. For example, *isleadbyte('我')* yields *.T.* because 我 is a Chinese character meaning *I*; it consists of two bytes. While *isleadbyte('I')* results in *.F.*

strconv(*string*,*n*) This function converts *string* into different types of character specified by *n*. The following table shows the value of *n* and the types of character conversion it makes.

Table 4.1 Codes for string conversion

N	Types of Character Converted
1	converts single-byte characters in *string* to double-byte characters.
2	converts double-byte characters in *string* to single-byte characters.
3	converts double-byte Katakana characters in *string* to double-byte Hiragana characters.

4	converts double-byte Hiragana characters in *string* to double-byte Katakana characters
5	converts double-byte characters in *string* to UNICODE .
6	converts UNICODE in *string* to double-byte characters.
7	converts *string* to locale-specific lowercase.
8	converts *string* to locale-specific uppercase.
9	converts double-byte characters in *string* to UTF-8
10	converts UNICODE characters in *string* to UTF-8
11	converts UTF-8 characters in *string* to double-byte characters.
12	converts UTF-8 characters in *string* to UNICODE characters.
13	converts single-byte characters in *string* to encoded base64 binary.
14	converts single-byte characters in *string* to decoded base64 binary
15	converts single-byte characters in *string* to encoded hexBinary.
16	converts single-byte characters in *string* to decoded hexBinary.

Enter the following in the command window:

```
  word='eat' ↵
  ?len(word) ↵
  3

  word=strconv(word,1) ↵
  ?len(word) ↵
  6

  ?word ↵
   ｅａｔ

  isleadbyte(word) ↵
.T.

word=strconv(word,2) ↵
?len(word) ↵
3

?word ↵
Eat

isleadbyte(word) ↵
```

.F.

4.2 Low-level File Functions

The following are functions for low-level file handling; that is, these functions deals with files at the machine code level.

fcreat(*filename* [, *fileattributecode*]) This function creates a text file and returns a file handle number to the file. We can assign the file handle number to a variable so that the file can be accessed with the variable. If the file creation fails, the return value is -1. If the file already exists, it'll be overwritten without warning. *fileattributecode* specifies the file access attributes, which are listed below:

Code	Access attributes
0	(Default) Read/write
1	Read-only
2	Hidden

If we want to create a file with the read/write file access attribute, we can omit 0, using only *fcreat(filename*). Now type:

```
newfile=fcreate('test1.txt') ↵
?newfile ↵
6

newfile=fcreate('test2.txt') ↵
?newfile ↵
7

newfile=fcreate('test1.txt') ↵
?newfile ↵
-1
```

The second *newfile=fcreate('test1.txt')* returns -1 because *test.txt* created by the first *newfile=fcreate('test1.txt')* is now open.

fopen(*filename* [, *fileattributecode*]) This function opens a file with specified file access attributes. It returns a file handle number to the file opened. If the function successfully opens the file, it returns a positive number, otherwise it returns -1. The file handle number can be assigned to a variable so that the file can be accessed with the variable. The file attribute codes are as follows:

code	file access attribute
0	(Default) Read-only

1	Write-only
2	Read and Write

The default file access attribute is read-only, and 0 can be omitted.

```
opfile=fopen('d:\fox\texts\alice.txt') ↵
?opfile ↵
6

opfile=fopen('d:\fox\texts\lglass.txt') ↵
?opfile ↵
7

opfile=fopen('d:\fox\texts\alice.txt') ↵
?opfile ↵
-1

close all ↵
opfile=fopen('d:\fox\texts\alice.txt') ↵
?opfile ↵
6
```

fseek(*filehandlenumber,bytesmoved* [, *position*]) This function moves the file pointer within a file created or opened with *fcreat()* or *fopen()*. *filehandlenumber* is the file handle number returned by *fcreat()* or *fopen()*. *bytesmoved* specifies the distance measured in bytes the file pointer is moved from *position*. *position* specifies the position of the file pointer. The following lists the file pointer position codes:

Position code	destination
0	(Default) the beginning of the file.
1	The current file pointer position.
2	The end of the file.

To move the file pointer to the top, type:

```
close all ↵
opfile=fopen('d:\fox\texts\alice.txt') ↵
fpointer=fseek(opfile,0,0) ↵
?fpointer ↵
0
```

The second statement assigns the file handle number to *opfile*; *fseek(opfile,0,0)* moves the file pointer to the top of the file, which returns 0, meaning the file pointer is 0 byte away from the top of the file. Now enter the following:

```
fpointer=fseek(opfile,100,0) ↵
?fpointer ↵
100
```

This means the file pointer is moved 100 bytes forward from the beginning of the file.

```
?fpointer=fseek(opfile,50,1) ↵
?fpointer ↵
150
```

fseek(opfile,50,1) moves the file pointer 50 bytes forward from the current file pointer position, which is 100 bytes from the top, so the result is 150.

```
fpointer=fseek(opfile,0,2) ↵
?fpointer ↵
151707
```

This moves the file pointer all the way to the bottom of the file and it’s 151,707 bytes from the top of the file.

```
fpointer=fseek(opfile,60,0) ↵
?fpointer↵
60

fpointer=fseek(opfile,75,1) ↵
?fpointer ↵
135

fpointer=fseek(opfile,45,1) ↵
?fpointer ↵
180
```

fseek(opfile,60,0) moves the file pointer 60 bytes downwards from the top, while *fseek(opfile,75,1)* moves the file pointer 75 bytes downwards from the current position, which is 60 bytes from the top; *fseek(45,1)* moves the file pointer 45 bytes downwards from the current position, which is now 135 bytes from the top, putting the file pointer 180 bytes away from the top of the file. If *bytesmoved* is negative, the file pointer is moved backwards, i.e. towards the top of a file. Now the file pointer is 180 bytes away from the top of the file.

```
fpointer=fseek(opfile,-160,1) ↵
```

```
?fpointer ↵
20
```

The first statement moves the file pointer 160 bytes backwards from the current file pointer position, which is 180, and the distance between the file pointer and the top of the file is now 20.

fgests(*filehandlenumber* [, *numberofbytes*])   This function gets a string from a file created with *fcreate()* or opened with *fopen()*. *filehandlenumber* is the file handle number and *numberofbytes* specifies the length of the string measured in number of bytes this function can get starting from the current position of the file pointer. The maximum number of bytes this function can return is 8,192. However, it stops when a carriage return is encountered even if the number of bytes of the string is less than the specified number. If *numberofbytes* is omitted, the number of bytes this function returns is 256. After the action of *fgets()* is completed, the file pointer is placed at the place where it stops, or right behind the carriage return when one is encountered. Suppose we want to get 120 bytes of string from the place 9 bytes from the top of *alice.txt*, type:

```
close all ↵
fhandle=fopen('d:\fox\texts\alice.txt') ↵
fseek(fhandle,9,0) ↵
getstring=fgets(fhandle,120) ↵
?getstring↵
ALICE'S ADVENTURES IN WONDERLAND
```

The string is only 32 bytes long because it's followed by a carriage return so the function stops there.

fread(*filehandle*, *numberofbytes*)   Like *fgets()*, this function gets data from a file created with *fcreat()* or opened with *fopen()*. The difference is that the maximum number of characters it can get from the file is 65,535, and it doesn't stop at carriage returns. After the action of *fread()* is completed, the file pointer is positioned at the position 65,535 bytes from its starting position. Now type:

```
close all ↵
fhandle=fopen('d:\fox\texts\alice.txt') ↵
fseek(fhandle,9,0) ↵
getstring=fread(fhandle,120) ↵
?getstring↵
ALICE'S ADVENTURES IN WONDERLAND
```

CHAPTER 1

fputs(*filehandlenumber, string* [, *numberofcharacters*]) This function inputs *string* to a file created with *freat()* or opened with *fopen()*. *filehandlenumber* is the file handle number and *string* is the data to be inputted to the file. *numberofcharacters* specifies how many characters of *string* are to be inputted to the file, but this option is seldom used in actual practice. Now type:

```
close all ↵
fhandle=fcreate('test.txt') ↵
fputs(fhandle,'This is the fputs() function') ↵
fputs(fhandle,'It inputs strings with carriage returns.') ↵
close all ↵
modi file test.txt ↵
```
This is the fputs( ) function.
It inputs strings with carriage returns.

fwrite(*filehandlenumber, string* [, *numberofcharacters*]) Like the *fputs()* function, this function inputs *string* to a file created with *freat()* or opened with *fopen()*. *filehandlenumber* is the file handle number and *string* is the data to be written into the file. *numberofcharacters* specifies how many characters of *string* are to be written into the file, but it’s seldom used in actual practice. The difference between *fputs()* and *fwrite()* is that the latter doesn’t put carriage returns at the end of *string*.

```
close all ↵
fhandle=fcreate('test.txt') ↵
fwrite(fhandle,'This is the fputs() function.') ↵
fwrite(fhandle,'It inputs strings with carriage returns.') ↵
close all ↵
modi file test.txt ↵
```
This is the fputs() function.It inputs strings with carriage returns.

fclose(*filehandle*) This function closes a file created with *fcreate()* or opened with *fopen()*. It can be used instead of *close all*.

```
close all ↵
fhandle=fcreate('test.txt') ↵
fwrite(fhandle,'This tests the fclose() function.') ↵
fclose(fhandle) ↵
modi file test.txt ↵
```
This tests the fclose() function.

feof(*filehandle*) This function checks whether the file pointer is at the end of a file created with *fcreate()* or opened with *fopen()*. If the file pointer is at the end of the file, the return value is *.T.*, otherwise it is *.F.*

```
close all ↵
fhandle=fopen('d:\fox\texts\alice.txt') ↵
fseek(fhandle,0) ↵
?feof(fhandle) ↵
.F.

fseek(fhandle,0,2) ↵
?feof(fhandle) ↵
.T.
```

4.3 Set Up Relations Among Tables With a Common Field

In this section, we’ll learn how to establish relations among tables with a common field so that we can access these tables at the same time for string handling, number crunching, outputting the contents of the different fields in these tables to a text file or to a table, etc. First, we’ll look at the commands used in setting up such relations.

use *tablename* **alias** *aliasname* This command opens a table and gives it an alias. Foxpro tables can be given aliases. The work areas in which tables are open have default aliases, which are *a* through *j* for work areas 1 through 10 respectively. For work areas from 11 to 32,767 the default aliases are *w*11 through *w*32767. Aliases are very useful for linking tables with a common field open in different work areas. We’ll demonstrate the use of aliases after we have looked at how to establish relations among tables with a common field.

set relation to *fieldname* **into** | [*tablename*] [*tablealiase*] | This command links a table open in a work area, the parent table, to another table *tablename*, the child table, open in another work area through their common field *fieldname*. If *tablename* has an alias, *tablealiase* can be used instead. To establish such relations, both tables must be indexed first, and once such relations are established, the fields in the child table can be accessed in the work area where the parent table is open and its record pointer moves with the record pointer of the parent table. The alias of the child table or its work area must be put before the field of the child table when being accessed from the work area of the parent table. We’ll establish relations between two tables *w1* and *w2* in *d:\fox\table3* with *w2* as the parent table. *w1* has two fields, *word* and *freq*, the former containing a set of words and the latter word frequency. *w2* has two fields as well, *word* and *wlength*, the former containing the same set of words as in the word field of *w1*, the latter word length. Now set default to *d:\fox\table3* and type:

```
selec 1 ↵
use w1 ↵
index on word tag word ↵
selec 2 ↵
use w2 ↵
index on word tag word
set relation to word into w1 ↵
```

If we want to list the first twenty words with their frequency from *w1* and word length from *w2* to the screen from *w2*'s work area, type:

```
list word, a.freq, wlength for recno()<21 ↵
```

a.freq stands for the *freq* field of *w1* in work area 1, whose alias is *a*. The following is displayed on the screen:

```
RECORD#  WORD            A->FREQ   WLENGTH
      1  A                 25897         1
      2  A.m.                  9         4
      3  Aback                 1         5
      4  Abandon              62         7
      5  Abandonment           9        11
      6  Abate                 2         5
      7  Abbey                24         5
      8  Abbot                10         5
      9  Abbreviate            2        10
     10  Abbreviation          6        12
     11  Abdicate              2         8
     12  Abdomen               1         7
     13  Abdominal             1         9
     14  Aberrant              2         8
     15  Aberration            3        10
     16  Abet                  1         4
     17  Abeyance              2         8
     20  Abhor                 1         6
     18  Abhorrence            1        10
     19  Abhorrent             2         9
```

To output the above to a table called *temp* in *d:\fox\practice,* type:

```
copy to d:\fox\practice\temp field word, a.freq, wlength ↵
```

The following demonstrates the use of table aliases in establishing relations between two tables with a common field. Don't press Enter until after all the statements have been entered in the command window and highlighted:

```
sele 1
use w1 alia tbl1
inde on word tag word
selec 2
use w2
inde on word tag word
set relation to word into tbl1
list word, tbl1.freq,wlength for recno()<21
```

The result is as follows:

```
RECORD#  WORD            TBL1->FREQ   WLENGTH
      1  A                 25897           1
      2  A.m.                  9           4
      3  Aback                 1           5
      4  Abandon              62           7
      5  Abandonment           9          11
      6  Abate                 2           5
      7  Abbey                24           5
      8  Abbot                10           5
      9  Abbreviate            2          10
     10  Abbreviation          6          12
     11  Abdicate              2           8
     12  Abdomen               1           7
     13  Abdominal             1           9
     14  Aberrant              2           8
     15  Aberration            3          10
     16  Abet                  1           4
     17  Abeyance              2           8
     20  Abhor                 1           6
     18  Abhorrence            1          10
     19  Abhorrent             2           9
```

set relation to This command breaks relations between two tables.

set skip to | [[*tablename1*] [**,** *tablename2*]…] [[*tablealias1*] [, *tablealias2*…]] | This command links the parent table with two or more child tables; these tables must have a common field. However, for this command to work, the tables must be indexed and they must be linked with each other using the *set relation* command. In *d:\fox\table3* there is a table *w3*, which has the *word* field containing the same set of words as in *w1* and *w2*, and the *rng* field containing the range of the set of words. *setskip.prg* uses *w1* as the parent table and links the three tables together through their common field *word* to output *word*, *freq* of *w1*, *wlength* of *w2* and *rng* of *w3* to a new table *temp*.

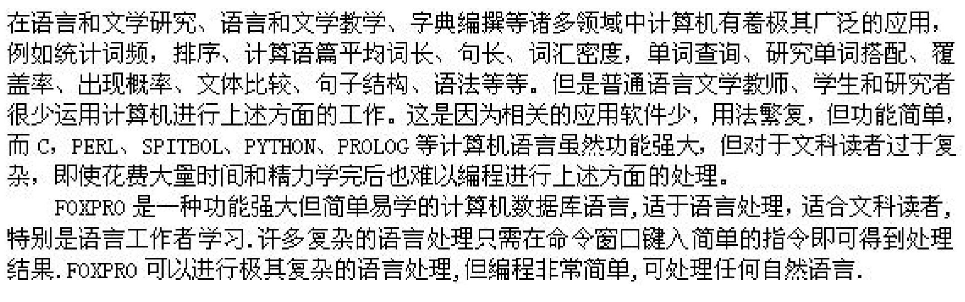
setskip.prg

```
set defa to d:\fox\practice
set safe off
close data
select 1
use d:\fox\table3\w1 alias tbl1
index on word tag word
sele 2
use d:\fox\table3\w2 alias tbl2
inde on word tag word
sele 3
use d:\fox\table3\w3 alias tbl3
inde on word tag word
select 2
set relation to word into tbl3
selec 1
set relation to word into tbl2
set skip to tbl2,tbl3
copy to temp field word,freq,tbl2.wlength,tbl3.rng
use temp
brow
```

4.4 Applications

4.4.1 Processing double-byte languages

In *d:\fox\texts* there is a short text in Chinese *chinese.txt*. It's a brief introduction to linguistic computing with Foxpro. It's as follows:



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计算机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。但是普通语言文学教师、学生和研究者很少运用计算机进行上述方面的工作。这是因为相关的应用软件少，用法繁复，但功能简单，而 C，PERL、SPITBOL、PYTHON、PROLOG 等计算机语言虽然功能强大，但对于文科读者过于复杂，即使花费大量时间和精力学完后也难以编程进行上述方面的处理。

　　FOXPRO 是一种功能强大但简单易学的计算机数据库语言，适于语言处理，适合文科读者，特别是语言工作者学习.许多复杂的语言处理只需在命令窗口键入简单的指令即可得到处理结果.FOXPRO 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.

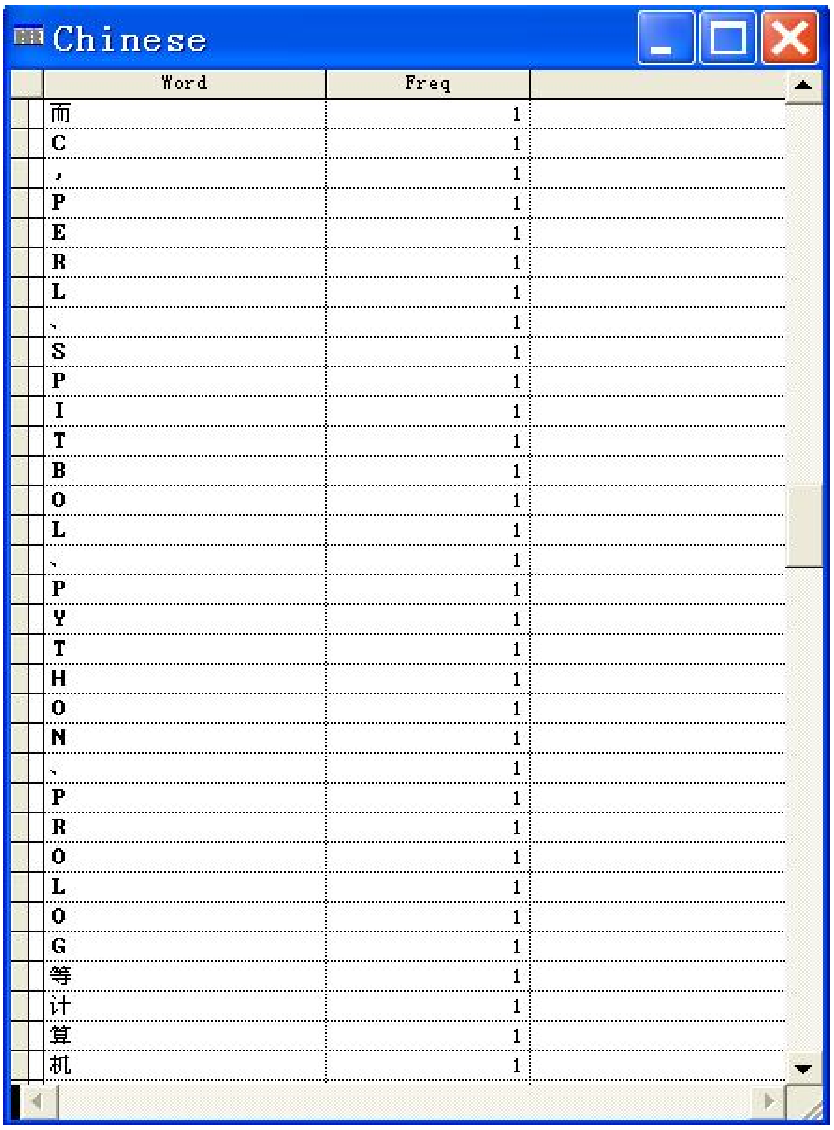
Figure 4.1 A short Chinese text

We'll write a program to tokenize this short text, turning it into individual Chinese characters, and then computing the frequency of these characters. We can't use the white space tokenizer to tokenize this text because there are no white spaces between Chinese characters in Chinese texts. One way to tokenize a Chinese text is to make continuous two-byte cuts starting from the top of a Chinese text since Chinese characters are two-byte long each. But the problem is many Chinese texts also have one-byte characters. Take the above short text as an example, there are English words such as *FOXPRO*, *PERL* and so on and a letter *C*. In addition, the punctuation marks in the first paragraph are Chinese punctuation marks, which are of double bytes, but those used in the second paragraph are the ones used in English texts, which are of single-byte. If we tokenize this text using two-byte cuts, the result would be a mess of garbled codes. If we convert all the single-byte characters in the text into double-byte characters, then we can safely use the two-byte cut tokenization. The program is as follows.

chinese1.prg

```
set defau to d:\fox\practice
set safe off
close data
creat table chinese (word c(15),freq n(5))
nothing=''
tokens=nothing
carriage=chr(13)
spaces=chr(32)
textput=filetostr('d:\fox\texts\chinese.txt')
for i=1 to 32
textinput=strtr(textput,chr(i),nothing)
endfor
textinput=strcon(textinput,1)
do while len(textinput)>0
cut=alltrim(substr(textinput,1,2))
textinput=stuff(textinput,1,2,nothing)
tokens=tokens+cut+carriage
enddo
strtofile(tokens,'temp.txt')
append from temp.txt sdf
replace all freq with 1
index on word tag word
total to temp on word
zap
append from temp FOR word<>spaces
brow
```

In this program, statements 10—12 remove unprintable characters. Statement 13 converts *textinput*, which holds the contents of the text, into a double-byte string. Statements 14—18 create a loop, in which two-byte characters are cut one at a time from *textinput* until the length of *textinput* becomes zero. Statement 15 assigns the first two bytes of *textinput* to *cut*. Statement 16 removes the two bytes that have just been assigned to *cut* from *textinput*, otherwise *cut* will be assigned the same two bytes again and again and the program will go into a dead loop. This program can handle a Chinese text of up to 250,000 words. Larger texts should be divided into smaller chunks. It can correctly handle Chinese texts mixed with single one-byte characters such as *1*, *a*, *Y* and so on. But if there are English words consisting more than one letter, these words will be broken into individual two-byte letters. The result is shown in Figure 4.2.


Chinese

Word	Freq
而	1
Ｃ	1
，	1
Ｐ	1
Ｅ	1
Ｒ	1
Ｌ	1
、	1
Ｓ	1
Ｐ	1
Ｉ	1
Ｔ	1
Ｂ	1
Ｏ	1
Ｌ	1
、	1
Ｐ	1
Ｙ	1
Ｔ	1
Ｈ	1
Ｏ	1
Ｎ	1
、	1
Ｐ	1
Ｒ	1
Ｏ	1
Ｌ	1
Ｏ	1
Ｇ	1
等	1
计	1
算	1
机	1

Figure 4.2 Part of the tokenized *chinese.txt* before totalling

Next, we’ll write a program that can properly handle Chinese texts mixed with English words formed by one-byte letters using the function *isleadbyte(string)*. The program is as follows:

chinese2.prg

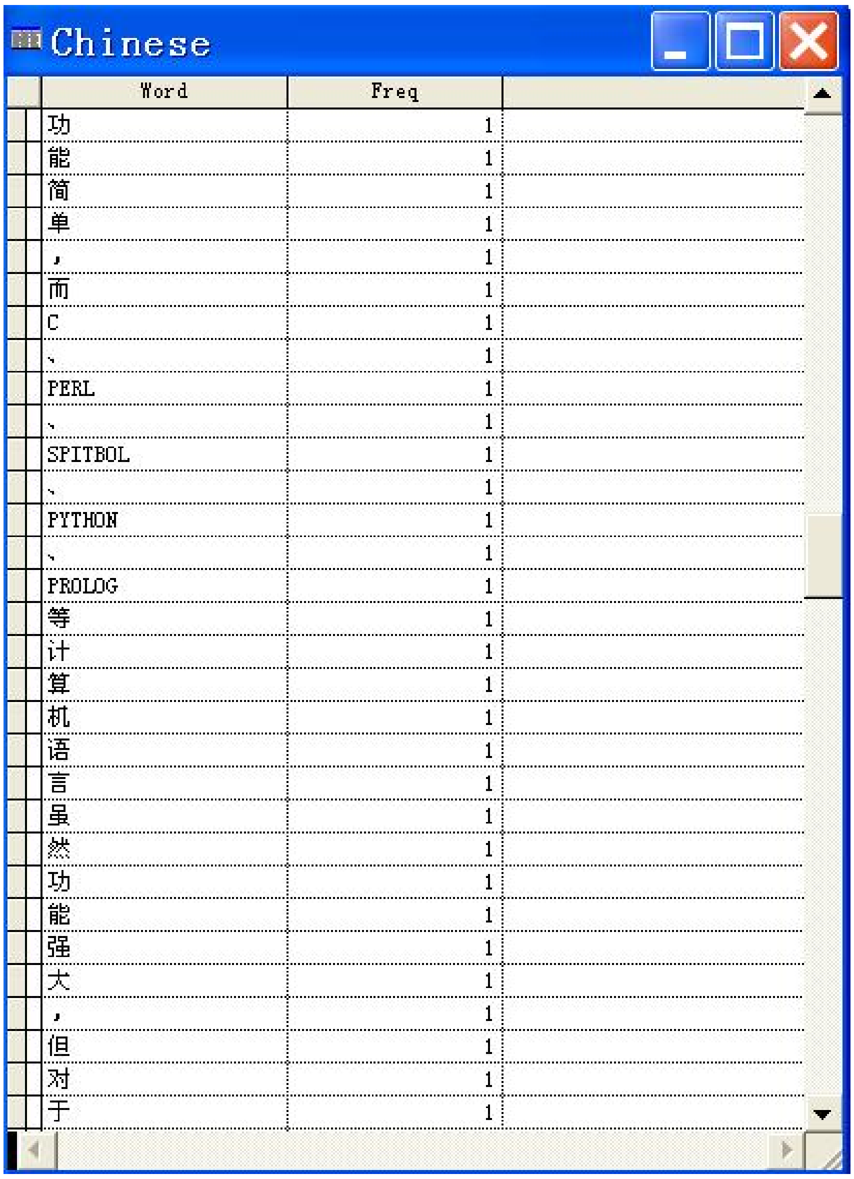
```
set defau to d:\fox\practice
set safe off
set talk on
clear
close data
creat table chinese (word c(15),freq n(5))
nothing=''
carriage=chr(13)
spaces=chr(32)
tokens=nothing
textinput=filetostr('d:\fox\texts\chinese.txt')
for i=1 to 12
textinput=strtr(textinput,chr(i),nothing)
endfor
do while len(alltr(textinput))>0
cut=substr(textinput,1,1)
byteplace=0
do while isleadbyte(cut)=.f. and len(alltr(textinput))>0
textinput=stuff(textinput,1,1,nothing)
if byteplace=0 or cut=',' or cut='.' Or cut='?' or cut='!' or cut=':' or cut='”' or cut=':'
tokens=tokens+carriage+cut
else
tokens=tokens+cut
endif
cut=substr(textinput,1,1)
byteplace=1
enddo
cut=substr(textinput,1,2)
textinput =stuff(textinput,1,2,nothing)
tokens=tokens+carriage+cut
enddo
tokens=strtr(tokens,spaces,carriage)
strtofile(tokens,'temp.txt')
append from temp.txt sdf
replace all freq with 1
index on word tag word
total to temp on word
```

```
38. zap
39. append from temp for word<>spaces
40. brow
```

This program turns a text in Chinese into a frequencied Chinese wordlist, while preserving single-byte strings intact, as shown in Figure 4.3. There is no limit to the length of texts to be processed. But to increase processing speed, long texts should be divided into shorter chunks, say, about 500,000 Chinese characters in length. Statements 12—14 remove non-printable characters. Statements 15—31 form a loop, within which single-byte and double-byte characters are cut from *textinput* and processed until *textinput* is exhausted. Tokenization takes place between statements 16—32. Statement 16 assigns one byte from *textinput* to *cut*, and statement 17 sets *byteplace* to 0. Statement 18 checks whether *cut* contains a single-byte character or a double-byte character. If it's half of a double-byte Chinese character, the program goes to statement 28, which assigns the complete double-byte Chinese character to *cut*. This double-byte character is subsequently removed from *textinput* in statement 29. *cut* is then added to *tokens* on a new line in statement 30. Then the program goes back to statement 16, and *cut* is assigned another byte and *byteplace* is again set to 0. If *cut* is a single-byte character, the program goes to statement 19, which removes this single-byte character from *textinput*. Statement 20 determines whether this character is the first character of a single-byte string (by checking the value of *byteplace*. If it's 0, it must be the first character of a single-byte string, such as *P* in *PERL*.), or one of the single-byte punctuation marks. If so, this single-byte character is added to *tokens* on a new line in statement 21; otherwise this character is placed on the same line with the previous character in statement 23. Statement 25 then assigns a new byte to *cut* from *textinput*, and *byteplace* is set to 1. If this character is still a single-byte character, the program goes to statement 19, which removes this byte from *textinput*. Then the program goes to statement 23 because the value of *byteplace* now is 1, and *cut* is placed on the same line with the previous single-byte character. If this byte is part of a double-byte character, the program goes to statement 28 to get the complete double-byte character, and statement 29 removes this character from *textinput*. Statement 30 adds this character to *tokens* on a new line. Then the program goes back to statement 16 to start another round of processing. After *textinput* is exhausted, statement 32 separates the possible strings such as “, PROLOG” in *tokens* into “,” and “PROLOG”. Statement 33 puts the contents of *tokens* to a temporary text file *temp.txt*, which is appended to the table *chinese*. Part of the result before totalling is shown in Figure 4.3.

4.4.2 Corpora handling

Corpora are now widely used in linguistics, translation, natural language processing, language teaching, etc. There are untagged corpora and tagged corpora. The first generation electronic corpora, the Brown Corpus and the LOB Corpus have both versions, while mega-corpora like the BNC have only the tagged version. In this section we'll learn how to write programs for handling corpora.



Word	Freq
功	1
能	1
简	1
单	1
，	1
而	1
C	1
、	1
PERL	1
、	1
SPITBOL	1
、	1
PYTHON	1
、	1
PROLOG	1
等	1
计	1
算	1
机	1
语	1
言	1
虽	1
然	1
功	1
能	1
强	1
大	1
，	1
但	1
对	1
于	1

Figure 4.3 Part of the tokenized *chinese.txt* before totalling obtained with *chinese2.prg*

The untagged LOB has reference codes as shown below:

```
A01      1 **[001 TEXT A01**]
A01      2 *<*'*7STOP ELECTING LIFE PEERS**'*>
A01      3 *<*4By TREVOR WILLIAMS*>
```

```
A01    4 |^A *0MOVE to stop \0Mr. Gaitskell from nominating any more
```

The characters from position 1 to 7 on the left are reference codes, respectively standing for text category, text number and line number of the text. For example, *A01 1* represents text category A, text 1, line 1. This type of reference codes is called fix field reference and was used in the first generation corpora the Brown Corpus and LOB Corpus. Now we'll write a program adding reference codes like the ones shown above to *text1.txt* in *d:\fox\texts*. We'll use *alice* as the text category, 01 as text number and 1, 2, 3… and so on as line numbers. The program is as follows:

fixfieldcode.prg

```
1.  set defa to d:\fox\practice
2.  set safe off
3.  close data
4.  nothing=''
5.  getlines=nothing
6.  carriage=chr(13)
7.  spaces=chr(32)
8.  creat table fixfieldcode (lines c(75))
9.  textinput=filetostr('d:\fox\texts\text1.txt')
10. do while len(textinput)>1
11. getlines=getlines+substr(textinput,1,at(carriage,textinput)+1)
12. textinput=stuff(textinput,1,at(carriage,textinput)+1,nothing)
13. enddo
14. strtofile(getlines,'temp.txt')
15. append from temp.txt sdf for len(alltrim(lines))>0
16. repl all lines with 'A01 '+padl((recno()),3,spaces)+spaces+lines
17. copy to fixfieldcode.txt sdf
18. modify file fixfieldcode.txt
```

In this program, statement 11cuts a line (including the carriage return at the end of the line, done by +1) one by one from *textinput* that contains text lines ending in a carriage return. Statement 12 removes the line assigned to *getlines*. Statement 16 adds reference codes to the lines, with the line numbers right justified with *padl((recn0(),3,spaces)*. Part of *text1.txt* with reference codes is shown below:

```
A01    1                 ALICE'S ADVENTURES IN WONDERLAND
A01    2                                CHAPTER I
A01    3                          Down the Rabbit-Hole
A01    4 Alice was beginning to get very tired of sitting by her sister
A01    5 on the bank, and of having nothing to do:    once or twice she had
A01    6 peeped into the book her sister was reading, but it had no
A01    7 pictures or conversations in it, `and what is the use of a book,'
```

A01 8 thought Alice `without pictures or conversation?'
A01 9 So she was considering in her own mind (as well as she could,
A01 10 for the hot day made her feel very sleepy and stupid), whether
A01 11 the pleasure of making a daisy-chain would be worth the trouble
A01 12 of getting up and picking the daisies, when suddenly a White
A01 13 Rabbit with pink eyes ran close by her.
A01 14 There was nothing so VERY remarkable in that; nor did Alice
A01 15 think it so VERY much out of the way to hear the Rabbit say to

4.4.3 Dealing with POS tags

Some corpora are tagged; that is, their words have POS tags (parts of speech tags). There are different POS tag sets. The following is a fragment of the tagged LOB:

```
B01   2 ^ editorial_NN ._.
B01   3 ^ dilemma_NN of_IN South_NP Africa_NP ._.
B01   4 ^ Prime_NPT Minister_NPT after_IN Prime_NPT
          Minister_NPT speaks_VBZ
B01   4   out_RP in_IN revulsion_NN
```

The following is a fragment of the BNC:

```
<s n="1"><w PNP>I <w VVD>began <w NN1-VVB>work <w
PRP>on <w AT0>the <w AJ0>big <w NN1>glass <w PRP>on <w
CRD>27 <w NP0>July <w CRD>1967<c PUN>, <w VVD>wrote <w
NP0-NN1>Harsnet<c PUN>.
CJC>and <w VVD>started <w TO0>to <w VVI>transcribe<c PUN>.
```

The following is a fragment of the Open ANC (Open American National Corpus, http://www.AmericanNationalCorpus.org/OANC) tagged with the XML codes:

```
<struct type="tok" from="58" to="60">
<feat name="id" value="2.5" />
<feat name="base" value="of" />
<feat name="msd" value="IO" />
    </struct>
<struct type="tok" from="61" to="68">
<feat name="id" value="2.6" />
<feat name="base" value="english" />
<feat name="msd" value="JJ" />
</struct><struct type="tok" from="69" to="79">
<feat name="id" value="2.7" />
<feat name="base" value="literature" />
<feat name="msd" value="NN1" />
    </struct>
```

Apart from corpora with POS tags, there are corpora that have syntactical codes for phrases, clauses and sentences. The ICE Corpus (the International Corpus of English) is an example. The following is a fragment of the ICE-GB Corpus (the British English sub-corpus of ICE):

```
[<#10:1> <sent>]
PU,CL(main,cop,past)
 SU,NP()
  NPHD,PRON(pers,sing) {It}
 A,AVP(excl)
  AVHD,ADV(excl) {just}
 VB,VP(cop,past)
  MVB,V(cop,past,neg) {wasn't}
 CS,NP()
  DT,DTP()
   DTCE,ART(indef) {an}
  NPHD,N(com,sing) {end}
 PUNC,PUNC(comma) {,}
```

A variety of information can be extracted from tagged corpora. Now we'll write a program processing tagged texts. In *d:\fox\texts* there is a tagged version of *text1.txt* called *text1_tagged*. It was tagged using the CLAWS5 tag set. A fragment of *text1_tagged.txt* is shown below:

```
<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>
```

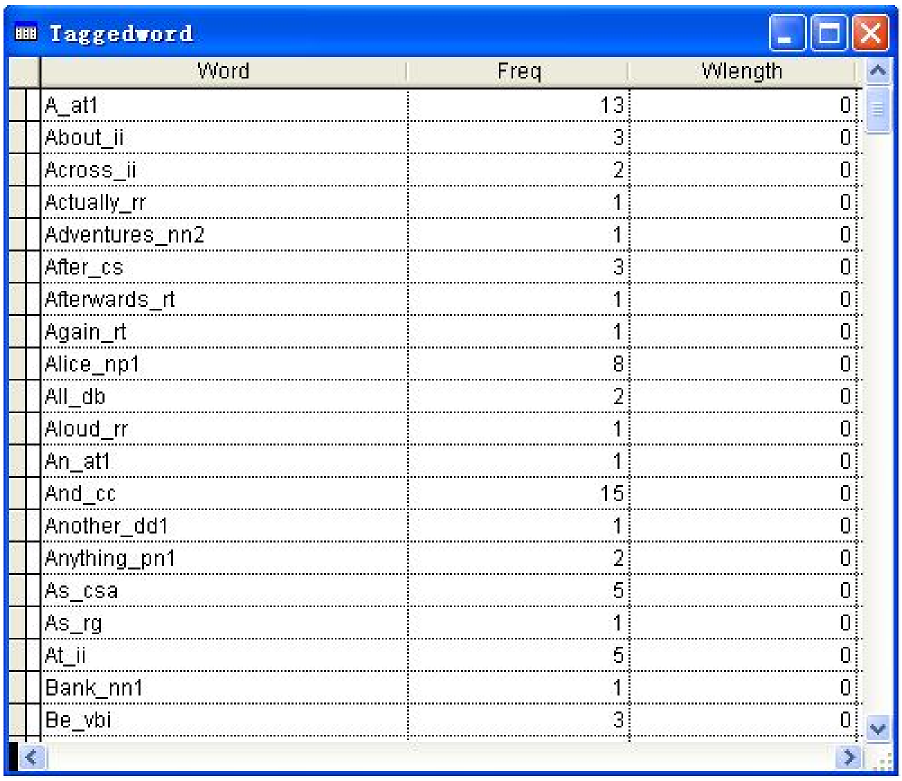
We'll first tokenize the text and make a wordlist with POS tags. The program is as follows:

taggedwords.prg

```
1. set defa to d:\fox\practice
2. close data
3. set talk off
4. set safety off
5. create table taggedword (word c(35), freq n(8), wlength n(3))
```

```
textinput=filetostr('d:\fox\texts\text1_tagged.txt')
nothing=''
carriage=chr(13)
spaces=chr(32)
textinput=strtran(textinput,'-',spaces)
textinput=strtr(textinput,spaces,carriage)
textinput=chrtran(textinput,'`,.?!:;()”',nothing)
textinput=strtran(textinput,”'”,nothing)
strtofile(textinput,'temp.txt')
append from temp.txt sdf for word<>spaces and word<>'_' and word<>'<'
replace all word with proper(word)
replace all freq with 1
index on word tag word
total to temp on word
zap
append from temp
brow
```

Part of the result is shown in Figure 4.4.



Word	Freq	Wlength
A_at1	13	0
About_ii	3	0
Across_ii	2	0
Actually_rr	1	0
Adventures_nn2	1	0
After_cs	3	0
Afterwards_rt	1	0
Again_rt	1	0
Alice_np1	8	0
All_db	2	0
Aloud_rr	1	0
An_at1	1	0
And_cc	15	0
Another_dd1	1	0
Anything_pn1	2	0
As_csa	5	0
As_rg	1	0
At_ii	5	0
Bank_nn1	1	0
Be_vbi	3	0

Figure 4.4 Part of wordlist of *text1_tagged.tx*t with POS tags

If we want to remove the POS tags, type in the command window:

```
replace all word with stuff(word,at('_',word),30,'') ↲
```

To remove the words, type:

```
replace all word with stuff(word,1,at('_',word),'') ↲
```

4.4.4 Making concordance

Now we'll write a program to make a concordance of every word in *alice.txt*. The concordance will be in the KWIC format, with a four-word context on either side of the key word. The program is as follows:

concord1.prg

```
set defa to d:\fox\practice
set safe off
set talk off
clear
clos data
nothing=''
linebreak=chr(10)
carriage=chr(13)
spaces=chr(32)
leftcontext=nothing
rightcontext=nothing
keyword=nothing
concordance=nothing
textinput=fileto('d:\fox\texts\alice.txt')
textinput=strtr(textinput,carriage+linebreak,spaces)
textinput =strtr(textinput,'--',spaces)
textinput=strtr(textinput,spaces+spaces,nothing)
textinput ='* * * * '+ textinput +' * * * * '&&note the space after each *
    and the space before the first * on the right of textinput
do while len(textinput)>1
texttocut=textinput
for i=1 to 4
spaceposition=at(spaces,texttocut)
cut=substr(texttocut,1,spaceposition)
texttocut=stuff(texttocut,1,spaceposition,nothing)
leftcontext=leftcontext+cut
endfor
```

```
27. spaceposition=at(spaces,texttocut)
28. keyword=substr(texttocut,1,spaceposition)
29. texttocut=stuff(texttocut,1,spaceposition,nothing)
30. for i=1 to 4
31. spaceposition=at(spaces,texttocut)
32. cut=substr(texttocut,1,spaceposition)
33. texttocut=stuff(texttocut,1,spaceposition,nothing)
34. rightcontext=rightcontext+cut
35. endfo
36. concordline=padl(leftcontext,40,spaces)+upper(keyword)+rightcontext
37. concordance=concordance+concordline+carriage
38. leftcontext=nothing
39. rightcontext=nothing
40. firstspace=at(spaces,textinput)
41. textinput=stuff(textinput,1,firstspace,nothing)
42. enddo
43. strtof(concordance,'concordance.txt')
44. modify file concordance.txt
```

In this program, statements 10—13 initialize *leftcontext*, *rightcontext*, *keyword* and *concordance*, which respectively hold the left four-word context, the key word, the right four-word context, and concordance lines. Statement 17 ensures there is only one space between words. Statement 18 adds four asterisks on either side of *textinput*, which serve as dummy words to provide a complete left four-word context and right four-word context respectively for the initial three words and the last three words of *textinput*. Statement 19 ensures that the program will loop between statement 19 and statement 42 until *textinput* is exhausted. Statement 20 assigns the contents of *textinput* to *texttocut*. Statements 21—26 produce the left four-word context, cut one by one from *texttocut*. Statement 28 cuts the keyword from *texttocut*, and statements 30—35 produce the right four-word context. Statement 36 puts the left four-word context, the keyword and the right four-word context to *concordline*, with the keyword 40 spaces from the leftmost of *concordline*. Statement 37 stores all the *concordline*s, putting each on a new line. Statements 38—39 empty *leftcontext* and *rightcontext* for the next round of processing. Statement 41 removes the first word of *textinput* so that the second word will be the keyword in the next round of processing.

The following is part of the result.

```
                                * * * * ALICE'S ADVENTURES IN WONDERLAND CHAPTER
                                  * * * ALICE'S ADVENTURES IN WONDERLAND CHAPTER I
                 * * ALICE'S ADVENTURES IN WONDERLAND CHAPTER I Down
                * ALICE'S ADVENTURES IN WONDERLAND CHAPTER I Down the
       ALICE'S ADVENTURES IN WONDERLAND CHAPTER I Down the Rabbit-Hole
       ADVENTURES IN WONDERLAND CHAPTER I Down the Rabbit-Hole Alice
                IN WONDERLAND CHAPTER I DOWN the Rabbit-Hole Alice was
```

```
      WONDERLAND CHAPTER I Down THE Rabbit-Hole Alice was beginning
             CHAPTER I Down the RABBIT-HOLE Alice was beginning to
         I Down the Rabbit-Hole ALICE was beginning to get
     Down the Rabbit-Hole Alice WAS beginning to get very
      the Rabbit-Hole Alice was BEGINNING to get very tired
Rabbit-Hole Alice was beginning TO get very tired of
         Alice was beginning to GET very tired of sitting
           was beginning to get VERY tired of sitting by
          beginning to get very TIRED of sitting by her
              to get very tired OF sitting by her sister
              get very tired of SITTING by her sister on
          very tired of sitting BY her sister on the
            tired of sitting by HER sister on the bank,
              of sitting by her SISTER on the bank, and
          sitting by her sister ON the bank, and of
               by her sister on THE bank, and of having
              her sister on the BANK, and of having nothing
            sister on the bank, AND of having nothing to
               on the bank, and OF having nothing to do:once
               the bank, and of HAVING nothing to do:once or
            bank, and of having NOTHING to do:once or twice
          and of having nothing TO do:once or twice she
           of having nothing to DO:ONCE or twice she had
      having nothing to do:once OR twice she had peeped
          nothing to do:once or TWICE she had peeped into
            to do:once or twice SHE had peeped into the
```

If the text to be processed is very long, say, about 100,000 words in length, *concord1.prg* is very slow. The following program uses low level file functions in making concordance for a text. It's much faster than the previous one.

concord2.prg

```
close all
set defa to d:\fox\practice
set safe off
set talk off
clear
nothing=''
carriage=chr(13)
spaces=chr(32)
leftcontext=nothing
rightcontext=nothing
keyword=nothing
textinput=fileto('d:\fox\texts\alice.txt')
textinput =strtr(textinput,'--',spaces)
textinput=strtr(textinput,spaces+spaces,nothing)
textinput ='* * * * '+ textinput +' * * * * '
```

```
16. textinput=strtran(textinput,spaces,carriage)
17. strtofile(textinput,'temp.txt')
18. store fopen('temp.txt') to fhandle
19. concordtext=fcreat('concordance.txt')
20. fseek(fhandle,0)
21. do while not feof(fhandle)
22. for i =1 to 4&&left context
23. word=fgets(fhandle) &&here fgets() gets a word from temp.txt. No
    number of bytes is specified in fgets() since it stops before a carriage
    return, and all the words in temp.txt have a carriage return after them
24. if i=1
25. position=fseek(fhandle,0,1)
26. else
27. fseek(fhandle,0,1)
28. endif
29. leftcontext=leftcontext+word+spaces
30. endfor
31. kword=fgets(fhandle)+spaces
32. fseek(fhandle,0,1)
33. for i = 1 to 4
34. word=fgets(fhandle)+spaces
35. fseek(fhandle,0,1)
36. rightcontext=rightcontext+word
37. endfor
38. concordline=padl(leftcontext,40,spaces)+upper(kword)+rightcontext
39. fputs(concordtext,concordline)
40. leftcontext=nothing
41. rightcontext=nothing
42. fseek(fhandle,position,0)
43. enddo
44. fclos(concordtext)
45. fclose(fhandle)
46. modify file concordance.txt
```

In this program, Statement 16 tokenizes the contents of *alice.txt* stored in *textinput*, with a carriage return at the end of each word. Statement 17 puts the contents of *textinput* in *temp.txt*, and statement 18 opens it with the low level file opening function *fopen()* and assigns its file handle to *fhandle*. Statement 19 creates a text file *concordtext.txt* using the low level file creation function *fcreate()*, and assigns its file handle to *concordtext*. Statement 20 moves the file pointer to the top of *temp.txt*. Statements 21—43 form a loop, in which words in *temp.txt* is cut one by one with a four-word context on either side. Statements 22—30 produce the left four-word context. Statement 23 cuts a word off *temp.txt*.

Statement 24 checks whether this word is the first word of the left context. If it is, statement 25 assigns the file pointer to *position* after the cut is made. Statement 27 shifts the file pointer downwards by the length of the word cut. Statement 29 adds the four words together to form the left four-word context. Statement 31 gets the key word, and statement 32 moves the file pointer downwards by the length of the key word. Statements 33 to 37 get the right four-word context. Statement 38 puts the key word 40 spaces from the left of *leftcontext* followed by *rightcontext*. Statement 39 stores this line of concordance in *concordtext.txt* created by statement 19. Statements 40 and 41 empty *leftcontext* and *rightcontext* to make room for the next round of processing. Statement 42 shifts the file pointer back to *position*, the beginning of the second word in *temp*, after which the program goes back to statement 22 to start making a concordance line for the second word.

4.4.5 Making annotated wordlists

In language teaching we often need to make annotated wordlists, which contain words, word frequency, range (in how many lessons they occur), in which lesson they occur and how many times they occur in that lesson. It's very time consuming and error-prone to make such wordlists manually. Now we'll write a program for making such wordlists, using the 48 text files *text1.txt—text48.txt* in *d:\fox\texts*, assuming they are the 48 lessons of an English course book. The program is as follows:

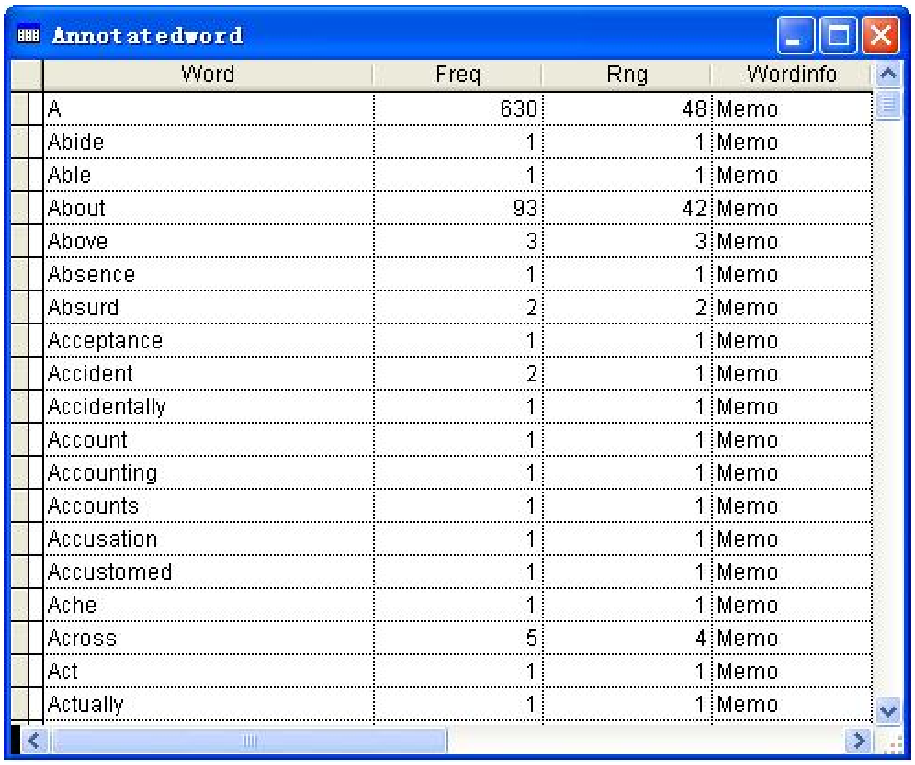
annoteword.prg

```
1.  set defa to d:\fox\practice
2.  set safe off
3.  set talk off
4.  clos data
5.  clear
6.  creat cursor wordlist(word c(25),freq n(6))
7.  nothing=''
8.  linebreak=chr(10)
9.  carriage=chr(13)
10. spaces=chr(32)
11. textinput=nothing
12. wordfield='(word c(25),'
13. freqfield=nothing
14. wordfield='(word c(25), rng n(3),'
15. for i=1 to 48
16. freqfield=freqfield+'L'+alltrim(str(i))+' n(3),'
17. endfor
annotation='wordinfo m(4))'
multifield=wordfield+freqfield+annotation
create table lexinfo &multifield
for i=1 to 48
textname='d:\fox\texts\text'+alltr(str(i))+'.txt'
textinput=textinput+filetostr('&textname')+carriage+'~~~~'
endfor
textinput=chrtr(textinput,',.`[?]_”!:;()*',nothing)
textinput=strtran(textinput,”'”,nothing)
textinput=strtran(textinput,'-',spaces)
textinput=strtr(textinput,spaces,carriage)
textinput=prop(textinput)
strtofi(textinput,'temp.txt')
select 1
append from temp.txt sdf for word<>spaces and word<>'~'
repl all freq with 1
index on word tag word
total to temp on word
zap
appe from temp
select 2
recordnumber=reccount()
for i=1 to 48
textchunk=substr(textinput,1,at('~~~~',textinput))&&get a text from textinput
textinput=stuff(textinput,1,at('~~~~',textinput),nothing)&&erase this text from input
strtofile(textchunk,'temp.txt')
append from temp.txt sdf for word<>spaces and word<>'~'
frequency='L'+alltrim(str(i))
replace all &frequency with 1 for recno()>recordnumber
index on word tag word
total to temp on word
zap
append from temp
replace all rng with rng+1 for &frequency>0
recordnumber=reccount()
endfor
go top
do while not eof()
freqinfo=nothing
for i=1 to 48
freqfield='L'+alltrim(str(i))
```

```
59. if &freqfield>0
60. freqinfo=freqinfo+freqfield+','+alltrim(str(&freqfield))+'; '
61. endif
62. endfor
63. replace wordinfo with 'Range: '+alltrim(str(rng))+'; '+freqinfo
64. freqinfo=nothing
65. skip
66. enddo
67. set relation to word into wordlist
68. copy to annotatedword fields word, a.freq,rng,wordinfo
69. use annotatedword
70. repl all wordinfo with word+'Freq: '+ alltr(str(freq))+'; '+wordinfo
71. brow
```

This program can be divided into four sections. The first section is between statement 1 and statement 20, for variable initialization and table creation. Statement 6 creates a temporary table *wordlist* that will be automatically deleted after the program has run. Statements 15—20 create a multiple field table, with a word field, a range field and 48 fields from *L1* to *L48 f*or holding word occurrences in each of the 48 lessons respectively. The second section is between statements 21—37. The 48 texts are put into *textinput* one by one separated with ~~~~. They are subsequently tokenized and turned into a frequencied wordlist. The third section is between statements 38—53 for extracting word range and word occurrences in individual lessons from *textinput*. Statement 38 accesses table *lexinfo*. Statement 39 assigns to *recordnumber* the current position of the record pointer in *lexinfo*. Statements 40—53 get texts from *textinput* one by one, calculate word range and word frequency and put them in their respective field. When *i* =1, *text1.txt* is cut from *textinput* and assigned to *textchunk*, which is subsequently tokenized and appended to *lexinfo*. Statement 45 assigns the string literal “L1” to *frequency*, and statement 46 replaces the field *L1* with 1 (using the macro operator &) for the newly appended words from *text1.txt.* Statement 51 calculates the range of the words in *lexinfo*. Statement 52 stores the current position of the record pointer after the words of *text1.txt* have been appended and totalled. When *i* = 2 statement 45 assigns “L2” to *frequency*. Statement 46 replaces the frequency field *L2* with 1 for the newly appended words from *text2.txt*, and statement 51 calculates the range of the words in *lexinfo*. Statement 52 assigns the current position of the record pointer to *recordnumber*. The process continues until *i* equals 48. The fourth section puts word range and word frequency in individual lessons in the memo field *wordinfo*. This section begins in statement 54. Statement 56 empties the variable *freqinfo*. Statements 57—66 gather word occurrences in individual lessons and put them in *freqinfo*, which is then put in the memo field *wordinfo*. Statement 64 empties the contents of *freqinfo* for the next word. Statement 67 links *lexinfo* to *wordlist*, and statement

68 copies the *word* field, the *rng* field and the *wordinfo* field in *lexinfo* and the *freq* field in *wordlist* (*a.freq* since *wordlist* is in work area 1 and its alias is *a*) to a new table *annotatedword*. Statement 70 replaces the memo field *wordinfo* in *annotatedword* with words, word frequency, word range and their occurrences in individual lessons. Figure 4.5 is part of *annotatedword*.



Annotatedword

Word	Freq	Rng	Wordinfo
A	630	48	Memo
Abide	1	1	Memo
Able	1	1	Memo
About	93	42	Memo
Above	3	3	Memo
Absence	1	1	Memo
Absurd	2	2	Memo
Acceptance	1	1	Memo
Accident	2	1	Memo
Accidentally	1	1	Memo
Account	1	1	Memo
Accounting	1	1	Memo
Accounts	1	1	Memo
Accusation	1	1	Memo
Accustomed	1	1	Memo
Ache	1	1	Memo
Across	5	4	Memo
Act	1	1	Memo
Actually	1	1	Memo

Figure 4.5 Part of *annotatedword*.

The following is the contents of the memo field *wordinfo* for *About*, *Above*, *Absence*, and *Absurd*:

About	*Freq: 93; Range: 42; L1,3; L2,1; L3,4; L4,1; L5,1; L6,2; L7,6; L8,2; L10,1; L11,4; L12,2; L13,3; L14,3; L15,3; L16,1; L18,3; L19,1; L20,1; L23,2; L24,1; L25,1; L26,1; L27,2; L29,1; L30,1; L31,4; L32,1; L33,4; L34,2; L35,2; L36,2; L37,3; L38,1; L39,5; L40,5; L41,1; L42,3; L44,1; L45,3; L46,2; L47,1; L48,2;*
Above	*Freq: 3; Range: 3; L14,1; L26,1; L43,1;*
Absence	*Freq: 1; Range: 1; L35,1;*
Absurd	*Freq: 2; Range: 2; L10,1; L23,1;*

4.4.6 Computing word sense concentration

Word sense diversification in the English language is very common. Many

English words belong to more than one word class and have a set of different meanings. Take the word *back* as an example, according to WordNet, it's a noun, verb, adjective and an adverb, and has 28 different senses. However, for a multi-sense word, if one of its meanings occurs much more often, then we say this meaning is the sense concentration of the word. A measure for word sense concentration is the Herfindahl´s concentration measure, also known as the Repeat rate, which is as follows:

$$R = \frac{1}{N^2}\sum_{i=1}^{S} f_i^2 ,$$

where f_i is the frequency of sense i of a word in a text or corpus, N the sum of the frequencies each of the senses has, and S the number of senses. For example, if a word has three different senses, and the frequency of each sense in a corpus is respectively 67, 1, 1, then the Repeat rate R is:

$$R = [67^2 + 1^2 + 1^2]/69^2 = 0.9433$$

Generally, the smaller the R, the more diverse the senses.

In *d:\fox\table3* there is a table *wordsense* containing 165 common English words with annotations taken from WordNet of the Princeton University (http://www.cogsci.Princeton.Edu/~ wn/); the annotations include word class, number of senses and the frequency of each of the senses in the Brown Corpus. Word annotations in the table are arranged in the following WordNet format:

```
*animal
The noun animal has 1 sense (first 1 from tagged texts)
1. (67) animal, animate being, beast, brute, creature, fauna
The adj animal has 2 senses (first 1 from tagged texts)
1. (1) animal, carnal, fleshly, sensual
2. animal -- (of the nature of or characteristic of or derived from an animal or
animals…)
```

The head word occupies a line and begins with an asterisk. The word class and the number of senses of the said word class are on the following line, while the frequency of a sense belonging to the class is placed on the third line with the number placed in brackets. However, if a sense does not exist in the Brown Corpus, no number is given on this line, as shown in the second sense of the adjective class of *animal*. In this case we can regard the frequency of this sense as 1.

Now we'll write a program to extract from *wordsense* in *d:\fox\table3* the head words, their word classes, the number of senses each class has, the frequency of each of the senses, and compute the Repeat rate for each of the words. The program is as follows.

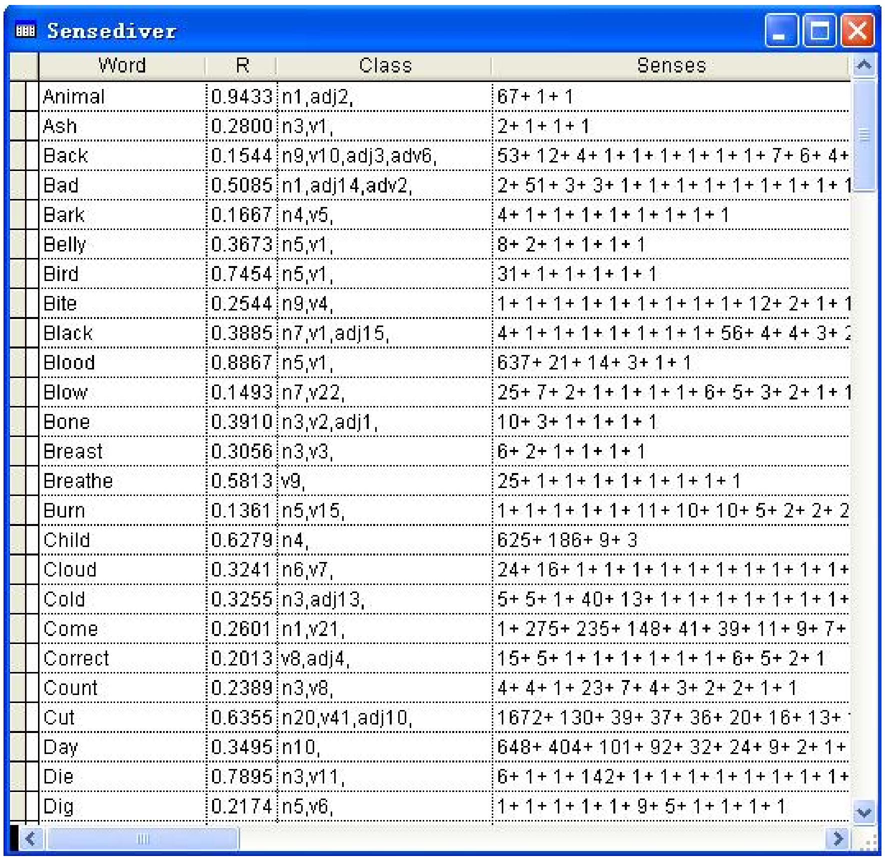
sensefocus.prg

```
set default to d:\fox\practice
clear
close data
set safety off
create table sensefocus (word c(20),r n(6,4),class c(25),sensefreq c(150))
create table sensetable(wordsense c(250))
append from d:\fox\table3\wordsense
replace all wordsense with lower(wordsense)
addfreq="
addwordclass="
scan for wordsense='*'
targetword=alltrim(wordsense)
skip
do while wordsense<>'*' and recno()<reccount()
if wordsense='the noun'
wordclass='n'
else
if wordsense='the verb'
wordclass='v'
else
if wordsense='the adj'
wordclass='adj'
else
if wordsense='the adv'
wordclass='adv'
endif
endif
endif
endif
sentence=stuff(wordsense,1,at('has ',wordsense)+3,")
sensenumber=substr(sentence,1,2)
addwordclass=addwordclass+wordclass+rtrim(sensenumber)+','
skip
do while alltrim(wordsense)<>'the noun' and    alltrim(wordsense)<>'the verb' and alltrim(wordsense)<>'the adj' and alltrim(wordsense)<>'the adv' and alltrim(wordsense)<>'*' and recno()<=reccount()
if '('$subs(wordsense,1,4)
sentence=stuff(wordsense,1,at('(',wordsense),")
freq=substr(sentence,1,at(')',sentence)-1)
addfreq=addfreq+freq+'+ '
else
```

```
40. if '('$left(wordsense,6)=.f.
41. freq='1'
42. addfreq=addfreq+freq+'+ '
43. endif
44. endif
45. skip
46. enddo
47. enddo
48. select 1
49. addfreq=stuff(addfreq,rat('+',addfreq),1,")&&remove the trailing +
50. n=evaluate(addfreq)
51. sumsquare='('+strtran(addfreq,'+','**2+')+'**2)'
52. append blan
53. replace word with targetword
54. replace r with evaluate(sumsquare)/n**2
55. replace class with addwordclass
56. replace sensefreq with addfreq
57. addwordclass="
58. addfreq="
59. sele 2
60. skip -1
61. endscan
62. select 1
63. replace all word with strtran(word,'*',")
64. replace all word with proper(word)
65. brow
```

In this program statements 5—6 create two tables *sensefocus* and *sensetable*. *sensefocus* has four fields: *word* for head words, *r* for Repeat rate, *class* for word class, and *sensefreq*, a character field, for sense frequencies in the form of 3+1+4+1, etc. *sensetable* has only one field for contents from *wordsense* in *d:\fox\table3*. Statements 9—10 initialize *addfreq* and *addwordclass*. The former holds sense frequencies of a word, with each frequency followed by a plus sign; the data type is character. The latter stores the different word classes of a word. Statements 11—61 scan for the head words and extract related information from their annotations. Once the head word is located, statement 12 assigns it to *targetword*, and the program moves between statements 14—47, until a line beginning with an asterisk is encountered, and searches for lines containing word class, number of senses of the class, and the frequency of senses belonging to the class, and extract such information. Statement 30 removes words preceding sense number from *wordsense* in *senstable* and assigns the remaining to *sentence*. For example, if *wordsense* contains the sentence *the noun has 15 senses*, statement 30 deletes *the noun has* and assigns the remaining *15 senses* to *sentence*. Statement

31 cuts the number off *sentence* and assigns it to *sensenumber*. Note that the number is actually a character. Statement 32 stores the types of word class and its sense number, adding a comma after each pair of word class and its number. Statements 34—46 extract the frequencies of a sense. Statements 35—37 get frequencies placed in brackets from *wordsense*, and statements 39—41 assign 1 to *freq* for cases where no frequency is given. Note that the frequency stored in *freq* is of character, too. *addfreq* in statement 38 and statement 42 pools the individual frequencies together, adding a plus sign after each of them. Statement 49 removes the trailing plus sign of *addfreq*, and statement 50 gets the sum of frequencies by turning the contents stored in *addfreq* into math operation using the *evaluate( )* function. Statement 51 prepares for computing the sum of squared frequencies. For example, if *addfreq* contains 22+1+3+2, then this statement turns it into (22**2+1**2+3**2+2**2), and statement 54 computes the Repeat rate by converting *sumsquare* into math operation using the *evaluate( )* function. Figure 4.6 is part of *sensfocus*.



Word	R	Class	Senses
Animal	0.9433	n1,adj2,	67+ 1+ 1
Ash	0.2800	n3,v1,	2+ 1+ 1+ 1
Back	0.1544	n9,v10,adj3,adv6,	53+ 12+ 4+ 1+ 1+ 1+ 1+ 1+ 1+ 7+ 6+ 4+
Bad	0.5085	n1,adj14,adv2,	2+ 51+ 3+ 3+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 1
Bark	0.1667	n4,v5,	4+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 1
Belly	0.3673	n5,v1,	8+ 2+ 1+ 1+ 1+ 1
Bird	0.7454	n5,v1,	31+ 1+ 1+ 1+ 1+ 1
Bite	0.2544	n9,v4,	1+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 12+ 2+ 1+ 1
Black	0.3885	n7,v1,adj15,	4+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 56+ 4+ 4+ 3+
Blood	0.8867	n5,v1,	637+ 21+ 14+ 3+ 1+ 1
Blow	0.1493	n7,v22,	25+ 7+ 2+ 1+ 1+ 1+ 1+ 6+ 5+ 3+ 2+ 1+ 1
Bone	0.3910	n3,v2,adj1,	10+ 3+ 1+ 1+ 1+ 1
Breast	0.3056	n3,v3,	6+ 2+ 1+ 1+ 1+ 1
Breathe	0.5813	v9,	25+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 1
Burn	0.1361	n5,v15,	1+ 1+ 1+ 1+ 1+ 11+ 10+ 10+ 5+ 2+ 2+ 2
Child	0.6279	n4,	625+ 186+ 9+ 3
Cloud	0.3241	n6,v7,	24+ 16+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 1+
Cold	0.3255	n3,adj13,	5+ 5+ 1+ 40+ 13+ 1+ 1+ 1+ 1+ 1+ 1+ 1+
Come	0.2601	n1,v21,	1+ 275+ 235+ 148+ 41+ 39+ 11+ 9+ 7+
Correct	0.2013	v8,adj4,	15+ 5+ 1+ 1+ 1+ 1+ 1+ 1+ 6+ 5+ 2+ 1
Count	0.2389	n3,v8,	4+ 4+ 1+ 23+ 7+ 4+ 3+ 2+ 2+ 1+ 1
Cut	0.6355	n20,v41,adj10,	1672+ 130+ 39+ 37+ 36+ 20+ 16+ 13+
Day	0.3495	n10,	648+ 404+ 101+ 92+ 32+ 24+ 9+ 2+ 1+
Die	0.7895	n3,v11,	6+ 1+ 1+ 142+ 1+ 1+ 1+ 1+ 1+ 1+ 1+ 1+
Dig	0.2174	n5,v6,	1+ 1+ 1+ 1+ 1+ 9+ 5+ 1+ 1+ 1+ 1

Figure 4.6 Part of *sensfocus*.

Exercises

1. Use *wordlist* in *d:\fox\table3* and copy it to *d:\fox\practice\test*. Enter a command in the command window to right justify the words, with their right end 25 characters from the leftmost position of the field and output the new contents to *temp.txt*.

2. In *d:\fox\table3* there is a table *wronglen*. In the *wlength* field the values of word length are all wrong. For example, the length of *aback* is given as 6. There must be an unseen character in the word field. Identify the character, remove it and then get the correct word length.

3. Write a short program and use the *set relation* command to put the *word* field, *rng* field, *wordinfo* field in *annotatedword* created in 4.4.5 and the *wlength* field in *aliceword* created in 2.4.1 to a table called *temp*.

4. Write a short program to centre-justify *To Autumn* (*poem.txt* in *d:\fox\texts*) using the function *padc()*, putting every line of the poem in the centre of a line 80 characters in length.

5. Write a program to remove *CHAPTER I, CHAPTER II, CHAPTER III…* in *alice.txt* and output the result to *d:\fox\practice\temp.txt.*

6. Use *postable* in *d:\fox\table3* and copy it to *d:\fox\practive\test*. Use *test* to do the following by entering commands in the command window:

a. Shift the POS tags to the right of the words and capitalize the first letter of the words.

b. Remove the words in *test*, keeping only the POS tags and combine the identical POS tags.

7. According to Kennedy (1998), nouns in the Brown Corpus and the LOB Corpus account for 26.8% and 25.2% of the total word tokens respectively. In *d:\fox\table1* there are 50 tables from *bncst1.dbf* to *bncst50.dbf* containing 50 wordlists with POS tags. These wordlists are made from 50 2000-word text samples randomly drawn from the BNC spoken text section. Write a program to combine all the 50 tables together, and then remove the words, keeping only the POS tags, and calculate the proportion of nouns.

8. In *d:\fox\texts* there is a file *multiplication.txt* containing the multiplication table without the products. Write a program using the *evaluate()* function in it and calculate the results and output the new multiplication table in the following format:

1 X 1 = 1

1 X 2 = 2
…

9. Write a program using the low level file functions to create a text file *temp.txt* in *d:\fox\practice* and put in it all the 48 texts in *d:\fox\texts*, adding section titles *TEXT 1, TEXT 2, TEXT 3*…in the middle of a new line on top of each text chunks in *temp.txt*.

10. In *d:\fox\texts* there is *text1* in the XML text format (*text1.xml*). Open it by typing *modify file d:\fox\texts\text1.xml* in the command window, examine it carefully and then write a program to remove the XML codes and non-textual characters, tokenize it and produce a wordlist for it.

5 Arrays, Procedures and User-defined Functions

In this chapter we'll learn how to create arrays, procedures and user-defined functions (UDF) and use them in programs. An array is actually a variable with data values arranged in rows and columns stored in it. So arrays can be regarded as tables without grids and field names; but unlike tables, which are stored on the hard disc, arrays are stored in memory, and like variables as soon as Foxpro is closed, the arrays created during a Foxpro session no longer exist. Procedures are sub-programs put at the end of the main program for performing repetitive tasks, and can be called in the main program when needed. This reduces program length and makes it more readable. User-defined functions, as the name suggests, are functions designed and written by the user for specific purposes. Although there are quite a lot of built-in functions provided by Foxpro and we have learned many of them, there are occasions when a function is needed for a special task but there are no such built-in functions in Foxpro. The creation and use of arrays, procedures and user-defined functions can make our program more flexible, concise and powerful.

5.1 Commands and Functions for Arrays

Like a table, an array has rows and columns, and data stored in an array are called array elements. If an array has 3,000 rows and 4 columns, the number of elements it contains is 12,000. For Foxpro 6.0, the total number of elements can't exceed 65,000. For higher versions the maximum number of elements is two gigabytes. The following commands and functions are for the creation and manipulation of arrays.

declare *arrayname1* (*rows* [, *columns*]) [, *arrayname2* (*rows* [, *columns*])] ..This command creates specified number of arrays with specified number of rows and columns. To create three arrays called *array1*, *array2* and *array3*, with 20 rows and 4 columns, 25 rows and 3 columns and 5 rows and 6 columns respectively, type:

```
    declare array1(20,4), array2(25,3),array3(5,6) ↵
```

The array elements can be referred to by the row and column they are in. For example, array1[1,1] is the element of *array1* in row 1 and column 1; array1[2,4] is the element in row 2 and column 4. We can also use the round brackets instead of the square brackets when referring to array elements, for example, array(1,1), array(1,2) etc. Values can be assigned to array elements by specifying their row and column numbers.

```
array1[1,1]='Foxpro' ↵
array1[1,2]='array' ↵
array1[2,1]='lab' ↵
array1[2,2]='practice' ↵
?array1[1,1] ↵
Foxpro

?array1[1,2] ↵
Array

?array1[2,1] ↵
lab

?array1[2,2] ↵
practice
```

display memory like | [*arrayname*] [*variablename*] | This command is used for checking the contents of an array or a variable.

```
display memory like array1 ↵
ARRAY1      Pub     A
(  1,  1)     C   "Foxpro"
(  1,  2)     C   "array"
(  1,  3)     L   .F.
(  1,  4)     L   .F.
(  2,  1)     C   "lab"
(  2,  2)     C   "practice"
(  2,  3)     L   .F.
(  2,  4)     L   .F.
```

In the first row of the result, *ARRAY1* is the name of the variable (remember arrays are variables?). *pub* means the property of this variable is public and can be accessed anywhere within the program it's created. *A* shows that *ARRAY1* is an array. In the following rows, (1, 1), (1, 2), (1, 3) etc are row 1 and column1, row 1 and column 2, row 3 and column 3 etc. Currently only array1[1,1], array1[1,2], array1[2,1] and array1[2,2] have data values, and the data type is *C*, meaning character; the rest store the string *.F.*, which means empty, and the data type is logical.

```
phrase='Foxpro arrays' ↵
```

```
display memory like phrase ↵
PHRASE              Priv         C     "Foxpro arrays"
```

The result means the variable name is *phrase*, whose property is private, and the type of data is *C*, meaning character, and its value is *Foxpro arrays*.

dimension arrayname1(rows [, columns]) [, arrayname2(rows [,columns])] ... This command is the same as the *declare* command. To create three arrays called *array1*, *array2* and *array3* with 3 rows and 5 columns each.

```
dimension array1(3,5),array2(3,5),array3(3,5) ↵
```

afields(*arrayname* | [, *workarea*] [, '*alias*']) | This command measures the number of fields of a table, and puts the names of the fields, the type of data the fields hold, their width and so on in an array *arrayname*. *workarea* is the work area where the table is open, and *alias* is the alias of the table.

```
use d:\fox\table3\wordlist in 2 alias w
?afields(fieldinfo,2) ↵
5

?afields(fieldinfo, 'w') ↵
5

?fieldinfo(1,1) ↵
WORD

go bottom ↵
a=fieldinfo(1,1) ↵
?&a ↵
Zzzzzing

go top ↵
?&a ↵
A

go 12 ↵
?&a ↵
Abate

?fieldinfo(1,2) ↵
C
```

```
?fieldinfo(1,3) ↵
25

?fieldinfo(2,1) ↵
FREQ

? fieldinfo(2,2) ↵
N
?fieldinfo(2,3) ↵
8

?fieldinfo(3,1) ↵
RNG

?fieldinfo(3,2) ↵
N

?fieldinfo(3,3) ↵
5

?fieldinfo(4,1) ↵
WLENGTH

?fieldinfo(4,2) ↵
N

?fieldinfo(4,3) ↵
4

?fieldinfo(5,1) ↵
NOTE

?fieldinfo(5,2) ↵
M

?fieldinfo(5,3) ↵
4
```

copy to array *arrayname* [**field** [*fieldname1*] [, *fieldname2…*]] [**for** *condition*] This command copies the specified contents of a table to an array. To copy the entire contents of *spgrowth* in *d:\fox\table3* to an array called *sparray*.

```
use d:\fox\table3\spgrowth ↵
copy to array sparray ↵
```

The above statements copy the entire contents of *spgrowth* to the array *sparray*.

alen(*arrayname*) This function measures the total number of elements of an array. The total number of elements of an array is the number of its rows multiplied by the number of its columns.

```
?alen(sparray) ↵
2000
```

Please note that once an array is created by the *declare* command, *dimension* command or *copy to array* command, the number of elements of the array remain fixed throughout the Foxpro session. In *d:\fox\table3* there is a wordlist table *spwordlist* containing 10,384 words, with four fields respectively holding words, frequency, range and length. Type the following; do not press enter until all the statements have been entered and highlighted:

```
use d:\fox\table3\spwordlist
copy to array sparray
?alen(sparray)
use d:\fox\table3\spgrowth
copy to array sparray
?alen(sparray)
```

The results are both 41,536.

copy structure to *tablename* This command copies the structure of an open table to a new table *tablename*.

```
use d:\fox\table3\wordlist ↵
copy structure to temp ↵
use temp ↵
brow ↵
```

The above statements create a new empty table *temp* with the same structure of *wordlist*.

append from array *arrayname* [**for** *condition*] [**fields** *fieldnames*] This command appends the contents of an array to a table.

```
use d:\fox\table3\spgrowth ↵
copy structure to temp ↵
copy to array sparray
use temp ↵
append from array sparray ↵
brow ↵
```

scatter [**fields** *fieldnames*] | [**to** *arrayname*] [**memvar**] [**name** *variablename*] | This command copies the current record in specified fields of a table to an array or a variable. *memvar* is a Foxpro system variable. To copy the current record to a non-system variable, *name* must be used before the variable. *fields fieldnames* specifies the fields of the record to copy. To copy all the fields of a record in a table, omit the *fields fieldnames* option. The *scatter* command is often used with the following *gather* command.

gather | [**from** *arrayname*] [**memvar**] [**name** *variablename*] | [**fields** *fieldnames*] This command appends to a table the record of another table put to an array or a variable by the *scatter* command. *fields fieldnames* specifies the fields to which the records are appended. To append the records to all the fields, omit *fields fieldnames*. If the *from arrayname* option is used, the field names of the two tables don't have to be the same but the data types must be the same. For the *memvar* and *name variablenames* options, the fields of the two tables must be the same, or nothing will be appended.

```
use d:\fox\table3\wordlist ↵
copy structure to temp ↵
scatter to testarray ↵
use temp ↵
append blank ↵
gather from testarray ↵
brow ↵
use d:\fox\table3\wordlist ↵
skip ↵
scatter memvar ↵
use temp ↵
append blank ↵
gather memvar ↵
brow ↵
use d:\fox\table3\wordlist ↵
skip ↵
scatter name fld ↵
```

```
use temp ↵
append blank ↵
gather name fld ↵
brow ↵
```

acopy(*sourcearrayname*, *targetarrayname*) This function copies the contents of an array to another array.

```
use d:\fox\table3\spgrowth ↵
copy to array array1 ↵
acopy(array1,array2) ↵
?alen(array1) ↵
2000

?alen(array2) ↵
2000
```

adel(*arrayname*,*elementnumber* [, 2]) This function deletes an element specified by *elementnumber* of an array. If *2* is used, then the entire column specified by *elementnumber* is deleted.

```
use d:\fox\table3\wordlist ↵
copy to temp for recn()<2000 ↵
use temp ↵
copy to array testarray ↵
?testarray(1,1) ↵
A

adel(testarray,1) ↵
?testarray(1,1) ↵
A.c.

adel(testarray,1,2) ↵
?testarray(1,1) ↵
32
```

adel(testarray,1,2) deletes the first column holding words in *testarray*, and *testarray(1,1)* yields 32, which is the frequency of *A.c.*

ascan(*arrayname*, *string*) This function searches an array for *string*. If the search is successful, the element number of the string in the array is returned, otherwise 0 is returned. For this function to work properly, the command *set* *exact on* should be used, otherwise if we search for *work*, the function may turn out the word *workable*.

```
set exact on ↵
use d:\fox\table3\wordlist ↵
copy to temp for recn()<2000 ↵
use temp ↵
copy to array testarray field word,freq ↵
?ascan(testarray,’Allow’) ↵
1213

?ascan(testarray,’Aallow’) ↵
0
```

adir(*arrayname* [, *files*]) This function puts file names of the current folder to an array, with their size, attribute and so on. The wild card * can be used in this function. This command is very useful for inputting the file names of a folder to a table.

```
set defa to d:\fox\texts ↵
adir(filearray, '*.txt') ↵
creat table d:\fox\practice\filelist(fname c(10),bytes c(10),dates c(10),time
c(10),attri c(10)) ↵
append from array filearray ↵
brow ↵
```

The above statements first put all the files with the *txt* extension to an array called *filearray*, then append the file names to a five-field table called *filelist*.

To put all the table names in *d:\fox\table2* to an array called *allfile* and then append the table names to *filelist*, type:

```
set defa to d:\fox\table2
adir(allfile) ↵
use d:\fox\practice\filelist ↵
zap ↵
append from array allfile↵
brow ↵
```

asort(*arrayname* [, *columnnumber* [, *numbertosort* [, *sortorder*]]]) This command sorts the elements of an array. *columnnumber* specifies which column to sort. *numbertosort* specifies how many of the elements in the column to sort; the default setting is 0, which means sorting the entire column. *sortorder* has two settings, 0, which is the default setting, and any positive integer larger than 0. The former sorts in ascending order and the latter in descending order. If *numbertosort* and *sortorder* are set to 0, they can both be omitted. However, if *sortorder* is set to a positive integer, *numbertosort* can't be omitted. If *arrayname* is used alone, all the elements in the first column are sorted in ascending order. To sort all the elements of the second column in descending order, change *columnnuber* to 2, *numbertosort* to 0, and *sortorder* to any positive integer, say 5. In *d:\fox\table3* there is a table *sortarray*, which contain the following data:

```
A  5
B  4
C  3
D  2
E  1
```

Now copy the table to *temp* and type in the command window:

```
use temp ↵
copy to array test ↵
asort(test,2) ↵
zap ↵
appe from array test ↵
brow ↵
```

The result is as follows:

```
E  1
D  2
C  3
B  4
A  5
```

Now type:

```
copy to array test ↵
asort(test,1,3) ↵
zap ↵
appe from array test ↵
brow ↵
```

The result is as follows:

```
C  3
D  2
E  1
B  4
A  5
```

Now type:

```
copy to array test ↵
asort(test,1,0,1) ↵
zap ↵
appe from array test ↵
brow ↵
```

The result is as follows:

```
E  1
D  2
C  3
B  4
A  5
```

5.2 Procedures

In Chapter 2 we wrote three programs for lexical comparison between *Alice's Adventures in Wonderland* and *Through the Looking-glass*. The first two programs are almost the same except for a couple of statements. As a matter of fact, we can combine the three programs together and put the two programs that practically do the same thing into a sub-program, which can be called when it's needed. Sub-programs like this are called procedures, which are placed at the end of the main program. The form of a procedure is as follows:

> **procedure** *procedurename*
> [**private** *variablelist*]
> [**public** *variablelist*]
> s*tatements*
> **return**
> [**endproc**]

Since procedures are sub-programs, we should specify whether the variables in a procedure are public or private; that is, whether the variables in the procedure are recognized throughout the program, both main and sub, or only within the procedure. Suppose we want to write a procedure called *cleantext*, with three public variables *a1*, *a2*, *a3* and two private variables *b1*, *b2*, the initial part of the procedure is like the following:

```
procedure cleantext
public a1, a2, a3
private b1, b2
```

However, unless otherwise specified in the procedure, the variables in the main program are public, recognized both in the main program and in the procedure.

To call a procedure in the main program, put the following command in the main program where the procedure is needed:

do *procedurename*

Now we'll combine the three programs in 2.5.1 for lexical comparison between *alice.txt* and *lglass.txt*.

aliceglass.prg

```
set default to d:\fox\practice
set safety off
close data
create table awordlist (word c(25),freq n(10),wlength n(4))
create table lwordlist (word c(25),freq n(10),wlength n(4))
create table aliceglass (word c(25),freq n(12,5))
textinput=filetostr('d:\fox\texts\alice.txt')
do tokenizer
select 1
append from temp
textinput=filetostr('d:\fox\texts\lglass.txt')
do tokenizer
select 2
append from temp
select 3
append from awordlist
replace all freq with freq*100000
append from lwordlist
index on word tag word
total to temp on word
zap
append from temp
copy to sharedw for mod(freq,100000)>0 and freq>100000
copy to aliceonly for mod(freq,100000)=0
copy to lglassonly for freq<100000
use aliceonly
replace all freq with freq/100000
use sharedw
replace all freq with freq/100000
procedure tokenizer
create table temp(word c(25),freq n(10),wlength n(4))
nothing=''
```

```
33. spaces=chr(32)
34. carriage=chr(13)
35. textinput=strtran(textinput,'-',spaces)
36. textinput=strtran(textinput,spaces,carriage)
37. strtofile(textinput,'temp.txt')
38. append from temp.txt sdf
39. replace all word with chrtran(word,',.`[?]_”!:;()*',nothing)
40. replace all word with strtran(word,”'”,nothing)
41. delete all for isblank(word)=.t.
42. pack
43. strtofile(textinput,'temp.txt')
44. replace all word with prop(word)
45. replace all freq with 1
46. index on word tag word
47. total to temp1 on word
48. zap
49. append from temp1
50. replace all wlength with len(alltrim(word))
51. use
52. return
```

In this program, the procedure called *tokenizer* is between statements 30 and 52, and it’s called twice by statement 8 and statement 12 in the main program. After the procedure completes its task, the program goes back to the main program to the statement next to the statement that calls the procedure.

The way to call procedures can be used for a program to call another program. For example, we can turn the above procedure into a stand-alone program with the same name and call it in another program the same way as procedures are called.

5.3 User-defined Functions

Foxpro has many built-in functions, but these functions can’t always meet the needs of the user. When we need a function that Foxpro doesn’t have, we can create the function ourselves. Such functions are called user-defined functions (UDF). User-defined functions are similar to procedures in that they are both sub-programs put at the end of the main program and can be called any time they are needed. The difference lies in the way they are created, called and how results are passed to the main program. The following commands and statements are for function creation:

function *functionname*

parameters [*parameter1*] [, *parameter2*]…
statements
return [*value*]
[**endfunc**]

User-defined functions have the following two settings, normally put at the initial section of the main program that uses user-defined functions.

set udfparms to value

In this setting, a user-defined function can manipulate variables of the main program but their original values in the main program can't be changed. This is the default setting.

set udfparms to reference

In this setting a user-defined function can manipulate variables of the main program and their original values can be changed.

The way to use user-defined functions is the same as we use Foxpro built-in functions. Suppose we have created a function called *counta()* for counting the number of the letter *A* in a text stored in a variable called *textinput*, then *counta(textinput)* performs this task.

Now we'll write a program to make 48 wordlists for the 48 texts in *d:\fox\texts* and store them in 48 tables called *text1*, *text2…text48*. A user-defined function *tokenize* is used to tokenize these texts and remove punctuation marks, numbers and other non-word strings. The program is as follows:

multiwordlist.prg

```
set defa to d:\fox\practice
set udfparms to reference
set safe off
set talk off
close data
create cursor wordtable(word c(25),freq n(6),wlength n(4))
nothing=''
carriage=chr(13)
spaces=chr(32)
for i=1 to 48
texts='d:\fox\texts\text'+alltr(str(i))+'.txt'
textinput=filetos('&texts')
tokenize (textinput)
strtof(textinput,'temp.txt')
selec 1
```

```
16. append from temp.txt sdf for word<>spaces
17. replace all freq with 1
18. index on word tag word
19. total to temp on word
20. zap
21. appen from temp
22. replace all wlength with len(alltrim(word))
23. copy to 'text'+alltr(str(i))
24. zap
25. endfor
26. function tokenize
27. parameters strings
28. strings=chrtr(strings,',.;:`”!?-()[]0123456789*',spaces)
29. strings=strtr(strings,”'”,nothing)
30. strings=strtran(strings,spaces,carriage)
31. strings=prop(strings)
32. return
```

In this program, statement 13 calls the user-defined function *tokenize()* to tokenize *textinput*. Since we want the function to pass *textinput* back to the main program with changed value, i.e. tokenized with numbers and non-word strings removed etc, *udfparms* is set to reference in statement 2. The function is between statements 26—32. Statement 26 in the function declares the parameter *string*, which stands for *textinput*.

If instead of a variable, a function is called in the main program to deal with a text, a table or a field of a table, the name of the text, table or field should be put between two quotes, either single or double. In the function part of the program, after the declaration of parameters standing for a table or a field of a table created or used in the main program, the macro operator & must be put before the parameters when referring to the table or field. Look at the functions used in the following two programs demonstrating user-defined functions for dealing with texts, tables and fields of a table. The function *makeword()*in the first program turns a text into a frequencied wordlist, while the function *wordlength()* in the second program computes word length of a table. Note the use of the macro operator & in *wordlength()*.

functext.prg

```
1. set defa to d:\fox\practice
2. set safe off
3. set udfpar to reference
4. clos data
5. makeword('d:\fox\texts\text1.txt')
6. copy to text1
```

```
7.  makeword('d:\fox\texts\text2.txt')
8.  copy to text2
9.  makeword('d:\fox\texts\text3.txt')
10. copy to text3
11. func makeword
12. parameters text
13. creat cursor wordtable(word c(25),freq n(5),wlength n(4))
14. nothing=''
15. carriage=chr(13)
16. spaces=chr(32)
17. textinput=filetos(text)
18. textinput=chrtr(textinput,',.:;`”!?-()[]0123456789*',spaces)
19. textinput =strtr(textinput,’”’,nothing)
20. textinput =strtran(textinput,spaces,carriage)
21. textinput =prop(textinput)
22. strtof(textinput,'temp.txt')
23. appe from temp.txt sdf for word<>spaces
24. repl all word with prop(word)
25. repl all freq with 1
26. inde on word tag word
27. tota to temp on word
28. zap
29. appe from temp
30. return
```

funcfield.prg

```
1.  set defa to d:\fox\practice
2.  set safe off
3.  set udfpar to reference
4.  clos data
5.  wordlength('text1','word','wlength')
6.  wordlength('text2','word','wlength')
7.  wordlength('text3','word','wlength')
8.  func wordlength
9.  parameters tables,words,length
10. use &tables
11. repl all &length with len(alltr(&words))
12. return
```

Next, we’ll write a program using a function to compute $\dfrac{\dfrac{15!}{4!}}{(23-6)!}$.

factorial.prg

```
1. ?factorial(15)/factorial(4)/factorial (23-6)
2. function factorial
3. parameters n
4. s=1
5. for i=1 to n
6. s=s*i
7. endfo
8. return s
```

The use of the function *factorial()* makes the program much shorter than without it, in which case three *for…endfor* loops have to be used respectively for 15!, 4! and (23−6)!. Statement 3 in the function declares the parameter *n*, which stands respectively for 15, 4 and 23−6. Since there is no variable of the main program represented in the function and *s* in the function is the result respectively of *factorial(15)*, *factorial(4)* and *factorial(23-6)*, *udfparms* is the default setting, and the value of *s* must be returned to the main program.

5.4 The *do case* command and *iff()* Function

The *do case* command is very important in setting multiple conditions for directing program flow, while the *iff()* function has the function of two *if* commands. The *do case* command is in the following form:

do case
case *condition1*
statements
case *condition2*
statements
case *condition3*
statements
……
endcase

The statements under the condition evaluated as *.T.* are carried out. *docase.prg* demonstrates the use of the *do case* command. The program picks out words in *wordlist* in *d:\fox\table3* ending in *ability*, *ism*, *ment*, *ness*, *ship*, *sion*, *tion* with their frequency and puts them in a text file in two columns in descending order of frequency, with the frequency column right justified.

docase.prg

```
1. set defa to d:\fox\practice
2. set safe off
```

```
clos data
set talk off
clear
carriage=chr(13)
wordend1=padr('__BILITY',28)+'FREQUENCY'+ carriage
wordend2=padr('__ISM',28)+'FREQUENCY'+ carriage
wordend3=padr('__MENT',28)+'FREQUENCY'+ carriage
wordend4=padr('__NESS',28)+'FREQUENCY'+ carriage
wordend5=padr('__SHIP',28)+'FREQUENCY'+ carriage
wordend6=padr('__SION',28)+'FREQUENCY'+ carriage
wordend7=padr('__TION',28)+'FREQUENCY'+ carriage
number1=0
number2=0
number3=0
number4=0
number5=0
number6=0
number7=0
use d:\fox\table3\wordlist
index on freq tag freq desc
copy to temp
use temp
do while not eof()
do case
case right(rtrim(word),6)='bility'
number1=number1+1
wordend1=wordend1+padr(rtrim(word),24)+padl(rtrim(str(freq)),10)+carriage
case right(rtrim(word),3)='ism'
number2=number2+1
wordend2=wordend2+padr(rtrim(word),24)+padl(rtrim(str(freq)),10)+carriage
case right(rtrim(word),4)='ment'
number3=number3+1
wordend3=wordend3+padr(rtrim(word),24)+padl(rtrim(str(freq)),10)+carriage
case right(rtrim(word),4)='ness'
number4=number4+1
wordend4=wordend4+padr(rtrim(word),24)+padl(rtrim(str(freq)),10)+carriage
case right(rtrim(word),4)='ship'
number5=number5+1
wordend5=wordend5+padr(rtrim(word),24)+padl(rtrim(str(freq)),10)+
    carriage
42. case right(rtrim(word),4)='sion'
43. number6=number6+1
44. wordend6=wordend6+padr(rtrim(word),24)+padl(rtrim(str(freq)),10)+
    carriage
45. case right(rtrim(word),4)='tion'
46. number7=number7+1
47. wordend7=wordend7+padr(rtrim(word),24)+padl(rtrim(str(freq)),10)+
    carriage
48. endcase
49. skip
50. enddo
51. wordends=wordend1+'NUMBER OF __BILITY: '+;
    ltrim(str(number1))+carriage+carriage+wordend2+'NUMBER OF;
    __ISM: '+ltrim(str(number2))+carriage+carriage+wordend3+;
    'NUMBER OF __MENT:'+ltrim(str(number3))+carriage+carriage+;
    wordend4+'NUMBER OF__NESS: '+ltrim(str(number4))+carriage+;
    carriage+wordend5+'NUMBER OF__SHIP: '+ltrim(str(number5))+;
    carriage+carriage+wordend6+'NUMBER    OF __SION: '+;
    ltrim(str(number6))+carriage+carriage+wordend7+'NUMBER OF;
    __TION: '+ltrim(str(number7))+carriage+carriage
52. wordends=wordends+'TOTAL NUMBER OF WORDS WITH;
    ABOVE ENDINGS:'+ltrim(str(number1+number2+number3+;
    number4+number5+number6+number7))
53. strtofi(wordends,'temp.txt')
54. modi file temp.txt
```

In this program statements 6—20 initialize variables for storing words with specified endings and their frequency respectively. Statements 22—23 arrange the frequency field in descending order and then copy *wordlist* to *temp*. Statements 25—50 create a loop, in which the record pointer moves down from the top to the bottom of *temp*, picking out along the way words satisfying the conditions set by the *case* statements between *do case* and *endcase*. If a word fails to satisfy any of the cases, the record pointer moves to the next word, which is evaluated by the cases again. After the record pointer reaches the bottom of *temp*, statements 51 and 52 put all the words with the specified endings and their frequency together, with their subtotal and a blank line between each type of ending and the grand total at the end. Statement 51 is too long and put in seven lines, with semi-colons at the end of each line. In Foxpro, a long statement can be broken into several lines with semi-colons plus carriage returns. The result is stored in *temp.txt*, which is opened by statement 54. The following is part of the result:

```
__BILITY            FREQUENCY
```

```
Responsibility    138
Ability           107
Possibility       91
Liability         58
Availability      19
Probability       18
Stability         17
Profitability     14
Disability        14
Flexibility       13
Inability         11
Reliability       10
Visibility        9
Capability        9
Accountability    9
Compatibility     8
Credibility       7
Vulnerability     6
Permeability      6
Viability         5
Suitability       5
Feasibility       5
Accessibility     5
Acceptability     5
Mobility          4
……
NUMBER OF __BILITY: 72
```

iif(*condition, statement1*, *statement2*) This function evaluates *statement1* and *statement2* under *condition*; if *statement1* is *.T.* under *condition*, it's carried out, else *statement2* is carried out.

```
a = 9 ↵
?iif(a<9,'Correct', 'Wrong') ↵
Wrong
```

5.5 Some Commands and Functions for Miscellaneous Purposes

In this section we'll look at some commands and functions for miscellaneous purposes. These commands and functions are for creating program-generated message to the screen; suspending and resuming a program; checking for work area, table name and field name; and getting the current time and date generated by the computer clock, etc.

@rownnumber, *columnnumber* [**say** *contents*] [**get** *variable*] This command outputs *contents* or *variable* to the screen at the specified position.

```
@10,50 say 4/24 ↵
```

The result 0.17 is outputted to the screen at the 10th row and 50th column.

```
@20,10 say ‘Tokenizing the text now. Please wait…’
```

The message *Tokenizing the text now. Please wait…* is outputted to the screen at row 20 and column 10.

```
a=log(80) ↵
@10,70 say 'The result is: ' get a ↵
```

The result is: 4.38 is outputted to the screen at row 10 and column 70.

wait *message* **window at** *rownumber*, *columnnumber* **timeout** *seconds* This command outputs *message* to a message box at the specified position. The message box disappears after the specified time.

```
wait 'Processing the text now. Please wait....' window at 15,40 timeout 4 ↵
```

The message appears in a message box at row 15 and column 40 on the screen, and stays there for 4 seconds.

suspend This command is used for temporarily stops a running program at the place where the command is issued, often used for checking the results produced so far or for debugging purposes.

resume This command restarts a program from where it’s temporarily stopped by the *suspend* command.

```
for i=1 to 100
?i
if i=40
suspend
endif
endfor
```

When the value of *i* reaches 40, the computer stops outputting the value of *i* to the screen. Now type:

```
resume ↵
```

outputting of *i* to the screen resumes.

dbf() This function returns the name of an open table and its path, often used in procedures and functions.

```
close data ↵
use d:\fox\table3\wordlist ↵
?dbf() ↵
D:\FOX\TABLE3\WORDLIST.DBF
```

field(*n*) This function returns the name of the n^{th} field of a table.

```
close data ↵
use d:\fox\table3\wordlist ↵
?field(1) ↵
WORD

?field(2) ↵
FREQ
```

time() This function outputs the current time of the computer clock.

```
?time()
```

This returns the current time of the computer clock. We can also assign the value of *time()* to a variable.

date() This function outputs the current date set in the computer.

```
?date()
```

This returns the current date of the computer clock. Like *time()*, we can assign it to a variable.

string **function** '**v***n*' This command continuously cuts *n* characters from the left of *string* and puts them to the screen with carriage return until there are no more characters remaining in *string*. If there are fewer than *n* characters left in string, the remaining characters are put to the screen.

```
a='Linguistics' ↵
?a function 'v1' ↵
L
i
n
g
u
i
s
t
i
c
s

?a function 'v5' ↵
Lingu
istic
s
```

set alternate to *filename* This command creates a file that captures any output directed to the screen. It must be used with the following command:

set alternate on The command opens the file created with *set alternate to* so that it can receive the output directed to the screen.

```
set alternate to test.txt ↵
set alternate on ↵
a='Linguistics' ↵
?a function 'v1' ↵
```

To view the contents of *test.txt*, type:

```
close all ↵
modify file test.txt ↵
```

difference(*string1*,*string2*) This function compares the sound patterns of *string1* with *string2* and the return value is a similarity scale from 1 to 4, with 4 having the highest similarity. It mainly compares consonants; it's not case sensitive and ignores non-alphabetic characters within *string1* and *string2*.

```
?difference('foxpro', 'f') ↵
1

?difference('foxpro', 'fox') ↵
2

?difference('foxpro', 'foxp') ↵
3

?difference('foxpro', 'foxpr') ↵
4
```

soundex(*string*) This function evaluates the sound patterns, mainly that of consonants, of a string and returns the pattern in the form of the first letter of the string and a number. It can be used to extract words with similar pronunciation from a text or a wordlist. This function is not case sensitive and ignores non-alphabetic characters including spaces within the string.

```
?soundex('foxpro') ↵
F216

?soundex('FoxPro') ↵
F216

?soundex('fox234_p?r@o') ↵
F216

?soundex('foxpr') ↵
F216

?soundex('foxp') ↵
F210

?soundex('fox') ↵
F200

?soundex('fo') ↵
F000

?soundex('f') ↵
F000
```

Now use *wordlist* and enter the following:

```
copy to temp for soundex(word)=soundex('cut') ↵
```

Words with similar sound patterns are copied to *temp*.

run [*externalcommand*] [*externalprogram*] This function executes external commands or programs inside Foxpro. Now use *awordlist* in *d:\fox\practice* and copy it to *d:\practice\temp* and enter the following:

```
set defa to d:\fox\practice ↵
run rename temp.dbf testrun.dbf ↵
```

The above statement executes the DOS *rename* command from within Foxpro, and *temp.dbf* is renamed *testrun.dbf*. The exclamation mark *!* can do the same as *run*.

cd *path* This command changes directory. If we are now in *d:\fox\practice*, to change to *d:\fox\texts*, type:

```
cd d:\fox\texts ↵
```

curdir() This function returns the current directory. Now type:

```
?curdir() ↵
```

quit This command shuts down Foxpro.

5.6 Application

In this section we'll write three programs for application in language processing using arrays, procedures, functions and the functions and commands we learned in 5.4.

5.6.1 Simulation of LNRE

LNRE (Large Number of Rare Events) refers to the phenomenon that, contrary to our intuition, in samples of natural language, whatever their sizes, a large percentage of words are hapax legomena; the number of words in the vocabulary of a language seems inexhaustible. In a mega-corpus like the BNC, its word frequency distribution is still in the LNRE zone. We have computed the vocabulary growth of the 100-million-word BNC at a 100000-word interval, but at the right end of the vocabulary growth curve it's still on the rise, as shown in Figure 5.1.

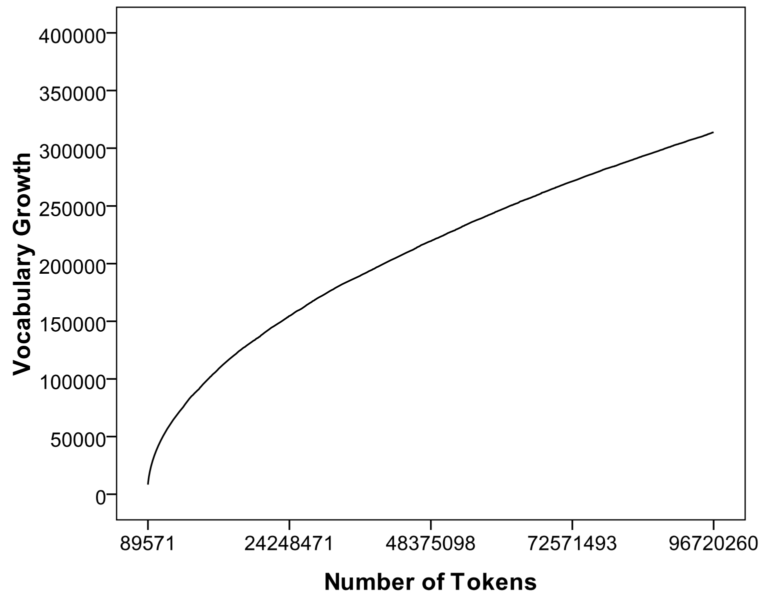


Figure 5.1 Vocabulary growth of the BNC at a 100000-word interval

According to Kornai, this lexical inexhaustiveness of a language is contributed by the infinite number of proper names, foreign words, typos, numeral-noun combinations and those generated by productive morphological processes. In theory, at a given point in time, the vocabulary of a language must be finite; only it's impossible for us to collect all the instances of the language use, written or spoken, at that particular time. However, we can simulate this theoretical linguistic situation in which we can collect all the instances of language use at a particular point of time. We now assume the 2,615 word types in *alice.txt* are the entire set of vocabulary of a language at a given point in time, and their frequencies as their actual occurrences in the language at that given point of time. We'll write a program to continuously draw replaceable samples of 50 words randomly from the 26,636 word tokens of *alice.txt*, assuming these small samples are the instances of language use of the language. We'll compute the vocabulary growth as the number of the random samples increases, and see how many samples have to be drawn before the vocabulary growth goes beyond the LNRE zone, and the vocabulary growth becomes zero, suggesting that all the 2,615 word types have been sampled. The program is as follows:

lnre.prg

```
set defa to d:\fox\practice
set talk off
clos data
```

```
set deci to 18
set safe off
creat cursor alicetokens(word c(25),freq n(4))
creat cursor wordtable(word c(25))
creat table simulatelnre(tokennum n(10), vocgrowth n(6))
tokenincrease=0
nothing=''
carriage=chr(13)
spaces=chr(32)
textinput=filetostr('d:\fox\texts\alice.txt')
textinput=strtran(textinput,'-',spaces)
textinput=chrtran(textinput,',.`[?]_”!:;()*',nothing)
textinput=strtran(textinput,”’”,nothing)
textinput=strtran(textinput,spaces,carriage)
textinput=proper(textinput)
strtofile(textinput,'temp.txt')
select 1
append from temp.txt sdf for word<>spaces
tokennumber=reccou()
copy to array randarray
rand(-34)
select 2
do while recc()<2615
vocsize1=reccou()
for i=1 to tokennumber
randarray(i,2)=rand()
endfor
asort(randarray,2)
appe from array randarray for recn()<vocsize1+51
inde on word tag word
total to temp on word
zap
appe from temp
vocsize2=reccou()
tokenincrease=tokenincrease+50
sele 3
appe blan
repl tokennum with tokenincrease
repla vocgrowth with vocsize2
@10,50 say 'the vocabulary size now is: ' get vocsize2
sele 2
enddo
```

In this program, statement 4 sets the decimal place to 18 for the purpose of generating multi-digit random numbers to reduce the possibility of generating identical random numbers. Statement 9 initializes the variable *tokenincrease*, which holds the simulated cumulative number of tokens repeatedly drawn from *alice.txt*. Statement 21 appends all the word tokens of *alice.txt* to a temporary table *alicetokens*, which is open in work area 1. Statement 22 measures the total number of word tokens in *alicetokens*. Statement 23 copies the contents of *alicetokens* to a two-column array *randarray*. Statement 24 maximizes randomness by putting a negative number in the *rand()* function. The LNRE simulation is between statement 25 and the end of the program. Statement 25 accesses *wordtable*, which continuously appends random samples, each 50 words in size, drawn from *randarray* and computes the vocabulary growth. Statement 26 checks whether the entire set of vocabulary of *alice.txt* has been exhausted. Statement 27 measures the vocabulary size before a new sample is appended. Statements 28—32 put random numbers to the second column of *randarray*, sort it to randomize the sequence of the words in the first column of *randarray*. The logic behind it is as follows. If we want to randomize the word sequence of the first 25 words of *alice.txt*, We first put the 25 words in a two-column array with multi-digit random numbers in the second column as follows:

Alices	*0.6758961889427160*
Adventures	*0.7731172381900250*
In	*0.2714417849201710*
Wonderland	*0.0945006739348170*
Chapter	*0.0143621934112160*
I	*0.4986768574453890*
Down	*0.2750961391720920*
The	*0.0994334695860740*
Rabbit	*0.1057327471207830*
Hole	*0.9042704734019940*
Alice	*0.9436191015411170*
Was	*0.9195503368973730*
Beginning	*0.8418385328259320*
To	*0.6537958388216790*
Get	*0.2603731083218010*
Very	*0.1618356527760620*
Tired	*0.4136778663378210*
Of	*0.4023807174526160*
Sitting	*0.9838708948809650*
By	*0.3818097133189440*
Her	*0.7362824191804980*
Sister	*0.1399497785605490*
On	*0.7782076245639470*
The	*0.4119589095935230*

Bank	*0.9448720037471500*

By sorting the random number column, the sequence of the words in the first column is randomized:

Chapter	*0.0143621934112160*
Wonderland	*0.0945006739348170*
The	*0.0994334695860740*
Rabbit	*0.1057327471207830*
Sister	*0.1399497785605490*
Very	*0.1618356527760620*
Get	*0.2603731083218010*
In	*0.2714417849201710*
Down	*0.2750961391720920*
By	*0.3818097133189440*
Of	*0.4023807174526160*
The	*0.4119589095935230*
Tired	*0.4136778663378210*
I	*0.4986768574453890*
To	*0.6537958388216790*
Alices	*0.6758961889427160*
Her	*0.7362824191804980*
Adventures	*0.7731172381900250*
On	*0.7782076245639470*
Beginning	*0.8418385328259320*
Hole	*0.9042704734019940*
Was	*0.9195503368973730*
Alice	*0.9436191015411170*
Bank	*0.9448720037471500*
Sitting	*0.9838708948809650*

Statement 32 appends 50 randomized words from *randarray*, and statements 33—37 measure the vocabulary growth after new random samples are added and identical words combined. Statement 38 increases the number of word tokens by 50. The vocabulary growth and the cumulative number of tokens are then appended to *simulatelnre* in statements 39—42.

The result of the LNRE simulation is displayed in Figure 5.2, which shows the entire LNRE zone of the vocabulary growth curve. The asymptotic property of the growth curve doesn't appear until around 100,000 tokens, nearly half of the entire language use, after which the curve still creeps upwards, though very, very slowly, until the cumulative number of tokens reaches 224,150, the total number of tokens of the entire language use of the hypothetical language at a given point of time.

5.6.2 Lemmatization

Lemmatization, according to Sinclair, is the process of gathering word-forms and turning them into lemmas. A lemma in turn is a set of lexical forms having the same stem, the same major part-of-speech, and the same word-sense. For example, words such as *go*, *goes*, *going*, *went*, *gone* and *conference*, *conferences* and so on are different word forms of the lemma *go* and *conference*. Automatic lemmatisation with high accuracy is difficult to achieve. For example, a computer lemmatisation program used in Chujo's study on vocabulary levels of English textbooks and tests had only a 45% accuracy).

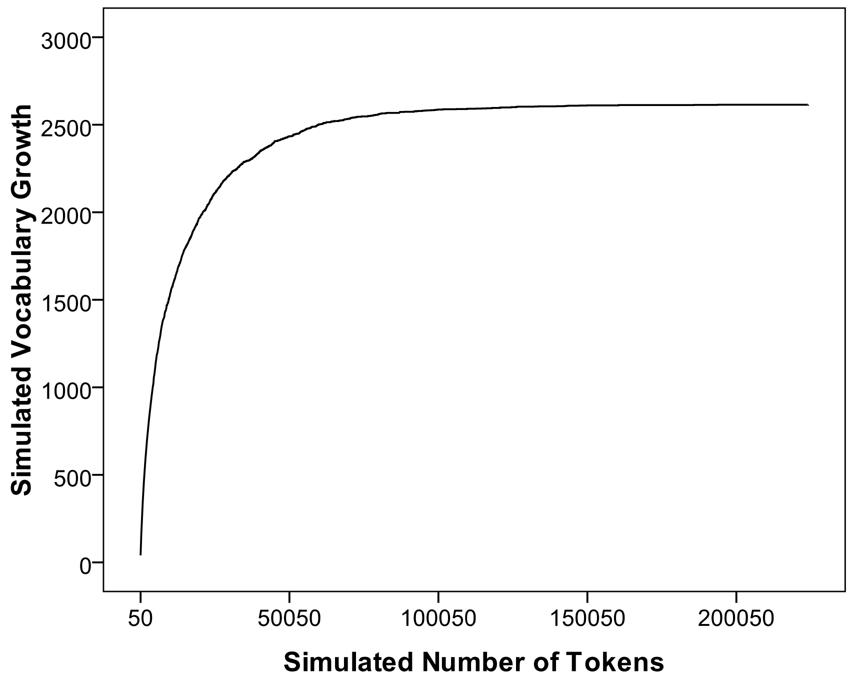
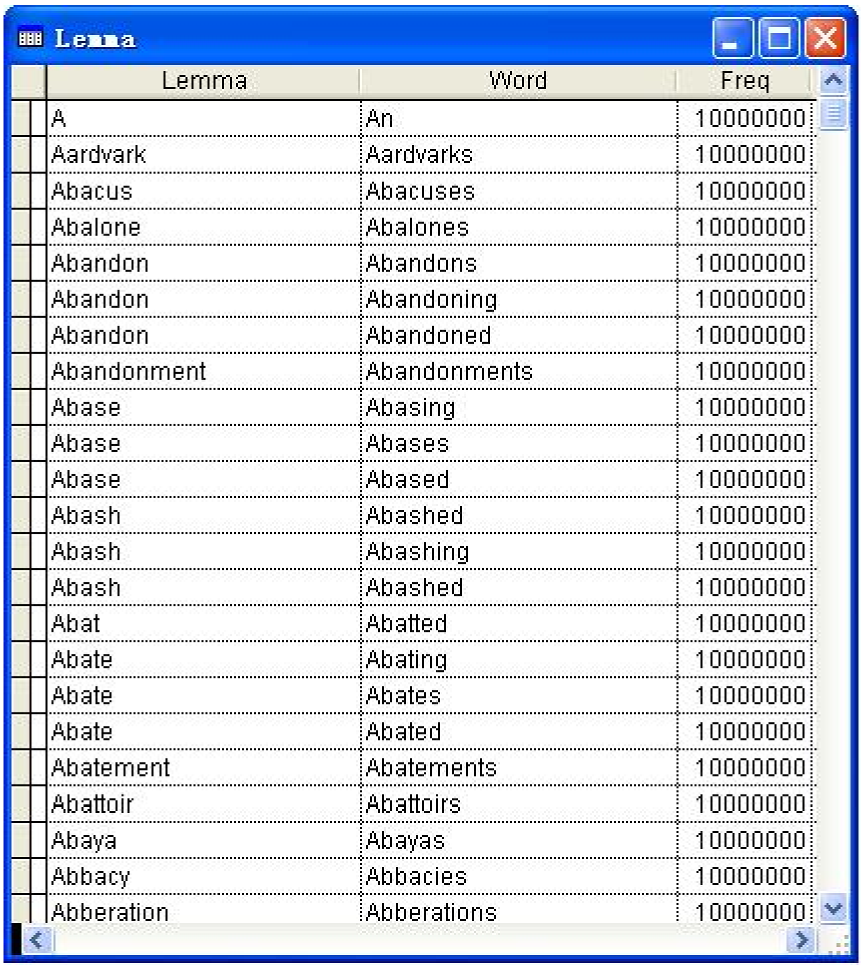


Figure 5.2 Simulation of LNRE

One way of lemmatization is by the use of a list of word forms with their corresponding lemmas. In *d:\fox\table3* there is such a wordlist table *lemma* containing 47,529 word forms and their corresponding lemmas. Figure 5.3 is part of the table. If we append an unlemmatized frequencied wordlist containing words such as *Abased*, *Abashing* etc, to the word field of the table, after totaling, identical word forms in the word field will be combined together, and their frequency will be greater than 10,000,000. For example, if the frequency of *Abased* and *Abashing* in the unlemmatized wordlist is respectively 3 and 7, now they'll become 10,000,003 and 10,000,007. Lemmatization is done by replacing the words in the word field whose frequency is greater than 10,000,000 with the lemmas in the lemma field. The words from the unlemmatized wordlist are very easy to pick out because their frequency is not 10,000,000.

The following short program uses *lemma* to lemmatize *awordlist* in *d:\fox\practice* created in 2.5.1.


Lemma

Lemma	Word	Freq
A	An	10000000
Aardvark	Aardvarks	10000000
Abacus	Abacuses	10000000
Abalone	Abalones	10000000
Abandon	Abandons	10000000
Abandon	Abandoning	10000000
Abandon	Abandoned	10000000
Abandonment	Abandonments	10000000
Abase	Abasing	10000000
Abase	Abases	10000000
Abase	Abased	10000000
Abash	Abashed	10000000
Abash	Abashing	10000000
Abash	Abashed	10000000
Abat	Abatted	10000000
Abate	Abating	10000000
Abate	Abates	10000000
Abate	Abated	10000000
Abatement	Abatements	10000000
Abattoir	Abattoirs	10000000
Abaya	Abayas	10000000
Abbacy	Abbacies	10000000
Abberation	Abberations	10000000

Figure 5.3 Part of the table *lemma*

alicelemma.prg
```
1. set default to d:\fox\practice
2. close data
3. set safety off
4. create cursor lemmatization(lemma c(50),word c(50),freq n(10))
5. append from d:\fox\table3\lemma
6. append from awordlist
7. index on word tag word
8. total to temp on word
9. zap
10. append from temp
11. replace all word with lemma for freq>10000000 && lemmatize words
    of awordlist
```

```
12. replace all freq with mod(freq,10000000) for freq>10000000 &&Return
    the lemmatized words to their real frequency
13. copy to alicelemma fields word,freq for freq<10000000 &&pick out the
    words from alwordlist
14. use alicelemma
15. index on word tag word
16. total to temp on word &&combine lemmatized words
17. zap
18. append from temp
19. brow
```

The lemmatization rate of this program is not very high for large unlemmatized wordlists since it can’t lemmatize word forms that *lemma* doesn’t have. For example, the word *wag* and *wags* in *awordlist* are not combined because there aren't such words in *lemma*.

Here we’ll use the comparison algorithm in a lemmatization program. The algorithm is similar to the Porter Stemmer algorithm, except that it uses a list of 33,818 common lemmas and a list of inflexional word endings. Lemmatisation is done in an alphabetically sorted wordlist by comparing a word with its following word. If the word compared is identical with the comparing word except for the ending, the ending is checked against a list of inflexional word endings stored in *wordending.txt*. If a match is found in the list of endings, the comparison succeeds and the word compared is replaced by the comparing word. If the comparing word has an inflectional ending, such as *ed*, *ing*, etc, the ending is removed before comparison, and if the comparison is successful, the word compared is replaced by the comparing word. For example, if the comparing word is *debated*, and the compared words are *debating* and *debates*, *ed* of *debated*, *ing* of *debating* and *es* of *debates* are removed and then compared; the comparison would be successful because the words are identical after the removal of the endings and these endings can find matches in *wordending.txt*. After the comparison, *debating* and *debates* are replaced by *debated*, instead of *debat* because *debat* is not a word. This means some of the words with inflexional endings can not be returned to their normalized form, but are replaced by the preceding word with an inflectional ending. However, if there are normalized forms for such words in the lemma list, they will be returned to their normalized forms. Words such as *replated*, *replating*, *replates* whose normalized form is not included in the lemma list are all lemmatised into *replated*. As for irregular verbs, nouns with irregular plural endings and some adjectives that have irregular comparative degree and superlative degree forms, the program uses a list of such verbs, nouns and adjectives and their normalized forms for their lemmatization. These words are stored in *irr_stopword.txt*, which also contains words or non-words that can cause problems during lemmatization. For example, if there is a typo such as *achiev* in a wordlist, and its following word is *achieved*, then the following word would be lemmatized into *achiev*. Words that can cause problems of this kind are put in *irr_stopword.txt* to prevent such errors in the lemmatization process. The user can add more words that can cause problems in *irr_stopword.txt*. The following is the program. Since it has more than 200 statements, explanatory notes are used between or after some statements for easy reading. The main program is from statement 1 to statement 13. There are five procedures and a user-defined function.

getlemma.prg

```
*This program can be used within a program or as a stand alone program
*for lemmatization. The first and second fields of the table to be
*lemmatized must be fields holding words and frequency respectively.
*The frequency field MUST not be empty. Data in other fields will be *lost.
This program uses three text files: irr_stopword.txt, wordending.txt
*and lemma.txt. They should be put in the same folder with the program.
1.  set safe off
2.  set talk off
3.  set exact off
4.  clear
*The following detects the work area of the table to be lemmatized and
*assigns it to workarea_1.
5.  workarea_1=alltr(str(sele()))
*The following three statements respectively detect and assign the name *of
the table to be lemmatized, the first two field names to tablename,
*wordfield and freqfield.
6.  tablename=dbf()
7.  wordfield=field(1)
8.  freqfield=field(2)
9.  @10,30 say 'lemmatizing &tablename now. please wait...'
10. totalize(wordfield) &&this function indexes and totals wordfield of
    tablename
11. do irregular &&this procedure gets the stop words and lemmatizes
    irregular words
12. do regular &&this procedure lemmatizes words with normal endings
13. @10,30 say 'lemmatization of &tablename is completed. All the
    lemmatized words are stored in lemma_log.dbf.'
*The following procedure lemmatizes words with irregular endings and
*stops certain words that might cause lemmatization errors.
14. procedure irregular
15. public workarea_2,workarea_3,workarea_4
*lem_temp is a temporary table for lemmatization.
16. creat table lem_temp(lemma c(60),&wordfield c(60), &freqfield
    n(20),freq_1 n(4),freq_2 n(4))
```

```
*Get the work area of lem_temp.
17. workarea_2=alltr(str(sele()))
*lemma_log keeps the lemmatized words and their corresponding
*unlemmatized forms for checking after the program completes
*lemmatization.
18. creat tabl lemma_log (lemma c(60),&wordfield c(60),&freqfield
    n(20),freq_1 n(4),freq_2 n(4))
*Get the work area of lemma_log.
19. workarea_3=alltr(str(sele()))
*The table word_ending holds word endings such as s, es, ed, ing, ies ,ect.
20. creat table word_ending(wordend1 c(14),wordend2 c(14),note c(40))
*Get the work area of the table word_ending.
21. workarea_4=alltr(str(sele()))
*wordending.txt is a text file containing different types of inflectional
*endings.
22. appe from wordending.txt deli with ','
23. sele &workarea_2&& access table lem_temp
*irr_stopword.txt contains irregular words and those words that may result
*in errors. It has two columns separated by a comma. The first column
*contains lemmas, which will be appended to the lemma field of
*lem_temp; the second their corresponding word forms, which will be
*appended to the word field. For stop words, both the first and the second
*column contain the same form.
24. appe from irr_stopword.txt deli with ','
25. repl all freq_1 with 1
*Append words from the table to be lemmatized to the field wordfield.
26. appe from &tablename
*The following mark the newly appended words in wordfield from
*tablename with 1 in freq_2.
27. repl all freq_2 with 1 for freq_1=0
28. totalize(wordfield)
*After totalling, those words in wordfield identical with those in lemma
*have freq_1=1 and freq_2=1. These words are to be lemmatized and
*stopped.
29. sele &workarea_3&& access lemma_log, which is currently empty
*Append words from lem_temp whose freq_1 and freq_2 are both 1 but
*lemma<>&wordfield to keep record of those words lemmatized. Words
*with freq_1=1 and freq_2=1 and lemma=wordfield are stop words and
*should not be included in lemma_log.
30. appe from lem_temp fiel lemma,&wordfield,&freqfield,freq_1,freq_2
    for freq_1>0 and freq_2>0 and lemma<>&wordfield &&prevent stop
    words such as achiev, we, she from being appended as lemmatized
    words.
```

```
31. sele &workarea_2&&lem_temp.
*The following lemmatize the irregular words.
32. repl all &wordfield with lemma for freq_1>0 and freq_2>0
33. totalize(wordfield)
*The following mark all the lemmatized words and stop words from
*&tablename with four *'s to exclude them from the following
*lemmatization process for regular words.
34. repl all &wordfield with alltr(&wordfield)+'****' for freq_1>0 and
    freq_2>0
35. sele &workarea_1&&the main table to be lemmatized, tablename.
36. zap&&remove the old contents
*Get its words back from lem_temp minus the lemmatized irregular words
*and stop words.
37. appe from lem_temp for '*'$&wordfield=.f. and &freqfield>0
*Append from lemma.txt.
38. appe from lemma.txt sdf
*The following is for putting words like accompany before accompanied
*in sorting so as to lemmatize the latter with the former since 2# is smaller
*than any letter. We can use any characters smaller than any letters, such
*as 0*, 1^.
39. repl all &wordfield with strtr(&wordfield,'y','2#')
40. totalize(wordfield)
41. dele tag all
*Change 2# back to y.
42. repl all &wordfield with strtr(&wordfield,'2#','y')
43. return
*The following procedure lemmatizes words with regular endings.
44. procedure regular
45. go top
46. do while .not. eof()
47. recordpointer=recno()&&get position of record pointer.
48. worda=alltrim(&wordfield)
49. do getwordendinga&&get the ending of first word
50. skip
51. wordb=alltrim(&wordfield)
52. do getwordendingb &&get the ending of the following word
*The following cases are for lemmatizing words with different types of
*endings.
53. do case
54. case wordb=worda and wordawordend1<>'e' and wordawordend2<>'ed'
    and wordawordend1<>'y' and wordalen>2 or wordaend3='eed'
55. do while wordb=worda
56. if wordawordend1=='s'
```

```
57. endtype='x'&&the x types of ending include words ending in ch, x, s, z
    sh and so on
58. wordslen=len(worda)
59. frequency=&freqfield
60. do lemmatize
61. else
62. if wordawordend1=='x'
63. endtype='x'
64. wordslen=len(worda)
65. frequency=&freqfield
66. do lemmatize
67. else
68. if wordawordend1=='z'
69. endtype='x'
70. wordslen=len(worda)
71. frequency=&freqfield
72. do lemmatize
73. else
74. if wordawordend2=='ch'
75. endtype='x'
76. wordslen=len(worda)
77. frequency=&freqfield
78. do lemmatize
79. else
80. if wordawordend2=='sh'
81. endtype='x'
82. wordslen=len(worda)
83. frequency=&freqfield
84. do lemmatize
85. else
86. if wordawordend1==left(wordbends,1)&&words such as
    acquit,acquitted
87. endtype='0' && endtype='0' means words not ending in e, y, ing, ed, es,
    f, fe, ying, ied or the x type of ending such as x, s, ch, sh, etc
88. wordslen=len(worda)+1
89. frequency=&freqfield
90. do lemmatize
91. else
92. if wordawordend1=='o' and wordbwordend2<>'os'&&for potatoes, but
    not photos.
93. endtype='e'
94. wordslen=len(worda)
95. frequency=&freqfield
```

```
96.  do lemmatize
97.  else&&words such as work,works, photo, photos.
98.  endtype='0'
99.  wordslen=len(worda)
100. frequency=&freqfield
101. do lemmatize
102. endif
103. endif
104. endif
105. endif
106. endif
107. endif
108. endif
109. enddo
110. case wordb=worda1 and (wordawordend1=='e' or
     wordawordend1=='y' or wordawordend1=='f')and worda1len>2
111. do while wordb=worda1
112. if wordawordend1=='e'
113. endtype='e'
114. wordslen=len(worda1)
115. frequency=&freqfield
116. do lemmatize
117. else
118. if wordawordend1=='y'
119. endtype='y'
120. wordslen=len(worda1)
121. frequency=&freqfield
122. do lemmatize
123. else
124. if wordawordend1=='f'
125. endtype='f'
126. wordslen=len(worda1)
127. frequency=&freqfield
128. do lemmatize
129. endif
130. endif
131. endif
132. enddo
133. case wordb=worda2 and (wordawordend2=='ed' or
     wordawordend2=='fe' or wordawordend2=='es') and
     wordaend3<>'ied' and worda2len>2
134. do while wordb=worda2
135. if wordawordend2=='ed'
```

```
endtype='ed'
wordslen=len(worda2)
frequency=&freqfield
do lemmatize
else
if wordawordend2=='fe'
endtype='fe'
wordslen=len(worda2)
frequency=&freqfield
do lemmatize
else
if wordawordend2=='es'
endtype='es'
wordslen=len(worda2)
frequency=&freqfield
do lemmatize
endif
endif
endif
enddo
case wordb=worda3 and (wordaend3=='ied' or wordaend3=='ing')
and worda3len>2
do while wordb=worda3
if wordaend3=='ied'
endtype='ied'
wordslen=len(worda3)
frequency=&freqfield
do lemmatize
else
if wordaend3=='ing'
endtype='ing'
wordslen=len(worda3)
frequency=&freqfield
do lemmatize
endif
endif
enddo
case wordb=worda4 and wordaend4=='ying' and worda4len>2
do while wordb=worda4
endtype='ying'
wordslen=len(worda4)
frequency=&freqfield
do lemmatize
```

```
enddo
endcase
if recordpointer<reccount()
go recordpointer+1
endif
enddo
sele &workarea_1&&tablename
dele all for &freqfield=0&&remove those words from lemma.txt
pack
*Get back those lemmatized irregular words and stop words.
appe from lem_temp for '*'$&wordfield and &freqfield>0
repl all &wordfield with strtr(&wordfield,'*','')&&remove *.
totalize(wordfield)
sele &workarea_3&&lemma_log
dele all for &freqfield=0
pack
sort to temporary_table on lemma
zap
appe from temporary_table
selec &workarea_1&&tablename
return
procedure lemmatize
sele &workarea_4
*Searching for corresponding ending.
loca for right(wordb,wordblen-wordslen)==alltr(wordend2) and
    alltr(wordend1)==endtype
*In the following statements, if a match is hit, the lemmatized word is put
*in lemma_log in workarea_3.
if found() and right(wordb,wordblen-wordslen)==alltr(wordend2)
sele &workarea_3&&lemma_log.
appe blan
repl lemma with worda
repl    &wordfield with wordb&&the word form before
    lemmatization.
repl &freqfield with frequency
sele &workarea_1&&tablename
*The following is the most important statement, which does the
*lemmatization.
repl &wordfield with worda
endif
sele &workarea_1
if not eof()
skip
```

```
wordb=alltrim(&wordfield)
do getwordendingb
endif
return
procedure getwordendinga
*Declaring public variables whose values are recognized both in the main
*program and sub-programs.
public wordalen,worda1len,worda2len,worda3len,worda4len,worda1,
    worda2,worda3,worda4, wordaend,wordawordend1, wordawordend2,
    wordaend3,wordaend4
wordalen=len(worda)
wordawordend1=right(worda,1)
wordawordend2=right(worda,2)
wordaend3=right(worda,3)
wordaend4=right(worda,4)
worda1=left(worda,wordalen-1)
worda1len=len(worda1)
worda2=left(worda,wordalen-2)
worda2len=len(worda2)
worda3=left(worda,wordalen-3)
worda3len=len(worda3)
worda4=left(worda,wordalen-4)
worda4len=len(worda4)
return
procedure getwordendingb
public wordbends, wordblen, wordb1len, wordb2len, wordb3len,
    wordb4len, wordb1, wordb2, wordb3, wordb4, wordbwordend1,
    wordbwordend2, wordbend3,wordbend4
wordblen=len(wordb)
wordbends=right(wordb,wordblen-wordalen)
wordbwordend1=right(wordb,1)
wordbwordend2=right(wordb,2)
wordbend3=right(wordb,3)
wordbend4=right(wordb,4)
wordb1=left(wordb,wordblen-1)
wordb1len=len(wordb1)
wordb2=left(wordb,wordblen-2)
wordb2len=len(wordb2)
wordb3=left(wordb,wordblen-3)
wordb3len=len(wordb3)
wordb4=left(wordb,wordblen-4)
wordb4len=len(wordb4)
return
```

```
function totalize
parameters fieldx
repl all &fieldx with prop(&fieldx)
index on &fieldx tag &fieldx
total to temporary_table on &fieldx
zap
append from temporary_table
return
```

Now save the program in *d:\fox\practice* and copy *irr_stopword.txt*, *wordending.txt* and *lemma.txt* to this folder from *d:\fox\texts*. Use *bnc3* in *d:\fox\table2*, copy it to *d:\fox\practice\test* and type the following:

```
set defa to d:\fox\practice ↵
use test ↵
do getlemma ↵
```

The program will start running. To check the lemmatization result, type:

```
close data ↵
use lemma_log ↵
brow ↵
```

The words in the *word* field are those before lemmatization, and those in the *lemma* field are their lemmatized forms. To view the lemmatized table, type:

```
use test ↵
brow ↵
```

Attention should be paid to the following when using this program:
1. Some lemmatization schemes make distinction in parts of speech of the words to be lemmatized. The verb *man* and the noun *man* are regarded as two different words. This program doesn't make such distinctions.
2. In this program, words like *saw*, a tool with toothed edge, is regarded as the past tense of *see*, and is lemmatized into *see*. To prevent cases like this, such words should be marked before lemmatization.
3. If a text has words like *brownbagged*, *brownbagging* and *brownbags* but not *brownbag,* they will be lemmatized into *brownbagged* since *lemma.txt* doesn't have the normalized form *brownbag*. This is not a problem for just one text. But if we lemmatize more than one text for, say, lexical comparison, this can result in inaccuracy, especially for small texts. For example, if *texta* has *brownbag* and *brownbagging*, and *textb* has *brownbagged* and *brownbags*, *brownbagging* in

texta will be lemmatized into *brownbag*, while *brownbags* in *textb* will be lemmatized into *brownbagged*. To prevent this from happening, in lemmatizing more than one text or table, first put all the texts or tables together and lemmatize them. Then lemmatize them again one by one using *lemma_log* the way *alicelemma.prg* does. This way words of the same stem in these texts or tables will have the same lemmatized form.

5.6.3 Extracting lexical information from multiple texts or tables

In this section, we'll look at a program that can extract lexical information from multiple texts or tables, such as vocabulary growth, the growth of words with frequency between 1—15, and the vocabulary size and number of words with frequency between 1—15 in each of the texts or tables, etc. The program uses the 50 tables *bncst1*, *bncst2…bncst50* in *d:\fox\table2*. Each of the tables contains word types with POS tags from a text chunk about 2,000 words in length randomly sampled from the BNC spoken text section. The program is as follows:

lexinfo.prg

```
*This program extracts lexical information from multiple tables as these
*tables are sampled and put together one by one, such as vocabulary
*growth, the growth of words with frequency 1 to 15, vocabulary size of
*individual tables and their respective number of words with frequency 1
*to 15.
1.  set defau to d:\fox\practice
2.  set safety off
3.  set talk off
4.  clos data
5.  clear
6.  set deci to 16&&generating random numbers for randomizing the
     sequence of tables
7.  starttime=time()&&get the starting time
*The following creates lexinfo1 for holding table names, number of word
*tokens, table vocabulary size and the number of words with frequency
*1—15 in each table
8.  create table lexinfo1(tablename c(30),tabletoken n(6),tablevoc n(4),;
    f1 n(5),f2 n(5),f3 n(5),f4 n(5),f5 n(5),f6 n(5),f7 n(5),f8 n(5),f9 n(5),;
    f10 n(5),f11 n(5),f12 n(5),f13 n(5),f14 n(5),f15 n(5))
*The following creates lexinfo2 for holding the same set of table names as
*in lexinfo1, cumulative number of tokens as these tables are sampled and
*put together one by one, vocabulary growth, and the growth of words
*from frequency 1—15.
9.  creat table lexinfo2(tablename c(30), cumutoken n(8),vocgrowth;
    n(6), fgrowth1 n(5),fgrowth2 n(5),fgrowth3 n(5), fgrowth4 n(5),;
    fgrowth5 n(5), fgrowth6 n(5), fgrowth7 n(5),fgrowth8 n(5),fgrowth9;
    n(5), fgrowth10 n(5),fgrowth11 n(5), fgrowth12 n(5), fgrowth13 n(5),;
    fgrowth14 n(5),fgrowth15 n(5))
*temp1 is for removing the POS tags and calculating number of tokens
*and vocabulary size of individual tables.
10. creat table temp1(word c(25),freq n(8))
*temp2 is for calculating cumulative number of tokens, vocabulary growth
*and the growth of words from frequency 1—15.
11. creat table temp2 (word c(25),freq n(8))
12. close databases
13. nothing=''
14. spaces=chr(32)
*Put the 50 bnc tables bncst1—bncst50 in array bnctable.
15. adir(bnctable,'d:\fox\table1\bncst*.dbf')
16. rand(-45) &&maximize randomness.
17. for i=1 to 50
18. bnctable(i,2)=rand() &&replace the second column with random
     numbers
19. endfor
*Sort the second column of array bnctable to randomize the order of the
*50 tables.
20. asort(bnctable,2)
21. sele 1
22. use temp1
23. for i=1 to 50
24. if mod(i,10)=0 &&send message to screen at an interval of 10
25. @10,70 say 'number of tables processed. ' get i
26. endif
*Assign table names held in the first column of the array bnctable to
*tabletoappend.
27. tabletoappend='d:\fox\table1\'+bnctable(i,1)
28. appe from &tabletoappend
29. repl all word with subs(word,1,at(spaces,word)) &&remove POS tags
30. totalize('word','freq') &&the function totalize() totals words in temp1
31. sum freq to tokennumber &&get number of tokens of individual table
*Measure vocabulary size of individual tables.
32. vocsize=recc()
33. sele 2
34. use lexinfo1
*The following statement mainly gets words of frequency 1--15 of
*individual tables and appends it to lexinfo1 using the function getfreq().
35. getfreq('freq','f','tabletoken','tablevoc')
```

```
sele 1
use temp2
append from temp1
*The following computes vocabulary growth and the growth of words
*with frequency 1—15.
totalize('word','freq')
sum freq to tokennumber &&cumulative number of tokens
vocsize=reccount() &&vocabulary growth
select 2
use lexinfo2
*The following function mainly computes growth of words with
*frequency 1—15.
getfreq('freq','fgrowth','cumutoken','vocgrowth')
select 1
use temp1
zap
endfor
?starttime&&output starting time to screen
?time()&&output ending time to screen
function totalize
parameters wordfield,freqfield
inde on &wordfield tag &wordfield
total to temp on &wordfield
zap
appe from temp
return
function getfreq
parameters freqfield,freqfieldx,tokens,vocabulary
select 2
append blan
select 1
for ii=1 to 15
count to freqnumber for &freqfield=ii
sele 2
fld=freqfieldx+alltr(str(ii))
repl &fld with freqnumber
repl tablename with tabletoappend
repl &tokens with tokennumber&&individual token
repl &vocabulary with vocsize
sele 1
endfor
return
```

Information obtained with this program is very useful in quantitative linguistic research. Figure 5.4 displays the vocabulary growth and the growth of words with frequency between 1 to 15 of the 50 tables. Figure 5.5 displays the relationship between vocabulary size and the number of words occurring 1—4 times in the 50 tables.

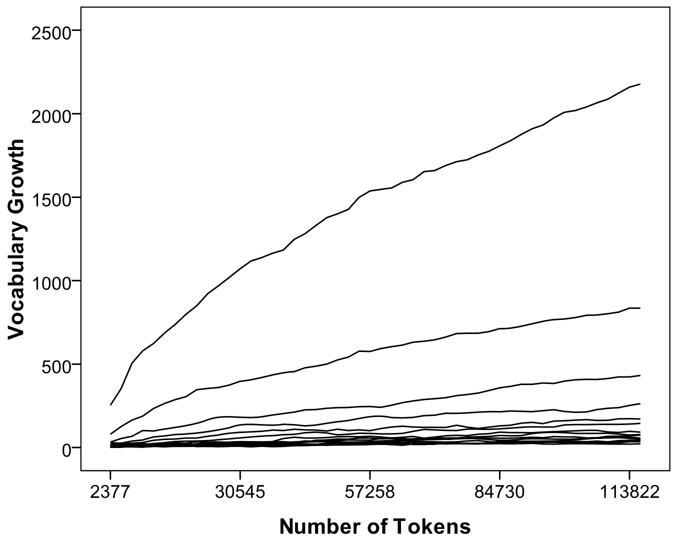


Figure 5.4 Vocabulary growth and the growth of words with frequency between 1 to 15 of the 50 tables

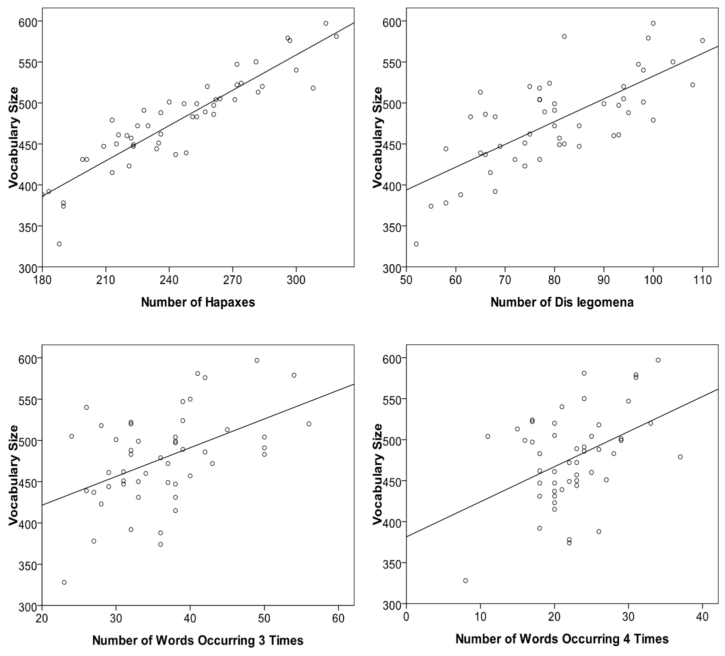


Figure 5.5 The relationship between vocabulary size and the number of words occurring 1—4 times in the 50 tables. The small circles are vocabulary sizes of the individual tables.

5.6.4 Extracting information on word class distribution

Quirk et al classifies English words into the following word classes: noun, adjective, full verb, adverb, preposition, pronoun, determiner, conjunction, modal verb, primary verb, numeral and interjection. The distributions of word classes tend to be different in different registers, for example, in spoken English and written English. In *d:\fox\table2* there are 50 wordlist tables with POS tags, named from *bncwt1* to *bncwt50*. These wordlists were made from 50 2000-word samples randomly drawn from the written text section of the BNC. We'll write a program to calculate the number of words belonging to the different word classes in each of the tables, as well as the cumulative number of words belonging to different word classes as the tables are put together one by one. The BNC uses a set of 57 POS tags because it subcategorizes words of the same class. We'll ignore these fine distinctions and reduce these POS tags to *NN*, noun; *VV*, verb; *AJ*, adjective; *AV*, adverb; *DT*, determiner; *PN*, pronoun; *PRP*, preposition; *CJ*, conjunction; *VM0*, modal auxiliary; *CRD*, numeral; *NP0*, proper noun; *TO0*, the infinitive marker *to*; *ITJ*, interjection; *UNC*, unknown category. We'll combine *AT*, article, *DPS*, possession pronoun, into *DT*; *ORD*, ordinal number, into *CRD*; *PRF*, *of*, into *PRP*; *XX0*, *not*, into *AV*; and *EX0*, existential *there*, into *PN*. The program is as follows.

wordclass.prg

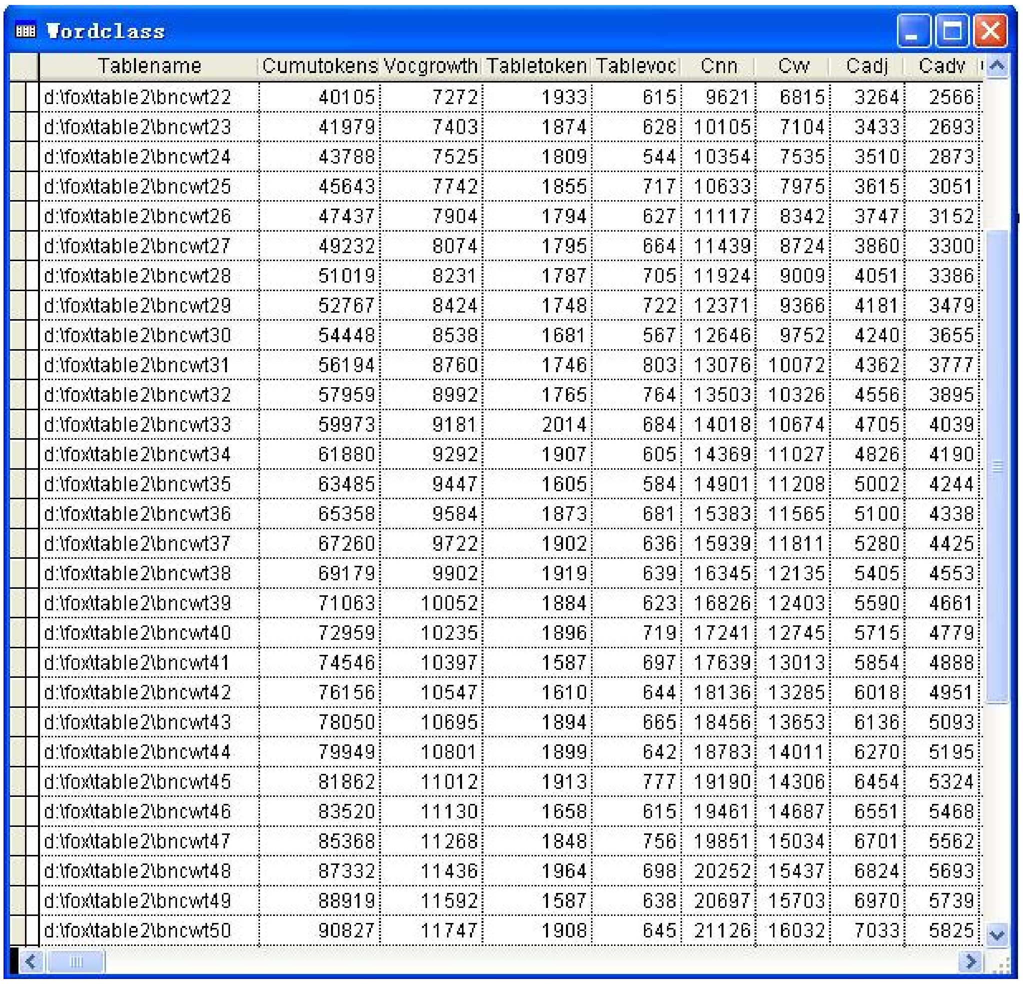
```
*This program calculates vocabulary growth, cumulative number of
*nouns, verbs and so on of 50 BNC tables, and those of individual tables.
1.  set defa to d:\fox\practice
2.  clos data
3.  set safe off
4.  set talk off
5.  clear
*In wordclass, tablename, cumutokens, vocgrowth, tabletoken, tablevoc
*are respectively for table names, cumulative number of tokens,
*vocabulary growth, number of tokens in each table, vocabulary size of
*each table. The rest of the fields are for number of words belonging to a
*particular word class. Those with c at beginning are for cumulative
*number of words belonging to a class. For example, cnn means
*cumulative number of nouns, while nn refers to number of nouns in each
*of the 50 tables.
6.  creat table wordclass(tablename c(30),cumutokens n(8),vocgrowth;
      n(6),tabletoken n(4),tablevoc n(4),cnn n(8),cvv n(8),cadj n(8),cadv;
      n(8),cdet n(8),cpron n(8),cprep n(8),ccj n(8),cmodl n(8),ccrd n(8), ;
      cnp0 n(8),ct00 n(6),cintj n(6),cunc n(6),nn n(4),vv n(3),adj n(3),adv;
      n(3),det n(3),pron n(3),prep n(3),cj n(3),modl n(3),crd n(3), np0 n(3),;
      t00 n(3),intj n(3),unc n(3))
```

```
*Create a 33-element array holding the above fields that are now empty.
7.  scatter to wordclassarray
8.  creat table temp1(word c(40), freq n(8),marker n(2))&&for loading the
     50 tables one at a time.
9.  creat table temp2(word c(40),freq n(8),marker n(2))&&for calculating
     vocabulary growth.
10. creat table temp3(word c(40),freq n(8),marker n(2))&&for cumulative
     number of words belonging to different classes
*Assign word class tags to postag1.
11. postags1='NN,V,AJ,AV,DT,PN,PRP,CJ,VM0,CRD,NP0,TO0,ITJ,UNC
     ,'
*Assign pairs of POS tags to be combined to postag2. The first tag of a
*pair is to be replaced with the second tag of the pair.
12. postags2='XX0 AV,AT DT,DPS DT,ORD CRD,PRF PRP,XX0
     AV,EX0 PN,'
*Create an array to hold the 14 word classes NN, V, AJ etc.
13. dimen posarray1(14)
14. for i=1 to 14
*Cut the POS tags one at a time from postag1 and put it to the array
*posarray1.
15. posarray1(i)=subs(postags1,1,at(',',postags1)-1)
16. postags1=stuff(postags1,1,at(',',postags1),'')
17. endfor
*Create an array for storing the POS tags to be combined.
18. dimen posarray2(7)
19. for i=1 to 7
*Cut each POS tag pairs from postag2 and put it to the array posarray2.
20. posarray2(i)=subs(postags2,1,at(',',postags2)-1)
21. postags2=stuff(postags2,1,at(',',postags2),'')
22. endfor
23. cumutokennumber=0&&initialize cumutokennumber
*Process the 50 tables one by one.
24. for i=1 to 50
25. tablename='d:\fox\table2\bncwt'+alltr(str(i))
*The following sends message to the screen at a 10-table interval.
26. if mod(i,10)=0
27. @10,70 say 'Number of tables processed: ' get i
28. endif
29. select 2&&temp1
30. appe from &tablename
31. sum freq to tabletoken&&tokens of each table.
*Get cumulative number of tokens.
32. cumutokennumber=cumutokennumber+tabletoken
```

```
*Put table name and cumulative number of tokens to the first element and
*the second element of wordclassarray.
33. wordclassarray(1)=tablename
34. wordclassarray(2)=cumutokennumber
*The following replace AT with DT, XX0 with AV etc. When ii=1,
*posarray2 (1) has the POS tag pair 'XX0 AV' and subs (posarray2(ii),;
*1,at(' ',posarray2(ii))-1) gets XX0, and stuff(posarray2(ii),1,at(' ‘ , ;
posarray2(ii)),'') gets AV.
*The former is replaced by the latter.
35. for ii=1 to 7
36. repla all word with strtr(word,subs(posarray2(ii),1,at('
    ',posarray2(ii))-1),stuff(posarray2(ii),1,at(' ',posarray2(ii)),''))
37. endfor
38. selec 3&&temp2, for voc growth.
39. appe from temp1
40. repl all word with left(word,rat(' ',rtrim(word))) for freq>0&&remove
    POS tags
41. totalize('word')
*Get vocabulary growth.
42. vocincrease=recc()
43. count to tablevocsize for freq>0&&get individual vocabulary size
*Store vocabulary growth, tokens, vocabulary size of each table to
    wordclassarray.
44. wordclassarray(3)=vocincrease
45. wordclassarray(4)=tabletoken
46. wordclassarray(5)=tablevocsize
47. repl all freq with 0&&for loading the next table
48. selec 4&&temp3, for cumulative number of word classes
49. appe from temp1
50. totalize('word')
*The following keeps record of how many elements in wordclassarray;
*have been loaded.
51. element=5
*Function getwordnumber sums the numbers of words belonging to a ;
*word class and stores them in wordclassarray.
52. getwordnumber('freq','word','posarray1','wordclassarray','marker')
*The marker field is for showing whether a table is newly loaded or old.;
*Words whose marker field is 1 are newly loaded.
53. repl all marker with 1
54. selec 2&&temp1.
*The following keeps records of how many elements in wordclassarray;
*have been loaded.
55. element=19
```

```
56. getwordnumber('freq','word','posarray1','wordclassarray','marker')
57. selec 1
58. appe blan
*Append the contents of the 33-element array to the 33 fields of
*wordclass.
59. gather from wordclassarray
60. selec 2&&temp1
61. zap&&empty it for the next table.
62. endfor
63. select 1&&access table wordclass.
*cvv includes cmodle. So the number of cmodl must be subtracted from it.
64. repl all cvv with cvv-cmodl
*The same as above.
65. repl all vv with vv-modl
66. function totalize
67. parameters wordfield
68. inde on &wordfield tag &wordfield
69. tota to temp on &wordfield
70. zap
71. appe from temp
72. return
73. function getwordnumber
74. parameters freqfield,wordfield,posarray,classarray,markfield
*The following removes words but keeps the POS tags.
75. repl all &wordfield with stuff(&wordfield,1,at(' ',&wordfield),") for
     &markfield=0
*The following removes words like "as" in "as well as AV0" but keeps AV0
76. repl all &wordfield with stuff(&wordfield,1,rat(' ',rtrim(&wordfield)),")
     for &markfield=0
77. repl all &wordfield with stuff(&wordfield,at('-',&wordfield),10,")    for
     '-'$&wordfield and &markfield=0&&remove hyphen and the tag after it
*The following statements get the number of words belonging to each of;
*the 14 word classes. The variable element contains the number of;
*elements wordclassarray stores, so new information should be stored;
*from element+ii.
78. for ii=1+element to 14+element
79. sum &freqfield to wordclassnumber for &wordfield =
     &posarray(ii-element)&&posarray(ii) has elements from 1 to 14!
80. &classarray(ii)=wordclassnumber
81. endfor
82. return
```

Part of the result is shown in Figure 5.6.


Wordclass

Tablename	Cumutokens	Vocgrowth	Tabletoken	Tablevoc	Cnn	Cw	Cadj	Cadv
d:\fox\table2\bncwt22	40105	7272	1933	615	9621	6815	3264	2566
d:\fox\table2\bncwt23	41979	7403	1874	628	10105	7104	3433	2693
d:\fox\table2\bncwt24	43788	7525	1809	544	10354	7535	3510	2873
d:\fox\table2\bncwt25	45643	7742	1855	717	10633	7975	3615	3051
d:\fox\table2\bncwt26	47437	7904	1794	627	11117	8342	3747	3152
d:\fox\table2\bncwt27	49232	8074	1795	664	11439	8724	3860	3300
d:\fox\table2\bncwt28	51019	8231	1787	705	11924	9009	4051	3386
d:\fox\table2\bncwt29	52767	8424	1748	722	12371	9366	4181	3479
d:\fox\table2\bncwt30	54448	8538	1681	567	12646	9752	4240	3655
d:\fox\table2\bncwt31	56194	8760	1746	803	13076	10072	4362	3777
d:\fox\table2\bncwt32	57959	8992	1765	764	13503	10326	4556	3895
d:\fox\table2\bncwt33	59973	9181	2014	684	14018	10674	4705	4039
d:\fox\table2\bncwt34	61880	9292	1907	605	14369	11027	4826	4190
d:\fox\table2\bncwt35	63485	9447	1605	584	14901	11208	5002	4244
d:\fox\table2\bncwt36	65358	9584	1873	681	15383	11565	5100	4338
d:\fox\table2\bncwt37	67260	9722	1902	636	15939	11811	5280	4425
d:\fox\table2\bncwt38	69179	9902	1919	639	16345	12135	5405	4553
d:\fox\table2\bncwt39	71063	10052	1884	623	16826	12403	5590	4661
d:\fox\table2\bncwt40	72959	10235	1896	719	17241	12745	5715	4779
d:\fox\table2\bncwt41	74546	10397	1587	697	17639	13013	5854	4888
d:\fox\table2\bncwt42	76156	10547	1610	644	18136	13285	6018	4951
d:\fox\table2\bncwt43	78050	10695	1894	665	18456	13653	6136	5093
d:\fox\table2\bncwt44	79949	10801	1899	642	18783	14011	6270	5195
d:\fox\table2\bncwt45	81862	11012	1913	777	19190	14306	6454	5324
d:\fox\table2\bncwt46	83520	11130	1658	615	19461	14687	6551	5468
d:\fox\table2\bncwt47	85368	11268	1848	756	19851	15034	6701	5562
d:\fox\table2\bncwt48	87332	11436	1964	698	20252	15437	6824	5693
d:\fox\table2\bncwt49	88919	11592	1587	638	20697	15703	6970	5739
d:\fox\table2\bncwt50	90827	11747	1908	645	21126	16032	7033	5825

Figure 5.6 Part of the table wordclass

Exercises

1. Write a short program *alicearray.prg* to create an array of 2,615 rows and two columns and put in it one at a time the 2,615 words and their frequency in *awordlist*. Sort the second column (the one holding frequency) in descending order. Create a table temp and append the contents of *alicearray* to it.

2. Rewrite the program *awordlist.prg* in 2.5.1. Turn statements 9—16 into a procedure called tokenizer, and statements 17—20 into a function totalize(). Save it as *awordlistb.prg*.

3. In his *Word Frequency Distributions*, Baayen compares word frequency distributions with the outcomes of casting a fair die repeatedly. The former is a LNRE phenomenon while the latter is not. It doesn't take many throws before the six possible outcomes have all appeared. Write a program to simulate 50 throws

of a fair die, putting the outcome in a table of each throw and the number of different outcomes as the number of throws increases. Use an array for the die throw simulation.

4. In taking samples from a number of different texts, we should randomize the starting sampling point within a text to prevent sampling from the same position, say, the beginning of a text. Use the 48 text chunks in *d:\fox\texts* as the source texts and write a program to draw 48 samples with a length of about 150 words from each of the texts. The samples should be drawn from four different positions within the source text.

5. In 2.5.1 there are three programs *awordlist.prg*, *lwordlist.prg* and *compare.prg*. Now write one program that can do the same things the three programs do. The program should have a function that turns *alice.tx*t and *lglass.txt* into two lemmatized tables (use the program *getlemma.prg*), and a procedure that compares the two tables, picking out the shared words and unique words of the two tables.

6. Write a program to do the following: put the names of the 50 tables in *fox\table2* from bncwlem1 to bncwlem50 into an array, turn them into 25 pairs randomly and calculate the number of words each of the pairs shares; create a table and put the names of the tables of each pair, the vocabulary size of each of the tables and the number of words they share into it. Use an array where possible in the program.

6 Interactive Programming, Program Packaging and Foxpro Graphs

Interactive programming refers to writing programs that interact with the user. Such programs can ask for task-related information from, or send such information to the user during program execution. For a bulky program that needs supporting files, i.e. the lemmatization program, we can put the program and the supporting files together in a package to make it portable so that it can be easily moved around from one folder to another, or from one computer to another. Foxpro has a graph wizard, which can turn data contained in a table into different types of graph. This can be done by using Foxpro's graph wizard following a few simple fool-proof directions. In this chapter we'll learn how to do all the above.

6.1 Writing Interactive Programs

6.1.1 Commands for keyboard input

accept [*string*] **to** *variable*   This command prints *string* on the screen and stores the input by the user through the keyboard in *variable*.

```
accept 'This program tokenizes a text. Please specify the name of the file: 'to
filename ↵
d:\fox\texts\alice.txt ↵
?filename ↵
d:\fox\texts\alice.txt
```

Now type the following. After completing the first statement, move the down key to start a new line and type the second statement. Highlight the two statements and press Enter:

```
accept 'What is your age?: ' to age
?'Your age is: ' +age
```

The computer pauses for input from the user after it executes the first statement. When a number is entered followed by Enter, the computer executes the second statement and the number inputted by the user is printed on the screen.

input [*string*] **to** *variable*   This command does the same thing as the *accept* command except that the input entered through the keyboard must have quotation marks on either side unless it's a number.

```
input 'This program tokenizes a text. Please specify the name of the file: ' to
filename ↵
d:\fox\texts\alice.txt ↵
variable 'D' is not found

'd:\fox\texts\alice.txt' ↵
?filename ↵
d:\fox\texts\alice.txt

input 'What is your age?: ' to age ↵
23 ↵

?age ↵
23
```

6.1.2 Application

Now we'll write an interactive program that tokenizes a text file and makes a wordlist for it. In addition, it also does the following:

1. asks the user for the name and the path of the text file to be processed;
2. if the file doesn't exist, the program asks the user to re-enter the source file;
3. asks the user for the name and the path of the table for storing the wordlist;
4. tells the user where the results are stored.

The program is as follows:

interactive.prg

```
*The following statements interact with the user, asking for the path, and
*the name of the source text, the name of output text file etc. If the source
*text does not exist or wrongly entered, the program will ask the user to
*re-enter until it finds the source text.
1. @10,30 say 'This program tokenizes a text and turn it into a frequencied
   wordlist.'
2. accept space(30)+'Please specify the path of the source text and its name
   with file extension (e.g. d:\fox\practice\text1.txt): ' to sorcetext
3. clear
4. @10,30 say 'The source text is: '+sorcetext
5. clear
6. do while file(sorcetext)=.f.
7. @10,30 accept space(30)+sorcetext+ ' does not exist. Please reenter the
   name of the source text. Press ESCAPE to exit: ' to sorcetext
8. clear
9. enddo
```

```
10. @10,30 say 'The source text is '+ sorcetext
11. accept space(30)+'Please specify the drive and folder for the result file
     (e.g., d:\fox\practice): ' to drivefolder
12. set defa to &drivefolder
13. accept space(30)+'Please specify the name of the table for the wordlist
     without file extension: ' to outputtable
14. set safe off
15. set talk off
16. close data
17. clear
18. create table temp1(word c(30),freq n(4))
19. nothing=''
20. spaces=chr(32)
21. carriage=chr(13)
22. textinput=filetostr('&sorcetext')
23. textinput=strtran(textinput,'-',spaces)
24. textinput=strtran(textinput,spaces,carriage)
25. strtofile(textinput,'temp.txt')
26. append from temp.txt sdf for word<>spaces
27. replace all word with chrtran(word,',.`[?]_”!:;()*',nothing)
28. replace all word with strtran(word,”’”,nothing)
29. replace all word with prop(word)
30. replace all freq with 1
31. index on word tag word
32. total to temp2 on word
33. zap
34. append from temp2 for word<>spaces
35. copy to &outputtable
36. clear
37. @11,40 say 'The result is stored in '+drivefolder+'\'+upper(outputtable)
```

In this program, statements 1—13 are for interaction between the user and the computer, while the rest are very similar to *awordlist.prg*.

6.2 Program Packaging

In 5.6.2 we wrote a program *getlemma.prg*. It’s very long and has three supporting files: *irr_stopword.txt*, *wordending.txt* and *lemma.txt*. It’s very inconvenient to move programs like *getlemma.prg* around from one folder to another or from one computer to another. To make such programs portable, we can put the program and the supporting files into one single package, which is called a project in Foxpro. The command to start packaging is as followings:

modify project *projectname* Suppose we want to put *getlemma.prg* and its three supporting files into a single package called *lemmatizer*, do the following. Type the following in the command window:

modify project lemmatizer ↵

The project manager window appears, as shown in Figure 6.1

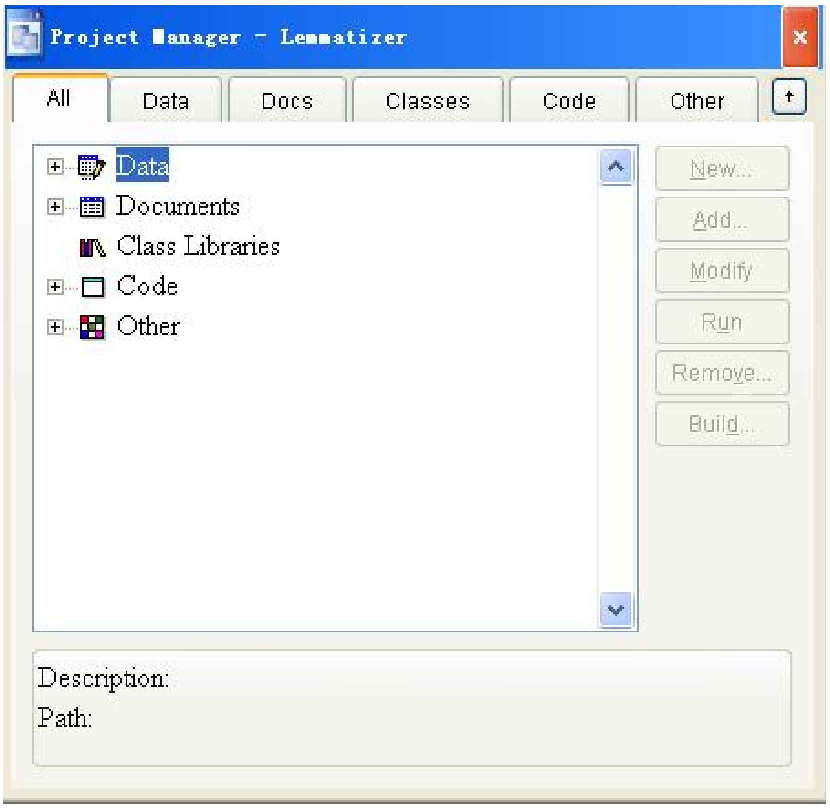


Figure 6.1. The Project Manager window

2. Click on all the + signs on the left of *Data*, *Documents*, *Class Libraries*, *Code* and *Other* to expand these items, as shown in Figure 6.2.
3. Click on *Programs* under *Code*, and then *Add*, select the program *getlemma.prg* and click on *OK*.
4. Click on *Text files* under *Other* and then *Add*, press the *Ctrl* key and select *irr_stopword.txt*, *wordending.txt* and *lemma.txt* and click on *OK*.
5. Click on *Build* and the following build options window appears, as shown in Figure 6.3. Select *Wind32 executable/COM server (exe)*, *Recompile All Files* and *Display Errors*, then click on *OK*. The program *getlemma.prg* and its supporting files *irr_stopword.txt*, *wordending.txt* and *lemma.txt* are packaged into a single stand-alone project called *lemmatizer* with the file extension *exe*. It has an icon—a little fox head. To lemmatize a table containing a word field and field,

put *lemmatizer.exe* in the folder where the table is open and type the following in the command window:

```
do lemmatizer ↵
```

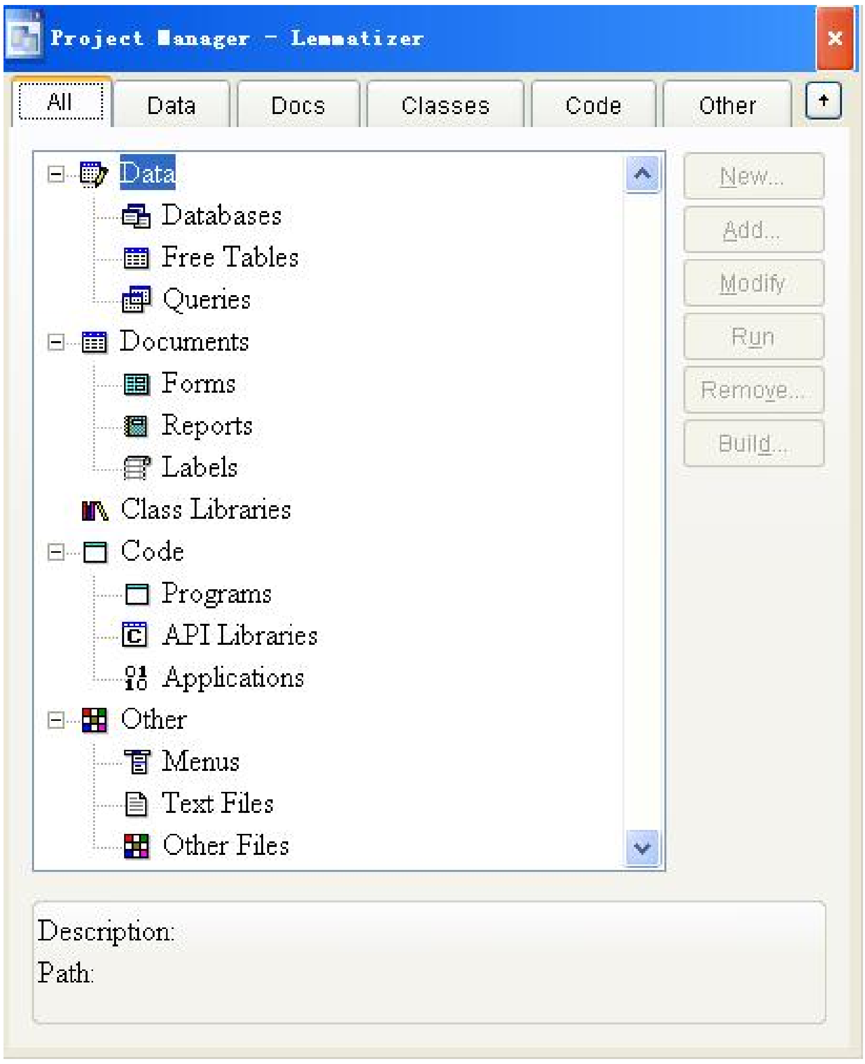
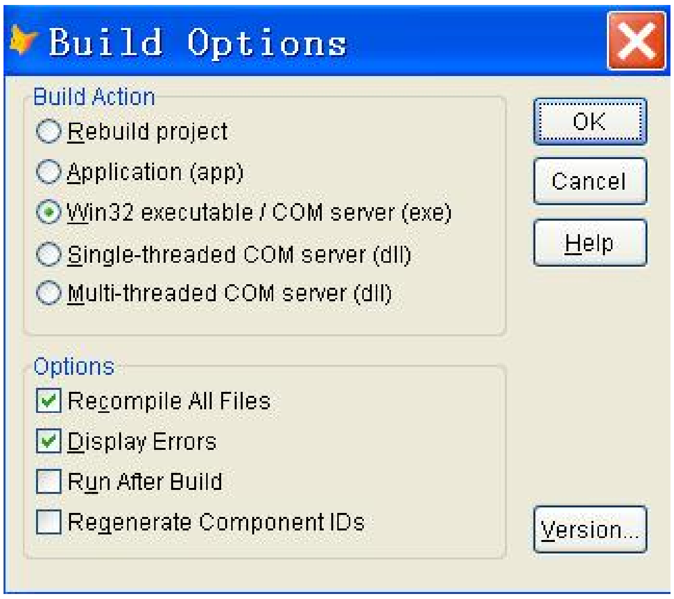


Figure 6.2 Expanded Project Manager window

This packaged program can also be used within another program by simply issuing the statement *do lemmatizer* in the program where it's needed. If *lemmatizer.exe* is not in the same folder of the table to be lemmatized, its path must be specified. For example, if the table to be lemmatized is in *d:\fox\practice*, but *lemmatizer.exe* is in *d:\fox\progs*, the following should be entered:

```
do d:\fox\progs\lemmatizer
```


Build Options

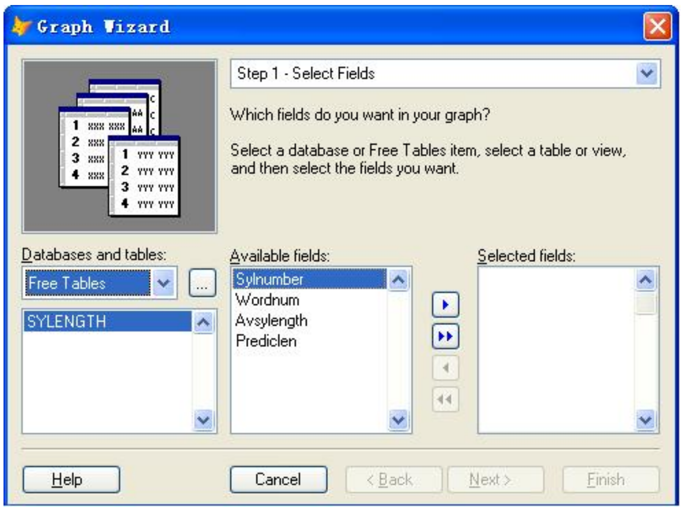
Build Action
- ◯ Rebuild project
- ◯ Application (app)
- ● Win32 executable / COM server (exe)
- ◯ Single-threaded COM server (dll)
- ◯ Multi-threaded COM server (dll)

Options
- ✓ Recompile All Files
- ✓ Display Errors
- ☐ Run After Build
- ☐ Regenerate Component IDs

Figure 6.3 The Build Options window

6.3 Foxpro Graphs

Foxpro has a graph wizard, which can turn data in a table into different types of graph using Microsoft Graph. We’ll use the table *sylength* in *d:\fox\practice* (produced by *syllable.prg* in 3.3.5) to make a pie chart of the distribution of word length in syllables. The procedure for making a graph is as follows:
1. Open the table *sylength*.
2. On the main Foxpro menu bar, select *Tools—Wizard—Query—Graph Wizard* and then click on *OK*. The *Select Fields* window appears as shown in Figure 6.4.


Graph Wizard

Step 1 - Select Fields

Which fields do you want in your graph?

Select a database or Free Tables item, select a table or view, and then select the fields you want.

Databases and tables: Free Tables

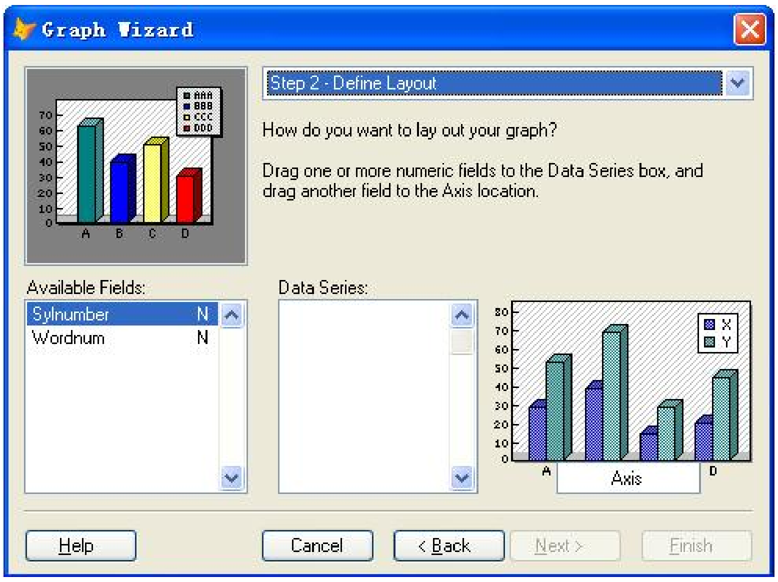
SYLENGTH

Available fields:
Sylnumber
Wordnum
Avsylength
Prediclen

Selected fields:

Figure 6.4 The Select Field window

Click on *Sylnumber* to highlight it and then move it to the *Selected Fields* box; do the same to *Wordnum*. Click on *Next* to get the *Define Layout* window, as shown in Figure 6.5.



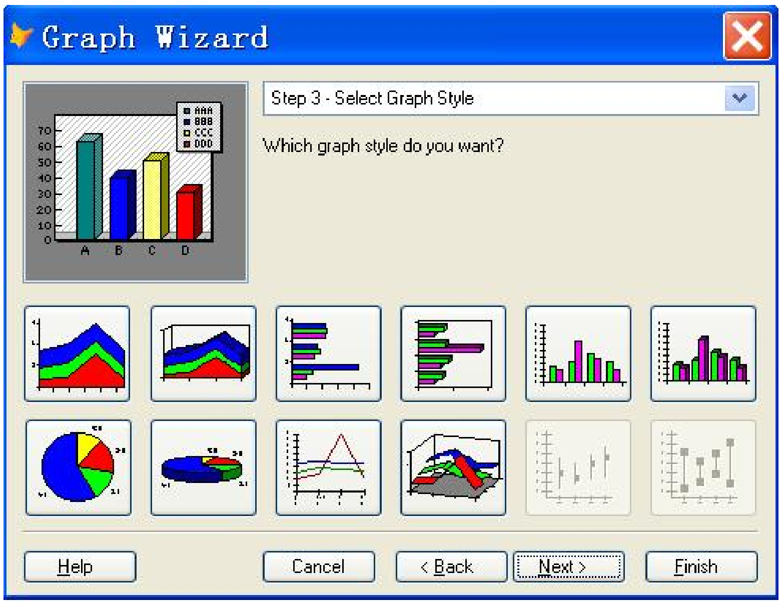
How do you want to lay out your graph?

Drag one or more numeric fields to the Data Series box, and drag another field to the Axis location.

Sylnumber	N
Wordnum	N

Figure 6.5 The Define Layout window

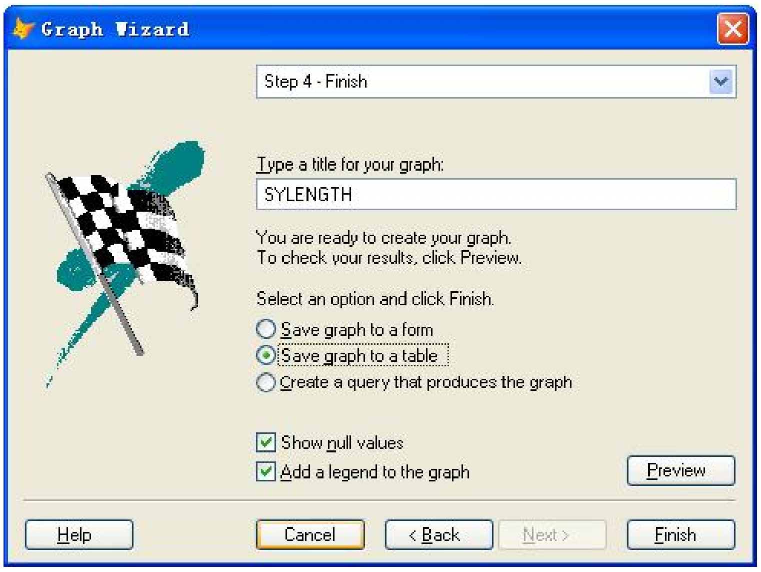
3. Point cursor at *Sylnumber* in the *Available Fields* box, hold down the left mouse button and drag *Sylnumber* to the *Axis* slot, and then drag *Wordnum* into the *Data Series* box. Click on *OK*. The *Select Graph Style* window appears, as shown in Figure 6.6.



Which graph style do you want?

Figure 6.6 The Select Graph Style window

Click on 3-D pie chart and then on *Next*, the Finish window appears, as shown in Figure 6.7.


Graph Wizard

Step 4 - Finish

Type a title for your graph: SYLENGTH

You are ready to create your graph.
To check your results, click Preview.

Select an option and click Finish.

- Save graph to a form
- [✓ Save graph to a table]
- Create a query that produces the graph

[✓] Show null values
[✓] Add a legend to the graph

Figure 6.7 The Finish window

4. Select *Save graph to a table, Show null values* and *Add a legend to the graph* and click on *Preview*. Click on *Return to Wizard* if the graph is satisfactory. Then click on *Finish* and save the graph in a table called *vfpgraph.dbf*, a default name provided by Foxpro. Of course we can give the table any other name as we wish. The graph is stored in the field *olegraph*, which is a general field. To use or edit the graph, open the table and double click on the general field, the graph appears. To put the graph in a word document, simply click on *Copy* in *Edit* on the Foxpro menu bar and paste it in the word document. It can be edited there as well. The 3-D pie chart of word length distribution is shown in Figure 6.8.

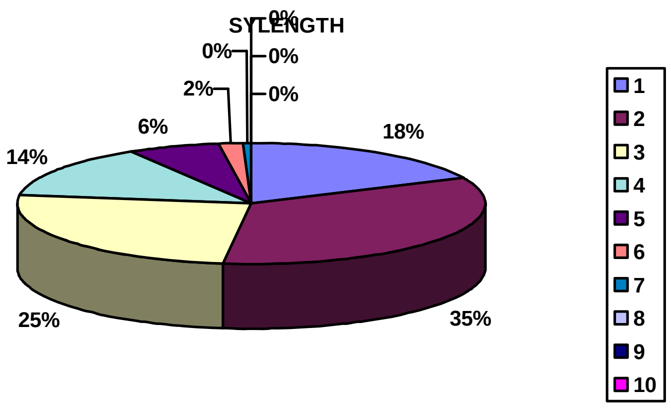


Figure 6.8 The 3-D pie chart of word length distribution

Microsoft Graph provides a variety of graphs to choose from, and each has several variations. Apart from the different types of graph, we can also change the colour, the heading, font style, line or bar style and so on of a graph. The reader can do them by selecting the appropriate items of the graph editing box.

Exercises

1. Write a short interactive program to make a wordlist for *lglass.txt* in *d:\fox\texts*. The program also does the following:

a. asks the user for the path and name of the file to be processed;
b. asks the user for the path and name of the file for storing the results;
c. turns the file into a frequencied wordlist;
d. tells the user where and in what file the result is stored.

2. In *d:\fox\progs* there is a programs *stemword.prg*, which can remove word derivational suffixes. It has the following supporting files: *stopword.txt*, *wordend.txt*, and *wordstem.txt*. These supporting files are all in *d:\fox\texts*. Package this program with its supporting files.

3. Use *wordclass* in *d:\fox\practice* produced by *wordclass.prg* in 5.6.4 and draw growth curves of nouns, verbs, adjectives and adverbs using Foxpro Graph Wizard.

4. In *d:\fox\texts* there is a text file *words.txt* containing 23,926 lemmas obtained from 500 2000-word random samples from the BNC written text section. Write a program called *bncletters.prg* to compute the frequency of the 26 English letters, put the letter frequency in descending order in a table called *letterfreq* and draw a bar chart of the distribution of the 26 letters.

Appendix

I. Model Answers to Selected Exercises

Exercises of Chapter 1

Exercise 3

a. ttr=100*1200/2000 ↵
   ?ttr ↵
   60

b. ttrx=(400 + 1200-(1000*1200)/2000)/2000 ↵
   ?ttrx ↵
   0.5

Exercise 4

The predicted number of *CN* formed with words of syllable length 1:
30.2693*1**-2.3212 = 30.2693.
The predicted number of *CN* formed with words of syllable length 2:
30.2693*2**-2.3212 = 6.0569.
The predicted number of *CN* formed with words of syllable length 3:
30.2693*3**-2.3212 = 2.3632.
The predicted number of *CN* formed with words of syllable length 4:
30.2693*4**-2.3212 = 1.213.
The predicted number of *CN* formed with words of syllable length 5:
30.2693*5**-2.3212 = 0.722.

Exercise 5

First set decimal to 6.

a. Add-one smoothing
   Smoothed probability of *inside out*:
   ?(3+1)/(23+13500) ↵
   0.00296
   Smoothed probability of *happy time*
   ?(2+1)/(45+13500) ↵
   0.000221

b. Good-Turing estimation
   Smoothed probability of *run rampant*
   ?(1+1)*2331/10043/145 ↵
   0.003201
   Smoothed probability of *strong tea*
   ?(3+1)*523/1125/76
   0.024466

Exercise 6

Use the following combined functions to do the transformation: *rtod (asin (sqrt( )))*. For example, to transform 12%, type:

```
rtod(asin(sqrt(0.12))) ↲
```

The result is 20.27. The rest of the data after the transformation are 22.79, 24.35, 26.57, 27.97, 31.31, 33.21, 35.67, 36.27, 38.65, 39.23, 41.55

Exercise 7

Tuldava:

```
v=1000000*2.71828**(-0.009152*log(1000000)**2.3057) ↲
?v ↲
20279.17083
```

Guiraud, Sánchez & Cantos:

```
v=65.7365677*sqrt(1000000) ↲
?v ↲
65736.5677
```

Exercise 8

a.
```
h=100*(log(98000)/(1-(3473/98000))) ↲
?h ↲
1191.4975
```

b.
```
h=100*(log(182000)/(1-(4336/182000))) ↲
?h ↲
1240.7357
```

Exercise 9

First, enter the following in the command window:

```
modify command arclength ↲
```

an empty file called *arclength.prg* appears. Enter the following in the file (note the carriage return after each semi-colon); save it and then run it by clicking on the red exclamation mark on the menu bar.

```
?((1635-872)**2+1)**(1/2)+((872-825)**2+1)**(1/2)+((825-730)**2+1)**;
(1/2)+((730-687)**2+1)**(1/2)+((687-540)**2+1)**(1/2)+((540-531)**2+;
1)**(1/2)+((531-528)**2+1)**(1/2)+((528-513)**2+1)**(1/2)+((513-410);
**2+1)**(1/2)+((410-398)**2+1)**(1/2)+((398-367)**2+1)**(1/2)+((367-;
364)**2+1)**(1/2)+((364-315)**2+1)**(1/2)+((315-274)**2+1)**(1/2)+(;
(274-263)**2+1)**(1/2)+((263-247)**2+1)**(1/2)+((247-211)**2+1)**(;
1/2)+((211-194)**2+1)**(1/2)+((194-182)**2+1)**(1/2)
1453.69
```

Exercise 10

```
?100938.3*2248**3-7754/5.56+(3400/2578)**(1/4)*(1102-331)**(12-8) ↲
```

1147065721994803

Exercises of Chapter 2

Exercise 3

```
use wordlist ↵
copy to temp for mod(recn(),2)=0 ↵
```

Exercise 4

```
use d:\fox\table3\wordlist ↵
copy to temp for word='Ex' ↵
copy to temp for right(alltrim(word),2)='ed' ↵
```

Exercise 5

```
create table zipf(word c(25),freq n(7),rank n(5)) ↵
append from wordlist field word,freq ↵
index on freq tag freq descending ↵
copy to temp ↵
zap ↵
append from temp ↵
replace all rank with recno()↵
```

Exercise 6

textchunk.prg

```
1.  set default to d:\fox\practice
2.  set safe off
3.  create table textchunk(names c(25),contents m(4))
4.  for i=1 to 48
5.  texts='d:\fox\texts\text'+alltrim(str(i))+'.txt'
6.  append blank
7.  replace names with texts
8.  append memo contents from &texts
9.  endfor
10. brow
```

Exercise 7

binomial.prg

```
1.  n=6
2.  r=3
3.  p=0.5
4.  n1=1
5.  r1=1
```

```
6.  nr=1
7.  for j=1 to n
8.  n1=n1*j
9.  endfo
10. for k=1 to r
11. r1=r1*k
12. endfo
13. for m=1to(n-r)
14. nr=nr*m
15. endfor
16. b=(n1/(nr*r1))*p**r*(1-p)**(n-r)
17. ?b
```

Exercise 8

Enter the following in the command window:

```
create table fit(tokens n(6),vocgrowth n(6),brunet n(6),herdan n(8,2),guiraud
n(8,2),orlov n(8,2))
append from d:\fox\practice\vocincrease field tokens,vocgrowth
replace all brunet with 0.03315956*log(tokens)**6.017229305
replace all herdan with 65.73656*tokens**0.4291
replace all guiraud with 24.706408821*sqrt(tokens)
replace all orlov with (132000*(log(132000)-log(tokens))*tokens)/((log (132000)
+1.483699120)*(132000-tokens))
```

Exercise 9

vocinfo.prg

```
1.  set default to d:\fox\practice
2.  set safe off
3.  close data
4.  clear
5.  set talk off
6.  nothing="
7.  carriage=chr(13)
8.  spaces=' '
9.  freqfield=nothing
10. for i=1 to 48
11. freqfield=freqfield+'freq'+alltrim(str(i))+' n(6),'
12. endfor
13. tablename='vocinfo(word c(25),'+'totalfreq n(6),'+'rng
    n(6),'+freqfield+'wlength n(4))'
14. create table &tablename
15. recordnumber=0
16. for i=1 to 48
```

```
17. texts='d:\fox\texts\text'+alltr(str(i))+'.txt'
18. frequency='freq'+alltrim(str(i))
19. textinput=filetostr('&texts')
20. textinput=strtran(textinput,'-',spaces)
21. textinput=strtran(textinput,spaces,carriage)
22. strtofile(textinput,'temp.txt')
23. append from temp.txt sdf
24. replace all &frequency with 1 for recno()>recordnumber
25. recordnumber=reccount()
26. endfor
27. delete all for word=spaces
28. pack
29. replace all word with chrtran(word,',.`[?]_”!:;()*',nothing)
30. replace all word with strtran(word,”'”,nothing)
31. replace all word with prop(word)
32. replace all totalfreq with 1
33. index on word tag word
34. total to temp on word
35. zap
36. append from temp
37. delete all for word=spaces
38. pack
39. replace all wlength with len(alltrim(word))
40. freqfield=nothing
41. for i=1 to 48
42. freqfield=freqfield+'round('+'freq'+alltr(str(i))+'/('+'freq'+alltr(str(i))+'+1),0)+'
43. endfor
44. freqfield=left(freqfield,len(freqfield)-1)
45. repl all rng with &freqfield
46. copy to temp field word,freq44 for freq44>0
```

Exercise 10

```
a. inde on right(alltrim(word),len(alltrim(word))-1) tag word
b. inde on right(alltr(word),1) tag word
```

Exercises of Chapter 3

Exercise 1

80incr.prg

```
1. clos data
2. set defa to d:\fox\practice
3. set safe off
```

```
set deci to 4
clear
addfield="&&no space between the quotes
ex=0
use d:\fox\table3\80vgrowth
copy to 80vgrowth
use 80vgrowth
alter table 80vgrowth add sdv n(8,4) add mean n(8,4) add lower n(8,4) add upper n(8,4)
do while not eof()
for i=1 to 80
addfield=addfield+'incr'+alltr(str(i))+'+'
endfor
addfield=left(addfield,rat('+',addfield)-1)
meangrowth=(&addfield)/80
for i=1 to 80
fieldname='incr'+alltr(str(i))
ex=ex+(&fieldname-meangrowth)**2
endfor
replace sdv with sqrt(ex/80)
replace mean with meangrowth
skip
addfield="&&empty addfield
ex=0
enddo
replace all lower with mean-1.96*sdv
replace all upper with mean+1.96*sdv
```

Exercise 2

First create a two-field table called *arclength*. Name the first field *fr*, 6 digits in width; the second *mathresult*, 10 digits in width, with 4 decimal places. Then input the rank frequencies in *fr* field. The program is as follows.

arclengthb.prg

```
close data
use arclength
do while not eof()
freqrankr=fr
skip
freqrankn=fr
compute=((freqrankr-freqrankn)**2+1)**(1/2)
replace mathresult with compute
enddo
sum mathresult to summathresult
```

```
?summathresult
```

Exercise 3

semiaux.prg

```
set default to d:\fox\practice
set safety off
clear
close data
create table semiaux(aux c(30),freq n(4))
create table sentence(sent1 c(250),sent2 c(250),sent3 c(250),sent m(4))
auxexpl=''&&no space between the quotes
number=0
tabs=chr(9)
linebreak=chr(10)
carriage=chr(13)
spaces=chr(32)
textinput=filetostr('d:\fox\texts\alice.txt')+filetostr('d:\fox\texts\lglass.txt')
textinput=strtr(textinput,carriage+linebreak,spaces)
textinput=strtr(textinput,spaces+spaces,carriage)
textinput=strtr(textinput,'.)',').'+carriage)
textinput=strtr(textinput,”!'”,”'!”+carriage)
textinput=strtr(textinput,”?'”,”'?”+carriage)
textinput=strtr(textinput,'.','.'+carriage)
textinput=strtr(textinput,'?','?'+carriage)
textinput=strtr(textinput,'!','!'+carriage)
strtofil(textinput,'temp.txt')
select 2
append from temp.txt sdf for sent1<>spaces
replace all sent with alltr(sent1+sent2+sent3)
select 1
append from d:\fox\texts\semiaux.txt sdf
go top
do while not eof()
auxiliary=lower(alltr(aux))
select sent from sentence where like('*'+auxiliary+'*',lower(sent)) into table temp
counter=reccount()
if counter>0
number=number+1
auxexpl=auxexpl+'('+alltr(str(number))+'). '+upper(auxiliary)+carriage
replace all sent with strtr(lower(sent),lower(auxiliary),'** '+upper(auxiliary)+' **')
go top
```

```
do while not eof()
getsemiaux=alltr(sent)
auxexpl=auxexpl+tabs+alltr(str(recn()))+'. '+getsemiaux+carriage
skip
enddo
auxexpl=auxexpl+carriage
endif
select 1
replace freq with counter
skip
enddo
strtofile(auxexpl,'result.txt')
modify file result.txt
```

Exercise 4

morethan.prg

```
set default to d:\fox\practice
set safety off
close data
morethantext=''
number=0
number=0
linebreak=chr(10)
carriage=chr(13)
spaces=chr(32)
create table sentence(sent1 c(250),sent2 c(250),sent3 c(250),sent m(4))
textinput=filetostr('d:\fox\texts\alice.txt')+filetostr('d:\fox\texts\lglass.txt')
textinput=strtr(textinput,carriage+linebreak,spaces)
textinput=strtr(textinput,spaces+spaces,carriage)
textinput=strtr(textinput,'.)',').'+carriage)
textinput=strtr(textinput,”!'”,”'!”+carriage)
textinput=strtr(textinput,”?'”,”'?”+carriage)
textinput=strtr(textinput,'.','.'+carriage)
textinput=strtr(textinput,'?','?'+carriage)
textinput=strtr(textinput,'!','!'+carriage)
strtofil(textinput,'temp.txt')
append from temp.txt sdf for sent1<>spaces
replace all sent with alltr(sent1+sent2+sent3)
select sent from sentence where like('*more*than *',lower(sent)) into table temp &&note the space between than and *
replace all sent with strtr(sent,'more', '**MORE')
replace all sent with strtr(sent,'More', '**MORE')
replace all sent with strtr(sent,'than', 'THAN**')
```

```
go top
do while not eof()
morethantext=morethantext+alltr(str(recn()))+'. '+alltr(sent)+carriage
skip
enddo
strtofile(morethantext,'more.txt')
modif file more.txt
```

Exercise 5

a. copy to new for like('*ship',alltr(word)) or like('*hood',alltr(word)) or like('*craft',alltr(word)) or like('*dom',alltr(word)) ↵
b. replace all word with replicate(' ',25/2-len(alltrim(word))/2)+alltr(word) ↵
c. replace all word with replicate(' ',25-len(alltrim(word)))+alltr(word) ↵
d. replace all word with alltrim(word) ↵

Exercise 6

likelihood_get

```
set default to d:\fox\practice
set safe off
set talk off
set decimal to 8
clear
create table wordtoken(word c(25),freq n(8))
create table kwictable (context c(120),freq n(5))
create table likehood (context c(25),freq1 n(4),freq2 n(4),lkhratio n(14,8))
close data
nothing=''
kwic=nothing
carriage=chr(13)
spaces=chr(32)
textinput=filetostr('d:\fox\texts\lglass.txt')
textinput=strtran(textinput,'-',spaces)
textinput =strtran(textinput,spaces,carriage)
strtofile(textinput,'temp.txt')
select 1
use wordtoken
append from temp.txt sdf for word<>spaces
n=reccount()&&the total number of word tokens, needed in likelihood ratio
go top
scan for lower(alltr(word))=='get' or lower(alltr(word))=='gets' or lower(alltr(word))=='getting' or lower(alltr(word))=='got'
replace word with upper(word)
keyword=alltrim(word)
```

```
skip -5
for i=1 to 11
kwic=kwic+alltrim(word)+spaces
skip
endfor
sele 2
use kwictable
append blank
keywordposition=at(keyword, kwic)
replace context with replicate(spaces,45-keywordposition)+kwic
kwic=nothing
sele 1
endscan
sele 2
inde on righ(context,75) tag context
copy to get.txt sdf field context
copy to temp
select 3
use temp
replace all context with strtr(context,left(context,45),nothing)
replace all context with strtran(context,left(context,at(spaces,context)),nothing)
replace all context with left(context,at(spaces,context))
select 3
use likehood
append from temp
replace all context with chrtr(context,'.,:;”()-`*[?]_!',nothing)
replace all context with strtr(context,”'”,nothing)
replace all context with proper(context)
replace all freq2 with 1
index on context tag context
total to temp on context
zap
append from temp
select 1&&access the table wordtoken
replace all freq with 1
index on word tag word
total to temp on word
zap
append from temp
replace all word with chrtr(word,'.,:;()-[?]_`*”!',nothing)
replace all word with strtr(word,”'”,nothing)
replace all word with prop(word)
```

```
inde on word tag word
total to temp on word
zap
append from temp for word<>spaces
sum freq to c1 for alltr(word)=='Get' or alltr(word)=='Gets' or alltr(word)=='Getting' or alltr(word)=='Got'
select 3 &&access the table likehood
go top
do while not eof()
getword=alltr(context)
select 1
locate for alltr(word)==getword
collocatefreq=freq
select 3
replace freq1 with collocatefreq
skip
enddo
select 3
dele all for freq1=0
pack
go top
do while not eof()
c12=freq2
c2=freq1
p=c2/n
p1=c12/c1
if c2-c12=0
p2=(c2+0.01-c12)/(n-c1)&&0.01 is added in cases c1=c2, p2 will be 0 and the program will crash!
else
p2=(c2-c12)/(n-c1)
endif
lkhvalue=log(p)*c12+log(1-p)*(c1-c12)+log(p)*(c2-c12)+log(1-p)*((n-c1)-(c2-c12))-log(p1)*c12-log(1-p1)*(c1-c12)-log(p2)*(c2-c12)-log(1-p2)*((n-c1)-(c2-c12))
repl lkhratio with lkhvalue*-2
skip
enddo
index on lkhratio tag lkhratio descending
brow
```

Exercise 7

t_test.prg

```
set default to d:\fox\practice
set safe off
set talk off
set decimal to 8
clear
create table wordtoken(word c(25),freq n(8))
create table kwictable (context c(120),freq n(5))
create table tablemake (context c(25),freq1 n(4),freq2 n(4),tvalue n(14,8))
close data
nothing=''
kwic=nothing
carriage=chr(13)
spaces=chr(32)
textinput=filetostr('d:\fox\texts\lglass.txt')
textinput=strtran(textinput,'-',spaces)
textinput =strtran(textinput,spaces,carriage)
strtofile(textinput,'temp.txt')
select 1
use wordtoken
append from temp.txt sdf for word<>spaces
n=reccount()&&the total number of word tokens, needed in likelihood ratio
go top
scan for lower(alltr(word))=='make' or lower(alltr(word))=='makes' or lower(alltr(word))=='making' or lower(alltr(word))=='made'
replace word with upper(word)
keyword=alltrim(word)
skip -4
for i=1 to 9
kwic=kwic+alltrim(word)+spaces
skip
endfor
sele 2
use kwictable
append blank
keywordposition=at(keyword, kwic)
replace context with replicate(spaces,40-keywordposition)+kwic
kwic=nothing
sele 1
endscan
sele 2
inde on righ(context,80) tag context
copy to make.txt sdf field context
copy to temp
```

43. select 3
44. use temp
45. replace all context with strtr(context,left(context,40),nothing)
46. replace all context with strtran(context,left(context,at(spaces,context)),nothing)
47. replace all context with left(context,at(spaces,context))
48. select 3
49. use tablemake
50. append from temp
51. replace all context with chrtr(context,'.,:;”()-`*[?]_!',nothing)
52. replace all context with strtr(context,”'”,nothing)
53. replace all context with proper(context)
54. replace all freq2 with 1
55. index on context tag context
56. total to temp on context
57. zap
58. append from temp
59. select 1&&access the table wordtoken
60. replace all freq with 1
61. index on word tag word
62. total to temp on word
63. zap
64. append from temp
65. replace all word with chrtr(word,'.,:;()-[?]_`*”!',nothing)
66. replace all word with strtr(word,”'”,nothing)
67. replace all word with prop(word)
68. inde on word tag word
69. total to temp on word
70. zap
71. append from temp for word<>spaces
72. sum freq to makefreq for alltr(word)=='Make' or alltr(word)=='Makes' or alltr(word)=='Making' or alltr(word)=='Made'
73. select 3 &&access the table likehood
74. go top
75. do while not eof()
76. getword=alltr(context)
77. select 1
78. locate for alltr(word)==getword
79. collocatefreq=freq
80. select 3
81. replace freq1 with collocatefreq
82. skip
83. enddo

```
select 3
dele all for freq1=0
pack
go top
do while not eof()
xbar=freq2/n
mu=(freq1/n)*(makefreq/n)
s2=xbar
repl tvalue with (xbar-mu)/sqrt(s2/n)
skip
enddo
inde on tvalue tag tvalue descending
brow
```

Exercise 9

hapdis.prg

```
set default to d:\fox\practice
set safe off
set talk off
close data
clear
use multitext
create table hapdislego (texts c(10),vocsize n(6,2),hapsize n(6,2),dissize n(6,2), hdratio n(6,2),mhlength n(6,2),mdislength n(6,2))
for i=1 to 48
select 1 &&access multitex open in work area 1
wordfield='text'+alltr(str(i))
freqfield='freq'+alltr(str(i))
count to hapaxnumber for &freqfield=1
count to disnumber for &freqfield=2
count to vocnumber for &freqfield>0
ratio=disnumber/hapaxnumber
average wlength to meanhaplength for &freqfield=1
average wlength to meandislength for &freqfield=2
sele 2&&access texthapax open in work area 2
append blank
replace texts with wordfield
replace vocsize with vocnumber
replace hapsize with hapaxnumber
replace dissize with disnumber
replace hdratio with ratio
repl mhlength with meanhaplength
replace mdislength with meandislength
```

```
endfor
set talk on
calculate avg(vocsize),avg(hapsize),avg(dissize), avg(hdratio), avg(mhlength), avg (mdislength)
calculate min(vocsize),min(hapsize),min(dissize),min(hdratio), min(mhlength), min (mdislength)
calculate max(vocsize),max(hapsize),max(dissize),max(hdratio), max (mhlength), max(mdislength)
calculate std(vocsize),std(hapsize),std(dissize)
```

Exercise 10

wordcoverage.prg

```
set default to d:\fox\practice
set safe off
close data
clear
set talk off
nothing=''
carriage=chr(13)
spaces=chr(32)
create table temp1(word c(25),freq n(8))
append from d:\fox\table3\wordlistb
replace all freq with 100000&&for picking out covered words
create table temp2(word c(25),freq n(8))
create table wordcoverage(textname c(25),tokens n(5),coveredw n(6),coverage n(6,2))
select 2 && access temp2
for i=1 to 48
texts='d:\fox\texts\text'+alltr(str(i))+'.txt'
textinput=filetostr('&texts')
textinput=strtran(textinput,'-',spaces)
textinput=chrtran(textinput,',.`[?]_”!:;()*',nothing)
textinput=strtr(textinput,”'”,nothing)
textinput=strtran(textinput,spaces,carriage)
strtofile(textinput,'temp.txt')
append from temp.txt sdf for word<>spaces
tokennumber=reccount()&&get total number of tokens of a text
replace all freq with 1
replace all word with proper(word)
append from temp1&&wordlistb
index on word tag word
total to temp on word
zap
```

```
31. append from temp
32. sum mod(freq,100000) to wordscovered for freq>100000&&words in temp2
    with frequency>100000 are covered words. mod(freq,100000) gets their real
    frequency
33. coverageratio=wordscovered/tokennumber
34. select 3&&access wordcoverage
35. append blan
36. replace textname with texts
37. replace tokens with tokennumber
38. replace coveredw with wordscovered
39. replace coverage with coverageratio
40. select 2
41. zap&&remove old contents for another text
42. endfor
43. select 3
44. set talk on
45. calculate avg(tokens), avg(coveredw),avg(coverage)
46. calculate min(coveredw),min(coverage)
47. calculate max(coveredw),max(coverage)
48. calculate std(coveredw),std(coverage)
```

Exercises of Chapter 4

Exercise 1

```
repl all word with padl(rtrim(word),25,' ') ↵
```
or
```
repl all word with space(25-len(rtrim(word)))+word ↵
```
or
```
repl all word with replic(' ',(25-len(rtrim(word))))+word ↵
```

Exercise 2

use *asc()* to detect the unseen character, then remove it and correct the mistake in word length.

Exercise 3

The program is as follows:

setrelation.prg

```
1.  set defa to d:\fox\practice
2.  set safe off
3.  select 1
4.  use aliceword
5.  index on word tag word
6.  select 2
```

```
7.  use annotatedword
8.  index on word tag word
9.  set relation to word into aliceword
10. copy to temp fields word,freq,rng,a.wlength,wordinfo
11. use temp
12. browse
```

Exercise 4

cjustb.prg

```
1.  set defa to d:\fox\practice
2.  close data
3.  set safe off
4.  create table poem2(lines c(80))
5.  spaces=chr(32)
6.  append from d:\fox\texts\poem.txt sdf
7.  replace all lines with padc(alltrim(lines),80,spaces)
8.  copy to cjustify2.txt sdf
9.  modify file cjustify2.txt
```

Exercise 5

removechapter.prg

```
1.  set default to d:\fox\practice
2.  nothing=''
3.  textinput=filetostr('d:\fox\texts\alice.txt')
4.  do while 'CHAPTER'$textinput
5.  textinput=stuff(textinput,at('CHAPTER',textinput),11,nothing)
6.  enddo
7.  strtofile(textinput,'temp.txt')
8.  modify file temp.txt
```

Exercise 6

a.

```
repl all word with alltrim(subs(word,at(' ',word),25-len(rtrim(word))))+' '+substr
(word,1,at(' ',word)) ↵
```

b.

```
zap
appe from d:\fox\table3\postable
repl all word with stuff(word,1,at(' ',word),'') ↵
inde on word tag word ↵
total to temp on word ↵
zap ↵
appe from temp ↵
```

Exercise 7

nouns.prg

```
set defa to d:\fox\practice
set safe off
clos data
creat table postag(word c(40),freq n(8))
spaces=chr(32)
for i=1 to 50
tablename='d:\fox\table1\bncst'+alltr(str(i))
appe from &tablename for word<>spaces
endfor
repl all word with alltrim(subs(word,rat(spaces,rtrim(word)),40-len(rtrim(word))))
inde on word tag word
tota to temp on word
zap
append from temp for word<>spaces
sum freq to nouns for 'NN'$word
sum freq to tokens
?nouns/tokens
```

Exercise 8

product.prg

```
set default to d:\fox\practice
create table multiplication(numbers c(25),result n(6))
append from d:\fox\texts\multiplication.txt sdf
replace all numbers with strtr(numbers,'X','*')
replace all result with evaluate(numbers)
replace all numbers with rtrim(strtr(numbers,'*','X'))+' '+'='+ltrim (str (result))
copy to temp.txt field numbers sdf
modi file temp.txt
```

Exercise 9

fputs.prg

```
set defa to d:\fox\practice
close data
newtext=fcreat('temp.txt')
carriage=chr(13)
spaces=chr(32)
for i=1 to 48
texttitle='TEXT '+ alltrim(str(i))
texts='d:\fox\texts\text'+alltr(str(i))+'.txt'
```

```
fhandle=fopen('&texts')
flsize=fseek(fhandle,0,2)
fseek(fhandle,0)
gettexts=padl(texttitle,35,spaces)+carriage+fread(fhandle,flsize)+carriage
fputs(newtext,gettexts)
endfor
close all
modi file temp.txt
```

Exercise 10

xmltext.prg

```
set default to d:\fox\practice
set safety off
close data
create table sentence(sent1 c(250),sent2 c(250),sent3 c(250))
create table wordlist(word c(25),freq n(5))
nothing=''
carriage=chr(13)
spaces=chr(32)
textinput=filetostr('d:\fox\texts\text1.xml')
textinput=strtr(textinput,'<',carriage)
strtofil(textinput,'temp.txt')
select 1
append from temp.txt sdf
replace all sent1 with alltrim(sent1)
delete all for sent1<>'w:t>'
pack
replace all sent1 with stuff(sent1,1,at('>',sent1),nothing)
copy to temp.txt fields sent1,sent2,sent3 sdf
textinput=filetostr('temp.txt')
textinput=strtran(textinput,'-',spaces)
textinput=strtran(textinput,spaces,carriage)
strtofile(textinput,'temp.txt')
select 2
append from temp.txt sdf for word<>spaces
replace all word with chrtran(word,',.`[?]_”!:;()*',nothing)
replace all word with strtran(word,”'”,nothing)
replace all word with proper(word)
replace all freq with 1
index on word tag word
total to temp on word
zap
append from temp
```

```
brow
```

Exercises of Chapter 5

Exercise 1
alicearray.prg

```
set default to d:\fox\practice
create table temp(word c(25),freq n(4))
dimension alicearray(2615,2)
use awordlist
for i=1 to 2615
alicearray(i,1)=word
alicearray(i,2)=freq
skip
endfor
*Sort the 2615 elements in the second column in descending order.
asort(alicearray,2,2615,1)
use temp
append from array alicearray
```

Exercise 2
awordlistb.prg

```
set default to d:\fox\practice
set safety off
close data
create table awordlist (word c(25),freq n(10),wlength n(4))
nothing=''
spaces=chr(32)
carriage=chr(13)
textinput=filetostr('d:\fox\texts\alice.txt')
do tokenizer
totalize('word')
replace all wlength with len(alltrim(word))
delete all for len(alltrim(word))=0    &&this removes empty records
pack
copy to awordlist.txt sdf
procedure tokenizer
textinput=strtran(textinput,'-',spaces)
textinput=strtran(textinput,spaces,carriage)
strtofile(textinput,'temp.txt')
append from temp.txt sdf
replace all word with chrtran(word,',.`[?]_”!:;()*',nothing)
replace all word with strtran(word,”'”,nothing)
replace all word with prop(word)
replace all freq with 1
return
function totalize
parameters wordfield
index on &wordfield tag &wordfield
total to temp on &wordfield
zap
append from temp
return
```

Exercise 3

dicecast.prg

```
*This program simulates die casts using an array.
clear
set defa to d:\fox\practice
clos data
set deci to 1
set safe off
*Create a temporary table for recording each outcome of a throw.
creat cursor castdice(points c(4))
*Table dice is for number of throws, die points of each throw and the
*cumulative number of each of the six die points as the number of throws
*increases.
creat table dice(casts n(4),points c(1),increase n(4))
*Creating a 6 x 2 array for the six different points of a die and six random
*numbers.
dimension dice(6,2)
for throw=1 to 50 &&50 throws
for dicepoint=1 to 6 &&for the six different die points
dice(dicepoint,1)=rand()
dice(dicepoint,2)=dicepoint
endfor
*Simulate die throwing.
asort(dice)
sele 1
appen blan
*Taking dice(1,2) as the outcome of a throw.
repl points with alltr(str(dice(1,2)))
inde on points tag points
tota to temp on points
zap
appe from temp
```

```
*Calculates the growth of the different types of outcomes.
22. differentpoints=recc()
23. sele 2
24. appe blan
25. repl increase with differentpoints
26. repl points with alltr(str(dice(1,2)))
27. endfo
28. sele 2
29. repl all casts with recn()
30. brow
```

Exercise 4

randomposition.prg

```
*This program uses do case to select text chunks of about 100 words in length
*from different places of the source texts randomly. The 48 randomly drawn
*samples are then named sample1.txt, sample2.txt, sample3.txt...sample48.txt
*and outputted.
1.  set defa to d:\fox\practice
2.  set safe off
3.  clos data
4.  clear
5.  set decimals to 0
6.  nothing=''
7.  for i=1 to 48
8.  textname='d:\fox\texts\text'+alltr(str(i))+'.txt'
*outputsample is the variable for the name of sample1.txt, sample2.txt etc.
9.  outputsample='sample'+alltr(str(i))+'.txt'
10. textinput=filetos('&textname')
*Meaure length of textinput.
11. textlength=len(textinput)
*Divide textlenth into 4 parts.
12. chunklength=int(textlength/4)
*Generate random number between 0 and 100.
13. randomnumber=rand()*100
*The following do case statements randomly sample text chunks at 4 different
*positions of the source texts.
14. do case
15. case randomnumber<26
16. randomsample=subs(textinput,1,chunklength)&&sampling from beginning
    of a text if the random number<26
*The following ensures that randomsample does not end in the middle of a 、
*word.
17. randomsample=stuff(randomsample,rat(' ',randomsample),1000,nothing)
```

```
case randomnumber>25 and randomnumber<51
randomsample=subs(textinput,chunklength,chunklength)&&sampling a chunklength away from beginning
randomsample=stuff(randomsample,1,at(' ',randomsample), nothing) && ensure the sample begins with a complete word
randomsample=stuff(randomsample,rat(' ',randomsample),1000,nothing)
case randomnumber>50 and randomnumber<76
randomsample=subs(textinput,chunklength*2,chunklength)
randomsample=stuff(randomsample,1,at(' ',randomsample),nothing)
randomsample=stuff(randomsample,rat(' ',randomsample),1000,nothing)
case randomnumber>75
randomsample=subs(textinput,chunklength*3,textlength)
randomsample=stuff(randomsample,1,at(' ',randomsample),nothing)
randomsample=stuff(randomsample,rat(' ',randomsample),1000,nothing)
endcase
strtof(randomsample,'&outputsample')
endfor
```

Exercise 5

lexcompare.prg

```
set default to d:\fox\practice
set safety off
close data
gettext('d:\fox\texts\alice.txt')
copy to awordlist
use awordlist
do getlemma
gettext('d:\fox\texts\lglass.txt')
copy to lwordlist
use lwordlist
do getlemma
do compareword
function gettext
parameters filename
create cursor wordlist (word c(30),freq n(10),wlength n(4))
nothing=''
spaces=chr(32)
carriage=chr(13)
textinput=filetostr(filename)
textinput=strtran(textinput,'-',spaces)
textinput=strtran(textinput,spaces,carriage)
strtofile(textinput,'temp.txt')
append from temp.txt sdf
```

```
24. replace all word with chrtran(word,',.`[?]_”!:;()*',nothing)
25. replace all word with strtran(word,”'”,nothing)&&there is a single quote
    between the double quotes
26. replace all word with prop(word)
27. replace all freq with 1
28. index on word tag word
29. total to temp on word
30. zap
31. append from temp for word<>spaces
32. replace all wlength with len(alltrim(word))
33. return
34. procedure compareword
35. create table aliceglass (word c(25),freq n(12,5))
36. append from awordlist
37. replace all freq with freq*100000
38. append from lwordlist
39. index on word tag word
40. total to temp on word
41. zap
42. append from temp
43. copy to shareword for mod(freq,100000)>0 and freq>100000
44. copy to aliceonly for mod(freq,100000)=0
45. copy to lglassonly for freq<100000
46. copy to lglassonly.txt for freq<100000
47. use aliceonly
48. replace all freq with freq/100000
49. copy to aliceonly.txt sdf
50. use shareword
51. replace all freq with freq/100000
52. copy to shareword.txt sdf
53. return
```

Exercise 6

overlap.prg

```
*This program measures vocabulary overlap between 25 pairs of bnc tables.
1.  set defa to d:\fox\practice
2.  set safe off
3.  close data
4.  clear
5.  set deci to 16
*tablename is for two table names, voc1 and voc2 are for their vocabulary
*sizes, overlap for the vocabulary overlap.
6.  create table overlap(tablename c(70),voc1 n(5),voc2 n(5),overlap n(5))
```

```
*Create a temporary table for measuring voc1, voc2 and overlap.
7.  creat cursor temp(word c(25),marker n(10))
*Put all the table names into tablearray.
8.  adir(tablearray,'d:\fox\table2\bncwlem*.dbf')
9.  rand(-551)
*Randomize the order of tables.
10. for i=1 to 50
11. tablearray(i,2)=rand()
12. endfor
13. asort(tablearray,2)
14. for i=1 to 50
*The following initializes twotablename for holding two table names.
15. twotablename=''
*Loading two tables.
16. for ii=0 to 1
*In the following, when i=1 and ii=0, tablearray(1,1) is assigned to
*tabletoappend. When i=1 and ii=1, tablearray(2,1) is assigned to
*tabletoappend, etc.
17. i=i+ii
18. tabletoappend='d:\fox\table2\'+tablearray(i,1)
*The following puts two table names together separated by ":".
19. twotablename=twotablename+tabletoappend+' : '
20. append from &tabletoappend&&append to temp in work area 1
*Get vocabulary size of first table.
21. if ii=0
22. table1voc=reccou()
23. endif
24. endfor
*Get number of words in temp after the second table is appended.
25. twotablewords=reccou()
*Get vocabulary size of the second table.
26. table2voc=twotablewords-table1voc
27. inde on word tag word
28. total to temp1 on word
29. zap
30. appe from temp1
*Get vocabulary size of the two tables after totalling.
31. twotablevoc=reccou()
*Get vocabulary overlap.
32. vocoverlap=twotablewords-twotablevoc
33. sele 1 &&access table overlap
34. append blan
35. repl tablename with twotablename
```

```
36. repl overlap with vocoverlap
37. repl voc1 with table1voc
38. repl voc2 with table2voc
39. sele 2
40. zap
41. endfor
42. sele 1
*Remove ":" at end of the two table names.
43. repl all tablename with stuff(tablename,rat(':',tablename),1,'')
44. brow
```

Exercises of Chapter 6

Exercise 4

bncletters.prg

```
1.  set default to d:\fox\practice
2.  close data
3.  set safety off
4.  create table letterfreq(letters c(2),freq n(8))
5.  textinput=filetostr('d:\fox\texts\words.txt')
6.  textinput=upper(textinput)
7.  for i=65 to 90
8.  letterfrequency=occurs(chr(i),textinput)
9.  append blank
10. replace letters with chr(i)
11. replace freq with letterfrequency
12. endfor
13. index on freq tag freq desc
14. browse
```

II. Foxpro Operators, Commands and Functions Covered in This Book

Index