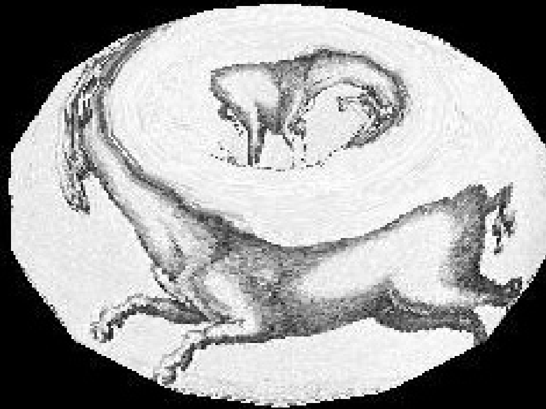
ICON PROGRAMMING FOR HUMANISTS



2nd Edition

ALAN D. CORRÉ

Icon Programming for Humanists

Alan D. Corré

University of Wisconsin-Milwaukee

Goal-Directed Press, Moscow, Idaho

The first edition of this book was previously published by: Pearson Education Inc. (Prentice-Hall).

Copyright © 2010 by Alan D. Corré. An individual is licensed to make copies of this document for personal use. No changes may be made to the text of this document without the express permission of the copyright holder. All rights reserved.

First printing, 9/3/2010. Send comments and errata to jeffery@cs.uidaho.edu.

This document was prepared using OpenOffice.org.

For Nita

Rabbi Judah ben Idi said in the name of Rabbi Johanan: We know from references in Scripture that the Shekina left Israel by ten stages: from the Ark-cover to the Cherub and from the Cherub to the threshold of the Holy of Holies and from the threshold to the court and from the court to the altar and from the altar to the roof of the Temple and from the roof to the wall and from the wall to the city and from the city to the mountain and from the mountain to the desert and from the desert it ascended and abode in its own place, as it is said: “I will go and return to my place.” (Hosea 5.15)

Babylonian Talmud, *Rosh Hashana*, 31a.

And numbers too, I found them,
The key to sciences;
And letters in their synthesis---
Secret of all memory, sweet mother of the arts.

Aeschylus, Prometheus Bound

Contents

Foreword to the Second Edition....ix
Preface....xi
1. Introduction....1
  1.1 The Icon Programming Language....1
  1.2 Basic Concepts....2
  1.3 Mathematical Concepts....2
2. Distributions....7
  2.1 Getting Started on a Program....7
  2.2 Separating Data and Program....11
  2.3 Distribution....12
3. Storing and Editing Programs in Files....17
  3.1 Making Information that Lasts....17
  3.2 The Emacs Editor....17
  3.3 Running an Icon Program From a File....18
4. Graphs....21
  4.1 Histograms....21
  4.2 Reading Data From a File....23
  4.3 Refining the Histogram....26
5. Measures of Central Tendency....31
  5.1 The Mode....31
  5.2 The Arithmetic Mean....37
  5.3 The Median....38
  5.4 Variants on the Program....39
6. Icon and Unicode....41
  6.1 Introduction to Unicode....41
  6.2 Working with Russian, Tamil, and Hebrew....42
    6.2.1 Russian....43
    6.2.2 Tamil....45
    6.2.3 Hebrew....49
  6.3 Further Study....50
  6.4 Hex to Dec....51
7. Standard Deviation....53
  7.1 Working with Sentences....53
  7.2 Figuring the Standard Deviation....56
  7.3 Word Frequency....59
8. Correlation....63
  8.1 The Spearman Rank Correlation....63
  8.2 Scattergrams....66
9. Pearson's Coefficient of Correlation....71
  9.1 Planning the Program....71
  9.2 Getting Information From the User....72
  9.3 Figuring the Coefficient....73
  9.4 Creating the Table—Pattern Matching....74
  9.5 Matching Against a List of Words....74
  9.6 More on Matching....77
  9.7 Sets....80
10. Programming a Nursery Rhyme....83
  10.1 Ten Green Bottles....83
  10.2 The House that Jack Built....85

10.3 Randomizing Jack....88
11. Creating a Data Base....91
11.1 A Grade Program....91
11.2 A Vocabulary Program....99
12 Icon and Markup Languages....103
12.1 A Brief History of Markup Languages....103
12.2 Manipulating texts encoded according to the TEI....104
12.2.1. The Input Text....105
12.2.2. The Program....106
12.2.3. The Output....107
12.3 Expanding the above Program....108
12.4 Stripping Tags and Character Entities from a Text....112
13. Conclusion....115
13.1 Word Processing....115
13.2 Other Icon Features....117
13.3 Co-Expressions....117
13.4 File Handling....118
Appendix A: Character Sets....119
Appendix B: Some Hints on Running Icon Programs....121
Icon for Apple OS X....121
Icon for Windows....122
Index....123

Foreword to the Second Edition

Books on text processing are few and far between, even more so now than in past decades. For a long time it has been almost impossible to obtain a copy of Alan Corré's *Icon Programming for Humanists*. This edition solves that problem. The text was scanned and run through an Optical Character Recognition program from a printed copy of the first edition. Thereafter the text was scrutinized and revised by the author and myself. Chapters 6 and 12 are entirely new, introducing important topics in character sets and text encoding, respectively.

The radical change about this edition is that the author generously placed the text under a free electronic download license. Among other things, going online will make it easy to correct remaining OCR and other errors as they are found.

This edition adopts the code format standards of the other Icon books. Modernization includes the deletion or replacement of DOS-era references with terminology and tools more appropriate to Linux, OS X, and Windows Vista. I have not taken the code examples and turned them into Unicon programs, despite my personal bias towards this modern successor for Icon.

The text of the second edition has benefited from comments and suggestions by Phillip Thomas and Alan Saichek. We are grateful to these individuals for their contributions.

Clinton L. Jeffery

Preface

In 1958 the *Science Education Newsletter* confidently declared: “Computers are here to stay.” Just about the same time, a professional journal wrote that the IBM 702 computer “has a large internal storage capacity,” which turns out to be the equivalent of 125 punched cards. The Newsletter was right of course, but the most enthusiastic futurist at that time could not have imagined the situation just fifty years later. Now a vast number of desks carry a small computer with many, many times the computing power of that free-standing IBM machine, and at a fraction of the price. And airplanes routinely carry almost as many laptops as passengers.

Icon Programming for Humanists was first published by Prentice-Hall (now Pearson Education Inc.) in 1990 with the intent of providing a task-oriented introduction to a computer language especially suited to those whose main concern is the written word. Since that time the pace of change has quickened in the world of communication. Public phone booths and corner mail boxes disappear as cellular phones and email usurp the functions they formerly served. Pens and pencils are replaced by word processors even for small tasks.

But some things do not change. The texts of the Hebrew Bible, the New Testament and the Koran are essentially the same as they were a thousand years ago, and are an essential ingredient of many of the hopes, fears, and conflicts of the world as we know it today. These are simple examples of the written word, which encapsulates a great deal of human culture over a great stretch of time. Many texts are becoming available in machine-readable form; a new chapter in this edition deals with the initiative to standardize such texts. The Icon Programming Language, now in its ninth incarnation, is still ready to lighten the tasks of the student of the Humanities.

Icon is a very “high” language, which means that it has many remarkable features and capabilities built in, which take it far from the primitive language of zeros and ones that the computer utilizes, and makes it approach being an *application.* But it nevertheless retains its status as a computer language, because it possesses the characteristic of universality. Humanists, using this language to further their researches, can do so in ways never previously envisaged. It can do most things that other languages can do, and often much more compactly.

This book teaches the principles of Icon in a very task-oriented fashion. Someone commented that if you say “Pass the salt” in correct French in an American university you get an A. If you do the same thing in France you get the salt. There is an attempt to apply this thinking here. The emphasis is on projects which might interest the student of texts and language, and Icon features are instilled incidentally to this. Actual programs are exemplified and analyzed, since by imitation students can come to devise their own projects and programs to fulfill them. A number of the illustrations come naturally enough from the field of Stylistics which is particularly apt for computerized approaches. This book assumes an acquaintance with the concepts of elementary statistics appropriate for such work, and the reader unfamiliar with these may wish to become familiar with them first. Kenny's book referred to in the first chapter gives a clear description of these principles and may be used with profit.

I am gratified that Professor Clint Jeffery, who has made so many valuable contributions to the development of computer science, suggested an updated reissue of this book, which has been out of print for some years. It is my hope that it will continue to be of use in furthering the knowledge and utilization of the Icon Programming Language in its classic form. Like the first edition, this book is dedicated to Nita, my wife of fifty years.

Alan D. Corré

1. Introduction

1.1 The Icon Programming Language

Why Icon? Icon is one programming language among hundreds, but is probably the best for tasks which involve the study or manipulation of natural language. Programming languages form a bridge between human beings and computers, enabling man to convey instructions to machine. Two of the earliest languages, FORTRAN and LISP, continue to be used for mathematical and artificial intelligence applications, and the great difference in character of these two languages indicated early on that there were many ways in which this man-machine communication could be carried on. In the late sixties the SNOBOL-4 language, which was deliberately designed to be different from its predecessors, appeared. Computers were originally intended to perform mathematical calculations but it became apparent that by encoding the letters of the alphabet, natural language texts could also be manipulated. SNOBOL-4 dealt primarily with strings of characters rather than numbers, and along with other original features this gave it special power in dealing with natural language. It also had the advantage of a particularly fine manual, both in content and appearance, perhaps the best computer manual ever produced[1]. Two things have happened since SNOBOL-4 appeared, one of a theoretical, one of a practical nature. First, there has been an emphasis on writing programs which are divided into smaller units systematically connected into a unified whole. This makes programs much easier to grasp for the human reader, and easier to modify subsequently. A long program which is not “structured” becomes a tangled web of which it is dangerous to interfere with any strand. The popularity of the Pascal language which emphasized this approach and in fact imposed it upon its users, attests to the manner in which this notion has become accepted in the programming community. The “C” language, and its successor C++, used by professional programmers, both adhere to principles of structured programming.

Secondly, interactive computing, in which the user engages in dialog with the machine, rather than furnishing it with a batch of data, has become the norm. Originally computer time and computer memory were both at a great premium. This is now true only on the smallest of embedded systems; affordable computing is within the reach of all. The design of SNOBOL-4 preceded these changes, and although it has been updated to take account of them, the need was felt for a language which would have features similar to SNOBOL-4, but would have a character similar to other modern, structured languages. The Icon language was evolved by Ralph Griswold and others to achieve this end. Government support was given to its development, and hence the language has the additional advantage of being in the public domain. It is available free, and runs on most modern computing platforms.

This book aims to show how Icon can be used fruitfully in literary computing, and hence does not duplicate the standard reference on the language[2] which is an indispensable guide to using Icon. We emphasize the writing of programs which manipulate or study texts, and follow largely the kind of procedures which are mapped out for manual use in Kenny's excellent introduction to statistics for students of literature and the humanities.[3] Many of the programs developed here are in fact automations of procedures suggested by Kenny.

It is not necessary to have any previous knowledge of computer programming in order to be able to use this book, but you need to have available the programs which convert the source code you will write into a form which the computer will “understand” and execute. The appropriate implementation of Icon for your platform can be downloaded from http://www.cs.arizona.edu/icon. Programs should be saved as “text only.” If you are using the Macintosh OS X, you should use the terminal, for which at least a

modest understanding of Unix is needed. Apple provides a pre-built Emacs editor which is fine for creating Icon programs. It is suggested that you use this text in front of your keyboard, and try out the various programs that are suggested. You may experiment by making modifications and seeing what happens. This will provide a discovery procedure in learning; if you learn nothing else you will probably discover that you do not think as logically as you thought you did. To have a dialog with the computer can be a humbling experience. Except in the rare instances where there is some hardware problem, or you encounter a bug in the software, the computer is excruciatingly logical.

1.2 Basic Concepts

Before we can use computers to aid us in our task we need to understand a few basic concepts. These may be divided into two sets: concepts familiar from daily life and concepts of a mathematical character. The concepts familiar from daily life are:

1. Sequence
2. Condition
3. Repetition

Let us illustrate these by reference to a cooking recipe (which is genuine but not guaranteed):

Peel vegetables.

If the beef is frozen, defrost at power 3.

Form into patties.

Layer half of patties and half of vegetables, repeat.

Microwave until vegetables are tender.

“Peel vegetables” and “Form into patties” are instructions to be carried out as they occur in the sequence of the directions. The instruction to defrost, by contrast, is dependent on the beef being frozen. If it is not, there is no need to obey the instruction, since no alternative action is specified. This illustrates the second concept, condition or choice, and it is this capacity to choose which gives the computer its great power, since it can act differently according to conditions. The last two instructions contain repetitions, often called “iteration” or “a loop.” In the first case, the repetition is determined numerically; it must be done two (and not three or ten) times. The loop also contains a type of condition. After performing the action, if the action has been done *less* than twice it is repeated. If it has been done twice the repetitive act ceases. In the last action a particular condition is constantly checked. Are the vegetables tender?—no, cook some more; yes—stop cooking.

These three concepts——sequence, condition, and repetition——are all that is needed for structuring a program. The most complex operations depend on these simple ideas.

1.3 Mathematical Concepts

The two concepts which are borrowed from mathematics may be a little more difficult to grasp. Let us first consider the following two statements:

John has four children.
John has four letters.

Both statements may be true, but in the first *John* represents a flesh-and-blood real person, whereas in the second we are speaking of the string of marks on paper or the screen. The second *John* would often be written in this context surrounded by double quote marks, and Icon uses the double quotes to surround literal strings of this kind. The first *John* is a *variable* which stands for its *value*, in this case a particular individual whose name happens to be John. In Icon any word that begins with an alphabetic letter or underscore and continues with any amount of letters, numbers, or underscores, can be a variable, provided it is not one of a small number of words which have been appropriated by Icon for special purposes. The underscore is used rather than the dash to avoid confusion with the use of the latter for subtraction and negative numbers. Thus

```
year := 2009
```

associates (by means of the := operator, called the assignment operator) the value 2009 to the variable year, the colon followed immediately by the equals sign being the symbol which assigns the value of what is on the right to the variable on the left. Note that this book uses a sans serif font to represent Icon code. So

```
last_year := year - 1
```

would assign the value 2008 to the variable last_year, which it would retain until it is specifically altered by a new assignment. When it gets this new value, the old value is lost. It is perhaps best to regard the variable as a kind of box or container into which a value can be poured. This image is not so far from the truth, since the variable represents a location in the memory of the computer which we can more readily handle by giving it a name which we can easily remember.

We mentioned that the ability of the computer to handle conditions is one of the things which gives it its great power. The ability to use variables similarly gives great power to the computer, since it makes it possible for programs to apply to a great deal of different data simply by feeding this data to a variable as appropriate. Programming at its best results in a *general* program which processes *specific* data.

The other mathematical concept widely used in programming is that of the *function.* In mathematics the function establishes a connection between one set of numbers and another set. Thus the function which we call *square* associates the numbers

```
1, 2, 3, 4, 5...
```

with the numbers

```
1, 4, 9, 16, 25...
```

The function *square* applied to 2, for example, gives ("returns," "produces") 4, and 4 is the *value* of the function. In a similar manner, the Icon function reverse() associates the strings

```
"cram", "did", "trap", "number2"...
```

with the strings

```
"marc", "did", "part", "2rebmun"...
```

So the value of

```
reverse("cram")
```

is "marc". The *function* is followed by parentheses which enclose the *argument* which is to be processed by the function. This argument may be a variable; for example

```
word := "cram"
reverse(word)
```

gives the same result. In addition to giving values, functions often have (side-)*effects*. It is most important to understand this distinction. For example, the value of

```
write("cram")
```

is "cram"—but in addition this function writes the word "cram" on the screen, and directs subsequent writing to be on the next line, or, in other words, it issues a carriage return. In this instance the *effect* is normally much more significant than the value, but we shall find that the value can be used too.

A function need not always produce a result. Sometimes it does not produce anything, or to put it another way, it *fails*. As an example, the function integer() produces a whole number from its argument —if it can. The value of integer(1.5) is 1. If there is a decimal, it is simply truncated, or cut off. But integer("a") does not produce anything—it fails. There is just no way the character a can be made into a whole number. This is important, because failure is, so to say, infectious. If the result of a function that fails is passed to another function as an argument, that function fails too, and any *effects* it may have do not occur. So if we write

```
write(integer("a"))
```

the integer() function fails, and so write() fails too, and nothing at all happens. On the other hand

```
write("")
```

does write a zero-length string, which is indeed nothing to speak of, but write() has succeeded, and so the effect of the carriage return will be seen in the fact that the cursor—that moving block or underline that you see on the screen—will move down. This difference between "writing nothing" (succeeding) and "not writing anything" (failing) may seem arcane or cabalistic, but it has significant results. It is one of the things that makes Icon tick. We may note in passing that write(integer("a")) is an example of a nested function: whatever is produced by integer() is immediately passed to write().

Since we are here talking about nothing, it will be useful to refer here to some other manifestations of this strange entity, which, let us recall, is not the same as not anything. Before so doing, we may like to consider an example from English grammar which will help to explain what we are talking about. In English the plural of "horse" is "horses" and the plural of "ox" is "oxen." We can then say that to form the plural in English, we add *-s* and in a few cases which can be specified *-en*. What about "sheep"? We can say that the plural is the same as the singular. But we can *regularize* the situation by saying that "sheep" adds *-0* which is "nothing" or, more technically, a "zero morpheme." This legal fiction is often very useful, because now *all* words add a morpheme to form the plural, and not just some, and it makes exceptional circumstances less awkward to handle. This is one of the "uses of nothing" as one linguist put it. Now we noted before that "John" is a string of four characters. We can subtract four characters from it and be *left* with the string of zero length which we just referred to. A function which produces a zero-length string succeeds, even though it may not seem to produce anything of great significance. This empty string is often useful however as a kind of starter to which other pieces ("substrings") may be added. The empty string is different from the *blank* or *space*, which is a real character and has a length of one. Blanks are obvious when they are in between words, but is there a blank at the end of a line on the screen or a piece of paper? The computer may sometimes want to know the answer to that question, but we can delay the answer for now. It is an historical accident that the blank is used as a word divider; some ancient scripts used a bar or some other marker that looks more like a "real" character. In addition Icon has an entity called *null*. This represents the initial value of a variable (year for example) before some value (such as 2009) is assigned to it. A function which produces null is in fact succeeding, even though there is not much you can do with null. The whole point of null is that it *cannot* be used in computations—whether numbers, strings of characters, or anything else is involved —since thereby if we inadvertently try to use something having that value in a computation while we

are developing a program, Icon will pick it up and let us know about it by flashing an appropriate message *on* the screen. The notation for the null value is &null. The ampersand at the beginning indicates that &null is a *keyword*, one of several useful values which Icon specifies for us. &null may be used in assignments; thus if the value of the variable year is 1987, the command

```
year := &null
```

restores it to its initial null value. We shall find later that alternating between a null value and some other value can have distinct uses. To sum up, we must learn to distinguish *nothing at all* (which is what functions which fail produce) from a skinny motley crew consisting of the empty string, the blank (or blanks), and the null value which may be lightweights but at least they are not failures.

Notes

1 R.E. Griswold, J.F. Poage, and I.P. Polonsky, The SNOBOL-4 Programming Language (Englewood Cliffs, N.J.: Prentice Hall, 1971).

2 Ralph E. Griswold and Madge T. Griswold, The Icon Programming Language, Third Edition. Peer to Peer Communications, San Jose, 1997. http://www.cs.arizona.edu/icon/lb3.htm

3 Anthony Kenny, The Computation of Style (Oxford, 1982)

2. Distributions

2.1 Getting Started on a Program

This chapter will consider how Icon can be used to quantify aspects of a text. We shall do this by a simple study of word length, taking first a very brief text:

One went and came. *W.B. Yeats*

We shall then show how to expand these principles for long texts. Our task is

1. To isolate individual words.
2. To count the number of letters each has.
3. To display the numbers on the screen.

Icon possesses a scanning facility which is able to go through a string such as a line of text that was read in from a file and process it. A *string* is defined as a linear sequence of characters treated as a unit. We invoke this facility by following our brief text with a space and a question mark thus:

```
“One went and came.” ?
```

The string must be between double quotes in the program because otherwise Icon would understand the words as variables. Now let us imagine that there is a pointer positioned initially at the beginning of the line immediately before the upper case O. This pointer indicates the point at which we are working and will gradually move through the string until our task is complete.

Icon has a function upto() which will locate the position in our string immediately before any of the characters in its argument. Thus if we write

```
"One went and came." ? upto(' .')
```

upto() will locate the first occurrence of a blank or a period in our string. The characters concerned, blank and period, are delineated by single quote marks. In general in Icon, the double quotes enclose a *string*—where the order of letters is significant—while single quotes enclose a *set of characters* or *cset* —where the order is irrelevant. (A *set* is a collection of items which is considered as a unity. Each item occurs once only.) At this point, the *value* of upto(' .') is 4, which is the position immediately before the blank. Check this by counting from position 1, which is at the beginning of the string, right before the first character ("O"). Remember that the blank or space is a character in its own right. We shall now use the Icon function tab(), the effect of which is to move up that imaginary pointer to the position of its argument and the value of which is the characters (in this case the first word) in between the old and new positions.

```
“One went and came.”
↑   ↑
1   4
```

(Positions 2 and 3 are on either side of the *n* in *One.*)

This gives us:

```
"One went and came." ?  tab(upto(' .'))
```

Since tab() has been used, the imaginary pointer has been moved up to position 4 and we shall proceed from there. At this point the value of the expression tab(upto(' .')) is the string "One"—and we wish to know its length. The length of a string can be obtained in Icon simply by prefixing an asterisk:

```
"One went and came." ? *tab(upto(' .'))
```

To check the evaluation of what follows the question mark, we must move from the innermost parentheses out. The value of upto() which is 4 is passed to tab(), the value of which is "One" and the length of this is reported by the asterisk. Note that so far all this activity is purely internal to the computer; we would not be aware of what has been going on. To find out, we use the function write() which prints out the result on the screen:

```
"One went and came." ? write(*tab(upto(' .'))
```

Let us review what this means. We have first the string we are processing between double quotes. Then comes the question mark which means that the string will be scanned and processed by what follows. We understand what follows by considering the innermost function first. The function upto() returns a position in the line (of little interest to us) which is immediately fed to tab() which moves the pointer (its *effect)* and produces the first word (its *value).* Observe that tab() has both an effect and a value. This word is turned over to the asterisk which produces its length, and finally this length is printed on the screen.

It is true that this is a lot of work to count three letters, but its power is that it can be applied generally and perform the tedious job of counting words by the millions. Before we do that, however, let us run this program which can tell us the length of a single word.

In Icon, programs are arranged in *procedures* which perform separate jobs and collectively make up the program. There is no absolute rule as to what should be included in a particular procedure, but arranging programs in this modular fashion greatly facilitates understanding and modification of the program at a later date. So we arrange the material we have prepared so far as follows:

```
procedure word_length()
  "One went and came." ? write(*tab(upto(' .')))
  return
end
```

Each procedure begins with the word procedure and is given a name we choose which must begin with a letter or underscore followed by any number of letters, numerals or underscores. It is best to give simple, descriptive names. The name is followed by the parentheses ( ) which will be used later on to pass information to the procedure, by including between them as many variables (called arguments) as are needed to pass that information. The procedure ends with the word end. The indentations simply help to set off the body of the procedure from the title and the end marker, and aid visually to see the structure of the program. Later, as the procedures become more complex, further indentation will be used, rendering it always possible to see which statements belong together since they will be written at the same level of indentation. The word return will be followed by the final value of the procedure if it has one, and indicates that the procedure has concluded successfully, and the program flow is to return from whence it came.

One thing is left to do. Every Icon program has a main procedure which acts like a traffic cop. By looking at it we can see the skeleton structure of the program, since each procedure mentioned or "called" in the main procedure is invoked in turn. No real business should be done in the main

procedure; it should simply control the flow of traffic. Since our program is quite simple, the main procedure will invoke only one other procedure to help. The entire program now looks like this:

```
procedure main()
  word_length()
end
procedure word_length()
  “One went and came.” ? write(*tab(upto(' .')))
  return
end
```

Procedures do not need to be written in any particular order, but it is usual to place the main procedure first, since it gives a conspectus of the program to the reader. When the program is executed, it always begins with the main procedure and follows the pattern laid out there.

Let us now try to run this little program. See Appendix B for help if Icon is not yet installed on your computer.

When you are sure that the Icon system is available on your computer, enter:

```
icont - -x
```

icont is a command to the operating system of your computer which invokes the Icon translator. This translates the program written in Icon into language (“machine code”) which the computer can use. The first hyphen tells Icon to take the input from the keyboard, as you are going to type in the program. Be sure to leave a space after it. The -x gives an instruction to execute the program as soon as it is translated, to put it into effect immediately. Press return, and in a few moments you will see the message Translating: which means that Icon is ready to translate your program to a form which it can understand and run. You will then see the message STDIN: which is the name that Icon gives to a translation of the program which it creates on disk in a form that the computer can understand. (The abbreviation stands for "standard input" which is usually the keyboard. The colon is not part of the name. The word may be in small letters.)

Copy in the entire preceding program. After each procedure, Icon will tell you if you have made an error. If all is well, it will repeat the name of the procedure possibly along with its size. If you do make a mistake, it is probably better to start over. Press *control-C* (hold down the *ctrl* key and press *c* and try again. After you have finished you must type in a character to tell the system that you are finished. This depends on the system you are using. It may be a *control-D* (hold down the *ctrl* key and press *d)* or *control-z* (which is the *F6* key on some terminals, or you may hold down the *ctrl* key and press *z.)* You may need to press the *Return* key also. This should ultimately result in the answer: 3. Since Icon has recorded on disk the copy of the program that the computer can read under the name *STDIN*, you can later repeat the program by typing in that word (stdin) at your command line prompt. (On some computers you may have to precede it by the word *iconx.)*

Let us now proceed to process the rest of the line. You will remember that the imaginary pointer was positioned right after the last letter of the first word. We now have to move it over the intervening blank. Note that the blank counts as a character. The function upto(), which we have already used, gives the position immediately before any member of its argument that it finds in the string. The function many() is the converse of this; it gives the position it finds immediately after *all* members of its argument that occur consecutively after the pointer. So

```
many('. ')
```

will skip over any periods or spaces which it finds immediately after the pointer and stop right *after* the last one. In this case the value of many('. ') is 5, which is the position after the space. If we feed this result to tab():

```
tab(many('. '))
```

tab() will move the pointer to the point immediately after the first blank. Its value is itself a space, but this is of no interest to us and we do not use it. Since the value is not assigned to any variable, or used as the argument to any function, it is in effect discarded. tab() is used for its *effect* of moving up the pointer. However, we wish to repeat this process until the end of the line. We achieve this by inserting while in front of the write() in our program and do at the end of the line:

```
procedure main()
  word_length()
end
procedure word_length()
  "One went and came." ? while write(*tab(upto('. '))) do tab(many('. '))
end
```

This means: *while* (or *as long as*) the instruction to write out a length can be carried out (or *succeeds*), *do* what follows. Let us trace this step by step. Before we introduced this "while-loop" our pointer was positioned as follows:

```
“One went and came.”
↑    ↑
1    5
```

Icon now repeats its initial activity. It finds the next blank, gives the value 9 to tab(), moves the pointer to just before the next blank:

```
“One went and came.”
↑        ↑
1        9
```

and produces the word "went". The asterisk figures its length at 4 and write() writes it to the screen. This process continues until the pointer reaches a point before the period:

```
“One went and came.”
↑                 ↑
1                 18
```

The function many() then produces 19 after detecting the period, and this 19 is fed to tab(). Now the pointer is at the end. When the next attempt to "write" occurs, the innermost function upto() *fails* because upto() is unable to detect a blank or a period which constitute the argument of upto(). The failure is "inherited" by all the functions in a chain leading to write(), and none of the failed commands is carried out. At this point the repetitive action ceases; since the expression between while and do has failed, and since there are no more commands, the program ends. The length of each word of the sentence should now be on your screen.

2.2 Separating Data and Program

Data may be defined as information presented in a form suitable for processing by your computer. So far we have been working on an extremely small text in which the text was written right into the program. While this was a useful way to get started, in general it is not a good idea to do this. It is better for programs to be of general application, and to achieve this we need to leave the data outside of the program which processes the data. One way to do this is to input the information from the keyboard, and this type of interactive computing where the machine responds instantly is very popular nowadays. Before we do so we have to handle one problem. We have used upto() to detect the presence of a blank or a period. That was fine in the sentence "One went and came." because each word was marked off by one or the other. But in a continuous text many lines do not end in any particular marker, so the last word in the line would not be noticed unless it happened to end in a period, or someone had inadvertently typed in a space after it, which would not of course be visible on the screen. We can easily solve this by replacing upto() by many() and using many() to span all the alphabetic letters of which a word may consist. This would necessitate writing an argument with at least 52 characters in it, but there is a simpler way. The *keyword* &ucase stands for a set of *all* the uppercase (capital) letters and &lcase represents all the lowercase (small) letters. You can use &letters for the union of these two sets. We can combine any two csets using a double plus sign (++) which is used for adding sets to each other. The single plus sign is reserved for arithmetic addition. Using distinct operators helps avoid programming errors. Note that elements only occur once in a set, so if you add together two sets which share certain items, the resulting set will have these items once only. Also, you can only add sets to sets; an element that you wish to add in must be in a set itself, even if that set has only one member. So

```
  &lcase ++ 'Z'
```

would give you the set of the lowercase letters and uppercase *Z*. When we come to learn about sets of items other than single characters, we shall find that there is a function which can insert an item into a set. Let us now look at the modified program and consider what the differences are.

```
procedure main()
  while word_length(read())
end
procedure word_length(line)
  line ? while write(*tab(many(&letters))) do
    tab(many('., ;:'))
  return
end
```

The effect of the function read() is to get one line of input from the keyboard, and the value returned by read() is the string which was typed at the keyboard. This is passed to the procedure word_length(). As you see, this modified procedure now has a variable (line) in its title. This variable will assume the value of the *argument* of word_length and can be used in the program. We have here an example of a perpetual loop. So long as the user inputs a line, the program will process it. In this loop the do is not necessary since no action takes place in the loop; it simply serves to keep on calling procedure word_length(). To break out of such a loop you have to enter a character provided by the operating system to terminate programs. Try *control-C* (hold down the *ctrl* key and press *c.)* When you use these so-called control characters, it is not necessary to press the shift key also.

There is a more elegant way to end this program. As the first line of the second procedure include:

```
if line == "." then stop("Bye")
```

The double equals sign checks for string identity, so if the string brought in consists of only a period, it invokes the function stop() which concludes the program immediately, and writes the message which is its argument.

2.3 Distribution

So far we have simply been listing word lengths. Now we should like to do the same as before, but we shall calculate and state how many words the text has which contain one, two, three, and more letters. For this we need variables in which to store these values as we go along. It would be *possible* to have, let us say, ten variables, one each for one-letter words, two-letter words and so on. It is easier however to use the *list* which is a collection of variables which all have the same name, distinguished by a number called an *index* enclosed in square brackets. If we call the variable letters, then the number of one-letter words will be stored in letters[1], the number of two-letter words in letters[2] and so on. We can set up such a variable by entering

```
letters := list(10,0)
```

The word list may only be used for this purpose in Icon. It is a function that creates and returns a list with as many variables as are stated in the first argument, and each has the initial value stated in the second argument. This list is assigned to a variable which you choose. So in this case, letters has a list assigned to it with ten values (from [1] to [10]), each of which is initially 0. The arguments are divided by commas. We chose ten because we are assuming that there will be no words of more than ten letters. If we assume the number *will* be higher then we must have a higher number, otherwise the longer numbers will be unaccounted for. How can we store the desired information in this list?

Previously, we did not store the length of any individual word since we wrote it out immediately. Let us use the variable length to store this value which we can obtain from the value of the write() command. At this point the program looks like this:

```
procedure main()
  while word_length(read())
end
procedure word_length(line)
  letters := list(10,0)
  line ? while length := write(*tab(many(&letters))) do
    tab(many(' .,;:!'))
  return
end
```

We could have written the second procedure differently. This way is less desirable, but it illustrates an important point:

```
procedure word_length(line)
 letters := list(10,0)
 line ? while length := (write(*tab(many(&ucase ++
   &lcase)))) do
    tab(many(' .,;:!'))
 return
end
```

The third line of the procedure became long, so we broke it into two. Since it is clearly incomplete (++ expects something to follow) Icon will look for the completion on the next line. Icon considers a line to be a complete statement unless it is clearly incomplete, such as a line which ends in a "+" or similar

operator, or a word like else which we shall find in conditional statements. All of these imply that the next line is also part of the statement. For example, the statement

```
n := 2 + 1
```

would assign the value 3 to n. If we wish to split the line, then

```
n := 2 +
  1
```

would have exactly the same effect. However,

```
n := 2
  + 1
```

would assign 2 and not 3 to n. Since the second line is a syntactically correct statement even though useless, it will not cause an error, but the result is not what the programmer intended, since the first line is understood by Icon as a complete statement. We now have a list called letters, and we have recorded the length of a word in length. In order to store length in letters we can write

```
letters[length] +:= 1
```

The symbols +:= written together mean that what is on the left is increased by the number on the right, or *incremented* by that number. Now let us say the word is "a". The variable length will have the value 1 and so the first variable in letters will be incremented. If the word is "please", then the sixth variable of letters, i.e. letters[6], will be incremented. Now this needs to be done each time we get a new word. However it is always assumed that the word do is followed by one and only one statement. This is covered by placing the *compound* statement inside curly brackets thus:

```
procedure main()
  while word_length(read())
end
procedure word_length(line)
  letters := list(10,0)
  line ? while length := write(*tab(many(&letters))) do {
    tab(many(' .,;:!'))
    letters[length] +:= 1
    }
  return
end
```

At this point letters has all the information stored, but no use is being made of it. Now we wish to print out a table of the length of the letters in our text. It would be a good idea to put this in a separate procedure since it is conceptually a separate item. But we have a problem. When procedures finish, all the variables they use are destroyed. This is done largely to save memory. We might fill up the memory of the computer with variables which we are not currently using. With the increase in the size of computer memory this is less of a problem than it used to be, but wasting memory is still worth avoiding. We can solve this by making letters a *global* variable, one that can be used by *all* procedures in the program and does not disappear when the procedure finishes for the time being. We do this right at the beginning by prefixing the word global. However there is an additional problem which is a little subtle. Each time the procedure word_length() is called, the assignment of the list to letters will be done again, with the result that the old list will be destroyed. This could be handled by making this the first statement in procedure main(), but a better solution is simply to place the word initial before it which means it will only be done the first time the procedure is called and not subsequently:

```
global letters
procedure main()
```

```
  while word_length(read( ))
end
procedure word_length(line)
  initial letters := list(10,0)
  line? while length := (write(*tab(many(&letters)))) do {
    tab(many(' .,;:!'))
    letters[length] +:= 1
    }
  return
end
```

It now remains to write a procedure to write out the table, include this procedure in the "traffic cop" main procedure and our task will be complete. Here it is:

```
global letters
procedure main()
  while word_length(read())
  printout()
end
procedure word_length(line)
initial letters := list(10,0)
  line ? while length := (write(*tab(many(&letters)))) do {
    tab(many(' .,;:!'))
    letters[length] +:= 1
    }
  return
end
procedure printout()
  every n:= 1 to 10 do
    write("There are ",letters[n]," ",n,"-letter words.")
  return
end
```

The explanation of printout() is as follows. You will recall that a while-loop executes repeatedly so long as the expression following the while succeeds. An every-loop by contrast functions so long as the expression following it *produces a result*. every-loops are used with Icon expressions called *generators* which can produce a group of results rather than only one. The expression 1 to 10 is such a generator, producing the whole numbers from 1 to 10, one at a time. Here n gets the values 1 to 10 in sequence. The command write() can take any number of arguments which are separated by commas, and will be written out one after the other. So the following pattern will be written ten times:

There are [the number of words of a specific length] [the specific length] -letter words.

For example:

There are 4 5-letter words.

SUMMARY OF ICON FEATURES INCLUDED IN THIS CHAPTER

1. ? invokes the line-scanning facility.

2. upto()
   value the position in the string after the first letter found in its argument.
3. tab()
   value the string between the old and new positions of the pointer.
   effect moves up the pointer to the position in its argument.
4. The asterisk returns the length of the following string.
5. Each program has a *main procedure* which executes when the program is run.
6. *icont* - *-x* invokes the Icon translator, expects input from the keyboard, and has the program immediately executed.
7. many()
   value the position in the string after all letters in its argument.
8. write()
   value its last argument.
   effect writes its arguments to the screen.
9. &ucase and &lcase are sets of the upper- and lowercase characters respectively. &letters is the union of these two sets
10. read()
    value a string read in from the keyboard.
    effect reads *in* a string from the keyboard.
11. A *list* is a *collection* of variables that share a name and are differentiated by an integer index that begins with 1, for example, letters[7] .
12. The successful *conclusion* of a procedure is indicated by return, which is followed by the value of the whole procedure if it has one. Note that here return refers to a word written into a program, and not the key of that name on the keyboard.
13. A *global variable* is accessible to all procedures. It must be declared at the beginning of the program.
14. A *generator* is an expression or a function which may produce a group of results rather than one result.
15. A *while-loop* continues so long as its control statement succeeds. An *every-loop* is used with *generators,* and continues so long as the generator produces a result.
16. A command preceded by the word initial is done only the first time the procedure in which it occurs is used. Such commands should come at the beginning of the procedure.

3. Storing and Editing Programs in Files

3.1 Making Information that Lasts

So far we have typed in programs from the terminal, and when data had to be input, we have typed that in too. This is satisfactory only for small programs and small amounts of data. There are two disadvantages. First, it is difficult to correct errors, either in program or data. If something goes wrong we have to do it over. Secondly, once the program is over, everything has disappeared. It is often desirable to retain both programs and data for future use. Also, they may be modified if necessary without having to start from the beginning. This is achieved by placing the program in a *file* and having it use data that is also in a file. A *file* in this context is a stream of information, and most often refers to one that has been fixed electronically onto a disk. However, the stream of information coming in from the keyboard is also technically a file, and so is the one appearing on your screen. It may be possible for you to hold the disk on which a file is registered in your hand, or it may be hidden away in a "hard" or "fixed" disk in your computer, or if you are using a time-sharing system it may be miles away and quite invisible to you. Formerly these files were often created by punching holes in cards that encoded the programs and data, and were transferred to disk by card reader machines. The most common method now is to enter the data using a keyboard and screen, and a program called an *editor* is used for this purpose. Word processors developed from these editors, but word processors are oriented towards ultimately printing the materials they handle and so possess features (word-wrap and justification, for example) which are unnecessary for editors. Many different editors are available with different capabilities, and choice is a matter of personal preference. Most word processing programs have a “text-only mode" or “save as text” option which allows them to serve as a program editor. The regular document mode is not suitable because it employs a binary file format that stores formatting, layout, and font information that is not allowed in program source code files. In the following section some hints are given on utilizing a commonly used and readily available editor. Any can be used provided that it uses the ASCII character set referred to in Appendix A.

Editors perform two major functions:

- They place new material into a file.
- They modify or correct material which already exists.

Of course, they are marvelous tools for many other tedious tasks, such as searching and replacing words or phrases.

3.2 The Emacs Editor

Emacs is an editor commonly found on the UNIX operating system, but it is not part of UNIX. (Actually the most widely-used version of Emacs belongs to GNU. GNU stands for: "GNU'S NOT UNIX." When you ask what *that* GNU stands for, you discover that you have a definition of a type known as recursive. Remember the cereal boxes carrying a picture of a boy holding the cereal box?) Emacs is often available on time-sharing systems, and may be copied freely, subject only to certain conditions which are stated in the documentation. Emacs is also available for personal computers. On Apple OS X and Linux systems, Emacs may be included in the software. To start Emacs you can type

```
emacs average.icn
```

which creates a file of that name, and you may proceed immediately to enter your text. You should use lowercase letters on the command line. You make the file permanent by holding down the *Ctrl* key and pressing *x*, releasing the *Ctrl* key and then pressing *s* (for *save).* On some systems you may need to continue pressing *Ctrl.* (If you find that your computer "freezes" when you press *Ctrl-S*, pressing *Ctrl-Q* should get it going again.) You leave Emacs by holding down the *Ctrl* key and pressing both *x* and then c.

Emacs has an enormous number of features, far more than most users ever need. Some of the most important will be described here. However, Emacs has a built-in tutorial which will acquaint you with the main features, and also an information tree which can be used as a reference. There is a convenient “cheat-sheet” at http://ccrma.stanford.edu/guides/package/emacs/emacs.html

In order to use the tutorial, type in

```
emacs
```

without naming a file. Then type in *Ctrl-H* (hold down the *Ctrl* key and press *h)* followed by *T.* The *Ctrl-H* summons up the "Help" facility, and *t* stands for *Tutorial.* You then simply follow instructions and are led through the main features.

Here are some of the major features of Emacs. *Control-V* (for *view)* enables you to page down through the file. Pressing the escape key and then *V* does the same thing in the opposite direction. It is worth pointing out that the control key modifies another key, much as the shift key does, so it must be held down while pressing the next key. The escape key, however, produces a character in its own right even though it is not normally visible on the screen, so it should be released before pressing the following key. Cursor control is achieved through *control-F (forward), control-B (back), control-P (previous line),* and *control-N (next line).* You can also use the arrows on the keypad. *Control-L* retypes what is currently on the screen around the current position of the cursor. Escape followed by the "less than" sign (which points back) takes you to the beginning of the file, and followed by the "greater than" sign (which points forward) takes you to the end. The backspace key *deletes* the character immediately before the cursor, and the delete key or *control-D* (for *delete) deletes* the character over the cursor. *Control-K* (for *kill)* deletes to the end of the line. This may be reversed by *control-Y* (for *yank).* This last feature may be used to move lines around the screen; kill the line, move the cursor to another place, and then yank it. The work you are doing is being held in a buffer, which is temporary storage. To make it permanent at any time press *control-X* and then *s* (for *save).*

Emacs does have automatic save features which mean that you are unlikely to lose too much work if there is some kind of breakdown involving your computer, but it is advisable to save explicitly from time to time. You can leave Emacs by pressing *control-X* and then *control-C.* Your implementation may allow you to leave Emacs temporarily (probably to tryout a program) by pressing *control-Z* and then you can return to the place you were with the command `fg` or possibly *%emacs*.

These commands are sufficient to get started with Emacs. You may also wish to look at the information tree. You can do this by summoning the help facility *(control-H,* or a local substitution) and then entering *i* for *info.*

3.3 Running an Icon Program From a File

We now have to learn how to handle a file containing a program so that the program can be run, and later how we bring data which is stored in a separate file into our program. This distinction between program and data is fundamental. In knitting we may compare the knitting pattern to the program and the yarn to the data. The yarn is manipulated on the basis of the pattern. In cooking we may compare

the recipe to the program and the ingredients to the data. The ingredients are reformulated in accordance with the recipe. Formerly programs and data were kept in different types of files, but that is no longer necessary. Files containing Icon programs must end in a period and the three letters *icn.* There are differing conventions as to the format of filenames, but generally if you keep them reasonably short and stick to alphabetic and numeric symbols, you will come up with valid filenames. Unlike the variables in an Icon program, filenames may begin with a numeric symbol. The Icon system takes this file and converts it into a machine-readable form. It takes the file name, chops off the period and "extension" as it is called; the three letters *icn* and this truncated form of the name become the name of the machine-readable file. In some implementations of Icon, the machine readable file ends in the extension *icx.* The original file (called the source file) continues to exist of course, and may be subsequently modified and reprocessed if necessary. The earlier machine-readable file will be replaced by doing so.

Let us say that we have created a file called *average.icn* which contains an Icon program. We type in

```
icont average
```

which submits the file *average.icn* to the Icon translator. Some intermediate files are created by the Icon system, but since they are later deleted, the programmer need not normally know anything about them. We may then make this translated file operate by typing in

```
./average
```

whereby Icon executes the translated file which it has created. On some systems you say “iconx average”, since the Icon executable files are in fact executed by a virtual machine named iconx. Icon files normally do not stand alone; the Icon system must be present when they are executing. Once an Icon program has been translated into an executable machine code file, the program may be rerun at any time by repeating the execution command (“./average” or “iconx average”). We may also perform the two steps on one line thus:

```
icont average.icn -x
```

where the -x argument tells the system to execute the file as soon as it has been processed. We shall learn how to handle data placed on a file in the next chapter.

This chapter covered writing Icon programs in a text editor and storing them in named files. While it is desirable to get accustomed to using a full-featured programmer's text editor, which ultimately saves time, it is quite acceptable to use the simple editors Notepad (on Windows) or TextEdit (on Apple OS) for writing programs. Even word processors such as Word can be used, but in all cases it is necessary to ensure that the output is in simple text format, without the hidden extras that Word and similar programs insert by default for their own purposes.

4. Graphs

4.1 Histograms

Now that we can record both data and programs in a permanent form, let us see how we may construct a frequency distribution graph in the form of a histogram and get it onto the screen. We want to represent the *word lengths* (from 1 up) along the horizontal axis so that the first spot represents words of one letter, the second spot words of two letters and so on. The vertical axis represents the *number of words* of each type (from 0 on up) starting from the bottom of the grid. Using "X" to fill a particular spot in the grid (since, for the moment anyway, we do not have access to the traditional rectangle) we get a figure that may look like this. Icon also has elegant 2D graphics facilities; see “Graphics Programming in Icon”, by Griswold, Jeffery, and Townsend.

```
   X X
   X X
   X X
   X XX
   X XX
   X XX
   XXXXX
   XXXXX
   XXXXX
  XXXXXXX
  XXXXXXXX
 XXXXXXXXXX X
 XXXXXXXXXXXX
XXXXXXXXXXXXX X
XXXXXXXXXXXXXXXX
```

There ought, of course, to be some indication of what the *X's* signify, but we can ignore that for the moment. When we draw such a figure by hand it is natural to start at the base at the 0,0 coordinates and move left-to-right and upwards. Although techniques (known as "cursor control") can mimic this, it is more natural for the computer to start printing out at the top and move down in its normal manner. Let us assume for the moment that there will be a maximum of thirty words in any column (up-to-down) and a maximum of ten letters per word in any row (left-to-right). Starting at the row representing thirty, we need to move along it ten times, inserting an X if there are thirty words for that particular column and a blank if there are not. This will then be repeated for row twenty-nine, inserting an X if there are twenty-nine *or more* words for that particular column, and a blank if there are not. We can express the movement down the columns by

```
every n := 30 to 1 by -1 do
```

Each time this statement is reached, every will call up the next value counting *down* by -1 so each time the value of n is reduced by 1. The expression 30 to 1 by -1 is an Icon specialty called a generator, which we met in the last chapter counting *up* from 1 to 10. It produces a *group* of values {30,29,28...3,2,1} and every effectuates a loop that keeps on going just so long as the generator continues to produce something (which in this case is assigned to n and can be used inside the loop). We can express the movement along the rows by

```
every p := 1 to 10 do
```

It is not necessary to express the "by +1" (although we may put it in if we wish) since this is assumed if no other value is mentioned. Such an assumed value is known as a *default*. The number of the iteration at any particular time is stored in the variable p, just as before. We now place the "row loop" inside the "column loop":

```
every n := 30 to 1 by -1 do
  every p := 1 to 10 do
```

In this way the p-loop will itself be performed thirty times. During each iteration of the p-loop the value of n will be constant, while that of p increases each time. This procedure is then repeated, with the value of n being reduced by 1. Now we have to extract the information contained in the list letters. At each point on the grid we want to put in an X if the p-th item in the list contains a number *equal to or greater than* n. This is a good example how variables enable the computer to be a *general* machine and is expressed by the command

```
  if letters[p] >= n then writes("X") else writes(" ")
```

There are several things to notice. First, the function writes() writes its argument to the screen, but unlike write() does *not* then proceed to the next line. When we get to the end of each row we are going to have to use write(), with its argument omitted, not to write anything, but simply to proceed to the next line. Second, the blank must be written expressly and comes in the other leg of the if-statement, beginning with the word else. Finally we need to note the symbol >= which means "is greater than or equal to." We now have:

```
  every n := 30 to 1 by -1 do {
    every p := 1 to 10 do
      if letters[p] >= n then writes("X") else writes(" ")
    write()
    }
```

The curly brackets are needed because the outer loop has within it two statements (the inner loop and the command to proceed to the next line when that loop finishes). Now do expects to be followed by only one command; if there are more than one, they must be enclosed within curly brackets. Notice how the indentation indicates the structure. The second every statement is controlled by the first, and so is indented. The if statement is controlled by the second every statement, and so is indented further. The write() statement is controlled by the *first* every statement, and so has the same indentation as the second every statement. Indentation is optional, but very helpful when looking at programs subsequently. The complete procedure now looks like this:

```
procedure histogram()
  every n := 30 to 1 by -1 do {
    every p := 1 to 10 do
      if letters[p] >= n then writes("X") else writes(" ")
    write()
    }
end
```

If we call this procedure from the main procedure, our program looks like the following. The function getch() waits for the user to strike any key. Reading user input after read() has already failed (indicating end-of-file) is a bit underhanded, but it works on most interactive terminals.

```
global letters
procedure main()
  while word_length(read())
  write("Hit any key to continue")
  getch()
```

```
  printout()
  write("Hit Return to continue.")
  getch()
  histogram()
end
procedure word_length(line)
local length
initial letters := list(10,0)
  line ? while length := (write(*tab(many(&letters)))) do {
    tab(many(' .,;:!'))
    letters[length] +:= 1
    }
  return
end
procedure printout()
local n
  every n:= 1 to 10 do
    write("There are ",letters[n]," ",n,"-letter words.")
  return
end
procedure histogram()
local n, p
  every n := 30 to 1 by -1 do {
    every p := 1 to 10 do
      if letters[p] >= n then writes("X") else writes(" ")
    write()
    }
  return
end
```

getch() expects to get a single character from the keyboard, thus dividing up the sections of the program for easier reading. There is also a function getche() which expects a single character, and echoes it to the keyboard. That is not needed here, but is useful if you wish the user to enter “y” (yes) or “n” (no), as its value can be used to input the user's choice to the program, for example:

```
 write("Do you wish to continue? y/n")
 answer := getche()
 if answer == "n" then stop()
```

Note that we specified n and p as *local variables* which function only in the procedure histogram, and we added return at the end of the procedure. The procedure would have worked quite well without these, since variables are assumed to function only within their procedure unless they are specified as global, and return is similarly unnecessary, because the entire program concludes at this point anyway, but it is good practice to specify local variables, and to make the procedure return properly. Local variables are specified by placing the word local at the beginning of the procedure followed by the names of the variables separated by commas.

4.2 Reading Data From a File

We have already learned how to record the program in a permanent form and run it from the file in which it has been preserved. It is also possible, and often desirable, to record the data in a file. The program then reads the data from the file instead of reading from the keyboard. Using Emacs, or any other editor which uses the ASCII character set, create a file called *sample.txt* and type in a paragraph

of text from a novel, poem, or any other source. We are distinguishing Icon source files by the "extension" *icn* and files containing texts by the extension *txt*. Instead of using the keyboard we wish to pull in the information from *sample.txt*, and we need some way of referring to this file in the program. The name itself will not do, since its value is simply a string of characters. Icon possesses a function open() which takes such a string as its argument, readies the file by that name for reading, and produces a value, which, unlike the string that names the file, *can* be used to reference it. This is then assigned to a variable of our choice:

```
  infile := open("sample.txt")
```

The main procedure now reads as follows:

```
procedure main()
local infile
  infile := open("sample.txt")
  while word_length(read(infile))
  close(infile)
  write("Hit any key to continue")
  getch()
  printout()
  write("Hit any key to continue")
  getch()
  histogram()
end
```

Material will now be read in, a line at a time, from the input file which is called "sample.txt" and is represented by the variable infile. When read() attempts to produce material from the file after the last line has been read, read() fails and the program moves on to the next item. When we have finished using the file we close it, using the function close(). In this instance there is no need to assign the value of close(), if any, to a variable since it would not be useful for any purpose. It is a good idea to close a file when it is not needed any more, since there may be limits on the number of files that can be open at one time. If this is omitted, however, Icon automatically closes all files when the program finishes.

The program as it stands at this point can only read from a file called *sample.txt*, which places a limit on the usefulness of the program. It is not general enough. It is worthwhile then to set up a separate procedure which will allow the user to specify what file is to be used. The *value* of this procedure will be the name of the file which the user wishes to use. Previously we have not used the values returned by procedures, but in this case it will be convenient to do so:

```
procedure getfilename()
  local filename
  write("What is the name of the input file?")
  filename := read()
  return filename
end
```

In this procedure we *prompted* the user to give the name of the file, and then returned it as the value of the procedure as a whole. In the preceding program, then, we replace the command

```
  infile := open("sample.txt")
```

by

```
  infile := open(getfilename())
```

After getfilename() returns the name of a file, we attempt to open it and assign it to infile. There is one further refinement which is desirable. What would happen if the user specified a file which does not exist? As it stands there eventually would be a *run-time error;* that is to say, the Icon system would stop the program and issue a message stating that it attempted to read from a file that does not exist. These messages, however, are meant for you as a program developer, not for the end user. To avoid the user getting a message which might not be understood, instead of the command

```
infile := open(getfilename())
```

we can write

```
(infile := open(getfilename()) | stop("File does not exist!")
```

In this case Icon will try first to execute the command, now within parentheses on the left. The upright bar specifies an alternative ("or") in the event that that command cannot be fulfilled. The function stop() simply stops the program dead in its tracks and writes on the screen the message, if any, which is its argument. This enables us to stop the program gracefully in a manner of our own choosing if continuance is impossible, rather than having the Icon system throw up its hands in disgust and declare that the programmer did not allow for the user's carelessness.

There is an alternative possibility. We can set up a loop that will only function if the attempt to open a file is unsuccessful. It will then solicit a new attempt by the user. When eventually the user enters an appropriate filename, the loop ceases:

```
while not (infile := open(getfilename())) do
   write("File cannot be opened. Please try again.")
```

In effect this means: "If the attempt to open the file is not successful, then enter the loop; otherwise continue." After the logical expression not it is a good idea to put what it controls in parentheses, otherwise it may attach itself to part of what follows with unexpected results. If we wish, we may use the word until instead of while not. Our program has now expanded to the following:

```
global letters
procedure main()
  local infile
  while not (infile := open(getfilename())) do
     write("File cannot be opened. Please try again.")
  while word_length(read(infile))
  getch()
  printout()
  getch()
  histogram()
end
procedure word_length(line)
  local length
  initial letters := list(10,0)
  line? while length := (write(*tab(many(&letters)))) do {
     tab(many(' .,;:!'))
     letters[length] +:= 1
     }
  return
end
procedure printout()
  local n
  every n:= 1 to 10 do
     write("There are ",letters[n]," ",n,"-letter words.")
```

```
  return
end
procedure histogram()
local n, p
  every n := 30 to 1 by -1 do {
    every p := 1 to 10 do
      if letters[p] >= n then writes("X")
      else writes(" ")
    write()
    }
  return
end
procedure getfilename()
local filename
  write("What is the name of the input file?")
  filename := read()
  return filename
end
```

4.3 Refining the Histogram

If you try the preceding program you will see that the histogram is very crude. It is squeezed into the left-hand side of the screen. If a text is chosen that has more than twenty or so examples of a word of a particular length, the top of the histogram will run off the screen. And the X's are not very attractive. Let us try to remedy these failings in turn. Icon possesses a function right() which produces the string which is its first argument padded out by blanks on the left so that it fills up a string the length of its second argument. Thus write("hi") will produce

```
hi
```

on the screen. write(right("hi" , 10)) will produce

```
        hi
```

on the screen, that is "hi" preceded by eight blanks to make up a total of ten. Accordingly, right before the p-loop in histogram() we can add

```
writes(right(" ",10))
```

which will place ten blanks at the beginning of each line. If we would like to start the line with the value at the point of the vertical axis we could use the corresponding function writes(left (n, 2)) which will write the value of n left-justified in two spaces. Thus whether the numeral printed out has one or two digits, the columns of the histogram will be in proper order. The second problem can be solved by scaling. If we take the statement

```
every n := 30 to 1 by -1 do {
```

and change the steps, the value of a particular *X* is augmented. Thus 85 to 5 by -5 would allow for a total of 85 occurrences of words of a particular length, and each *X* would be equivalent to five occurrences. This will occupy only 18 lines on the screen and will of course treat occurrences of less than five as a non-event. These numbers can be adjusted in any way convenient.

Finding something to replace the *X* is a little more complicated because it involves using characters which cannot be input from the keyboard, and is dependent on the particular kind of screen you are using. All the characters of the ASCII character set have a number. Icon has a function char() which

takes a number as its argument and returns the corresponding character as its value. So the value of char(65) is the uppercase *A.* As an exercise let us try to write a procedure to mimic what char() does. Let us call it chr(). Now Icon has several standard sets of characters. Such useful standard values are called *keywords* and are preceded by the character &. Thus &ucase is the set of all the capital letters from "A" to "Z". &lcase is the set of all the small letters from "a" to "z". &letters is the two previous sets combined. &digits is the set of numbers from "0" to "9". &ascii is the set of 128 ASCII characters from 0 to 127. &cset is the set of all the 8-bit characters from 0 to 255. This latter is known as the extended ASCII set. The characters from 128 to 255 may or may not have a graphic representation, according to the local conditions under which you are working. All we need to do is to find the correct character in the set and return it as the value of the function. This is achieved by the following:

```
procedure chr(n)
  if integer(n) then
    &cset ? {
      move(n)
      return move(1)
      }
end
```

&cset is actually a set, in which the *order* of the components is not significant, but Icon converts this set to a string for this purpose. These automatic conversions are characteristic of Icon. They contribute to Icon's status as a "high-level" programming language and certainly save time for the programmer. Function move() moves that imaginary pointer as many slots as have been put into the procedure. Since the first character is number 0, the value of move(n) is a string from the beginning of the set right up to the character we wish to capture. Since this string is of no particular interest to us, no use is made of it and it is discarded. But the pointer has been moved up, and by moving it up one more and returning that one-character string as the value of the function, our goal has been achieved. Note carefully the difference between move() and tab(). The first moves up the pointer by adding the number of its argument to the current position of the pointer in the string. The second moves it to the position in the string stated in its argument. So if you are starting at the beginning of a string, move(2) brings you to position 3 (since the first position is numbered 1), whereas tab(2) brings you to position 2. Try running the following:

```
procedure main()
  printout()
end
procedure printout()
local n
  every n := 0 to 255 do {
    writes(right(n,3) ,"=" ,char(n)," ")
    if n % 13 = 0 then write()
    }
end
```

Now, if you like, you can change char to chr, include the chr() function in the program and satisfy yourself that it works in exactly the same way.

Using the right() function in printout() ensures that most of the characters are lined up appropriately. The result is not completely satisfactory because when some characters are written to the screen they cause special movements of the cursor, such as number 8 which causes a backspace. The percent sign represents the arithmetic modulo, i.e., it gives the remainder of n divided by thirteen. This ensures that a maximum of thirteen items are included on each line, each item including the ASCII number, an

equal sign, the corresponding character, and a space. For clearer examination of the characters 128 and over, change

```
every n := 0 to 255 do {
```

to

```
every n := 128 to 255 do {
```

and

```
if n % 13 = 0 then
```

to

```
if n % 13 =  0 then { write(); write() }
```

The semicolon causes the following command to be considered as if it were a separate command written on another line. The extra pair of curly brackets is necessary so that both occurrences of write() are part of the then clause. Try leaving them out, and figure out why the effect on the screen is what it is. Also, try changing   n % 13 = 0

to

```
n % 13 = 10
```

and figure out why it produces a better-looking result on the screen.

Returning to our original problem, if we are lucky, we now have a choice of graphic-type characters that we might use to replace the *X*. 171, 177, 178, 219 may be possible candidates. Substitute char(171) (or another number) for the "X". In implementations that do not use the ASCII characters 128 to 255 for any graphic representation on the screen, you may try characters 24 and 26 which sometimes are represented by a rectangle on the screen. Unfortunately Icon on the Apple terminal has no graphic representation for the ASCII characters lower than 32 or higher than 126. However, by "writing" any of the characters 7 through 13 certain things happen with which you can experiment. For example, writing char(7) caused a bell to ring on the old teletype machines; strangely, this is represented on the Apple by a dull thud. It might be used to draw the users attention:

```
write(char(7),char(7),char(7),"Listen up!")
```

SUMMARY OF ICON FEATURES

1. The expression by -1 may be used in Icon to count down a loop.

2. A group of commands included inside curly brackets is treated as though the whole was a single command.

3. return is followed by the value of the function, if any, and indicates that the function has succeeded.

4. open()
   value a *file variable* which may be used in a program to reference a file.
   effect prepares a file for use in a program.

5. close()
   value its value is not useful, since the file it references is closed.
   effect closes a file, and prevents it from being further used in the program.

6. The upright bar (|) is the logical “or”. If the expression preceding it fails, the one following it is tried.

7. stop()
   effect concludes the program as soon as the message, if any, in its argument has been printed on the screen.

8. right() and left() control positioning on the screen by padding out the string which is the first argument by the number of blanks stated in the second argument. An optional third argument may use some other string instead of a blank

9. getch() and getche() both make the program await the input of a character by the user. The second echoes the character entered to the screen. The term “echo” means that what appears on the screen has been input by the user, and is not output from the program.

10. Keywords are preceded by an ampersand sign and contain useful values. Keywords presented back in chapter 2 included &ucase and &lcase, the upper- and lowercase character sets, and &letters, the set of upper and lower case letters. &digits is the set of numeric digits from 0 to 9. &ascii is the set of the first 128 ASCII characters. &cset is the set of all 256 characters.

11. The percent sign produces the remainder of the numeral before it divided by the numeral after it.

12. move()
    value the string between the old and new positions of the pointer.
    effect moves up the pointer in a string being scanned the number of places stated in its argument.

5. Measures of Central Tendency

5.1 The Mode

The mode represents the most common value, even though values almost as common may be very different from it. Our task is to take a text, divide it into three sections, and find the most common word-length in each. The task can be divided up as follows, and these subtasks will correspond to procedures in the program:

1. Get the name of the file containing the data and make sure that it actually exists.

2. Count the number of words in the file and divide by three.

3. Find out how many words of length 1 to 20 there are in the first third, find the most common value (the mode), and print out the results on the screen neatly; repeat this operation for the remaining two-thirds.

At this point it may prove convenient to devise a procedure which pulls a single word from a file every time it is called until the end of the file is reached. This procedure becomes in effect a "black box" as it is called; at one end an entire file enters, at the other the words pour out in sequence. Once we have the procedure, we can forget about how it works, and this is what the black box is: a *machine* which works without requiring the owner to know how. We just have to be sure to give it the right input and use the output appropriately. Our procedure operates much as a food processor turns vegetables into soup, although turning soup into vegetables might be a closer analogy. We first need to define the characters which may occur in words and the punctuation marks (including the space) which mark them off. This may be achieved tentatively by:

```
chars := (&letters ++ &digits ++ '\'-')
punct :=' .,?";:!'
```

The double plus signs are used for adding sets, an operation that is often called a set *union*. We add the set of letters to the numerals, the dash, and the apostrophe. Note that the apostrophe (or "single quote") is preceded by a backslash. If we did not put it in, Icon would assume we meant it as the closing marker of the set which is marked off by single quote marks. There is a slight problem here: Does the word "boy's", for example, contain four letters or five? This is really a matter of definition, involving perhaps a distinction between letters and characters, and for our purposes it will be simpler to assume five in this case. We can cope with this problem later if necessary. Another problem will occur if single quote marks are used in the file much as double quote marks are used. This problem too can be coped with if necessary, but for the moment we may want to be sure that our data uses single quote marks only as apostrophes. We let the procedure know the name of the relevant file by including a variable in the parameters of the procedure, i.e., in the parentheses which follow the procedure name. The procedure heading will then be:

```
procedure getword(filename)
```

This file is opened by:

```
filvar := open(filename)
```

The function open() takes the filename, prepares the file for reading and assigns a *file value* to the variable filvar. Note the difference between the value of filename and filvar. The first is a string of characters which is the *name* of the file. The second is the file itself. We assume that a check has

already been made that a file of this name actually exists, and will discuss this later. We are now free to read a line from the file and chop it up into words much as we did before. Instead of using return to precede the value that will be the value of the procedure, we use suspend. This returns a value just like return, but leaves everything in the procedure in place so that it can continue when called again in a loop. Normally, when a procedure returns, all the variables it has employed disappear, thus making room in the memory for storing other necessary information. But in this case a loop like

```
every write(getword(filename))
```

will produce every word in the file and write it to the screen. Here now is our complete procedure:

```
procedure getword(filename)
#This procedure produces one word at a time from the file
local chars, punct, filvar, line, word
  chars := (&letters ++ &digits ++ '\'-')
  punct:='.,?";:!'
  filvar := open(filename)
  while line := read(filvar) do
    line ? {
      tab(many(' ')) #skip leading blanks
      while word := tab(many(chars)) do {
        tab(many(punct))
        suspend word
        }
      }
    close(filvar)
end
```

This procedure should always be used to process the entire file. If it is stopped at some point in the middle, and later resumed, the results may be unexpected.

For the first time we have added a *comment.* When Icon sees the pound sign # (also known as the number sign or crosshatch) it ignores it and the rest of the line. This enables us to make comments to aid the understanding of the program. These comments are written in normal English and can be as long as desired, provided each line or part of line begins with #. Another comment explains that we move up the imaginary pointer until just after any blanks that begin a line. This refers to indentation of course. When the file has been completely read, an attempt to read again fails, and the file is closed.

The preceding procedure makes it an easy matter to count the number of words in a file:

```
procedure countfile(filename)
#Counts the number of words in a file
local total
  total := 0
  every getword(filename) do total +:= 1
  return total
end
```

Here we pass the name of the file (previously checked) to this procedure which simply calls getword() and counts the number of times that it works. This is done by setting the value of a local variable (called here total) to zero and incrementing it on each turn of the loop. The value of total becomes the value of the procedure. This can be captured by assigning it to a variable, or using it as the argument for some other function.

We mentioned that we have to have some way of getting a filename from the user and checking that the file actually exists. We suggested two ways of doing this: checking and stopping the program if the file does not exist, or alternatively keep on asking the *user* for a filename until a valid filename is received. It is possible to combine these two approaches. We can give the user a fixed number of tries, and then if it still isn't right, halt the program:

```
procedure get_valid_filename()
#This procedure gives user three chances to select a file
local filename
  every 1 to 3 do {
    writes("What is the name of the input file? ”)
    filename := read()
    if close(open(filename)) then return filename else
      write("Unable to open file.")
    }
  stop("Bye!")
end
```

Let us first look inside the every loop. The program solicits a filename which will be typed on the same line as the question since writes() was used, and does not issue a carriage return. (There is a blank at the end of the question to make a small division.) Whatever is read in from the keyboard is stored in the variable filename. The command

```
  close(open(filename))
```

attempts to open the file and immediately close it. The value produced by open() is passed immediately to close(). If the attempt is successful, then the string of characters which constitutes the name of the file is immediately returned as the value of the procedure, jumping clean out of the loop and the procedure. If this is *not* the case, a message is given to the user and the process is repeated. This can only be repeated a total of three times. If the third attempt fails, the loop is left normally, and the program is stopped with a farewell. You might ask why the file is not left open, since presumably we plan to use it. It is probably best to allow the procedures that need it to open their own files and close them when finished. Closing the file enables the next procedure which needs the file to start at the beginning. If a file is left open after use, we remain at the point reached in the file, which will generally be the "end of file" or *EOF*, as it is known. In this case there would be no problem, since the file has not been read, but it is best to get accustomed to closing files as soon as they are no longer needed.

Now we have to consider a procedure to do the actual work of processing the information we wish to display. The procedure getmodes(filename) receives the name of a valid filename from the main program, and will read as follows:

```
procedure getmodes(filename)
#Does the main work of the program
local filelength, limit, section, counter, letters, word
  filelength := countfile(filename)
  limit := (filelength / 3)
#Section is the numeric name of one of the sections of the file
  section := 0
#Counter triggers a printout when it reaches limit
  counter := 0
  letters := list(20,0)
  every word := getword(filename) do {
    #Increment the appropriate element in the
    #list of word-lengths
```

```
    letters[*word] +:= 1
    counter +:= 1
    if counter = limit then {
      #Increment the section number, pass it to
      #printout() with the list
      printout(section +:= 1, letters)
      #Reinitialize letters and counter
      letters := list(20,0)
      counter := 0
      }
    }
  return
end
```

The procedure contains a rather large number of local variables. The variable filelength holds the number of words in the file, and is found by using the procedure countfile. This number is divided into three and the result stored in the variable limit which is the number of words in each of the three sections we shall consider. The expression

```
  limit := (filelength / 3)
```

contains an example of integer division. If one whole number is divided into another whole number, the result will be a whole number with any remainder discarded. Note that the number is not rounded to the nearest number, it is rounded down. To compute a result with a decimal, use 3.0 rather than 3. This is not necessary here where we have no need to handle fractions of words. In this instance the variable filelength is not used much and we could save a variable by replacing the two commands

```
  filelength := countfile(filename)
  limit := (filelength / 3)
```

by

```
  limit := (countfile(filename) / 3)
```

The variable section will be used to number the sections when we print them out. We shall make its initial value 0, then right before each time we use it, we shall increase its value by one. Similarly, the variable counter is initially zero, and will have its value increased by one each time we pull a word. When it reaches the limit, which is one-third of the file, it will trigger a printout of results so far, enabling the process to start over. The variable letters is a list of twenty variables, which will be used to hold the number of occurrences of a particular word length. Thus if there are five occurrences of four-letter words, the value of letters[4] will be 5; if there are ten occurrences of two-letter words, the value of letters[2] will be 10. The initial value of each member of the list is 0, and there are twenty members because we suppose that is the maximum likely length of any word in English. (The number would have to be higher for German, for example.) The variable word holds the current word that we are dealing with. Now let us see how the procedure works. A word is extracted from the file by getword(). Its length is ascertained by prefixing an asterisk and this is used as the index of the list

```
  letters[*word]
```

which is increased by one. Each time we do this we check if the number of times it has been done is equal to the limit of one-third of the length of the file. When this occurs, we take time out to print what we have achieved so far by calling printout(), and when that has been done, return to getmodes(), reset the counter and all the elements of the list to 0 and start over with the next section of the text.

Now let us see what this diversion to printout() achieves. This procedure receives two pieces of information: the number of the section (from one to three) on which it is currently working, and the list containing the information which getmodes() has acquired. There are three local variables:

- greatest—holds the greatest value in the list.
- n—holds the current index from 1 to 20.
- mode—holds the index which points to the greatest value at any particular time, ultimately pointing to the greatest value of the entire list.

We simply print out the values in the list from the top down, and compare greatest which is initially zero with the particular value. If the value is greater, it replaces the current value of greatest and that particular index becomes the mode, and is stored in the variable mode. At the end we print that out too, and go back to getmodes(). Our complete program is now as follows:

```
#This program divides a text into three parts, finds the
#number of words of length 1-20, prints out a list of
#each and gives the mode in each case.
procedure main()
  getmodes(get_valid_filename())
end

procedure get_valid_filename()
#This procedure gives user three chances to open a file
local filename
  every 1 to 3 do {
    writes("What is the name of the input file? ")
    filename := read()
    if close(open(filename) then return filename else
      write("Unable to open file.")
    }
  stop("Bye!")
end

procedure getword(filename)
#This procedure produces one word at a time from the file
local chars, punct, filvar, line, word
  chars := (&letters ++ &digits ++ '\'-')
  punct := ' .,?\";:!'
  filvar := open(filename)
  while line := read(filvar) do
    line? {
      tab(many(' ')) #skip leading blanks
      while word := tab(many(chars)) do {
        tab(many(punct))
        suspend word
      }
    }
  close(filvar)
end

procedure countfile(filename)
```

```
#Counts the number of words in a file
local total
  total := 0
  every getword(filename) do total +:= 1
  return total
end

procedure getmodes(filename)
#Does the main work of the program
local filelength, limit, section, counter, letters, word
  filelength := countfile(filename)
  limit := (filelength / 3)
#Section is the numeric name of one of the sections of the file
  section := 0
#Counter triggers a printout when it reaches limit
  counter := 0
  letters := list(20,0)
  every word := getword(filename) do {
    #Increment the appropriate element in the list of word-lengths
    letters[*word] +:= 1
    counter +:= 1
    if counter = limit then {
#Increment the section number, pass it to printout() with the list
      printout(section +:= 1, letters)
#Reinitialize letters and counter
      letters := list(20, 0)
      counter := 0
      }
    }   #end of the every loop
  return
end
procedure printout(p, L)
#prints out the info. p is the section no., L is the list.
local greatest, n, mode
#Assume the lowest possible value for the most occurrences #& correct it
  greatest := 0
  write("Section no. ", p, ".")
  write("Length     Number Such")
  every n := 20 to 1 by -1 do {
#Arrange info in columns
    write(right(n, 2),right(L[n], 12))
#Update greatest and mode if necessary
    if L[n] > greatest then {
      greatest := L[n]
      mode := n
      }
    } #end of the every loop
  writes(" ", "Mode = ", mode)
  write(right("Press any key to continue.", 52))
  getch()
  return
end
```

5.2 The Arithmetic Mean

The substitution of the arithmetic mean or average for the mode is quite simple. The procedure printout() needs to know the number of words with which it is dealing so that it can figure the mean. We achieve this by making limit a global variable, which means that it can be used by all procedures in the program. So we declare this variable at the very beginning of the program preceded by the word global. We remove it as a local variable in getmodes() which perhaps now we should call getmeans(). It would be possible to pass this value to printout(), but since it never changes there is not much point, and it might as well be stored once for all in limit. The beginning of our program will now read as follows:

```
#This program divides a text into three parts, finds the
#number of words of length 1-20, prints out a list of
#each and gives the mean in each case.
global limit
procedure main()
  getmeans(get_valid_filename())
end
```

We set up a local variable in printout(), letter_total, to hold the total number of letters. letter_total is initialized to 0, and at each turn of the loop we increase it by the value of L[n] multiplied by n (i.e., the number of words multiplied by the number of letters in those words). So our revised printout() will look like this:

```
procedure printout(p, L)
#prints out the info. p is the section no., L is the list.
local letter_total, n
  letter_total := 0
  write("Section no. ", p, ".")
  write ("Length              Number Such")
  every n := 20 to 1 by -1 do {
    # Arrange info in columns and figure letter total
    write(right(n, 2), right(L[n], 12))
    letter_total +:= (n * L[n])
    }
  #Write out the mean
  writes(" ", "Mean = ", real(letter_total) / limit)
  write(right("Press any key to continue.", 52))
  getch()
  return
end
```

Note that we convert one of the items in the division to a real number (i.e., one with a decimal point) so that the result will be precise. If arithmetic functions are performed on numbers of which at least one is real, the answer will be real. The function real() converts an integer, or a string if feasible, into a real number. One question arises. What would happen if the file were empty? It might appear that we should divide by zero and thereby cause a run-time error. In point of fact there is no problem, because getword() will fail as soon as it is called, with the result that printout() will not be called at all. However, this is something to watch for. It is always a good idea to test a program by offering it unusual data, such as an empty file.

5.3 The Median

In order to figure the median, only printout() needs to be changed from the form it took in the previous version of our program, although we may wish to change the name getmeans() to getmedians( ). First we must get a list of cumulative frequencies, based on the list of frequencies which was prepared in getmedians(). We set up this totally distinct list in the same way as before:

```
lcf := list(20, 0)
```

We now have a list called lcf containing twenty elements initialized to zero. The first value in the list is identical with the first value in the frequencies list:

```
lcf[1] := L[1]
```

We then increment each value of the cumulative list by the next value in the frequencies list, and that becomes the next value of the cumulative list:

```
every n := 2 to 20 do
  lcf [n] := (lcf [n - 1] + L[n])
```

Our list of cumulative frequencies is now complete. The parentheses are put in to ensure that the addition is done before any assignment is made to lcf[n]. Items within parentheses are always computed first, and their use may avoid an incorrect computation. It is often safer to put in parentheses than try to remember the priorities that Icon assigns, even though it is quite consistent in the manner in which this is done. We now compute the value which contains the median. We set an index n to zero and increase it so long as the corresponding value in the list of cumulative frequencies is less than half of the total:

```
#calculate vcm
n := 0
while lcf[n +:= 1] < (limit / 2.0)
vcm := n
```

As soon as lcf[n] overshoots one-half of the total, the loop stops. This while loop contains only the expression which controls the loop, so the word do which normally introduces the body of the loop is missing altogether. This is in order, since we are interested in the value of n which is part of the controlling expression and increases until it becomes the number we are seeking. It is then the number of the value containing the median and is stored in vcm. We can now compute the median by subtracting 0.5 from the value containing the median (since the upper range of the next value down is in between the two values) and add it to the cumulative frequency of the lower value subtracted from one-half of the total divided by the frequency of the value containing the median. This gives us the following:

```
#calculate median
  median := ((vcm - 0.5) + ((limit / 2.0 - lcf[vcm - 1]) / L[vcm]))
```

with the entire procedure as follows:

```
procedure printout(p, L)
local lcf, vcm, n
#lcf is a list of cumulative frequencies,
#vcm is the value which contains the median, n is the index
#The procedure prints out the info. p is the section no., L is the list of frequencies.
#calculate lcf
  lcf := list(20,0)
  lcf [1] := L[1]
  every n := 2 to 20 do
    lcf[n] := (lcf[n - 1] + L[n])
#calculate vcm
```

```
   n := 0
   while lcf[n +:= 1] < (limit / 2.0)
   vcm := n
#calculate median
   median := ((vcm - 0.5) + ((limit / 2.0 - lcf[vcm - 1]) / L[vcm]))
   #Arrange info in columns
   write("Length      Number Such")
   every n := 20 to 1 by -1 do
      write(right(n, 2), right(L[n], 12))
   writes(" ","Median = ", median)
   write(right("Press any key to continue.", 52))
   getch()
   return
end
```

5.4 Variants on the Program

You may wish to try modifying these programs to do slightly different tasks. For example, you could remove the feature of splitting the text into three and allow it to find the various averages for an entire text. You could solicit the user to tell you into how many pieces the text should be split before proceeding. This would need a prompt for the user, and the response would be stored in a variable which would replace the fixed number three. Note that in this case it would be a good idea to check that the user enters a valid number. One way to do this is to use the function integer() which converts its argument to an integer if that is possible, and fails if it is not. Hence, if the user would enter *q* instead of 2 the function would fail and an error message could be issued. You might get one program to print out on the screen several averages. You might include more than one version of printout(), giving them different names to keep them distinct, and call each version. It would be a good idea to make a copy of a working program with a different name and use the copy for experimentation of this type.

SUMMARY OF ICON FEATURES

1. The union of sets may be achieved by using the double plus sign (++). The double minus (--) and double multiplication sign (**) may similarly be used for set difference and intersection.

2. If we wish to include a single quote in a set (itself delineated by single quotes) we must precede the single quote with a backslash(\).

3. The word suspend returns a value from a procedure, but leaves the current status of the procedure in place for another call.

4. Comments may be inserted freely in a program provided that on each line they are preceded by the number sign (#).

5. If one integer (whole number) is divided by another integer, any remainder is discarded. A calculation which contains a real number (one with a decimal point) will result in a real number.

6. Global variables, which are available to all procedures in the program, must be declared at the beginning of the program preceded by the word global.

7. The function real() converts its argument to a real number (one with a decimal point) if it is possible, and fails otherwise.

8. Normally multiplication and division have precedence over addition and subtraction, but this order may be changed by using parentheses. Operations within parentheses are always done first. Thus 2 + 5 * 8 evaluates to 42, while (2 + 5) * 8 evaluates to 56. It is a good idea to use parentheses freely, in non-arithmetic expressions also, for clarity and to avoid unexpected results.

6. Icon and Unicode

6.1 Introduction to Unicode

The "native script" of the Icon Programming Language is the extended character set of ASCII, which stands for *American Standard Code for Information Interchange,* comprised of 256 characters numbered from 0 to 255. The only alphanumeric characters recognized are those of the Roman alphabet, and the usual numerals used in the west. For example, character number 065 is capital A of the English alphabet. Clearly such a system, devised when computers used mainly English, is inadequate to cover even European languages, let alone world languages. There is now a system called *Unicode* which offers a character set having tens of thousands of characters, covering almost all of the scripts which are, or have been, used on this planet. Despite the limitations placed on Icon by its adherence to ASCII, Icon is capable of handling so-called *character entities*, which evolved to represent individual characters for use on new developments such as *Hypertext Markup Language* (HTML) used on the World Wide Web which has developed so dramatically over the past decade. HTML is used not only on the World Wide Web, but with other items such as email. HTML is still very much in development, and it is difficult to tell now what its future will be. It certainly has come far in the few years of its existence. Originally a character entity was of the form &#*nnn*; where *nnn* was a three-letter integer in the range 000—127 (several of these numbers were not in use) and this represented the same characters of the brief ASCII character set. There were only three so-called control characters, 009 (tab), 010 (linefeed), and 013 (return). The formerly dominant Netscape browser brought into use the extended ASCII character set, and new characters were included in the 3.2 HTML standard. For example, é represents the letter e with an acute accent (é). Some of these numeric entities can be replaced with more easily remembered *named entities*. So é is precisely equivalent to é. It must be said that the choice of entities was rather arbitrary: < (the "less than" sign) has an equivalent named entity <, whereas = (the "equals" sign) does not, probably because the "less than" sign was used as a pointy bracket in HTML to begin a "tag", which is used to give instructions to the browser, and they wished to avoid confusing the browser. By the same token Þ represents a sign used only in Icelandic, yet there is no representation for the a with a tilde (~) used in Portuguese, a language spoken by many millions more than Icelandic. (Could there have been an Icelander on the committee?) Unicode was elaborated in the 1990s to remedy this chaotic situation. It is a carefully thought out system, still in process of revision and improvement.

Unicode customarily uses hexadecimal numbers (base-16 or hex numbers) but in HTML you may use decimal numbers if you wish. In addition to the usual decimal digits 0—9, hex numbers use A—F for the numbers represented in decimal by 10—15. Thus the Russian capital A (which is quite similar to the English letter, but treated as a separate entity) is represented in Unicode for HTML as А using the notation explained in the previous paragraph. If you use hex numbers you must write А. The x tells the browser that this is a hex number. 0410 is the hexadecimal equivalent of 1040. Multiply 1 by 16, and add it to 4 multiplied by 16 squared to satisfy yourself of this. You can ascribe to the Arabs the fact that you naturally move from right to left when you do this. Arabic is written from right to left, so they placed the least significant digit on the right.

Unicode is quite complex, but it is possible to find on the World Wide Web as much help as you need, and we shall give some pointers to this presently. Here we shall try to show how Icon might be used with three non-Roman scripts, each of which presents different issues. One is a non-Roman, Indo-European script (Russian), one is Indic (Tamil), and one is Semitic (Hebrew). The three are dissimilar,

but typical of the varying scripts which Unicode includes. Even if you are unfamiliar with these languages, you should find the information of value, beyond the particular problem dealt with here.

It should be noted that character sets such as ASCII and Unicode are distinct from *fonts*. A font is defined as a collection of typeface of one style and size; italic and bold fonts are well-known, but there are many others. Character sets make no determination of what font should be used; there is an HTML tag which can be used to recommend a particular font to the browser, which will use it if it is available. Note also that Unicode may repeat items similar in appearance for different scripts. English capital A which we mentioned before occurs in the Russian character set as number 1040 (decimal) or 0410 (hex) and looks quite similar. (Spammers sometimes use this feature to beat filters!)

6.2 Working with Russian, Tamil, and Hebrew

Russian uses a subset of the Cyrillic alphabet which is based on the Greek and Hebrew alphabets. It is used by a large number of languages, Serbian and Mongolian among them, for example, although since the collapse of the Soviet Union where the use of Cyrillic was fostered, some have switched to Roman script. Russian has 33 characters, upper and lower case, which is seven more than English, so no one-to-one equivalence is possible. One character ("yo"), mainly used in texts meant for foreigners or children, is excluded, and will be handled separately. Modern microcomputers handle Russian readily, but there is an additional problem: the characters are keyboarded according to the customary Russian keyboard, which has an arrangement totally different from the English keyboard. Icon is able to convert an easily memorized set of correspondences of the individual's own choosing to the appropriate Unicode characters. This makes it possible to create a Russian HTML text in Unicode which can be used on the World Wide Web or in email. We shall call this a *qwerty representation*, in reference to the traditional arrangement of the top line of the English typewriter keyboard. (This arrangement was originally intended to *slow down* the typist using a mechanical typewriter to avoid the keys' jamming. This gave rise to the expression "qwerty effect" to signal an arrangement that one is stuck with even though the original reason has evaporated.) We use the upright bar(|) in front of an alphabetic letter to represent another similar, but distinct, letter. This doubles the reach of the English alphabet.

Tamil uses a syllabary, that is to say each symbol usually represents a syllable, rather than a single sound. It therefore has more than 100 characters, which are, however, partially similar. The native reader sees each of these characters as a single entity, but such *grapheme clusters* will be represented ingeniously in Unicode by only two Unicode characters, whatever cluster is involved. Here too we can create with Icon a mode of entering Tamil which suits our taste in transcription. In order to make the code consistent, the short vowel *a* will be represented by the same letter on the keyboard, and will actually insert an empty string.

There is an additional issue with Tamil. Tamil has more slots in Unicode than it actually needs, and more than 20 of these slots are left empty, occurring unevenly within the set. I suspect this is due to the fact that originally, as a syllabary, Tamil was given a large allotment, but it was found that not all were needed. Another qwerty effect, I suppose. This is handled in our list by insertion of the empty string ("") which becomes the key to the blank element. Each of the empty strings replaces the previous one in the table, so the table is finally increased in size only by 1.

Hebrew has two substantial problems. First, it *optionally* uses vowel representations which work essentially like Tamil, effectively creating syllables rather than letters. Modern Hebrew usually does not bother with them, since the native speaker finds the consonantal shp 'f th wrd sffcnt. (You were able to read that, wrn't y?) Unicode has a rich representation of this complex system, but we shall ignore it here. If it is important to you, you will become able to find ways to handle it yourself. Secondly,

Hebrew, like most Semitic languages, is written from right to left, yet both printing and scanning have traditionally moved from left to right in computers for the obvious reason that these innovations were created in the west. Computers handle this by expecting left-to-right input, which is processed by *prepending*, rather than *appending*, characters. Unicode is supposed to have a right-to-left mark (200E), but it does not appear to be implemented.

It must be pointed out that modern browsers all handle Unicode to some degree, but with a varying measure of completeness. There may even be differences in the same browser when it is used with different platforms. It is to be expected that this situation will gradually improve. In the meantime, it is well to test the results of your activities on various browsers. Let us now consider each language in turn.

6.2.1 Russian

Cyrillic characters in Unicode start at 0400 hex, 1024 decimal, but the specifically Russian characters are at 0410—044F plus 0401 and 0451 hex (1040—1103 plus 1025 and 1105 decimal). These appear in the following fully commented program.

```
#This program converts a text from Roman to Cyrillic (Russian) HTML
global chartable
procedure main()
  initialize()
  process()
end

procedure initialize()
local n,m, charlist
  # Create table. Maverick chars return ?? which is the default.
  chartable := table("??")
  # charlist is in the order of the Russian alphabet, capital letters first,
  # but yo (e with an umlaut) is excluded and dealt with under "special items"
  # below. In the input file zhe, che, shcha are preceded by a vertical bar.(|)
  # ~ (tilde) is used for the hard sign, and |~ for the capital hard sign.
  # ` (grave) is used for the soft sign, and |` for the capital soft sign.
  # The equivalents of sha and kha are chosen for their resemblance to the English letter.
  # Normally an Icon statement ends at the end of the line. If the statement is incomplete
  # however, the compiler will look to the next line, as in this long charlist. The list is
  # created here by placing the items between square brackets [ ].
  charlist := ["A","B","V","G","D","E","|Z","Z","I","J","K","L","M","N","O",
              "P","R","S","T","U","F","X","C","|C","W","|W","|~","Y","|`","|E","|U","|A",
              "a","b","v","g","d","e","|z","z","i","j","k","l","m","n","o",
              "p","r","s","t","u","f","x","c","|c","w","|w","~","y","`","|e","|u","|a"]

  #initialize list index counter
  m := 0
  #n holds a constantly augmenting unicode number; m uses the augmented assignment +:=
  #this type  of loop was explained in chapter 2.3. A character entity is created after :=
  every n := 1040 to 1103 do
    chartable[charlist[m +:= 1]] := "&#" || n || ";" #put 64 chars in table
  #Special items
  chartable["{"]  := "&#171;"  #Russian left angle quote  (left guillemet)
  chartable["}"]  := "&#187;"  #Russian right angle quote (right guillemet)
  chartable["|O"] := "&#1025;" #upper case Yo
```

```
   chartable["|o"] := "&#1105;" #lower case yo
   chartable["["]  := "<br />"  #HTML newline tag
   chartable["]"]  := "</p><p>" #HTML end old, begin new paragraph tags
   # table is complete
   return
end #initialize()

procedure process()
local filename,title,infilvar,outfilvar,line,newline,char
   # filename and title are provided by user. infilvar and outfilvar are
   # the input and output file variables. line is taken sequentially from
   # the input file, processed and written to newline which is then appended
   # to the new html file. Your input file can be created by any editor or
   # word processor, but be sure it is in text only format. It is preferable
   # for the filename not to have a suffix. Omit the underlines if you wish
   # the browser to take care of line breaks.

   writes("Enter the name of previously prepared file: ")
   filename := read()
   writes("What is the title of the text? ")
   title := read()
   # if the input file cannot be found, the program aborts.
   (infilvar := open(filename)) | stop(filename," not found.")
   # since all is well open a file for writing
   outfilvar := open(filename || ".html","w")
   # write to the output file a simple HTML header.
   write(outfilvar,"<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0Transitional//EN\">")
   write(outfilvar,"<html>")
   write(outfilvar,"<head>")
   write(outfilvar,"<title>",title,"</title>")
   write(outfilvar,"<body>")
   write(outfilvar,"<p>") # begin a paragraph
   # loop brings in lines from input file
   while line := read(infilvar) do {
      # initialize newline to the zero string
      newline := ""
      # scan the input line
      line ? {
         # loop processes the character, terminates at end of line when move() fails
         while char := move(1) do {
            if char << "A" then newline ||:= char  #punctuation,numerals.
            # << means: is lexically lower than "A" in the ASCII character set.
            else if char == "|" then newline ||:= chartable["|" || move(1)]
            else newline ||:= chartable[char]
            }
         }
      write(outfilvar,newline)
      }

   # write a simple HTML footer to the output file. Program end closes files.
   write(outfilvar,"</p>") # close last paragraph
   write(outfilvar,"</body>")
   write(outfilvar,"</html>")
```

```
   write("File ",filename,".html has been created.")
   return
end #process()
```

Let us compile this program, and submit to it the following brief Russian text in a file we call simply joke. You may omit the square bracket if you want the browser to take care of linebreaks.

```
K vra|cu prixodit vos`mides|atiletnij akt|or:[
--Doktor, ka|zdoe utro, vstav s posteli, |a  pervym delom |cita|u gazety.
I esli ne vi|zu v nix nekrolog o moej smerti, odeva|us` i idu v teatr.
```

(Meaning: An eighty year old actor goes to the doctor [and says:] "Doctor, first thing every morning I read the newspapers. And if I don't see in them an obituary about my death, I get dressed and go to the theater.")

Here is the output of the program:

```
!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0Transitional//EN">
<html>
<head>
<title>A Joke</title>
<body>
<p>
&#1050; &#1074;&#1088;&#1072;&#1095;&#1091; &#1087;&#1088;&#1080;&#1093;&#1086;\
&#1076;&#1080;&#1090; &#1074;&#1086;&#1089;&#1100;&#1084;&#1080;&#1076;&#1077;&\
#1089;&#1103;&#1090;&#1080;&#1083;&#1077;&#1090;&#1085;&#1080;&#1081; &#1072;&#\
1082;&#1090;&#1105;&#1088;:<br />
--&#1044;&#1086;&#1082;&#1090;&#1086;&#1088;, &#1082;&#1072;&#1078;&#1076;&#108\
6;&#1077; &#1091;&#1090;&#1088;&#1086;, &#1074;&#1089;&#1090;&#1072;&#1074; &#1\
089; &#1087;&#1086;&#1089;&#1090;&#1077;&#1083;&#1080;, &#1103;  &#1087;&#1077;\
&#1088;&#1074;&#1099;&#1084; &#1076;&#1077;&#1083;&#1086;&#1084; &#1095;&#1080;\
&#1090;&#1072;&#1102; &#1075;&#1072;&#1079;&#1077;&#1090;&#1099;.
&#1048; &#1077;&#1089;&#1083;&#1080; &#1085;&#1077; &#1074;&#1080;&#1078;&#1091\
; &#1074; &#1085;&#1080;&#1093; &#171;&#1085;&#1077;&#1082;&#1088;&#1086;&#1083\
;&#1086;&#1075;&#187; &#1086; &#1084;&#1086;&#1077;&#1081; &#1089;&#1084;&#1077\
;&#1088;&#1090;&#1080;, &#1086;&#1076;&#1077;&#1074;&#1072;&#1102;&#1089;&#1100\
; &#1080; &#1080;&#1076;&#1091; &#1074; &#1090;&#1077;&#1072;&#1090;&#1088;.
</p>
</body>
</html>
```

Two observations may be made about this output. First, the data is difficult to read unless you have an unusual memory. However, items such as punctuation, and HTML tags can help find a location if editing is called for. Secondly, the HTML coding is extremely lean and to the point, without the large amount of surplus coding that front-ends tend to insert. It results in this quite satisfactory result:

6.2.2 Tamil

Unicode saves us a lot of work by enabling us to represent the complex Tamil syllabary relatively simply. A single syllable may be represented by one, two, or three graphs on paper, but Unicode takes care of that for us automatically.

Here is a table of the Tamil characters with the Qwerty representation.

CONSONANTS

V	ஃ
c	ச
h	ஹ
j	ஜ
k	க
l	ல
L	ள
|r	ழ
m	ம
N	ந
|N	ண
n	ன
g	ங
|n	ஞ
p	ப
r	ர
R	ற
s	ஸ
S	ஷ
|s	ஶ
t	த
T	ட
v	வ
y	ய

VOWELS

In our transcription, vowels always follow the consonant, whatever the Tamil representation may be.

Regular Vowels

Initial short vowels: |a |e |i |o |u
Initial long vowels: |A |E |I |O |U
Medial/final short vowels: a e i o u
Medial/final long vowels: A E I O U

Diphthongs

Initial ai: |~ Initial au: |`
Medial/final ai: ~ Medial/final: `

Absence of any vowel (Virama)

} This must be placed after any unvoweled consonant.

Special Signs

Rupee sign: |R
Pound Sterling sign: |P
HTML Newline tag: [

HTML New paragraph tag:]

Tamil characters in Unicode start at 0B83 hex, 2947 decimal, but many slots are empty, as explained above. These characters appear in the following fully commented program.

```
#This program converts a text from Roman to Tamil HTML
global chartable
procedure main()
  initialize()
  process()
end

procedure initialize()
local n, m, charlist
  # create table. maverick chars return ??
  chartable := table("??")

  # The characters in charlist are explained in the preceding text. The empty
  # string ("") is used to fill one of the empty slots which will not be
  # referenced when the table is constructed.

  charlist := ["V","","|a","|A","|i","|I","|u","|U","","","","|e","|E","|~","",
              "|o","|O","|`","k","","","","g","c","","j","","|n","T","","","",
              "|N","t","","","","N","n","p","","","","m","y","r","R","l","L","|l",
              "v","|s","","s","h","","","","","A","i","I","u","U","","","","e",
              "E","~","","o","O","`","}"]

  #initialize list index counter
  m := 0
  #n holds constantly augmenting unicode number
  every n := 2947 to 3021 do
    chartable[charlist[m +:= 1]] := "&#" || n || ";" #put 75 chars in table but
  #length of table is now only 53 characters
  #Special items
  chartable["a"]  := ""  #zero length string--base form has short a
  chartable["|R"] := "&#3065;"  #rupee sign
  chartable["|P"] := "#163;"    #English pound sign
  chartable["["]  := "<br />"  #HTML newline tag
  chartable["]"]  := "</p><p>" #HTML end old, begin new paragraph tags
  #table is complete with 58 characters
  return
end #initialize()

procedure process()
local filename,title,infilvar,outfilvar,line,newline,char
  # filename and title are provided by user. infilvar and outfilvar are the input and output file variables.
  # line is taken sequentially from the input file, processed and written to newline which is then
  # appended to the new html file. Your input file can be created by any editor or word processor, but
  # be sure it is in text only format. It is preferable for the filename not to have a suffix. Omit the right
  # curly brackets if you wish the browser to take care of line breaks.

  writes("Enter the name of previously prepared file: ")
  filename := read()
  writes("What is the title of the text? ")
```

```
  title := read()

  # if the input file cannot be found, the program aborts.
  (infilvar := open(filename)) | stop(filename," not found.")

  # since all is well open a file for writing
  outfilvar := open(filename || ".html","w")

  # write to the output file a simple HTML header.
  write(outfilvar,"<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0Transitional//EN\">")
  write(outfilvar,"<html>")
  write(outfilvar,"<head>")
  write(outfilvar,"<title>",title,"</title>")
  write(outfilvar,"<body>")
  write(outfilvar,"<p>") # begin a paragraph
  # loop brings in lines from input file
  while line := read(infilvar) do {
    # initialize newline to the zero string
    newline := ""
    # scan the input line
    line ? {
      # loop processes the character, terminates at end of line when move() fails
      while char := move(1) do {
        if char << "A" then newline ||:= char  #punctuation,numerals.
        # << means: is lexically lower than "A" in the ASCII list.
        else if char == "|" then newline ||:= chartable["|" || move(1)]
        else newline ||:= chartable[char]
        }
      }
    write(outfilvar,newline)
    }

  # write a simple HTML footer to the output file. Program end closes files.
  write(outfilvar,"</p>") # close last paragraph
  write(outfilvar,"</body>")
  write(outfilvar,"</html>")
  write("File ",filename,".html has been created.")
  return
end #process()
```

Let us compile this program, and submit to it the following brief Tamil text in a file we call simply TV:

|eNt~ yAy}, |em} paran}, maR}Ru miyA var}k}kum}[
taN}t~, tAy}, tam}pi rAn}, Ranak} kVtilAn}[
muN}ti |en}nuL} pukuN}tanan}, yAvarum}[
ciN}t~ yAlum} |aRivaru|n} cel}vanE!

This stanza from the classical Tamil poem *Tiruvacakam* 5.47 is translated by G.U. Pope thus:

Father and Mother, Lord! To all besides.

Sire, Mother, Lord:—to him all these are not!

Erewhile within my inmost soul he entered,

Whom none can know, the Ever-blissful one!

We follow the same procedure as above, and the final result is as follows:

To learn more about Tamil Unicode, see http://www.unicode.org/faq/tamil.html

6.2.3 Hebrew

A basic program for Hebrew, omitting the vowel points, is very easy. Hebrew is caseless, i.e. it very sensibly does not bother with capitalization, which makes the lives of school kids much easier. (We should have dumped capitalization, like the poet e.e. cummings, when the computers came in, but we lost our opportunity.) So we only need 27 characters in total, and I will let you do this yourself.

Unicode expects input from left to right even for languages that are written and read from right to left. As soon as it spots a Hebrew (or Arabic etc.) character, it automatically *prepends* the characters rather than appending them. This achieves right-to-left output. You are liable to confuse the poor browser if you start or end a line with a punctuation character from the regular, left-to-right set, so be careful. There is a Unicode right-to-left mark (200F hex) but the browsers do not seem to know about it. Another thing. You must reverse parens () and similar items, because a beginning opening paren in English is a closing paren in Hebrew. Sorry. Oh yes. When you use <p> to start a new paragraph, replace it with <p align="right"> so that the output is ragged left.

For the Hebrew alphabet we can use the following, with the five final letters preceding their initial/medial sisters. I do not know why the finals get precedence in the computer world, but they do:

abgdhvzjTyXxlMmNnSoFfCcqrst

When two similar letters occur, I capitalize the less common one to save work. I have used *j*, as in Spanish, for the *het*, and *x* for *khaf*, as in Greek. Now I chose *o* for Hebrew *ayin*, because they were originally the same letter, despite appearances. When you realize that *ayin* means "eye" or "well" (the kind from which you draw water) you will see that Hebrew has strayed farther from the original pictogram than English, where the resemblance is still apparent. You can change these in any way you like, but be consistent. The Unicode decimal numbers for the 27 Hebrew letters go from 1488 (*alef*) through 1514 (*tav*). (By the way, if you subtract 1488 from 1514, you get 26 and not 27. Why is that? If you do not know, figure it out, because it will save you from some programming errors.) Since there are so few characters, we do not need to use the upright bar. Calling the table "chars", it is sufficient to write:

```
 while char := move(1) do
   if char << "A" then newline ||:= char
  else newline ||:= chars[char]
```

Now let's say you want to get fancier and mark the Hebrew letters *b x f* with a dot in the middle for their "hard" sound. There are two possible ways to do this. The Unicode character 1468 decimal, 05BC hex will put a dot in the middle of the letter right before it, so after we fill up our table we can have a statement:

```
chars[`] := "&#x05BC;"
```

If there is a plain *b* in our input text it will appear in the output as a *vet* (the soft pronunciation); if we write *b`* it will be *bet* with a dot in the middle, the hard pronunciation. Note that I chose the grave (`) arbitrarily. You can choose any letter you do not need.

Another way is to have a special letter for the hard pronunciation of these three letters, let us say *B K P*. Then we shall need to enter a separate item in our table:

```
chars["B"] := "&#1489;&#x05BC;"
```

Note that not every browser will insert the dot or *dagesh*. Some may simply ignore it.

Another thing you might consider is the following. The final letters XMNFC only occur at the end of words. When the program brings in a line from the input you could scan it for the letters which have final variants, check the next letter, and if it is not another letter, but a space or a period or something of the sort, or the end of the line, you could swap x for X and so on. (Hint: use the Icon function `upto('xmnfc')` with `every` to locate the position of each such letter, check what the next one is, and if necessary change it to the corresponding final letter, `X` etc.) This would be obeying the IBM *Polyanna Principle:* "Computers should work; people should think." On the other hand, you would lose a certain amount of flexibility: in Isaiah 9.7 (verse 6 in English bibles) there is an anomalous final *mem* in the middle of a word, (!) and you would lose the ability to represent this. There is often a tension between flexibility and ease of use; if you use a modern scanner to scan a document, or a camera to take a photo, and use the "auto" option it is so easy, but the machine makes decisions for you with which you may not be entirely happy. Or compare it to choosing the fixed meal in a fancy restaurant, or eating *à la carte*; your insistence on precise choice comes at a cost. 'Twas ever thus.

6.3 Further Study

Look first at:

http://www.unicode.org/charts/

There you will find all the myriads of Unicode. Click on Cyrillic, or Tamil, or Hebrew, or whatever language group you are interested in, and you will find the characters laid out before you. Great ingenuity has gone into all this. It is a tribute to what humans can achieve when they work together. Tamil is truly remarkable. You do not have the option of ignoring the vowels as you do in Hebrew; some vowels are attached to the consonant, some precede it, some follow it, some follow *and* precede it! All this Unicode takes in its stride by subtle modifications that are transparent to the user. I used the stratagem of the empty string("") in dealing with the short "a" vowel because that is really the base form, and it makes it possible to regularize the way we handle vowels. Sometimes "nothing" is quite useful. We have long got used to the idea of zero, but getting used to a string of zero length is still a bit difficult. The final result is the beautiful, curvaceous, Tamil script, used by 30 million people in India alone.

Alan Wood's Unicode Resources is also very helpful:

http://www.alanwood.net/unicode/

There are numerous links there, including tests you try on a particular browser to see what it does and does not recognize. Codes there are given both in decimal and hexadecimal.

6.4 Hex to Dec

Talking of hexadecimals, here is a little program which will convert hex numbers to decimals if you should need it. Note that when Icon scans the number from left to right it constantly decrements the number of the index. When it gets to the last digit it multiplies it by 16 to the zeroth power which is 1 for *all* numbers. This looks to be unnecessary, but it simplifies the programming (comparably to the zero string above!) and is worth thinking about apart from the modest achievement of the program.

```
#This program converts hex numbers to decimals
global A,a,B,b,C,c,D,d,E,e,F,f,impermissible
procedure main()
  initialize()
  process()
end #main()

procedure initialize()
#High hex numbers are normally caps, but lower case is ok.
#The value at the right is first assigned to the lower case letter, then
#that value is assigned in turn to the upper case letter.
  A := a := 10
  B := b := 11
  C := c := 12
  D := d := 13
  E := e := 14
  F := f := 15
#create a global character set excluding hex numbers
  impermissible := (&ascii -- &digits -- 'aAbBcCdDeEfF')
  return
end #initialize()

procedure process()
local hexno,length,result,n
  write("This program converts hex numbers to decimal.")
  writes("Enter hex number to be converted. Period (.) to finish: ")
#continue the program as long as input is not a period.
  while (hexno := read()) ~== "." do {
#check input for invalid characters, conclude program if found
    hexno ? if upto(impermissible) then stop(hexno," is an invalid entry.\nBye.")
#set *length* to the length of the hex humber, initialize *result* to 0 because you
# cannot increment the initial null value of an undefined variable.
    length := *hexno
    result := 0
    every n := 1 to length do
#pick up the digits in turn, left to right, increment the result. The augmented assignment
# "+:=" adds the value of the convert function to value of result, and assigns the new
#value to result.
      result +:= convert(hexno[n],length - n)
    write("Hex number ",hexno," = ",result," decimal.")
    writes("Enter hex number to be converted. Period (.) to finish: ")
    }
  write("Bye.")
end #process()
```

```
procedure convert(digit,index)
local outcome
# convert the digit to decimal, allowing for its place by power.
# variable(char) converts the alpha hex number to the variable of that name.
  outcome := (integer(digit) | variable(digit)) * (16 ^ index)
  return outcome
end #convert()
```

SUMMARY OF ICON FEATURES

1. table("??") creates a table comparable to the familiar printed tables. A table is a collection of pairs, each of which has a *key*, such as "b" corresponding to a *value*, such as its Unicode encoding. So chartable["A"] := "А" assigns the letter A on the keyboard as the Unicode first letter of the Russian alphabet. We chose "??" as the default value, which shows up if a key unknown to the table is entered. When the double question mark appears in the output, you know that you forgot something!

2. The function open() is used to work with files, which store information on some persistent media which may be available after the program execution is finished, for example in some later program run. The statement var := open(filename) opens an already existing file named filename, and stores that opened file in var, which may be used in subsequent read operations. Function open() takes an optional second argument "w" which allows the file to be written to. Data previously there will be destroyed. If the second argument to open() is "a", data that is written will be appended to data already there. Note that these arguments must be in double quotes, or must be a variable holding the string value "w" or "a".

3. *Augmentation* (increasing or decreasing an existing value) occurs frequently in programs, e.g. p := (p + 1). If the original value of p was 7 it will now be 8. Icon has an economical shorthand for this called *augmented assignment*: p +:= 1. This can be used for other operators. Thus, s1 := (s1 || s2) appends the string s2 to s1, and can be shortened to: s1 ||:= s2.

4. The symbol << means "lexically less than" and will succeed if the character on the left has an ASCII number less in the ASCII numbers. See Appendix B for the list in the Griswold book. Similarly, the symbol >> means "lexically greater than".

5. The function variable(s) converts the identifier (which is a string) called here "s" to a variable. We used this in the Hex to Dec program by converting "A", in the incoming string "0B8A" for example, to the *variable* A, to which we had assigned the decimal value 10, A being the equivalent of 10 in decimal, and similarly B,C,D,E,F.

7. Standard Deviation

7.1 Working with Sentences

Our main task in this chapter is to find the standard deviation of the sentence length of a text, sentence length being measured by the number of words the sentence contains. Let us begin by developing a procedure to produce sentences from a file analogous to the one which produced words from a file. In this way we can build up a library of procedures which may have various applications. Let us first define a sentence as a string of characters ending in a period, a question mark, or an exclamation point. This definition is probably inadequate, because it may not be allowing for quoted sentences, but it will do for now. This does show us incidentally that programming often forces us to consider issues which we tend to overlook, since the computer requires us to be totally explicit. Let us observe first that the sentence can be longer or shorter than the line. The line is of a length that happens to be convenient for human beings to handle on the printed page or screen, and hence text files are traditionally divided into lines, which are about the length of a punched card—i.e., eighty characters—although the computer can handle lines of any length. When we bring in a line for handling, we must allow for the fact that the sentence which we wish to isolate can potentially be shorter or longer than the line. We are going first to bring in lines from the file as long as there are lines to bring, so we can safely start off with the now familiar loop

```
while line := read(filvar)
```

which brings in lines from a file until there are no more to bring, whereupon it fails. We assume that we have already checked that filvar refers to the relevant file. A variable named markers will hold the set of characters that can finish the sentence and mark it off. The variable line will hold the current line of the file with which we are dealing. The variable sentence will hold the string of characters that will ultimately constitute a sentence by our definition, and will be returned as one of the values of the procedure. The variable substring will hold the string of characters from the beginning of the line up to a marker. This could be a complete sentence, or it could be the end section of a sentence that is several lines long. Now we set sentence to "", a valid string of zero length, and go looking for one of the sentence markers. So often as we find one, we have delineated a sentence, and can return it as one of the values of the procedure. Accordingly we try to find a marker, and if we do, add that section of the line, including the marker, to sentence, in case there is something there already from a previous line. This activity is analogous to incrementing a number. Recall that the statement

```
letters[n] +:= 1
```

adds 1 to letters[n] and stores the resulting number in letters[n]. Similarly

```
sentence ||:= substring
```

adds substring to sentence, or, as we say, concatenates it, and then stores the new augmented string in sentence. The double bar (||) is used for the concatenation of strings. Of course, initially we shall just be adding the section of the line to the zero-length string that sentence is at first. We then return this value, reset sentence to the zero-length string, skip over any blanks beginning the next sentence, and try again. If we fail and are at the end of the line, we can just go fetch another line (if any are left) and repeat the process. So we have so far:

```
sentence := ""
while line := read(filvar) do
  line ? while substring := tab(upto(markers)) do {
    sentence ||:= (substring || tab(many(markers)))
```

```
    suspend sentence
    tab(many(' '))
    sentence := ""
    }
```

But what if we are not at the end of a line? This would imply that there is something left in the line, or possibly the entire line which has not been taken care of. We can test for this by the function pos(). This function takes a single numeric argument and succeeds if its argument corresponds to the position of that imaginary pointer. 0 is the position right after the last character on the line, or perhaps more accurately, the initial position going from right to left. These positions start from 1 at the far left just before the first character and add one as you move right (1, 2, 3...), or from 0 at the far right just after the last character and subtract one as you move left (0, -1, -2...). It is possible then to indicate the position of the pointer positively from the left, beginning at 1, or negatively from the right, beginning at 0. Accordingly, pos(0) will succeed if the imaginary pointer is right at the end. If it is, we don't have to do anything. If it is not, we want to add what is left to the sentence, plus a space to allow for the fact that we don't normally put in a space at the end of the line:

```
  if not pos(0) then
    sentence ||:= (line[&pos:0] || " ")
```

&pos is a keyword which always contains the current position of the imaginary pointer, so

```
  line[&pos:0]
```

represents the portion of line from the position of the pointer up to the end. This is added to sentence, a space is added, and we are ready to start over. Observe that like do, the ? which marks the scanning facility expects one statement, so we must use the curly brackets if there are more than one. Here is the complete procedure:

```
procedure get_sentence(filename)
local filvar, sentence, line, substring, markers
  markers := '. !?'
  filvar := open(filename)
  sentence := ""
  while line := read(filvar) do line ? {
    while substring := tab(upto(markers)) do {
      sentence ||:= (substring || tab(many(markers)))
      suspend sentence
      tab(many(' '))
      sentence := ""
      }
    if not pos(0) then
      sentence ||:= (line[&pos:0] || " ")
    }
  close(filvar)
end
```

It is worthwhile working through this procedure one step at a time with various types of sentences to see how it works. Let us take, for example, a file which begins with a sentence that takes up the whole of the first line and concludes with a period in the middle of the second line. The value of sentence is an empty string. Variable line acquires the value of the first line of the file. The while loop looks for a marker and fails, so the loop is never entered, and the value of &pos remains at 1 as it was initially. Since it is *not* at the end of the line, the if-statement right after the loop applies, and the value of line[1:0]—which is the entire line—is added to the string of zero length, along with a blank, and stored in sentence. Notice the productive way in which a zero-length string can be used; it enables us to treat

the initial build-up of the string in exactly the same way as the subsequent ones. Initialization to zero in arithmetic computations gives analogous benefits. This represents a less easygoing approach than that of Icon's predecessor language SNOBOL-4, in which the initial value of a variable is the empty string, which can function too as zero. This casualness may often save time and effort, but also it can lead to subtle programming errors.

The next line is then fetched from the file. Again, an attempt is made at the entry of the while-loop to find a marker, and this time it succeeds, so the loop is entered. The characters up to the marker are stored in substring, then this plus the following marker are added to sentence. The sentence is now complete, and returned as the value of the procedure by suspend which puts everything on hold until the process can continue. When it does, any blanks following the sentence just processed are skipped, and &pos now has the value of the position right after those blanks. sentence is reset to the empty string so as to be ready for the next sentence. The loop is attempted again, but fails. The if-statement is checked and succeeds, because &pos does not indicate that the pointer is at the end. So the piece of the line from the beginning of the new sentence until the end of the line plus a blank are stored in sentence and we are ready to look for the rest of the line.

Let us consider a different possibility. Imagine our line consists of three short sentences, so the last character on the line is a period. The while-loop succeeds, and the first sentence is returned. The loop succeeds a second and third time, each time returning a sentence. On the fourth try the loop fails. Since the pointer is right at the end of the line the if-statement fails, and a new line is brought in for processing.

We now need the ability to count the words in a sentence, and to do so we modify and simplify two procedures we have used already:

```
procedure getword_from_sentence(sentence)
#This procedure produces one word at a time from the sentence

local chars, punct, word
  chars := (&letters ++ &digits ++ '\'-')
  punct:=' .,?";:!'
  sentence? {
    tab(many(' ')) #skip leading blanks
    while word := tab(many(chars)) do {
      tab(many(punct))
      suspend word
      }
    }
end
procedure countsentence(sentence) #Counts the number of words in a sentence
local total
  total := 0
  every getword_from_sentence(sentence) do
    total +:= 1
  return total
end
```

The procedure getword_from_sentence() is like getword(), but it does not have to concern itself with opening a file. The procedure countsentence() uses getword_from_sentence() to count the number of words in a sentence.

We are now ready for a procedure which will do the main work of the program. This will be simpler than our previous program in that it does not have to divide the text into parts, but it will also use a procedure to calculate the standard deviation as well as the mean.

7.2 Figuring the Standard Deviation

Let us consider the following procedure:

```
procedure getav(filename)
#Does the main work of the program
local counter, words, sentence
  words := list(20,0)
  counter := 0
  every sentence := get_sentence(filename) do {
    #Increment the appropriate element in the list of sentence lengths
    words[countsentence(sentence)] +:= 1
    counter +:= 1
    }
  printout (words , counter)
  return
end
```

Remember that the while-loop succeeds so long as the control statement at its head *succeeds*. The every-loop succeeds so long at the control statement at its head *produces a result*, and it prompts the statement to produce all the results in turn. Observe that a while-loop being controlled by a generator may go on perpetually, since only the first item that the generator may produce will be called up. This will be successful, but the next time around the same result will be produced, unless some effort is made within the loop to alter that situation. The variable sentence will hold the sentences in the file in succession. The number of words in the sentence is found by countsentence(sentence) and the corresponding element in the list words is increased by 1. We also have a counter which is increased in value each time the loop is entered and thus counts the total number of sentences in the file. When the job of getav() is done, it passes the information it has stored in the list words and the total number of sentences to printout(), which will figure the mean, call a procedure to figure the standard deviation, and print out the results on the screen. Let us look at these.

```
procedure printout(w,c)
#prints out the info. c is the total
#of sentences. w is the list.
local word_total, n
  write("Length     Number Such")
  word_total := 0
  every n := 20 to 1 by -1 do {
    #Arrange info in columns
    write(right(n, 2), right (w[n], 12))
    word_total +:= (n * w[n])
    }
  mean := real(word_total) / c
  writes(" ","Mean = ", mean)
  writes(right("Standard deviation = ", 42))
  write(standard_deviation(w,c,mean))
  return
end
procedure standard_deviation(w,c,mean)
```

```
local sum_of_squares, n, st_dev
  sum_of_squares := 0
  every n := 1 to 20 do
    sum_of_squares +:= ((n - mean) ^ 2) * w[n])
  st_dev := ((real(sum_of_squares) / c) ^ 0.5)
  return st_dev
end
```

printout() works in a similar way to the previous chapter, but it is only called once since it is processing the entire file. We already know the sentence total. We figure the word total by running through the list and multiplying the total words in a particular sentence by the number of times that that length of sentence occurs. We pass on this information to standard_deviation() which is ready to do the calculation. Our local variables are n which serves as an index; sum_of_squares (which speaks for itself) and st_dev which will be the final value that we want. Initially sum_of_squares is set to zero and n is set to 1. The mean supplied by printout() is subtracted from 1 and this is squared. In Icon the symbol caret or wedge (^) is used for exponentiation, and so n ^ 2 achieves this end. On some keyboards or screens the circumflex (ˆ) (most often) or the up-arrow (↑) may appear. All three represent the same character, ASCII 94. This is then multiplied by the number of times that that value is needed, which is stored in the first element of the list. It might be noted that if that number is zero the entire calculation is pointless because the ultimate result is zero. This could be taken care of by introducing a condition:

```
  if w[n] ~= 0 then
    sum_of_squares +:= (((n - mean) / 2) * w[n])
```

The Icon symbol ~= means "does not equal." (The symbol ~ is called a *tilde.)* This will inhibit the calculation if it is unnecessary. However, in this case the value of w[n] has to be fetched each time, causing additional computing if the value of w[n] is greater than zero, so it is probably not worthwhile to add this provision. Notice that there is still no need to insert a curly bracket after the do because the if-statement is a single statement, including its else-leg if it has one. Finally the standard deviation is figured by getting the mean and taking its square root (using a fractional exponent.)

Here is the complete program:

```
procedure main()
  getav(get_valid_filename())
end
procedure get_valid_filename()
local filename
#This procedure gives user three chances to select a file
  every 1 to 3 do {
    writes("What is the name of the input file?” )
    filename := read()
    if close(open(filename)) then return filename else
      write("Unable to open file.")
    }
  stop("Bye!")
end
procedure getword_from_sentence(sentence)
 #This procedure produces one word at a time from the sentence
local chars, punct, word
  chars := (&letters ++ &digits ++ '\'-')
  punct := ' .,?";:!'
  sentence ? {
```

```
    tab(many(' ')) #skip leading blanks
    while word := tab(many(chars)) do {
      tab(many(punct))
      suspend word
      }
    }
end
procedure countsentence(sentence)
#Counts the number of words in a sentence
local total
  total := 0
  every getword_from_sentence(sentence) do
    total +:= 1
  return total
end
procedure getav(filename)
#Does the main work of the program
local counter, words, sentence
  words := list(20,0)
  counter := 0
  every sentence := get_sentence(filename) do {
    #Increment the appropriate element in the list of sentence lengths
    words[countsentence(sentence)] +:= 1
    counter +:= 1
    }
  printout(words, counter)
  return
end
procedure printout(w, c) #prints out the info, c is the total # of sentences,
#w is the list,
local word_total, n, mean
  write("Length      Number Such")
  word_total := 0
  every n := 20 to 1 by -1 do {
#Arrange info in columns
    write(right(n,2),right (w[n],12))
    word_total +:= (n * w[n])
    }
  mean := real(word_total) / c
  writes(" ","Mean = ", mean)
  writes (right ("Standard deviation = ",   42))
  write(standard_deviation(w,c,mean))
  return
end
procedure get_sentence(filename)
local filvar, sentence, line, substring, markers
  markers := ' . !?'
  filvar := open(filename)
  sentence := ""
  while line := read(filvar) do
    #look for a marker
    line? {while substring := tab(upto(markers)) do {
      #if one is found add it to sentence plus the marker
```

```
        sentence ||:= (substring || tab(many(markers)))
        suspend sentence
        #skip blanks at beginning of next sentence
        tab(many(' '))
        sentence := ""
        }
      #if the line is not finished, append the rest to sentence
      if not pos(0) then
        sentence ||:= (line[&pos:0] II " ")
      }
   close(filvar)
end
procedure standard_deviation(w,c,mean)
   local sum_of_squares, n, st_dev
   sum_of _squares := 0
   every n := 1 to 20 do
      sum_of_squares +:= (((n - mean) ^ 2) * w[n])
   st_dev := ((real(sum_of_squares) / c) ^ 0.5)
   return st_dev
end
```

7.3 Word Frequency

Using the preceding procedures we have designed, it is not difficult to construct a program which will check the frequencies of particular words. Our main procedure will have a parameter word_list which means that when the program is run, the user can simply write the words right after summoning the program. So if the program is in a file called wordfreq. icn one might write

```
icont wordfreq.icn -x the by in
```

to translate and execute the program for those three words. Later we shall show how we can proceed if the user fails to enter one or more words. The main procedure is quite simple:

```
procedure main(word_list)
   process(get_valid_filename, word_list)
end
```

The first argument of process() is obtained just as before. The second, which is the user's word list must be passed on to process() which otherwise would know nothing about it. As an alternative, one could specify a global variable right at the very beginning of the program (wl let us say) and assign word_list to this variable. Then all procedures can use it. In this procedure we shall use a table. We set up our table by

```
   word_table := table(0)
```

If at this stage we check the value of, let us say, word_table["the"], it will return 0, but this does not mean that there is actually such a value in the table. The length of the table (found by word_table) is zero. Once some value is assigned to an element in the table the element comes into being, and the length of the table is increased by one. So

```
   word_table["the"] +:= 1
```

increases the *the-th* element by one. Tables make it very easy to tabulate numbers related to strings. Additionally both the values and the indexes of tables can very easily be sorted into order, alphabetical or otherwise. We shall therefore use getword() to bring in the words from the file. Each individual word will be checked against each word in the list furnished by the user. If a match is found, the

corresponding entry in the table will be incremented, and looking for further matches will cease. It should be pointed out that while this method of word checking works, it is cumbersome and tedious. A better method, more in the spirit of Icon, involving pattern matching will be discussed in the next chapter. When we are through we shall move to another procedure which will print out the results on the screen in alphabetical order. Here is the procedure:

```
procedure process(filename, word_list)
local filvar, counter, word, n, word_freq
#set up the frequency table
  word_freq := table(0)
#open the previously checked file
  filvar := open(filename)
#initialize the word counter
  counter := 0
#bring out the words in the file one at a time
  every word := getword(filvar) do {
#increment the word counter
    counter +:= 1
#check the word against the user's list
    every n := 1 to word_list do
      if word == word_list[n] then {
        #if found, increment the corresponding slot in the table
        word_freq[word] +:= 1
       #and break out of the inner loop--no point in further checks
       break
       }
    }
#close the file, pass the counter and the
#table to printout
  close(filvar)
  printout(counter, word_freq)
end
procedure printout(c, wf)
local wl, n
  write(); write(); write()
  write("Total number of words in file= ",c)
  writes ("Word")
  writes(right("0ccurrences",30))
  writes (right( "Percentage", 30)
#sort the table into a list
  wl := sort(wf)
#print list
  every n := 1 to wl do {
    writes(wl[n] [1])
    writes (right (wl[n][2],30)))
    writes(right((wl[n][2] * c) / 100.0,30))
    }
end
```

You will note that the function sort() takes a table and creates from it a list. This list itself contains a group of lists in which the first element is the entry in the table and the second is the value of the entry, the entries now being in order. So in the preceding program, wl[3][1] refers to the third sorted entry in the table (now a list), while wl[5][2] refers to the value of the fifth sorted entry. There is an alternate

way of handling this. If you add 3 as the second argument of sort() then a single long list will be created in which the entries and values of the tables will alternate, the entries being in sorted order. Try modifying the preceding program to use this feature. The second argument *2* will produce a list in which there is a group of lists sorted according to value rather than entry, and the second argument *4* will produce a single list in which entries and values alternate, but sorting is by value rather than entry. The function sort() may also be used to sort lists, in which case the order of items is simply rearranged. No second argument is available for sorting lists. Numbers will be sorted before strings. It should be noted that sorting is according to the order in the computer character set, so all uppercase letters will occur before all lowercase letters.

SUMMARY OF ICON FEATURES

1. In the line-scanning facility the position of the imaginary pointer at any time is stored in the keyword &pos. The line which is being scanned is stored in the keyword &subject.

2. Positions in the line may be numbered positively from the left (beginning at 1) or negatively from the right (beginning at 0).

3. The function pos() succeeds if the argument (positive or negative) corresponds to the position of the pointer in the line being scanned.

4. Exponentiation is expressed by the caret (∧). On some keyboards this character may appear as the up-arrow (↑) or circumflex (ˆ). Roots may be obtained by a fractional exponent. The following comparisons would succeed, in theory. In practice, the second one is dangerous, as a floating point number produced by computation often has the tiniest errors, such as 4.0000001 or 3.99999999.

```
(7 ^ 2) = 49
(16 ^ 0.5) = 4
```

5. The tilde (~) is used to negate an operator such as equals (~=). It is not to be confused with not which negates conditional expressions, making them succeed when they would fail and vice versa.

6. The function table() creates a structure in which values may be indexed by any data object. Its argument becomes the initial value of each element in the table.

7. The function sort() will accept a table as its argument and produce a sorted list as its result. With no second argument this will be a list of two-element lists, the first element being an entry from the table and the second its value, the entries being in order. If the second argument is 2, sorting is by value. If the second argument is *3* or *4*, a single list is created in which entries and values alternate, sorting being by entry (*3*) or value (*4*). Lists may also be sorted. Numbers will be sorted before strings, which are arranged in character set order.

8. Correlation

8.1 The Spearman Rank Correlation

In this section we shall develop a program to figure the Spearman Rank correlation coefficient. It is set up as though dealing with a correlation between the date of composition of plays and the average number of words per verse indicating a constant increase or decrease, but it could of course be used for many other purposes with slight modification. It is assumed that the values of words per verse will be read in order of date of composition. The values could be read in random order, provided that they were associated with some value that would rank them.

Here is a table of the information for ten plays of Corneille:

Date	*Mean Words per Verse*
1629	8.93
1632	9.02
1635	9.147
1640	9.26
1644	9.152
1650	9.2
1662	9.22
1666	9.32
1672	9.48
1674	9.53

Let us first decide to give the user the option of giving the values on the command line which calls the program or being solicited for them by the program. We can achieve this with a main procedure as follows:

```
#Calculates Spearman Rank Correlation Coefficient
procedure main(command_line)
#if the command line has data send it to process, otherwise let get_info() get it.
  if *command_line = 0 then
    process(get_info())
  else
    process (command_line)
end
```

Here the main procedure has a parameter command_line. If the user writes values after the instruction to the computer to call the program, these values will be stored in command_line in the form of a list. So we measure the length of the list. If it equals zero, then the list is empty, and we call a procedure get_info() which will solicit the missing information and return it as its value. This can then be supplied to the procedure process() which will do the main work of the program. Otherwise we simply supply what we got in the command line to process().

Now let us see how get_info() will function. We have to get the values into a list to replace command_line which the user left empty, so we will set up a variable dlist. Since we do not know how long the list will be, it will be best to set it up empty and allow it to expand to be as large as necessary. We achieve this by

```
  dlist := list(0)
```

which creates a list of zero length. This function may take a second argument which gives an initial value to each element of the list, but that would be pointless in this instance. Alternatively we may write

```
dlist := [ ]
```

which serves the same purpose. The square brackets enclose the list, and here they are enclosing an empty list. Icon has a function

```
put()
```

which puts the value of its second argument at the tail of the list that is its first argument. For example

```
put(dlist, 1.5)
```

would make 1.5 the first element of dlist. Executing subsequently

```
put(dlist, 2.0)
```

would make 2.0 the second element of dlist and so on. Notice that the elements are added at the *right* side of the list, that is, at its tail. We are actually creating an entity called a *queue,* because if subsequently we come to process the list from left to right, we shall meet first with the first element that was put in. We may think of it as a theater queue or line; the first person to come is the first to get a ticket. This is sometimes known by the acronym FIFO—first in, first out. We may mention in passing that there is a similar function push(), which adds an element to a list on the *left* side, so that if we process from left to right, the last element to go in is met with first. This is known as LIFO—last in, first out—and creates an entity called a *stack*, which may be compared to a stack of dishes; when you get one you use the one that was placed there last. So

```
put(dlist, read())
```

will add to the right of dlist whatever the user types in at the keyboard. We want to repeat this as many times as the user desires.

```
while put(dlist, read())
```

will keep on doing this so long as put() succeeds. But how can we get it to stop? There is no way that the computer can read the mind of the user, and know that a particular value is the last. (Compare this with the situation where a disk file is being read, and the computer finds a physical "end of file.") Icon has a function

```
numeric()
```

which converts its argument to a whole number (integer) or number with a decimal point (real number) *if possible*. If it is not possible it will fail. So

```
numeric(read())
```

will succeed if read() is bringing in numbers and fail otherwise. (We may note that when we type in 1.5 for instance we are really entering a string which at some point Icon will automatically convert to a real number. Here we are doing the conversion *explicitly* to achieve a particular effect.) All we need to do is to instruct the user to enter some non-numeric data (a period, for example). The function numeric() will then fail, and put() will be "infected" by this failure and fail too. So the loop will end. This will serve the additional purpose of excluding bad data; as soon as something other than a number is entered, no further data will be accepted. For safety's sake it will be a good idea to repeat back to the user the data that the program is working with so that any discrepancy can be noted. One final point. What if the user enters no data even when solicited, either just hitting return or entering a period immediately? It might be well to check if this has happened (by measuring the length of the list) and stop the program if it has. Otherwise we may be doing calculations with no data. In addition, a significant aspect of the "user-friendliness" of a program is its capacity to handle bad data appropriately, by making a new request for

good data, or bringing the program to an orderly halt, without cryptic messages from Icon or the operating system. It is worth noting that we have not really attempted to check what the user puts in if the command line is used. Perhaps we can assume that a user doing that will be extra careful; perhaps we should indeed devise some checks for that method too, but we shall not do that here. You may want to consider how that might be done. (The list could be scanned to see what is in it before processing it, for example.)

Our procedure now appears as follows:

```
procedure get_info()
local dlist
  dlist := [ ]
  write("Enter mean words per verse for each play in order of composition.")
  write("Hit return after each value. Single period + return to finish.")
  #add values to list at right end
  while put(dlist, numeric(read()))
  if *dlist = 0 then stop("Bye!")
  return dlist
end
```

We now proceed as follows with procedure process(drama_list) which will do the actual calculation of the coefficient. The parameter of the procedure will hold the list of values which the user has provided, and have been entered in chronological order by year. We can sort this into ascending order very easily by the Icon function sort() which takes a list as its argument and returns the sorted list as its value. So

```
  sort_list := sort(drama_list)
```

gives us a list of the same length as drama_list with the values in order. Now let us set up a table in which the indices are the *values* of sort_list and the values are the *position* of each value in sort_list. This is achieved by

```
  dtable := table(0)
  every n := 1 to *drama_list do
    dtable[sort_list[n]] := n
```

Be sure to understand this step; the value in sort_list becomes the index in dtable, and the current position, represented by n, becomes the value of that element of the table.

We now have to create a sum of the squares of the difference between each item's rank on the two scales. One scale is already ranked and is represented by a constantly incremented n. The other is represented by

```
  dtable[drama_list[n]]
```

The index to drama_list returns a value which, when passed to dtable returns the ranking in the sorted version of the list. Note the manner in which an indexed variable (drama_list[n]) can be the index of another variable (dtable[]). Having previously initialized a variable sum_of_squares to 0, we proceed to increment it by the new value squared:

```
  every n := 1 to *drama_list do
    sum_of_squares +:= ((n - dtable[drama_list[n]]) ^ 2)
```

Now we are ready to calculate the coefficient known as rho (ρ):

```
   rho := (1 - (6 * sum_of - squares) / (real(*drama_list) * ((*drama_list ^ 2) - 1)))
```

This can all be written on one long line even if it is wider than the screen or window in which it is displayed. Icon accepts statements even though they are too long for the screen.

If you do break it, do so at a point where the line is clearly unfinished, after an addition sign, for example. The value is then written to the screen. It is often desirable to use real numbers in division in order to ensure that remainders are not discarded by integer division. Converting one of the values to a real number by real() ensures this.

The complete procedure now looks as follows:

```
procedure process(drama_list)
local sort_list, sum_of_squares, n, dtable, rho
#initialize sum_of_squares and n
  n := sum_of_squares := 0
#sort the list
  sort_list := sort(drama_list)
#create a table of locations from sort_list
  dtable := table(0)
  write("Values are:")
  while write(drama_list[n +:= 1])
  every n := 1 to *drama_list do
    dtable[sort_list[n]] := n
#figure the differences and sum the squares
 every n := 1 to *drama_list do
    sum_of_squares +:= ((n - dtable[drama_list[n]]) - 2)
  rho := (1 - (6 * sum_of_squares) / (real(*drama_list) * ((*drama_list ^ 2) - 1))
  write("rho = ", rho)
  return
end
```

8.2 Scattergrams

The scattergram indicates visually whether there is a correlation between two sets of variables, and is often a useful precursor to the coefficient just mentioned in order to see if it is worth pursuing. Clearly we can have a program in which the same data is used to figure the Spearman coefficient and draw a scattergram, but we will do it rather differently here in order to illustrate some features of Icon that have not occurred before. You may want to consider developing an integrated program which does both. Let us set up a main procedure like this:

```
procedure main()
  get_values()
  figure()
  scatter()
end
```

The program is in three parts. One obtains the necessary data; one manipulates the data so that it can be used; and one puts the dots that constitute the scattergram on the screen. It is necessary to get the data in pairs, so we will allow the user to enter any amount of values, but the total must be even. It is not possible then to allow the user to enter a period at any time; it must be done *only* for one value of the pair. For this purpose a repeat loop may be useful. Unlike while which iterates so long as the control expression at the top succeeds, and every which iterates so long as the control expression at the top produces a result, repeat goes on forever. It has no control expression. It may be stopped by the expression break, and this is what we in fact do:

```
procedure get_values()
  values := [ ]
  repeat {
```

```
    writes ("X-value? Nonnumeric to finish. ")
    put(values,numeric(read())) | break
    writes ("Y-value? ")
    while not put(values,numeric(read())) do
      writes("You must enter a numeric value. Y-value?")
    }
  return
end
```

We first initialize the variable values to the empty list and then enter the repeat loop, using the curly bracket to direct Icon that all the following commands until the corresponding curly bracket are to be considered part of the loop. We first solicit the x-value from the user, and check that it is numeric. If it is, we put it in the list of values. If it is not, we break out of the loop, and the collection of data is done. The symbol | means *or* and implies that if the expression preceding it fails, then the one following should be tried—*either* put the value in the list *or* break out of the loop. This statement is equivalent to:

```
if not put(values,numeric(read())) then break
```

but it is more concise and easier to understand. Once the user has put in an x-value, then a y-value *must* be put in. So we set up a loop (inside the repeat loop) which is entered only if the user attempts to exit at that point by entering a non-numeric. This will recur until a proper value is entered, and the repeat loop starts over. This time we shall not return values as the value of the procedure; instead values will be declared a global variable which is accessible to all procedures in the program. As we mentioned already, variables of this type should be used sparingly. Since they can be altered anywhere in the program, they are prone to cause errors which are difficult to detect. Where possible, keep variables local, using them only in one procedure. The declaration of global variables must be done outside all procedures; the best place for them is right at the beginning of the program thus:

```
global values
```

These paired values must now be placed on a grid on the screen. How can we do this? The easiest, although not the only, way to do this is to move the cursor around the screen directly to the point concerned. Most terminals are capable of doing this, but the instructions may vary to achieve the necessary effects. The code example given here uses escape sequences that seem to work on MS Windows and Mac OS X terminal windows. It may require consultation with your terminal manual to find what the corresponding commands are for your terminal.

First we have to note that some kind of scaling is necessary. The text screen has 25 lines (the x-axis) and 80 columns (the y-axis). It is also possible to have a 25 x 40 screen. There are graphic screens on which points can be plotted with great accuracy, but these are not accessible to the current implementation of Icon. For this purpose we shall need to know the range of our data, and adapt it to the size of the grid which we have available. We will therefore declare the variables for the minimum values of x and y and the range of x and y as global variables. The maximum value will not be further needed and so may be declared a local variable in procedure figure(). The very first line in our program now looks like this:

```
global values, x_min, x_range, y_min, y_range
```

We are now going to run through the values supplied by the user and determine the minima and maxima. We first set both x_min and x_max to the first value, since at this point this value is both the maximum and the minimum. We do the same for the y value. Since we have already taken care of the first two values we can start looking at value three. We set a variable n to 3, and are going to increment it in steps of two, since the x and y values alternate. We now have a pair of or-statements:

```
(x_min := (x_min > values[n])) |  (x_max := (x_max < values[n]))
(y_min := (y_min >  values[n + 1])) |   (y_max := (y_max < values[n + 1]))}
```

Let us consider the x statement first. We are going to try to assign a value to x_min. If we succeed, then we shall skip the "or" part of the line. If we fail, we shall try to assign something to x_max. In other words, if we find a value of x lower than the current minimum, it will replace that minimum. If that does not work, we shall check if it can replace the maximum. Of course, both attempts may fail. The expression

```
x_min > values[n]
```

succeeds if the value of x_min so far is greater than the current element in values, and returns the second value—i.e., that of values[n]—and this then replaces the former minimum. We cannot use the more natural expression

```
values[n] < x_min
```

because then x_min would be returned, and no change would take place. Having less-than return the second value was a decision made by those who constructed the language. It allows expressions such as i < j < k to work as expected. When the loop finishes, we have ascertained the maximum and minimum values for the x- and y-axes. From these we can calculate the range, and then go on to scale these to fit the screen. Our procedure now reads as follows:

```
procedure figure()
local x_max, y_max, n
  x_min := x_max := values[1]
  y_min := y_max := values[2]
  every n := 3 to *values by 2 do {
    (x_min := (x_min > values[n])) | (x_max :=
        (x_max < values[n]))
    (y_min := (y_min > values[n + 1])) | (y_max :=
         (y_max < values[n + 1]))}
    x_range := x_max - x_min
     y_range := y_max - y_min
end
```

The long lines are broken in a place where they are clearly incomplete. This prevents semi-colons from being inserted at those newlines. The procedure for scaling is quite simple. We pass to it the particular value we wish to scale:

```
procedure scale(n,min,range,max)
  return integer((n - min) / real(range) * max)
end
```

and it returns the scaled value.

We can regard our screen as a grid numbered from 1 to 25 vertically and 1 to 80 horizontally. Executing a writes() statement

```
  writes("\e[1;1H")
```

moves the cursor to the top left corner of the screen, but writes nothing there.

```
  writes ("\e[25;80H")
```

moves the cursor to the bottom right corner. It is permissible to write this statement in pieces, provided that we do not separate the \e which represents the escape character. If the value of q is 25 and of trh 80 then an equivalent statement to the last is

```
  writes("\e[",q,";", trh,"H")
```

This feature enables us to *compute* the coordinates. The cursor is then sent to the computed coordinates, and a dot is printed there. Here is the complete program, including the procedure to place the dots appropriately on the screen:

```
global values, x_min, x_range, y_min, y_range
procedure main()
  get_values()
  figure()
  scatter()
end
procedure get_values()
  values := [ ]
  repeat {
    writes("X-value? Nonnumeric to finish.  " )
    put(values,numeric(read())) | break
    writes ("Y-value? ")
    while not put(values,numeric(read())) do
      writes("You must enter a numeric value. Y-value?  " )
    }
  return
end
procedure figure()
local x_max, y_max, n
  x_min := x_max := values[1]
  y_min := y_max := values[2]
  every n := 3 to *values by 2 do {
    (x_min := (x_min > values[n])) | (x_max := (x_max < values[n]))
    (y_min := (y_min > values[n + 1])) | (y_max := (y_max < values[n + 1]))
    }
  x_range := x_max - x_min
  y_range := y_max -  y_min
end
procedure scale(n,min,range,max)
  return integer((n - min) / real(range) * max )
end
procedure scatter()
local n
  writes("\e[2J")
  every n := 1 to *values by 2 do {
    writes("\e[", 26 - ((scale(values[n + 1], y_min, y_range, 24) + 1)),
           ";", scale(values[n], x_min, x_range, 79) + 1, "H")
    writes(".")
    }
end
```

SUMMARY OF ICON FEATURES

1. The main procedure may have a single argument. Then, if the program is run with following arguments these will be stored in the form of a list in the variable constituting the argument of the main procedure.

2. list(i,x) creates a list of i values with the initial value of x.

3. put(L,x) adds the value x to the right of the list L.

4. numeric(n) returns a numeric value for n if it can, and fails otherwise.

5. repeat initiates a "perpetual" loop concluded by break. The loop can also terminate if the whole procedure returns or fails or if the whole program is ended.

9. Pearson's Coefficient of Correlation

9.1 Planning the Program

In this chapter we shall develop a program to calculate Pearson's coefficient for word frequency in two texts. Kenny comments as follows (p. 83):

> Calculation of the Pearson product-moment coefficient by means of its definition formula can be a slow and laborious business, since it involves calculating the mean and standard deviation for each of the two distributions involved, and the conversion of all the values along each scale into z-scores. This will have become apparent to the reader on working through even the artificially simple exercises in the text…

Kenny goes on to point out that hand calculators often include a routine for calculating the coefficient. This program offers the possibility of calculating the coefficient easily, once the texts or a sample of them are available in machine-readable form—and preparing these is much less tedious and error prone than working through texts by hand and making the calculations.

Additionally, optical scanners are able to convert printed matter, or even written matter, into machine-readable form and are readily available.

Let us now consider what information we shall need, and how we shall proceed. From the user we shall need three pieces of information: a set of words, the occurrence of which is to be checked in the texts, and the names of two files which contain the texts. Our main procedure will call a procedure to get the set of words, and use twice the procedure to get a valid filename which we have used before. The results of these three procedures will be passed to a procedure which will systematically use this information to calculate the coefficient. Our main procedure then will look like this:

```
procedure main()
  process(get_words(), get_valid_filename(), get_valid_filename())
end
```

The second and third arguments of process() give the user three chances to designate a valid filename, and end the program if this effort does not succeed. The first one will get a list of words, and we can also use it to explain the program. The three pieces of information are passed to procedure process() which will do the main work of the program. If we wish to make the main procedure less succinct but more readable we can use three local variables thus:

```
procedure main()
local word_list,filename_1,filename_2
  word_list := get_words()
  filename_1 := get_valid_filename()
  filename_2 := get_valid_filename()
  process(word_list, filename_1, filename_2)
end
```

Both of these approaches are valid, and which one chooses is a matter of style. The procedure process() will proceed as follows:

1. Create a table for file *A* in which the index will be the words concerned, and the values will be the number of times the word occurs. This table will be created by another procedure which will get the words from the file (using a procedure we have used before) and compare them with the word list.

2. Create a similar table for file *B*. These tables will not necessarily be the same length, since some words may occur in only one file.

3. For ease of processing, convert both tables to lists of values in the same order. From this point on we do not need to know what the word is; we have to have paired values for the two files. The two lists will be in corresponding order.

4. Figure the mean of the values of file *A*, using a procedure for that purpose, then do the same for file *B*.

5. On the basis of step 4, figure the standard deviation for each text, using a procedure similar to one already used, but quite general this time.

6. On the basis of step 5, figure the z-score for each pair of values in the lists derived from the tables, multiply them together, and keep a running total.

7. The coefficient is figured by dividing the running total by the length of the word list, and is printed on the screen.

9.2 Getting Information From the User

Here is the procedure which will give the user some information, and also request some.

```
procedure get_words()
local word,word_list
  word_list := [ ]
  write(); write(); write()
  write(center("PEARSON PRODUCT-MOMENT COEFFICIENT",64))
  write(center("—————————————————— ",64))
  write()
  write("This program will figure the coefficient for a set of words")
  write("you will supply as they occur in two texts. Enter them one")
  write("'at a time, following each by pressing Return. When you are")
  write("through enter a single period and press Return. You will be")
  write("asked twice for an input file. Enter valid file names.")
  write("Please enter the words.")
  while (word := read()) ~== "." do
    put(word_list,word)
  return word_list
end
```

This procedure is quite simple. It has only two local variables: word which will hold the words as they come in from the user, and word_list into which the individual words will be placed and will ultimately be returned as the value of the procedure. We start by initializing word_list to an empty list, and then write some instructions to the screen. These could be formatted in any way you find esthetically pleasing; the function center() is helpful in doing this, and a little experimentation will indicate what works best. It is a good idea to start with clearing the screen, but how this is done depends on your particular terminal. It will normally involve writing some code to the screen. This will usually be a control character or some characters beginning with the escape character which is used to indicate that what follows are not the usual printing characters. The following will work on the Apple terminal, and usually work elsewhere:

```
  writes("\e[2J")
```

This is the *escape* character, represented in Icon by \e followed by a code arbitrarily set by the system. If this does not work, look in the manual for your terminal for the appropriate code for *clear screen* or *home and clear*, or ask someone who might know! Another approach would be to use

```
system("clear")
```

which instructs the system to use its command "clear." This functions slightly differently in that it also moves the cursor to the top of the screen. The user then types in a word at a time, following each by return. The end of the list is signaled by entering a single period or some other appropriate character. Our loop then is

```
  while (word := read()) ~== "." do
    put (word_list,word)
```

that is, so long as the word read is *not* a single period enter the loop which adds the word to the word list. Note that the item read is assigned to word, and it is the value of that assignment operation which is checked against a single period before the loop is entered. The possible length of the list is limited only by the computer's memory; it is unlikely that this will present a problem with a list of reasonable size. As soon as the loop is concluded, the word list is returned as the value of the procedure, and this is in turn passed by the main procedure to a procedure which will process the information. Two final points before we leave get_words(). First,

```
  word_list := list(0)
```

would be an acceptable alternative for initializing word_list. Second, it might be a good idea before returning the word list to check that it actually contains something; the user might have entered a period immediately. We could do this by checking if the length of the list is zero, in which case we stop the program:

```
  if *word_list = 0 then stop("Bye!")
```

9.3 Figuring the Coefficient

The procedure which we shall call process() will have the three arguments mentioned before, and a number of local variables. If the local variables are too many to place conveniently on one line we can repeat the word local.

For each of the two files we shall need variables to contain a table, a list, a mean, and a standard variation. In addition we shall need a variable to contain the running z-score product and finally one for the coefficient we are seeking. Here is the procedure:

```
procedure process(wl,fnm_a,fnm_b)
local table_a,table_b,list_a,list_b,n,a_mean,b_mean
local a_sd,b_sd,z_sc_product,coefficient
  z_sc_product := 0
  table_a := make_table(fnm_a,wl)
  table_b := make_table(fnm_b,wl)
  list_a := list(*wl,0)
  list_b := list(*wl,0)
  every n := 1 to *wl do {
    list_a[n] := table_a[wl[n]]
    list_b[n] := table_b[wl[n]]
    }
  a_mean := mean(list_a)
  b_mean := mean(list_b)
```

```
  a_sd := st_dv(a_mean,list_a)
  b_sd := st_dv(b_mean,list_b)
  every n := 1 to *wl do
    z_sc_product +:= z_sc(list_a[n] ,a_mean, a_sd) * z_sc(list_b[n],b_mean,b_sd)
  coefficient := z_sc_product / *wl
  write()
  write("Coefficient is ",coefficient)
end
```

We initialize the running z-score product to 0. Then we create a table for each file, using a procedure which we shall discuss soon. We then create a list for each file, which has as many elements as there are words in the word list, and the initial value of which is 0 in each case. We then make each word in the word list in turn the index of the table, and transfer the value to the new list. Notice that if a value is missing in the table because the word did not occur in the file, it will still return the value 0, and this will be entered in the list as an actual value. We then figure the means and the standard deviations; from this the sum of the products of the pairs of z-scores and then the coefficient. We now have to consider the procedures that contribute to this procedure.

9.4 Creating the Table—Pattern Matching

In order to create a table, we must get the words in turn from the file (which we already know how to do) and then compare them with the words in the word list, incrementing the appropriate entry in the table if a match is found. On a previous occasion we did a similar operation by using a loop which compares each word in turn with the members of the word list and jumps out if a match is made. Icon has a much better way of doing this kind of operation. In order to understand it we must examine the nature of the string scanning facility in Icon. When this facility is invoked by following the string, or string valued variable, by the question mark, two keywords are activated. &subject contains the original string that is being processed. &pos contains the position of the imaginary pointer which moves along the line as we process it, and is initially set to 1; that is, immediately left of the first character in the string. Both &subject and &pos are global; that is to say, both can be accessed by any other procedure. Accordingly it is possible to set up a procedure outside the one in which the string scanning facility is in operation which will utilize these keywords and give back information to the string scanning operation. Here is our procedure to make a table:

```
procedure make_table(flnm,L)
local t
  t := table(0)
  every word := getword(flnm) do
    word ? if is_in_list(L) then
      t[word] +:= 1
  return t
end
```

This procedure creates a table of which the initial values will all be 0. We then get one word at a time from the file, call up the scanning facility for that word, and determine if the word is in the list. If it is, the corresponding index has its value incremented by one. The procedure is _in_list() will succeed only if the word which is the subject of scanning is in the list provided to is_in_list() as its argument.

9.5 Matching Against a List of Words

For our purposes we can use a procedure referred to by Griswold (p. 219) slightly modified.

```
procedure is_in_list(L)
```

```
  suspend tab(match(!L) & pos(0)
end
```

The exclamation point prefixed to a variable produces the elements of that variable in turn. For example, if L is a file, it will produce each line in the file, and we could use an every loop to print out those lines or use them in some other way. In this case, L is a list. Now suspend which returns a result and leaves the procedure intact rather than departing from it entirely as return does, acts like every inasmuch as it prompts the production of every possible result. So what happens is as follows. Let us assume that the words in the word list are *he she we you* and the word against which we are matching them is *weed*. The expression !L first produces *he* and match() attempts to find this string starting at the current position of the pointer which is 1. match() fails and so tab() fails too. (Since this combination of tab() and match() is so common, Icon possess a shorthand notation for it which consists of placing the equal sign immediately before the string or the string valued variable.) !L now produces *she* which meets with the same fate. When *we* is produced match() succeeds and returns 3, which is the position in *weed* after the string *we*. tab() now moves the pointer up to position 3. So far we have a positive result. This is conjoined by & with the result of pos(0). However, this fails, since the pointer is not at the end, and the whole procedure fails. Since this scanning operation is complete however, the original position of the pointer at the beginning is restored, and another try is made with *you* which fails in the same way as the first two. If the word against which we are matching is *we* rather than *weed*, the third attempt will succeed and so the procedure will succeed since it is producing a result. This is not strictly a "matching procedure" as Icon defines it, since the matching procedure should produce the string between the original and the ultimate position of the pointer. However, in this case our interest is purely in a yes-no answer to the match, as we already have the string in which we are interested.

Here now is our complete program. Comments have been added. The procedures for computing in turn the mean, the standard deviation, and the z-scores should be self-explanatory.

```
# This program solicits a set of words and two filenames containing texts. It calculates
# Pearson's coefficient of correlation for the frequency of the words in the two texts.
procedure main()
  process(get_words(),get_valid_filename(),
    get_valid_filename())
end
#For each text make a table, the index of which is a word, and the value of
#which is the number of occurrences. Convert this info to a list of values
#for each text. Figure the mean, standard deviation, and thence the z-scores for the
#raw scores, and figure coefficient based on pairs of scores.

procedure process(wl,fnm_a,fnm_b)
local table_a,table_b,list_a,list_b,n,a_mean,b_mean
local a_sd,b_sd,z_sc_product,coefficient
  z_sc_product := 0
  table_a := make_table(fnm_a, wl)
  table_b := make_table(fnm_b, wl)
  list_a := list(*wl, 0)
  list_b := list(*wl, 0)
  every n := 1 to *wl do {
    list_a[n] := table_a[wl[n]]
    list_b[n] := table_b[wl[n]]
    }
  a_mean := mean(list_a)
  b_mean := mean(list_b)
```

```
  a_sd := st_dv(a_mean,list_a)
  b_sd := st_dv(b_mean,list_b)
  every n := 1 to *wl do
    z_sc_product +:= z_sc(list_a[n],a_mean,a_sd) * z_sc(list_b[n],b_mean,b_sd)
  coefficient := z_sc_product / *wl
  write()
  write("Coefficient is ",coefficient)
end
#Gives instructions & solicits word list from user
procedure get_words()
local word,word_list
  word_list := [ ]
  write(); write(); write()
  write(center("PEARSON PRODUCT-MOMENT COEFFICIENT",64))
  write(center("________________",64))
  write()
  write("This program will figure the coefficient for a set of words")
  write("you will supply as they occur in two texts, Enter them one")
  write("at a time, following each by pressing Return, When you are")
  write("through enter a single period and press Return. You will be")
  write("asked twice for an input file. Enter valid file names.")
  write("Please enter the words. ")
  while (word := read()) ~== "." do
    put (word_list,word)
  return word_list
end
#Gives user three chances to enter a valid filename
procedure get_valid_filename()
local filename
  every 1 to 3 do {
    writes("What is the name of the input file? ”)
    filename := read()
    if close(open(filename)) then {
      write("OK")
      return filename
      }
    else
      write("Unable to open file,")
    }
  stop("Bye! ")
end
#Figures the mean of a list of values
procedure mean(L)
local total,n
  total := 0
  every n := 1 to *L do
    total +:= L[n]
  return real(total) / *L
end
#Figures the standard deviation of a list of values based on mean
procedure st_dv(m, L)
local sum_of_squares, n
  sum_of_squares := 0
```

```
  every n := 1 to *L do
    sum_of_squares +:= ((L[n] - m) ^ 2)
   return (real(sum_of_squares) / *L)  ^ 0.5
end
#Figures the z-score based on raw score, mean, and standard deviation
procedure z_sc(raw_score,m,sd)
  return (raw_score - m) / real(sd)
end
#Succeeds if the subject matches any of a list of words
procedure is_in_list(L)
  suspend !L & pos(0)
end

#Creates a table of occurrences of list of words from a file
procedure make_table(flnm,L)
local t
  t := table(0)
  every word := getword(flnm) do
    word? if is_in_list(L) then
      t[word] +:= 1
  return t
end
#This procedure produces one word at a time from the file. Definition of word is somewhat primitive.
procedure getword(filename)
local chars, punct, filvar, line, word
  chars := (&letters ++ &digits ++ '\'-')
  punct := ' .,?";:!'
  filvar  := open(filename)
  while line := read(filvar) do line ? {
    tab(many(' ')) #skip leading blanks
    while word := tab(many(chars)) do {
      tab(many(punct))
      suspend word
      }
    }
  close(filvar)
end
```

9.6 More on Matching

Let us now consider a further possibility of using Icon's ability to match strings. You will recall that in order to figure the Pearson coefficient we need three pieces of information from the user:

1. The name of the file containing Text A.
2. The name of the file containing Text B.
3. A set of words, the occurrence of which is to be checked in the texts.

Assume now that instead of getting this information interactively as we have done using prompts, we shall allow the user to enter this information on the command line, making the assumption that the first two valid filenames on the line are the names of the files concerned, and all the other entries are the words that are going to be checked. Icon allows the main procedure to have an argument,

“command_list” for example, which makes available a list of strings to the program, consisting of items, separated by blanks, which follow the name of the program when it is called by the user. Note that this time we are not checking that the file *exists;* we may indeed wish to check that also, but here we are checking that the actual name of the file follows appropriate rules as to how it is spelled out. We could incorporate this feature into the main procedure by checking to see if the list coming in from the command line contains anything; if it does we use it, if not, we solicit the information interactively as before. For our purpose, a valid filename will be a number of characters deemed permissible for a filename, followed by a period followed by the ending *txt*, which is called the *extension*, and usually consists of three or four letters which hint at what the file contains. For example, a file containing a document may have the extension *doc*, while one containing an Icon program must end in *icn.* Rules for filenames are system dependent, and procedure isfile() below can be modified according to need.

For the moment we shall write a separate program which will simply take a list of strings from its command line, identify the two files, and print them out. We shall do this by taking the command line (which is a list of strings) and turning it over to a procedure which will return and print out a list of two strings that meet our needs if it finds them. If we wanted to use this technique in the preceding program, we could put the filenames into one list, the words for matching in another, place both lists in another list, and return it as the value of the procedure. This information can then be passed to procedure process().

Here is our main procedure:

```
procedure main(command_list)
local filelist
  filelist := getfiles(command_list)
  write("First file is ",filelist[1])
  write("Second file is ",filelist[2])
end
```

The procedure getfiles() returns a list of up to two files, which is stored in filelist. The write() commands will only be followed if the appropriate element in filelist exists, since the attempt to retrieve the contents of an element ([1] or [2] in this case) will fail if that element does not exist, and that failure will be "inherited" by write().

In getfiles() we shall proceed something like this:

- Prepare an empty list which will hold the valid filenames.
- Bring in a word from the command line list.
- If a procedure gives the OK, add it to the new list.
- Quit when two valid filenames have been found, or the old list ends.

Let us now convert this to Icon language. The first preceding item becomes:

```
 L_out := [ ]
```

We now need a *loop* to bring in each string from the command line in turn. Let us assume that in this procedure the command line has been stored in list L.

```
  every n := 1 to *L do
```

Now we invoke the string scan:

```
  L[n] ? if isfile() then put(L_out, L[n])
```

The procedure isfile() will succeed if L[n] is indeed a valid filename, and then it will be added to the new list. Otherwise we go on to try a new string. There is one additional thing. Once we have found two files, we should like to stop, and we can achieve that by setting up a flag. After the first file is found, the flag goes up; when the second is found the flag is checked, found to be up, and the procedure stops. Now if we establish a variable flag, its initial value is null. This we take to be the flag in its down condition. We can check this by prefixing a forward slash (/) to the variable. This will produce the variable flag if that variable is null; otherwise it will fail. So the command

```
/flag := "up"
```

will *succeed* the first time it is met with in the program, since the value of flag is null; however the value "up" will be assigned to flag which is produced by the expression /flag. Accordingly, the next time around

```
/flag := "up"
```

will *fail* because now flag *does* have a value, and is not null. You will notice that /flag produces a *variable* not a value, and it is this fact that allows a new value to be assigned to flag. Of course, we don't have to use the string "up". The arbitrary number 1 would do equally well. There is a similar prefix, the back slash (\), which succeeds if the variable it precedes does have a value. Finally we may note that an expression like if flag then... is useless, since it will always succeed, producing null or some value, and being successful in either case. Using this feature our procedure becomes:

```
procedure getfiles(L)
local L_out,n,flag
  L_out := [ ]
  every n := 1 to *L do
    L[n] ? if isfile() then {
      put (L_out, L[n])
      (/flag := 1) | break
      }
  return L_out
end
```

Since the flag statement fails the second time around, the alternative following the bar is taken account of, and the loop stops at the behest of break. The flag statement is put into parentheses to make sure it is properly grouped. Now we reach the crux of our discussion: the procedure isfile().

We first declare a static variable permissible. By declaring permissible as static it will retain its value however many times the procedure is called. There is then a command preceded by the word initial which means that it will be done only the first time the procedure is called. This avoids unnecessary computation of a value that will never change. This stores in permissible a set of all the valid characters for filenames. The function many() will then span all such characters and return the position immediately after the last of them. many() must find at least one character to succeed. We then need to find a period followed by *txt* at which point the string must end. This is expressed by

```
tab(match(".txt")
```

where match() looks for the string that is its argument immediately after the imaginary pointer, and tab() then moves the pointer beyond it. The equals sign may be used also thus:

```
=".txt"
```

The end of the string is symbolized by conjoining the pos(0) function, which ensures that the pointer is now at the end. So we have:

```
procedure isfile()
```

```
  static permissible
  initial permissible := &letters ++ &digits ++ '$&#%\'()-@^{}~` !_'
  return tab(many(permissible)) II =".txt" & pos(0)
end
```

Should we wish txt to be permissible in any of the eight possible combinations of lower- and uppercase letters we might substitute

```
  ="." II tab(any('Tt')) II tab(any('Xx')) II tab(any('Tt')) & pos(0)
```

The function any() will look for a single character in its argument immediately after the current position of the pointer, and return the position after it.

You may wish to try modifying the program for figuring the coefficient to allow an option of providing the information in the command line. You will need an if-statement in the main procedure such that an empty command line triggers the interactive solicitation of information, and a command line with information is processed such that the filenames and word list are correctly passed to process().

9.7 Sets

Icon has another structure which can often be fruitfully used in matching words. We have previously learned something about character sets or "csets" which are created in Icon by placing a group of characters between single quotes and are useful with such functions as upto() which looks for a point in the string where any member of the cset occurs. Icon can handle sets of items other than single characters. Set theory is a branch of mathematics that was elaborated by Georg Cantor, and a brief example will help explicate the kinds of situations in which it can be helpful. Let us imagine that committee A has members Jones, Brown, Rodriguez, and Sperber while committee B has members Robinson, Schwartz, Rodriguez, and Sperber. If the two committees meet together, the joint committee will have a membership of six and not eight. If we take away from the joint committee those members who only belong to one committee, a set of two persons will be left. Icon is able to mirror operations of this type. In order to create a set we start with a list

```
  [" Jones", "Brown", "Rodriguez", "Sperber"]
```

and convert it to a *set* by the function set():

```
  committee_A := set(["Jones", "Brown", "Rodriguez", "Sperber"])
```

Note that an identical individual item that recurs in the list will occur only once in the set, and unlike the list, the members of the set are in no particular order. We can check if an item is a member of a set by the function member(). So in this instance

```
  member(committee_A,"Jones")
```

would succeed and return the value "Jones". If the second argument had been "Schwartz" the function would have failed. This can readily be used to check the occurrences of a word list in a text. We set up a table to hold the occurrences of particular words:

```
  wordcount := table(0)
```

and a set of the words we are checking

```
  wordlist := set(["you","we","they"])
```

then as each word comes in and is stored in the variable word the statement

```
  wordcount[member(wordlist,word)] +:= 1
```

will create a table in which the entries are the members of the set (or as many of them as occur in the text) and the values are the number of occurrences. No if-statement is required. If the word is not in the set, the function member() will fail and no assignment will be made, since the entire statement will fail. If it is in the set, it is returned as the value of member() and becomes the index of the table. The value of that entry is then increased by one. The length of the table will be as many entries as have been found at least once, but if a non-occurring word is checked in the table it will return 0, since that is the initial value of table entries. This is a good example of how the useful Icon structures table and set can be used to produce concise code in which a single line does a great deal of work.

The equality of two Icon sets can be checked by a triple equal sign (===). We have already met the single sign being used for the equality of numbers and the double sign being used for the equality of strings. Further information on comparing values can be found on pages 128-130 of Griswold and Griswold's "The Icon Programming Language", 3rd edition. A member can be inserted in a set as follows:

```
insert(committee_A, "Jackson")
```

or we could put Jackson in his own set and unify the two.

```
committee_A ++:= set(["Jackson"])
```

Set union and difference is shown by the double plus sign or minus sign, and set intersection (which gives the set of members that belong to both) by the double asterisk. In the just-cited example committee_A and the new set are unified, and the result is assigned to committee_A which now has the new value of the combined sets. The function delete(committee_A,"Jackson") would remove Jackson from the set. Both insert() and delete() always succeed and cannot be used to check to see if a member is there or not; for this member() must be used.

SUMMARY OF ICON FEATURES

1. An empty list may be created by list(0) or a pair of empty square brackets ([ ]).

2. The word local declaring the names of variables may occur more than once.

3. &subject and &pos are global keywords (accessible to all procedures). The first contains the string being scanned. The second contains the current position of the pointer used in string scanning.

4. The exclamation point (!) prefixed to a variable produces the elements of that variable.

5. The equals sign (=) prefixed to a string or string-valued variable is equivalent to tab(match()).

6. The ampersand (&) conjoins statements. Both must succeed for the whole to succeed. The bar (|) alternates statements. If one succeeds, subsequent ones are not evaluated, and the whole succeeds.

7. The forward slash (/) prefixed to a variable succeeds and produces the variable if its value is null (the initial value of variables.) The back slash (\) prefixed to a variable succeeds and produces the variable if its value is something other than null. Otherwise it fails. A variable without either always succeeds because it always produces something.

8. If the Icon main procedure has a variable as its argument, that variable will contain a list valued at the strings written after the command that calls the program. So if the first line of the main procedure is procedure main(c) and the line calling the program is coefficient he she it, then c will be a three-string list consisting of "he", "she", and "it".

9. An attempt to access a list element that does not exist will cause failure (which will be “inherited” by any procedure calls in which such attempt occurs).

10. A variable declared static will retain its value from one call of the procedure to the next providing "memory" for the procedure. A statement (or compound statement inside curly brackets) preceded by the word initial will be executed on the first call of the procedure only. Statements like this must occur at the beginning of the procedure.

11. The double bar (||) concatenates (joins together) strings.

10. Programming a Nursery Rhyme

10.1 Ten Green Bottles

In this chapter we shall try to generate some rhymes, both as an exercise in becoming sensitive to structures in language and to learn some new Icon features.

The structure of the rhyme "Ten Green Bottles" is fairly obvious. It is a loop in which the number constantly decreases until it reaches zero—a rather fancy description of a child's rhyme. We need to set a variable to 10, decrease its value by one each time a bottle falls, and come out of the loop when 0 is reached:

```
procedure rhyme()
  local n
  n := 10
  while n > 0 do {
    write(n," green bottles hanging on the wall.")
    write(n," green bottles hanging on the wall.")
    write("Now if 1 green bottle should accidentally fall,")
    write("There'd be ",n -:= 1," green bottles hang_
      ing on the wall.")
    }
end
```

The symbols -:= written together decrease the value of the variable by what follows just as +:= increase it. A quoted string may be broken up for convenience between lines by using an underscore to denote that a continuation is to be expected. Now we just need to call this procedure and it will print out the entire rhyme. There are, however, some inelegant features occasioned by the conventions of the English language which we may wish to remove. First, we are supposed to spell out small numbers and not use the numeral.

There is no automatic connection in Icon between "1" and "one". The first is a one-character string representing an integer, a counting number, and the second is a three-character string which is the name in English of that integer. The first item may change according to the number system ("I" in the Roman and "" in the Hebrew system, for example), while the second may change according to language ("uno" in Spanish or "één" in Dutch), but they always point to the same underlying integer. We might make this connection by setting up a list with ten elements and then fill it with the appropriate words:

```
  numbers := list(10)
  numbers[1] := "One"
  numbers[2] := "Two"
```

and so on.

If this procedure were going to be called more than once it would be appropriate to declare numbers as static rather than local and initialize the elements in numbers in a compound initial statement (i.e., initial followed by the assignments inside curly brackets) since their value will never change. It is unthinkable that the string "Two" could ever be assigned to numbers[1] for instance. By declaring the variable as static and designating the statements as initial, the assignments are made once and retained from one call of the procedure to another. In this case it makes little difference, since the procedure is only called once. You might like to try writing a procedure that will return the name in English, or some other

language, when furnished with an integer between 1 and 100. How would you avoid spelling out the numbers between the tens greater than twenty? (Hint: Remember the modulo – %.)

We still have a problem in that we have not allowed for zero. Actually, if we run the program like this the last line will not be written because the attempt to access numbers[0] will fail, and so the write() will fail too. Then we can add the missing line after the loop. We begin by initializing the index to 10, and then enter the loop:

```
  n:= 10
  while n > 0 do {
    write(numbers[n]," green bottles hanging on the wall.")
    write(numbers[n] ," green bottles hanging on the wall.")
    write("Now if one green bottle should accidentally fall,")
    write("There'd be ",numbers[n  -:= 1]," green bottles hanging on the wall.")
    }
  write("There'd be no green bottles hanging on the wall.")
```

Notice how the indentation shows that the last command is outside the loop. By moving the line back we show that the flow of the program has moved up one level, and is not inside the loop. An alternative approach would be to use a table instead of a list. Then we could have a 0 index also. The conventions of English are giving us some new problems. We wrote in the numbers with capital letters, but the third line requires a small letter.

So we can set up a procedure to take care of this:

```
procedure lcase(s)
return map(s,&ucase,&lcase)
end
```

The function map(s1,s2,s3) produces a string in which each character of s1 that appears in s2 is replaced by the corresponding character in s3. The keywords represent csets, but these are automatically converted to strings by Icon.

This is a general procedure that takes a string and turns any uppercase letter into the corresponding lowercase letter, leaving everything else unchanged. So now we have

```
  write("There'd be ",lcase(numbers[n -:= 1])," green bottles hanging on the wall.")
```

Bearing in mind that write() writes to the screen all its arguments in order, notice what the second argument of write() does. It decreases the value of n by one, then uses the value of the assignment statement as the index of a list which produces a string which is then converted to all lowercase letters.

There is one other inelegant feature. When we get to "one", "bottle" does not need an s. (Curiously, with "no" it does!). We can, of course, avoid the issue by writing "bottle(s)" in the first instance, but somehow this seems in order for an official form but inappropriate for a nursery rhyme. Spelling out for a computer all the feelings and intuitions we have with respect to our own language is a truly enormous task. We can cope with this by setting up a procedure for plurals that will return s or a null string according to circumstances:

```
procedure plural(n)
  if n = 1 then return "" else return "s"
end
```

Notice too the way the null string is used. It regularizes the situation such that each word ends in something—even if that something is invisible! Here now is the complete program:

```
procedure main()
  rhyme()
end
procedure rhyme()
local n,numbers
  n := 10
  numbers := list(10)
  numbers[1] := "One"
  numbers[2] := "Two"
  numbers[3] := "Three"
  numbers[4] := "Four"
  numbers[5] := "Five"
  numbers[6] := "Six"
  numbers[7] := "Seven”
  numbers[8] := "Eight"
  numbers[9] := "Nine"
  numbers[10] := "Ten"
  while n > 0 do {
    write(numbers[n]," green bottle",plural(n)," hanging on the wall.")
    write(numbers[n]," green bottle" ,plural(n), " hanging on the wall.")
    write("Now if one green bottle should accidentally fall,")
    write("There'd be ",lcase(numbers[n -:= 1])," green bottle",
             plural(n)," hanging on the wall.")}
  write("There'd be no green bottles hanging on the wall.")
end
procedure plural(n)
  if n = 1 then return "" else return  "s"
end
procedure lcase(s)
  return map(s,&ucase,&lcase)
end
```

Two observations may be made on this program. First, it is generally not worthwhile to program small items. Once one gets beyond a very crude product, considerable work goes into programming, especially where natural language is concerned. Programming is most efficient when large amounts of data are to be processed, or the program can be used over and over on small amounts of data. For this reason it is hardly worthwhile to write a program to balance a checkbook, while in contrast word processors, which get a great deal of use, have been very successful. Second, this little program illustrates the *explicitness* that programming requires. Many human beings can be trusted to perform a task intelligently, making intuitive allowances for special circumstances. The computer must be told precisely what to do, without ambiguity. Dealing with natural language is particularly troublesome in this regard, since it has many ambiguous and unexpected features which are hard to anticipate. It is this fact which has made progress in machine translation from one natural language to another much slower than was initially expected.

10.2 The House that Jack Built

If we examine the structure of the nursery rhyme called “This is the House that Jack Built” we shall see that it consists of two loops, one within the other. The outer loop constantly brings in two new

elements, a noun or noun phrase and a verb or verb phrase, while the inner loop trots out all the previous nouns and verbs, so that the verses get progressively longer. To get started we need two nouns and a verb which are then set in a particular framework, then subsequently we need only one noun and one verb which are set in a slightly different framework, but this pattern repeats indefinitely.

Accordingly it will be well to have a procedure to get us going and generate the first phrase, and then pass this to a procedure which will generate the rest of the rhyme:

```
procedure get_started()
local str
  writes("This is the ")
  str := read()
  writes(“That ")
  str ||:= “ that “ || read()
  return str
end
```

This procedure has one local variable which will hold the string which will ultimately be the last few words of each verse. We prompt the user to enter “house” (or an equivalent) by printing out the words: “This is the ” and staying on the same line. Note the blank at the end of this phrase; this makes it unnecessary for the user to insert one. The function writes() does this for us since it does not issue a carriage return. This is stored in str. We then print “That” at the beginning of the next line to solicit “Jack built.” or the equivalent. This is added to the previous string along with the word “ that ” (surrounded by spaces). The symbols ||:= increase the string by what follows just as +:= do with integers. At this point we might want to consider simply continuing to make the string longer each time in the same way. This is possible, since we can even include carriage returns in the string. For this we use \n (for *newline).* Thus the string

```
"this is a \ntest"
```

would print out

```
this is a
test
```

(There are some other so-called “escape” characters of this type. \l is exactly the same as \n. This stands for *linefeed,* which is the ASCII name for this character. The previous symbol is the newer UNIX terminology for the same character. \r brings the cursor back to the beginning of the line and in effect overwrites what is there already. This is a simple carriage return without the linefeed which normally accompanies it. \b is a backspace.) However, it is not a good idea to create very long strings, even though it can be done. Long strings are clumsy to manipulate, and the string would have *the* and *that* repeated many times, using up storage space unnecessarily.

Let us observe first of all that in each repetition of the rhyme the words inserted later are mentioned earlier, “last in, first out.” This arrangement is typical of the *stack* (like the restaurant plates) and is represented by a list which uses push() to push new elements in at the *front* (the left side) of the list. There is also a function pop() for *removing* an element from the left side of the list, but since Jack is cumulative we do not need to use it. It will be best to set up a stack for nouns and a stack for verbs, adding the new "plates" to the stack as they come in.

```
procedure rhyme(conc)
local noun,verb,nouns,verbs,n
  nouns := [ ]
  verbs := [ ]
  repeat {
```

```
      n := 0
      write()
      writes("This is the ")
      noun := read()
      writes("That ")
      verb := read()
      writes("The ")
      while write(nouns[n +:= 1]) do
         writes("That ",verbs[n]," the ")
      push (nouns ,noun)
      push(verbs,verb)
      write(conc)
      }
end
```

Since the rhyme can theoretically go on forever, the outer loop is well represented by repeat which goes on indefinitely. Two stacks (empty lists) are created. At the beginning of the loop, an index variable is initialized to zero. The noun and verb are then solicited from the reader and stored in variables. The inner loop then looks for the first noun, and since the index is being constantly increased by 1, the loop will finish when it exhausts the stack, by trying to access an element that will not exist until the next time around. The first time the loop will not even be entered, since it will try to access an element (nouns[1]) which does not yet exist. When this loop finishes, the string which was created by the previous procedure is printed out. We add the new items to their respective stacks, and the repeat loop is ready to start again. The entire program now looks like this:

```
procedure main()
   rhyme(get_started())
end
procedure get_started()
local str
   writes("This is the ")
   str := read()
   writes(“That ")
   str ||:= “ that “ II read()
   return str
end
procedure rhyme(conc)
local noun,verb,nouns,verbs,n
   nouns := [ ]
   verbs := [ ]
   repeat {
      n := 0
      write()
      writes("This is the ")
      noun := read()
      writes("That “)
      verb := read()
      writes("The ")
      while write(nouns[n +:= 1]) do
         writes(“That ",verbs[n]," the ")
      push(nouns, noun)
      push(verbs, verb)
      write(conc)
```

```
    }
end
```

It would be possible to use only one stack, keeping both verbs and nouns on it in pairs. You might like to modify the program to use this approach.

10.3 Randomizing Jack

We can have a little fun with this rhyme by shuffling the stack with the result that we may find the malt biting the house and other improbable and amusing combinations. We can achieve this with a procedure mentioned in Griswold (p. 97 in the first edition, and in a different form in the third edition, p. 243.) They use it initially for strings, but it works equally well for lists.

In order to understand the procedure, we need to consider two useful Icon features that we have not met before. The question mark (?) prefixed to a variable produces a random element of that variable, if, like a list, it contains various elements. Do not confuse this with the question mark that invokes the scanning facility. That is a *binary operator* which is surrounded by spaces and connects the two expressions on either side. The one we are speaking of is a *unary operator* which affects only the value it precedes. If the value is a whole number greater than 0, it will produce a random number between one and that whole number. If the value is 0, it will produce a real number somewhere between zero and one. The other Icon feature in this procedure is as follows. You will recall that the expression

```
  x := y
```

assigns the value of y to x. Icon has an exchange operator (:=:). So

```
  x :=: y
```

assigns the value of y to x, and the value of x to y. The two variables exchange values. Now let us look at the procedure.

```
procedure shuffle(L)
local i
  i := *L
  while i >= 2 do {
    L[?i] :=: L[i]
    i -:= 1
    }
  return L
end
```

The local variable is set to the size of the list. If there are eight elements, the value of i will be 8. We then enter a loop in which an element selected at random will be exchanged for the last element in the list, number 8. The value of the local variable is then decreased by 1, so the exchange is with the last but one element, and this will be a string selected from the first seven elements. When the number of the variable drops below two the loop finishes, because no exchange is possible when only one element is involved. So

```
  shuffle(nouns)
```

will randomly rearrange the various elements in nouns. But how shall we activate this shuffling? An easy way would be to invite the user to add some character (say a plus sign) to any answer. We then check each answer to see if a plus sign is there. If it is, we shuffle the stack and delete the plus sign so that we are left with a regular word. Since this will be done in a separate procedure, we need to make the two stacks global, so that the new procedure can operate on them. While it is a good idea in general to avoid global variables, in this case it would be clumsy to pass the information between procedures.

While we are about it, we may as well give the user some way of concluding the program (since Jack is potentially infinite) let us say by entering a period in the answer, and arrange to use the same procedure for shuffling two different stacks. This can be achieved by an extra argument that indicates whether nouns or verbs are to be shuffled. The following will achieve what we wish:

```
procedure check(s,n)
local p
  if upto('.', s) then stop()
  if p := upto('+', s) then {
    s[p] := ""
    if n = 1 then shuffle(nouns)
    else shuffle(verbs)
    }
  return s
end
```

The procedure check() first looks for a period. If it finds one, the program ends. A string can appear as an argument of stop(), a commercial for a local builder let us say, which is printed on the screen before the program ends. Here upto() is being used outside of the string scanning facility, and so the string it is operating on must be specified as the second argument. Other string scanning functions may be used similarly. You may like to try modifying the program so that the string scanning facility is used in this case too. If no period is found, a check is made for a plus sign. If no plus sign is found, the procedure simply returns the original string unchanged. If a plus sign is found, it is first eliminated from the string. Strings can be indexed, so if the value of str is "steak" then the value of str[2] is "t". New values can be assigned to such indexed variables, and the size of the string changes automatically. So

```
str[2] := "tr"
```

would change "steak" to "streak" and

```
str[1] := ""
```

would change it to "teak". This last method is used to eliminate the plus sign which has now served its purpose. The procedure then checks the second argument. If it finds a "1" it scrambles the nouns; otherwise it scrambles the verbs. The original word, stripped of its plus sign, is now returned as the value of the function. To incorporate this in the program we just replace both occurrences of read() in procedure rhyme() by check(read(),1) and check(read(),2).

As an exercise you may like to try adding the possibility of unscrambling the stacks. To do this we need to set up two more stacks (we might call them nouns_bak and verbs_bak) which are filled at the same time as the regular stacks but are not affected by shuffle(). Then, when the user inserts some other symbol which check() can recognize, the scrambled stack is replaced entirely by the backup stack:

```
nouns := copy(nouns_bak)
```

The function copy() when used with lists, tables, and records creates an exact but *distinct* copy. If this were omitted, both nouns and nouns_bak would reference the same list, and the program would not work properly. This is why in 10.3 we did not need

```
nouns := shuffle(nouns)
```

Normally, when you pass an item to a procedure a copy of it is made, but this does not apply to lists, tables, and records (known collectively as "structures") where the actual item is accessed.

SUMMARY OF ICON FEATURES

1. The combination of symbols -:= decreases the variable on the left by the value on the right.

2. The combination of symbols ||:= concatenates the string variable on the left to the string or string variable on the right and assigns the result to the variable on the left.

3. It is permissible to break a string between lines if the first part of the string is followed immediately by an underscore.

4. The following "escape" characters may be used in strings:

   \n or \l newline

   \b backspace

   \r return to beginning of line

   Also, you may precede \, ' and " by the back slash (\\ \' \") if you want them to represent themselves literally and not the special functions they normally fulfill as marker or delineator.

5. Strings may be indexed to access individual characters in the string. Such indexed characters may have other values assigned to them, and the length of the string is automatically adjusted.

6. The function copy() makes a distinct copy of tables, lists, and records. In other cases it simply produces the value, and so the result is not different from the result of the original item.

7. The function map(s1,s2,s3) produces a string in which each character of s1 that appears in s2 is replaced by the corresponding character in s3.

11. Creating a Data Base

11.1 A Grade Program

In this chapter we shall consider the possibility of establishing a data base, which is one of the most useful features that the computer has for many users along with word processing programs and spread sheets. Despite horror stories of unfortunate sufferers who are dunned to pay up zero dollars, computers have made record keeping much simpler, at least when they are properly programmed and used. While many excellent commercial data base programs are available for general and specialized applications, it will be helpful to write a simple program of this type in order to see what it can do. Possibilities exist for research as well as practical purposes. Care must be employed in this kind of endeavor since security becomes important, both in terms of not losing crucial data and avoiding snooping or tampering by third parties. The latter is a complex issue which will not be dealt with here. Some systems include a program which will encrypt a file which cannot then be used until it is decrypted by means of a password. If you need this kind of security, you may want to check what is available on your system, or you may be able to obtain a program to keep your files secret. On time-sharing systems it is normally possible to make files private so that others cannot read them. The UNIX system has a program *chmod* which can make a file available only to its owner, and you can check the documentation to learn how to use this program. It should be pointed out that Icon keeps its records in ASCII which is useful insofar as the files can be read by an editor or an operating system command to “list” the file (i.e., print it out on the screen—the term comes from the days when program source files with commands that could be listed were the only files written in something like natural language). But it also means that unauthorized access is easier. Some programming languages create binary data files that can only be read by a program designed to do so. Asking for a social security number in these days of identity theft would be clearly unwise; a student number should be substituted. Building a simple data base will also give us the opportunity to use the Icon data type called *record*. In Icon the list is a very flexible structure which may expand and contract in length, be added to or subtracted from at either end, and hold varying types of data. The result of this is that the record is not quite as useful in Icon as it is in some languages where the data structures corresponding to the list (arrays, for example) are more restricted in their use. The record resembles a list in that its elements can be accessed by a numerical index. It resembles a table in that its elements can be accessed by name. It differs from both in that the names of its elements and hence its length are predetermined. The nature of the record has to be declared at the beginning of the program, and may be used by all procedures in the program, which are not allowed to have their own individual record declarations.

Note that the declaration of a record at the beginning of a program does not *create* a record. It specifies rather the attributes of one kind of record values when they are created in the course of the program. It is useful when we have an item that has various fixed attributes that are interesting to us, for example, age, sex, address, telephone number, and so on.

It would be possible to write a program simply to figure student grades, but that is rather trivial and better achieved with a pocket calculator. More significant is a program which will preserve information, can use that information to generate new information, can be updated from time to time, and can issue some kind of report. The techniques covered here can be used for many other data base applications.

Our program will have three sections:

- Initialization. This makes it possible to enter and preserve information that may reasonably not be expected to change: name, address, date of birth, and so on. Of course, someone can marry and change names or move, but we may disregard this for the moment.
- Updating. This makes it possible to enter information that was not initially available, such as the results of tests or assignments.
- Reporting. This makes it possible to retrieve the information that has been entered, perhaps after some calculations or other manipulations that make the data more useful.

The features which might be added to these are virtually infinite, as are the enhancements that might be added to these sections. We might want to bring in the use of a printer, perhaps for different types of reports. We will certainly wish to emend as well as update, dropping or adding students, for example. We may wish to make the files generated by the program available to another program, for example, one which would enable an individual student to access a restricted set of information of particular relevance. It is also desirable to add protections against the entry of inappropriate information, and help in case the user should need it. These last two items make the program "user-friendly." Above all, the program should be able to accept enhancements and modifications easily, so that the aforementioned three sections could be expanded to five, or twenty-five, simply by adding more procedures.

Our main program will call a procedure to get initial information from the user, and on that basis call the appropriate procedure:

```
procedure main()
  local result
  result := menu()
  case result of {
    "1" : initialize()
    “2” : update()
    “3” : report()
    default : write("Goodbye")
  }
end
procedure menu()
  write(center("Data Base Program", 80))
  write()
  write("Do you wish to")
  write("\t1. Initialize")
  write("\t2. Update")
  write("\t3. Report")
  write()
  writes("Enter 1,2 or 3 and press return.”)
  return read()
end
```

The main procedure calls a procedure menu() and stores the value of that procedure in result. Let us look at menu(). It prints out the name of the program and offers a "menu" of three options. The menu is a common way to offer options to the user. You may have a numbered list of the options, and invite the user to enter the appropriate number. An alternative method is to invite the user to press an arrow key and then the cursor moves between the various alternatives on the screen. Nowadays the mouse is used to select choices, but this option is not currently available in Icon. In order to implement the second possibility, you need to know what character is sent to the computer by the arrow keys (different keyboards may have different conventions), and then when the program detects this character it moves

the cursor appropriately, accepting the information from the user when finally the return key is pressed. Since the first method is easier to implement and less dependent on local conditions, we shall stick to that one. The \t is an Icon character that represents a horizontal tab, giving the effect here of a small indentation. The procedure menu() returns whatever the user types in, without running any checks on the validity of the answer. Now the main procedure takes this value, and submits it to a case statement. A case statement allows for a multi-way condition. The value that is being checked (here, result) appears between the words case and of. This is followed by a curly bracket and a set of lines divided by colons, which are checked through in order. If the value of response corresponds to the left-hand part of the first line, then the rest of the line after the colon is executed, and the program continues after the closing curly bracket of the case statement. If it does not correspond, then the next is tried and so on until the curly bracket is reached. It is possible to use the word default, which is executed if all else fails. The statement containing default is always tried last, wherever it occurs in the case statement, so you could put it first, if you wish, to emphasize the default. Note that "1", "2", and "3" come in as strings (as does everything from the keyboard) and must be checked as strings. Icon frequently makes automatic conversions of data. But in this case there is simply a comparison, and so the items must belong to the same data type to be considered identical. The main procedure simply shunts the flow to the appropriate procedure, or shuts down with a polite farewell if it is unable to do so. You may like to try modifying the program to allow more chances to the user. It is a little more difficult here than before, but it should be possible to find a way to do it. It would be possible to use getch() instead of read(), in which case “and press return” could be omitted, but in this case read() is preferable, because it gives the user the chance to change her mind and backspace if she happens to enter an incorrect answer.

Even at this early stage it is possible to check to see if the program is working. Add some dummy procedures as indicated next, and then give it a try:

```
procedure initialize()
  write("Space reserved for initialization.")
end
procedure update()
  write("Space reserved for updates.")
end
procedure report()
  write("Space reserved for reports.")
end
```

After we see that the basic flow is correct we can replace the dummies with procedures that actually do some work. Let us look first at initialize(). At the beginning of the program we must place a record declaration which gives the name of the record type (we shall call it students) and has as its arguments the names of the various elements which it contains. Here is the record declaration, placed first in the program:

```
record students(name,soc_sec,pwd,fstexam, secexam,final,total,grade)
```

and here is the procedure for initialization:

```
procedure initialize()
local filename,student,student_table,student_list
  student_table := table()
  writes("Enter the course number, e.g. 101 ”)
  filename := read()
  writes("Enter the last two digits of the current year ")
  filename ||:= read()
  writes("Enter a single digit for the semester ")
```

```
  filename ||:= read() II "1.dat"
  writes("The name of the file is ", filename)
  write(". Please note this for future use.")
  write("Enter information about students. Enter period(.) _
        only when asked for name to finish.")
  repeat {
    student := students("","","" ,0,0,0,0, "")
    write()
    writes("Name? ")
    student.name := read()
    if student.name == "." then break # jump out of the loop
    writes("Social security number? ")
    student.soc_sec := read()
    writes ("Password? ")
    student.pwd := read()
    student_table[student.name] := student
    }
  student_list := sort(student_table)
  write_to_file(filename,student_list)
end
```

We need a file in which to keep our information. We could simply ask the user to designate a filename, but here we construct a filename from the course number, year, and semester (which amounts to six digits) and then add a 1 (which will be increased for each update file) and the extension *.dat.* We then ask the user to note this name as the identification of this particular series of information. In this manner a series of files will be created automatically. The way we shall do it will work properly for nine files only. You may wish to consider how to modify the program to make it work for ninety-nine files, which should be possible without too much difficulty. We then enter a loop to structure the incoming information appropriately. We initialize a variable student as a record of the type of students. The initial values are zero for numerical items and the empty string for string items. We then fill in such information as we can. We do not even ask for the score on the first examination for example, and that stays at the 0 to which it was initialized. A password of the student's choosing is entered in case we wish to make the scores available on a limited basis to individuals possessing the password to their particular entry. Observe that the particular element of the record is accessed by placing a dot after the record variable and then the name of the element. This is much more understandable in this case than using a list, which is referenced only by numbers. We then place the information in a table indexed by the name. This preserves the information and enables us to reinitialize the record variable and use it for the next student. Additionally, it will be easy to alphabetize the list later using the Icon sort facility. The loop then starts over. Anytime a single period is detected in the first entry, the loop terminates. This kind of loop is used because it is necessary to reinitialize the variable each time, and this is not a suitable item to control a while loop. Actually a while loop could be used, and is in fact a more “structured” solution. Initialize the variable *before* entering the loop, and use the first reading of information to control the loop:

```
  while (student.name := read()) ~== "." do {
```

This reads in the student's name and then uses the *value* of the assignment statement to check that it is not a period, in which case the loop will finish. Then we have to reinitialize the record variable at the *bottom* of the loop, right after placing the information in the table. This means that the initialization of the record variable is written twice: once outside the loop and once inside. The tilde (~) makes the string-equals sign negative, that is, it means “does not equal…”

The table is then sorted. This converts the table into a list which itself contains a set of lists equal in number to the items in the table. The first item in each of these lists is the index of the table. The second item is the value of the table. These lists will appear in the order of the ASCII character set according to the index, now the first item in each list. (You can also sort by the value if you wish, by adding 2 as the second argument in the sort() function.) We pass this information and the filename to write_to_file() which will write out the information to the file and can be used by the other procedures for the same purpose. Here it is:

```
procedure write_to_file(f,s)
  local filvar,n,p,line
  filvar := open(f,"w")
  n := 0
  while (n +:= 1) <= *s do {
    line := ""
    every p := 1 to 8 do
      line ||:= s[n][2][p] || "%"
    write(filvar,line)
    }
  close(filvar)
end
```

This procedure first opens the file to which it will write the information. The second argument "w" specifies writing to the file. If this is omitted the file can be read only. Two loops are used to get the information in the student list into a format suitable for the ASCII file. The outer loop keeps going as many times as there are students which is indicated by the length of the student file. The inner loop runs eight times specifically since there are eight pieces of information on each student. The inner loop will first look at the first item of the second element (which is a record) of the first list, and start off a line with it. It will add a percent sign as a divider, then look at the second item of the second element of the first list and add this to the line. After eight times of doing this, it will write the line to the file, reinitialize the line to the empty string and then look at the first item of the second element of the second list, and so on until all students and their attributes have been recorded. We now have the information filed in a consistent manner.

Let us talk a little about the structures that are involved here. The student list is a list that itself consists of lists, as many as there are students in the class. Each of these lists has a fixed number of elements, namely 2. The first element is the name, and the second element is a record, which contains information relevant to the student. The format has been dictated by the manner in which the table sort in Icon works, and hence there is a duplication of the name. But it is convenient to keep this format, because anytime we wish we can easily use the sort facility again, for example, if we would later add new names. This is a nested structure—wheels within wheels. Icon cannot write such a nested structure to a file. We have to write it in a sequential fashion, using some arbitrary symbol as a marker between the different fields. When we bring back in the information from the file, we can restore it to its nested form that is convenient to work with when manipulating the information. Note that when we want to process all the information in the record, it is often convenient to index the record as though it is a list of fixed length, rather than referring to the elements by name.

Let us now consider a procedure to update the information. It will do this in a rather gross fashion, offering the user all the current information and giving the opportunity to enter the same information or to change it. You may wish to consider making this more specific, perhaps by offering a menu. Another possibility is to allow the original score to stand by allowing an empty string (which the user enters simply by pressing return) to represent the same score as before.

We first have to get the name of the current file, and we do this by asking the user for the original name, then constructing a name with the version number that that name contains constantly being increased until failure indicates that such a file does not yet exist:

```
procedure get_version()
local filename,n
  n := 0
  write("Enter the name of the ORIGINAL filename ")
  filename := read()
  while close(open(filename[1:7] || (n +:= 1) || ".dat"))
  return filename[1:7] || (n - 1) || ".dat"
end
```

Note that this assumes that none of the old files have been deleted, and allows for only eight updates. We now take this file, put the information back into the original structure, solicit replacement entries from the user, and then write out the new information to the next file in the series. A typical filename might be 2050713.dat where the 205 would be the course number, the 07 would be the year, the 1 would be the semester and the 3 would be the sequential number of the version, increased from the original 1. filename[1:7] produces the section of filename between positions 1 and 7, i.e., the first six characters of the string. Such ranges of parts of a string-valued variable may also be negative, starting from 0 for the position after the last character in the string and then counting back -1, -2, and so on. So filename[-1:0] would produce the last character in the string. The Icon line scanning facility is used to detect the percent sign in the file used as a marker, and the field between the old and new positions of the pointer is stored in each element of the record. We skip over the percent sign by ="%". The command move(1) would have the same effect in this case. Here is the procedure:

```
procedure update()
local filename,filvar,n,student_list,line,student, inner_list, current
  filename := get_version()
  filvar := open(filename)
  student_list := [ ]
  while line := read(filvar) do {
    student := students("", "", "", 0, 0, 0, 0, "")
    inner_list := list(2)
    n := 0
    line ? while student[n +:= 1] := tab(upto('%')) do
      ="%"
    inner_list[1] := student.name
    inner_list[2] := student
    put(student_list,inner_list)
    }
  close(filvar)
  n := 0
  while current := student_list[n +:= 1][2] do {
    write("Name: ",current.name)
    write("Current first exam = ",current.fstexam)
    writes("New score? ")
    current.fstexam := read()
    write("Current second exam = ",current.secexam)
    writes("New score? ")
    current.secexam := read()
    write("Current final exam = ",current.final)
    writes("New score? ")
```

```
      current.final := read()
      student_list[n][2] := current
      }
   filename[7] +:= 1
   write_to_file(filename,student_list)
end
```

The current file is first opened, and read using the file variable filvar. The variable student_list is initialized to an empty list, and we shall push on to it as many inner lists as there are students in the class. We initialize inner_list to a list of two elements, the first of which is to contain the student's name, and the second the student's record. From the first line we extract all the items and place them in a record variable which reproduces the original form of the record. When complete, this is placed in the second item of the inner list, and the name only in the first item. This is then pushed onto the student list, as just indicated. This is repeated for as many students as there are in the class. Our original nested structure has now been reproduced. The second part of the procedure extracts each record in this nested structure, stores it temporarily in a variable current, and uses this to display and maybe change the information. Since current is being used as a record, it is possible to add a period and the respective record field to it. When we are through, the same procedure as before is used to write the information to the new file.

We shall leave it to the reader to write a report procedure. This might take the exam scores and sum them, perhaps with some scaling, such as 40% for the final and 30% for the other exams. This figure will be stored in the total element of the record. The grade might be figured by a long if-statement:

```
   if current.total >= 92 then current.grade := ”A” else
   if current.total >= 89 then current.grade := ”A-” else
```

and so on. It is not necessary to state the range of the grade. As soon as it is found to be greater than or equal to one of the stated numbers, it gives a value to current.grade and skips out of the if-statement. Note that as one of the if-statements fits, then all the rest are skipped because they form part of the else-leg of that particular statement. The end will be something like:

```
   if current.total >= 64 then current.grade := "D-" else current.grade := "F"
```

The final else picks up all that remains, which is in effect any grade below 64.

You may wish to try writing a program for a student to access only one record. This will involve having the student enter a password. This will be compared with the password field on each line, and if a match is found make accessible the rest of the data on that line. This program is offered for illustration only, and no guarantees are given as to its effectiveness in practice.

Here now is the program at the point to which it has been developed:

```
record students(name, soc_sec, pwd, fstexam, secexam,final,total, grade)
procedure main()
local result
   result := menu()
   case result of {
      "1" :  initialize()
      "2" : update()
      "3" : report()
      default: write("Goodbye")
      }
end
```

```
procedure menu()
  write(center("Data Base  Program" ,80) )
  write()
  write("Do you wish to")
  write("\t1. Initialize")
  write("\t2. Update")
  write("\t3. Report")
  write()
  writes ("Enter 1,2 or 3 and press return. ”)
  return read()
end
procedure initialize()
  local filename, student, student_table, student_list
  student_table := table()
  writes("Enter the course number, e.g.
  filename := read()
  writes("Enter the last two digits of the current year ")
  filename ||:= read()
  writes("Enter a single digit for the semester ")
  filename ||:= read() II "1.dat"
  writes("The name of the file is ",filename)
  write(". Please note this for future use.")
  write("Enter information about students. Enter period(.) _
        when asked for name to finish.")
  repeat {
    student := students("","","",0,0,0,0,"")
    write()
    writes ("Name? ")
    student.name := read()
    if student.name == "." then break
    writes("Social security number? ")
    student.soc_sec := read()
    writes (“Password? ")
    student.pwd := read()
    student_table[student.name] := student
    }
  student_list := sort(student_table)
  write_to_file(filename, student_list)
end
procedure write_to_file(f,s)
  local filvar,n,p,line
  filvar := open(f,"w")
  n := 0
  while (n +:= 1) <= *s do {
    line := ""
    every p := 1 to 8 do
      line ||:= s[n] [2] [p] II "%"
    write(filvar,line)
    }
  close(filvar)
end
```

```
procedure report()
  write("Space reserved for report.")
end
procedure get_version()
local filename,n
  n := 0
  write("Enter the name of the ORIGINAL filename ")
  filename := read()
  while close(open(filename[1:7] II (n +:= 1) II ".dat"))
  return filename[1:7] II (n - 1) II ".dat"
end
procedure update()
local filename,filvar,n,student_list,line,student, inner_list,current
  filename := get_version()
  filvar := open(filename)
  student _list := []
  while line := read(filvar) do {
    student := students("","","",0,0,0,0,"")
    inner _list := list(2)
    n := 0
    line ? while student[n +:= 1] := tab(upto('%')) do =”%”
    inner_list[1] := student.name
    inner_list[2] := student
    put(student_list,inner_list)
    }
  close(filvar)
  n := 0
  while current := student_list[n +:= 1][2] do {
    write("Name: ",current.name)
    write("Current first exam = ",current.fstexam)
    writes("New score? ")
    current.fstexam := read()
    write("Current second exam = ",current.secexam)
    writes("New score? ")
    current.secexam := read()
    write("Current final exam = ",current.final)
    writes("New score? ")
    current.final := read()
    student_list[n][2] := current
    }
  filename[7] +:= 1
  write_to_file(filename, student_list)
end
```

11.2 A Vocabulary Program

Tables can be constructively used for purposes other than counting. Most people will agree that a major part of learning a foreign language is the absorption of thousands of words, along with their translation into the mother tongue. It was customary to ask students to compile word lists for this purpose. Although merely writing down the words may help somewhat to remember them, retrieval becomes difficult as the lists lengthen. We can uses tables to aid this effort. The data base in this instance can be quite simple: a word in the target language glossed by a translation in the mother tongue. We do not

need a program to create the data base itself; we can use a simple text editor for this purpose. For example we might set up a file *FRtime* for French and English time words:

```
jour%day
heure%hour
mois%month
an%year
janvier%January
February%février
mars%March
avril%April
mai%May
juin%June
juillet%July
août%August
septembre%September
October%octobre
novembre%November
décembre%December
```

The words and their translation are separated by a percent sign (%) or whatever marker you chose. For this purpose, the order does not matter, but you should avoid trailing blanks. You can use any character that can be keyboarded; thus you can type e-acute (é) by entering option-e followed by e, or alt followed by 130 on the keypad, according to the operating system you are using. The special character is encoded by the machine, but this is transparent to the programmer or the user. We can use this simple data base both as a two-way dictionary and a ready way of testing our knowledge.

It may be possible to use this program for non-Roman scripts. Thus, if the international feature is activated on the Apple operating system, foreign scripts can be keyboarded. In this case it is easier to type into the data base all the foreign words first on separate lines, and only then all the English corresponding words. The OS even takes care of the right-to-left orientation of Hebrew automatically by prepending each letter as it is written in.

In the following program, we first open the previously prepared vocabulary file, and create a table vocabulary from the data it contains. We take a line at a time and scan it, storing the first section in the variable *a*. We then pass over the marker, and capture the rest of the line, storing it in *b*. In the table *a* first becomes the *key* to the value *b*, and then *b* becomes the key to the value *a*, thus making the table bi-directional.

We then ask the user if they wish to be tested; if the user does not respond positively, we ask them to enter a word in this case in French or English. When the user does so, we call up the value of that word in the table; if the word does not exist in the table, we return the default value of the table, which is "not found".

If the user opts to be tested, we use a feature of Icon which extracts a pseudo-random key from the table. This is done by prefixing a question mark to the variable which represents the table thus: ?vocabulary. This item is then presented to the user, and their response is compared to the item. This is repeated as long as the user wishes, and then the program is concluded in the usual way. Here is the complete program.

```
global filvar, vocabulary
```

```
procedure main()
  initialize()
  create_table()
  process()
  finish()
end

procedure initialize()
local vocab
  writes("Name of vocabulary? ")
  vocab := read()
  (filvar := open(vocab)) | stop("Vocabulary not found.")
  return
end #initialize

procedure create_table()
local line,a,b
  vocabulary := table("not found")
  while line := read(filvar) do {
    line ? {
      a := tab(upto('%'))
      move(1)
      b := tab(0)
      }
    vocabulary[a] := b
    vocabulary[b] := a
    }
  return
end

procedure process()
local word
  write("Do you wish to be tested? y/n ")
  if getch() == "y" then test()
  else {
    writes("Enter word in English or target language: ")
    while   (word := read()) ~== "." do {
      write("The translation is ", vocabulary[word], ".")
      writes("Enter word in English or target language. Period to finish: ")
      }
    }
  return
end

procedure test()
local item,answer,total,correct
  total := correct := 0
  item :=  ?vocabulary
  writes("What is the translation of ",item,"? ")
  while (answer := read()) ~== "." do {
    total +:= 1
    if answer == vocabulary[item] then {
      correct +:= 1
```

```
      write("Correct.")
      }
    else write("Correct answer is: ",vocabulary[item],".")
    item := ?vocabulary
    writes("What is the translation of ",item,"? Period to finish:")
    }
  write("Your score is ",(correct * 100) / total,"%.")
  return
end

procedure finish()
  close(filvar)
  write ("Goodbye.")
end
```

SUMMARY OF ICON FEATURES

1. A record is declared globally at the beginning of the program. It has a fixed number of fields which are named and may be accessed by creating a variable of this type and adding to its name a period and the field name.

2. A case statement allows for conditions with multiple choices. It tests the value of a variable and if it finds a match, executes the statement following the colon after that match. A default condition may be included.

3. \t is the Icon character for a tab stop.

4. A substring of a string valued variable may be accessed by, for example, str[2:4] which produces the substring between positions 2 and 4 of str.

12 Icon and Markup Languages

12.1 A Brief History of Markup Languages

The word *markup* has several definitions in the American Heritage Dictionary. The fourth one is the one with which we are concerned, and reads as follows:

4a. The collection of detailed stylistic instructions written on a manuscript that is to be typeset.

4b. Computer Science: The collection of tags that describe the specifications of an electronic document, as for formatting. [A *tag* is a labeling device indicating the beginning or ending of an information unit; it is sometimes called a *sentinel*.]

Markup, however, is older than both typesetting and electronic documents by many centuries. Around the seventh century C.E. Hebrew scribes devised a complex system of marks to be added above or below each word in the Hebrew scriptures, in order to represent the traditional cantillation of these texts, that is, the form of high speech intermediate between speaking and singing, used in publicly declaiming them for the instruction of the public. These marks were also a form of punctuation, since the cantillation follows the sense of the words. Soon after, an additional set of vowel points was added, since vowels were not indicated in the original sacred text, and it was considered very important to ensure the correct enunciation. In the 1950s I observed a skilled operator using a Hebrew Linotype machine for a Bible text; he had to make three passes on each line, entering first the consonants, then the vowels, then the cantillation points. Finally he hit a button which justified the line, producing a single slug for printing that line. Syriac, another Semitic language, borrowed the Greek vowels, and positioned them above or below the Syriac consonantal letters. The vocalic markup of Arabic, besides adding vowels, implicitly corrects the pronunciation of the text from the urban dialect in which it was written to the higher-status dialect of the Bedouin. (Orthodox Muslims might not agree with this characterization.) The Indic scripts, like the Tamil syllabary previously mentioned, are another example of markup. The consonants were probably derived from the Aramaic alphabet, and vowels were added to each consonant, ultimately with simplifications, creating a complex alpha-syllabic system.

The markup of printer's proofs is a set of marginal symbols, coupled with underlining of the applicable words, to specify fonts, especially italics and boldface, and to make changes or corrections to the proof. Like the markup of ancient manuscripts, such items grew up in an unregulated manner. Symbols of this type are inconsistent in character, like traditional mathematical notation, and not well suited for computer handling, so a computer-friendly system was devised at IBM for use with their text formatter. This was called GML which stands for *Generalized Markup Language*, and also, not coincidentally, the initial letter of the names of the three individuals responsible for devising this scheme. These tags were introduced by a colon (:) which had the disadvantage of not being obvious on the page. This system was supplanted by SGML (*Standard Generalized Markup Language*) which enclosed tags between the symbols < and > which came to be regarded as pointy brackets, rather than their original mathematical significance. SGML has a steep learning curve, and in the 1990s several simplified versions were proposed. An especially significant application derived from SGML is HTML (*HyperText Markup Language*) with which millions have some contact on account of its use on the World Wide Web. HTML uses a subset of SGML tags, and adds the important capacity of linking to specific sections of a web site, as well as to other web sites. SGML also led to XML (*Extensible Markup Language*) which has a more manageable set of tags, as well as user-defined tags. An important development in 1987 was

TEI (*Text Encoding Initiative*) which concentrates on making available documents of scholarly interest for interchange and research. Further information on this initiative may be found at:

http://www.tei-c.org/index.xml

12.2 Manipulating texts encoded according to the TEI

To show how Icon may be used for extracting information from an encoded text, we offer a program to extract all the quotes from a text from Charlotte Brontë's famous novel *Jane Eyre*. This was encoded years ago by C.M. Sperberg-McQueen, who has been prominent in the development of the TEI.

The commencement of a quote is marked by the tag <q> and its end by </q>. This use of the forward slash to indicate an end tag is standard. We have not included much error-checking—as would be necessary in a commercial program—and the search for quotes is hardwired, something which will often occur in a one-shot program designed for a specific purpose, for example, the study of the local dialect used by the speakers. The program can easily be expanded to look for *any* markup tag, by soliciting the name of a tag from the user, and storing it in a variable. We shall give such an expanded program later. The program first looks for an initial page number, enclosed in a pb (page boundary) tag, and sends it to the output file. You may wish to ascertain for yourself why, if there is no pb tag, the program will produce no output, even though the program is still not erroneous. It then looks for a line with the appropriate <q> tag, and uses the scanning facility to scan for a </q> end tag, or another beginning tag. The quotes are numbered and sent to the output file, retaining the line breaks of the original text. Finally, the user is told the total number of quotes in the text.

Note that there are various possibilities for the occurrence of quotes in a particular line:

- There may be one complete quote in a line.
- There may be several complete quotes in a line.
- A quote may run over the end of the line, and continue in the next, as yet unprocessed, line.
- A line may start with an unfinished quote from the previous line.

All these possibilities must be allowed for. It is convenient to begin by attempting to process a line starting with an unfinished quote, even though in a well-formed, encoded text such an instance cannot be the first line of the data. This permits us to process the rest of the line in the usual way. We shall be helped by a *flag*, which is a way of informing later parts of a program what conditions—in this case “quotedness”— earlier parts have encountered. The use of the term reflects the use of physical flags to indicate the stop/go of the train conductor, or similar alternate states. We do this by selecting a variable which is defined by assigning “up”— or any other value— to it, and this means “quote”. By allowing it to retain its initial null value, or assigning the null value to it explicitly, we indicate the absence of a quote. A back slash prefixed to this variable tests whether the variable is defined, and the expression succeeds only if the variable is in fact defined. The fact that most Icon expressions can succeed or fail usually renders such “Boolean values” unnecessary, but sometimes they are useful.

The first example given here is very simple. In the current version of the TEI additional information may be included in the tag, using an *attribute*, which adds more specificity. For this more sophisticated approach to quotes see:

http://www.tei-c.org/Guidelines/Customization/Lite/U5-hilites.html#z635

which deals with quotation in detail. This is part of the current “Lite” version of the TEI which is supposed to cover 90% of what anyone would need to encode texts.

Here follow:

- The input text from Jane Eyre;
- The Icon program to process the text;
- The output of the program.

12.2.1. The Input Text

```
<teiHeader>
</teiHeader>
<text>
<body>
<pb n=’474’/>
<div1 type="chapter" n=’38’>
<p>Reader, I married him. A quiet wedding we had: he and I,
the parson and clerk, were alone present. When we got back
from church, I went into the kitchen of the manor-house,
where Mary was cooking the dinner, and John cleaning the
knives, and I said &mdash;</p>
<p><q>Mary, I have been married to Mr Rochester this
morning.</q> The housekeeper and her husband were of that
decent, phlegmatic order of people, to whom one may at any
time safely communicate a remarkable piece of news without
incurring the danger of having one’s ears pierced by some
shrill ejaculation and subsequently stunned by a torrent of
wordy wonderment. Mary did look up, and she did stare at
me; the ladle with which she was basting a pair of chickens
roasting at the fire, did for some three minutes hang
suspended in air, and for the same space of time John’s
knives also had rest from the polishing process; but Mary,
bending again over the roast, said only &mdash;</p>
<p><q>Have you, miss? Well, for sure!</q></p>
<p>A short time after she pursued, <q>I seed you go out with
the master, but I didn’t know you were gone to church to be
wed</q>; and she basted away. John, when I turned to him,
was grinning from ear to ear. <q>I telled Mary how it would
be,</q> he said: <q>I knew what Mr Edward</q> (John was an
old servant, and had known his master when he was the cadet
of the house, therefore he often gave him his Christian
name) &mdash; <q>I knew what Mr Edward would do; and I was
certain he would not wait long either: and he’s done right,
for aught I know. I wish you joy, miss!</q> and he politely
pulled his forelock.</p>
<p><q>Thank you, John. Mr Rochester told me to give you and
Mary this.</q></p>
<p>I put into his hand a five-pound note. Without waiting
to hear more, I left the kitchen. In passing the door of
that sanctum some time after, I caught the words &mdash;</p>
```

```
<p><q>She’ll happen do better for him nor ony o’ t’ grand
ladies.</q> And again, <q>If she ben’t one o’ th’
handsomest, she’s noan fa&agrave;l, and varry good-natured;
and i’ his een she’s fair beautiful, onybody may see
that.</q></p>
<p>I wrote to Moor House and to Cambridge immediately, to
say what I had done: fully explaining also why I had thus
acted. Diana and <pb n=’475’/> Mary approved the step
unreservedly. Diana announced that she would just give me
time to get over the honeymoon, and then she would come and
see me.</p>
<p><q>She had better not wait till then, Jane,</q> said Mr
Rochester, when I read her letter to him; <q>if she does,
she will be too late, for our honeymoon will shine our life
long: its beams will only fade over your grave or mine.</q></p>
<p>How St John received the news I don’t know: he never
answered the letter in which I communicated it: yet six
months after he wrote to me, without, however, mentioning Mr
Rochester’s name or alluding to my marriage. His letter was
then calm, and though very serious, kind. He has maintained
a regular, though not very frequent correspondence ever
since: he hopes I am happy, and trusts I am not of those who
live without God in the world, and only mind earthly things.</p>
</body>
</text>
```

12.2.2. The Program

```
# Extracts quotes in the TEI.

#can be used in any procedure
global inputFileVariable, outputFileVariable, quoteCounter

procedure main()   #the main procedure is a "traffic cop" outlining the program
  initialize()
  process()
  finish()
end

procedure initialize()        #Obtain the name of the input and output files
  writes("Name of Input File?> ")
  inputFileVariable := open(read(),"r") |  stop("File not found.")
  writes("Name of Output File?> ")
  outputFileVariable := open(read(),"w")
  quoteCounter := 0
end
procedure process()
local line, newline, quoteFlag      #can be used only in this procedure

  #read in a line, look for initial page number, write it to out file
  while line := read(inputFileVariable) do
    line ?
```

```
      if find("<pb") then {
        tab(upto(&digits))
        write(outputFileVariable,"Page ",tab(many(&digits)))
        break
        }

  #if the incoming line is already a quote, look for its end. When quoteFlag is defined,
  #that is, "up", a quote is being handled; when it is null, "down", regular text is involved.

  while line := read(inputFileVariable) do
  line ? {
    if \quoteFlag then {                    #the prefixed backslash is a non-null test
      if newline := tab(find("</q>")) then {
        move(4)
        quoteFlag := &null
        write(outputFileVariable,newline)
        }
      else {
        write(outputFileVariable, line)
        next
       }
     }

  #find every example of a quote in the rest of the line, or the new line,
  #write it to the out file
   every tab(find("<q>")) do {
    write(outputFileVariable,quoteCounter +:= 1) #increment the counter by 1
    move (3) #move the pointer over the tag
    quoteFlag := "up"
     if newline := tab(find("</q>")) then {
       quoteFlag := &null #the flag is now down
       write(outputFileVariable,newline)
       }
    else  write(outputFileVariable,line[&pos:0]) #&pos holds the position of the counter;
    }                                          #0 is the position after the last character
  }
end

#close files, tell user number of quotes found

procedure finish()
  close(inputFileVariable)
  close(outputFileVariable)
  write("The output file has been created.")
  write("Total number of quotes: ",quoteCounter,".") #Tell user how many quotes
  write("Goodbye.")
end
```

12.2.3. The Output

Page 474
1
Mary, I have been married to Mr Rochester this

morning.
2
Have you, miss? Well, for sure!
3
I seed you go out with
the master, but I didn't know you were gone to church to be
wed
4
I telled Mary how it would
be,
5
I knew what Mr Edward
6
I knew what Mr Edward would do; and I was
certain he would not wait long either: and he's done right,
for aught I know. I wish you joy, miss!
7
Thank you, John. Mr Rochester told me to give you and
Mary this.
8
She'll happen do better for him nor ony o' t' grand
ladies.
9
If she ben't one o' th'
handsomest, she's noan faàl, and varry good-natured;
and i' his een she's fair beautiful, onybody may see
that.
10
She had better not wait till then, Jane,
11
if she does,
she will be too late, for our honeymoon will shine our life
long: its beams will only fade over your grave or mine.

12.3 Expanding the above Program

Let us now proceed to expand the program in the following respects

- The user will be allowed to specify an arbitrary number of tags to be extracted. These tags may be nested in other tags.
- When the page changes, this is noted in the output.
- The user is notified as to the number of tags found, and the information is safely stored in a file named by the user. This file can be inspected as desired.

In order to have more than one suitable tag, let us tag the two occurrences of "did" on page 474 and the phrase "better not" on page 475 with the tag <emph> and its end tag, indicating some form of emphasis, perhaps by italic type.

Here is the program:

```
# Extracts tags in the TEI.
```

```
global inFileName,inputFileVariable,outputFileVariable,tagCounter

procedure main()   #the main procedure is a "traffic cop" outlining the program
  initialize()
  process()
  finish()
end

procedure initialPageNo()
#read in a line, look for initial page number, write it to outfile

  while line := read(inputFileVariable) do
    line ?
      if find("<pb") then {
        tab(upto(&digits))
        write(outputFileVariable,"Page ",tab(many(&digits)))
        break
        }
  return
end

procedure initialize()       #Obtain the name of the input and output files
  writes("Name of Input File?> ")
  inFileName := read()
  inputFileVariable := open(inFileName,"r") |  stop("File not found.")
  writes("Name of Output File?> ")
  outputFileVariable := open(read(),"a")
  initialPageNo()
  tagCounter := 0
end

procedure process()

local line, newline, tagFlag, tagName,pageNo

  writes("Enter the name only of the tag you are seeking. To conclude enter a single period.> ")
  while ((tagName := read()) ~== ".") do {
    # if the incoming line is already tagged, look for its end. When tagFlag is defined,
    # that is, "up", a tagged text unit is being handled; when it is null, "down",
    # regular text is involved.
    write(outputFileVariable,"\nTag = ", tagName)
    while line := read(inputFileVariable) do
    line ? {
      #Check if there is a page boundary in the line
      if tab(find("<pb")) then {
        tab(upto(&digits))
        pageNo := tab(many(&digits))
        write(outputFileVariable,"\nPage ",pageNo)  # \n creates a blank line
        &pos := 1
        } #restore the position of the pointer to the beginning of line
      if \tagFlag then {                              #the prefixed backslash is a non-null test
        if newline := tab(find("</" || tagName || ">")) then {
          tab(upto('>'))
```

```
          move(1)
          tagFlag := &null
          write(outputFileVariable,newline)
          }
        else {
          write(outputFileVariable, line)
          next
          }
        }

      # find every example of the tag in the rest of the line, or the new line,
      # write it to the out file
      every tab(find("<" || tagName || ">")) do {
        write(outputFileVariable,(tagCounter +:= 1)) #increment the counter by 1
        move (2 + *tagName) #move the pointer over the tag
        tagFlag := "up"
        if newline := tab(find("</" || tagName || ">")) then {
          tagFlag := &null #the flag is now down
          write(outputFileVariable,newline)
          }
       else write(outputFileVariable,line[&pos:0]) #&pos holds the position of the counter;
       }                                          #0 is the position after the last character
     }
  #Tell user how many tagged items
  write("Total number of ",tagName," = ",tagCounter,".")
  tagCounter := 0 #reset counter
  close(inputFileVariable) #go to the beginning of the input file
  inputFileVariable := open(inFileName,"r") #by reopening it

  writes("Enter the name only of the tag you are seeking. To conclude enter a single period.> ")
  }
end

#close files
procedure finish()
  close(inputFileVariable)
  close(outputFileVariable)
  write("The output file has been created.")
  write("Goodbye.")
end
```

The user is asked to state the name of a text file for processing; if that file cannot be found, the program concludes immediately. The user then provides the name of the output file, which should be a new file, since all additions are appended to this file. The user is then asked for the name of a tag; the program looks for such tags, writes their contents to the output file, numbering each, and lets the user know how many have been found. It then loops back to ask for another tag, first closing the input file, and then reopening it, which resets the file to the beginning. Since the output file has not been closed, and is in append mode, the material already placed there will not be overwritten. When the user has no more tags to request, a single period point brings the loop to a conclusion. Note that the first solicitation of a tag name occurs before the loop commences, and a similar solicitation occurs *within* the loop, right at the end. This gives full control to the user.

Here is the interchange between the machine and the user:

Name of Input File?> jane.dat
Name of Output File?> jane.out
Enter the name only of the tag you are seeking. To conclude enter a single period.> q
Total number of q = 11.
Enter the name only of the tag you are seeking. To conclude enter a single period.> emph
Total number of emph = 3.
Enter the name only of the tag you are seeking. To conclude enter a single period.> .
The output file has been created.
Goodbye.

And here is the stored output of the program:

Page 474

Tag = q
1
Mary, I have been married to Mr Rochester this
morning.
2
Have you, miss? Well, for sure!
3
I seed you go out with
the master, but I didn’t know you were gone to church to be
wed
4
I telled Mary how it would
be,
5
I knew what Mr Edward
6
I knew what Mr Edward would do; and I was
certain he would not wait long either: and he’s done right,
for aught I know. I wish you joy, miss!
7
Thank you, John. Mr Rochester told me to give you and
Mary this.
8
She’ll happen do better for him nor ony o’ t’ grand
ladies.
9
If she ben’t one o’ th’
handsomest, she’s noan faàl, and varry good-natured;
and i’ his een she’s fair beautiful, onybody may see
that.

Page 475
10
She had <emph>better not</emph> wait till then, Jane,
11
if she does,
she will be too late, for our honeymoon will shine our life
long: its beams will only fade over your grave or mine.

Tag = emph

Page 474
1
did
2
did
3
better not

12.4 Stripping Tags and Character Entities from a Text

TEI encoded texts convey much valuable information; however they may be inappropriate for statistical study, as suggested earlier in this book, since they contain many characters that are markup, and not found in the original text. This would entirely skew a word length study, for example. Here follows a program which achieves the following:

It removes all tags, both beginning and ending, and the entire "header" which precedes the actual text.

It deals with the problem created by "character entities", for example in the word "faàl" where Brontë attempts to represent a dialect word by utilizing a grave accent, which is not normally used in English. Something which is really a single character, is here eight characters. In this case the program looks for items beginning with the ampersand (&), which heralds the onset of a character entity. It then looks at the next character. If this is a named entity, like *à*, beginning with a vowel, or the letters *c* or *n* which frequently take a diacritical mark in Romance languages, the appropriate unmarked letter replaces it. If the ampersand is followed by the pound sign (#), indicating a numeric entity, or some other letter, as in *—* (which represents —) the entity is replaced by an asterisk (*), often used for an ellipsis, and is a single character. The program then seeks out the semi-colon which ends the character entity.

```
global ifv,ofv

procedure main()
  initialize()
  process()
  finish()
end

procedure initialize()
  writes("Name of Input File?> ")
  ifv := (open(read(), "r")) | stop("File not found.")
  writes("Name of Output File?> ")
  ofv := (open(read(),"w")) # a previously created file will be overwritten
  return
end

procedure process()
local line, newline
  line := skipHeader()
  until (match("</body>",line)) do  {
    newline := strip(line)
    write(ofv,newline)
```

```
    line := read(ifv)
    }
  return
end

procedure skipHeader()
local line
 (line := read(ifv)) | write("File is empty.")
 until match("<p>",line) do
  line := read(ifv)
return line
end

procedure strip(l)
local segment, vowel, nl

static markers, vowels

initial {
  markers := '<&'
  vowels := 'aeiouycnAEIOUYCN'
  }

  nl := ""
  l ? {
    while segment := tab(upto(markers)) do {
      nl ||:= segment
      if ="<" then {
        tab(upto('>'))
        move(1)
        }
      else {
        move(1)
        if vowel := tab(any(vowels)) then
          nl ||:= vowel
        else nl ||:= "*"
        tab(upto(';'))
        move(1)
        }
     }
     nl ||:= tab(0)
    }
  return nl
end

procedure finish()
 write("The output file has been created.")
 close(ifv)
 close(ofv)
end
```

We have abbreviated inputFileVariable in this program to ifv; both are acceptable names for variables. You can choose according to the power of your memory, or you can use underscores like this: input_file_variable.

procedure initialize() opens the input file, if possible, and the output file. We then proceed to procedure process() which in turn calls procedure skipHeader() which simply reads the header and throws it away until the tag <p> is reached. This means that a paragraph of real text is now available, and it is returned as the value of the procedure and stored in the variable line. The line on which it occurs is passed to procedure strip(l), which does the main work of the program. There is an important point to note here. It is inadvisable to change the original line, since altering its length (which Icon can do automatically) can cause much confusion. Rather, we build from it a new line, and return this new creation as the value of procedure strip(l). The variables which will hold two csets are designated as static, rather than local, because their value is always the same, and will be used numerous times (see 10.1). The new line we shall build, here called nl, is first initialized to the null string ("") and it will be built up steadily using the very convenient augmented concatenation (||:=) which appends the contents of the right side of the equation to the contents of the left side, and stores the result in the left side variable. We store the segment of the string prior to the markers < and &, and then eliminate the tags, or convert the character entity into a single letter. The remainder of the line (which will often be the entire line!) is covered by nl ||:= tab(0) which moves the pointer to the end of the line, and returns the text from the old place of the pointer. This is added to the new line and makes it complete. This new line is returned as the value of this hard-working procedure. process() writes this out to the output file, and this continues until the tag </body> signals the end of the text, and the loop stops. Files are closed by procedure finish().

SUMMARY OF ICON FEATURES

1. The forward slash (/) prefixed to a variable checks if the variable is null, that is to say, it has not been given a value, or its value has been replaced by the keyword &null. If it does have the null value it succeeds, otherwise it fails.

2. The backslash (\) prefixed to a variable is the reverse; it checks if the variable has a value other than null. If it has such a value it succeeds and produces the value. If not, it fails. This means that a statement like if x then... is useless, because it always succeeds.

13. Conclusion

13.1 Word Processing

In this chapter we shall deal with a few items which may help you in your programming. Let us talk first a little about the question of word processing. This has become a major use of computers, even though it was a by-product of writing computer programs. Computing was originally intended to perform complex mathematical operations accurately and speedily, hence its name. The fact that letters of the alphabet and other symbols can be encoded as numbers made it possible for computers to manipulate texts as well as mathematical entities.

Just half a century ago the word *computer* referred to a human being, and when the French came to need a word for the modern meaning of computer, they selected *ordinateur* which means *that which puts in order,* which is a better representation of the capabilities of the computer. There are many commercial word processors available, some of them inexpensive or even free, and it is not likely that you will want to compete with these products by writing your own program for word processing. However, it may occasionally be convenient to write a small program to perform some task that might be troublesome on your word processor. The SNOBOL-4 language became famous for its "one-line programs" which could do a limited task expeditiously, and Icon has similar capabilities. This is largely due to the fact that statements in Icon can “fail,” which is equivalent to building in a condition into the statement; if certain circumstances prevail then the statement succeeds, and if they do not, it fails. This can be exploited to write very concise mini-programs. For example, let us say you have a file which uses the British spelling *colour*, and you wish to change it to the American usage which omits the *u*. In all probability your word processor will have a command to make such a change throughout the file easily. But say you have a group of files that need such a change, and they have a set of related names such as file1, file2, and so on. In such a case it might be easier to write a little program to handle the change throughout. Let's consider the change in the spelling of the word first, assuming that we have a line in which the word occurs twice. The following loop will take care of it:

```
while line[find("colour",line) + 4] := ""
```

The inner function find() will look for the string we wish to change in the line. Since we are not using here the Icon scanning facility, the function requires as its second argument the string (in this case represented by the variable line) which is being searched. When we use Icon's scanning facility we do not need to do this, since the target string is cited right at the beginning prior to the question mark which initiates this facility. If find() succeeds, it returns the position at the beginning of the word. Used by itself as the index of the variable, this number represents the first letter in the word. By adding four to whatever this number is, we zero in on the offending *u*, and by assigning to it a null string we effectively delete it. This loop will keep on going as long as it finds the word *colour* in the line. When it has changed all occurrences, it will fail. Note that this loop has no “body.” All the work it needs to do is performed by the assignment which takes place in the control statement beginning with while. By enclosing this loop in another loop which reads in lines for processing, and reads them out modified if necessary, we cover the entire file:

```
while line := read(infile) do {
  while line[find("colour",line) + 4] := ""
  write(outfile,line)
  }
```

Now we enclose all of this in a loop which takes care of each file in turn.

Assuming we have twenty files, it will look like this:

```
every n := 1 to 20 do {
  infile := open("file" || n)
  outfile := open("file" || n || ".new","w")
```

followed by the same loop as before and the closing of the current files before the outermost loop is recommenced. You will now have the original set of files, and a modified set of files which have appended the extension *.new* to their names. There is a certain safety feature here; the original files are opened for reading only and so will undergo no change. The modified file was initially created by adding the second argument to the open() function. When you are satisfied that all is as it should be, you can delete the original files using *rm*, or *del*, or whatever your operating system uses. Here is the complete program:

```
# Omits the “u” in “colour” in twenty files
procedure main()
  process()
end
procedure process()
local n,infile,outfile,line
  every n := 1 to 20 do {
    infile := open("file" || n)
    outfile := open("file" || n || ".new","w")
    while line := read(infile) do {
      while line [find("colour" ,line) + 4] := ""
      write(outfile,line)
      }
    close(infile)
    close(outfile)
    }
end
```

Let us take another similar example. I have a set of files in which the names of the section headings are on a separate line and follow an indicator *\section{*, prescribed by the typesetting program, and each section heading is itself closed with a right curly bracket. I have each of these in small letters and wish to capitalize them all. The following line will achieve this:

```
line := line[1: match("\section{" ,line)] || map(line[10:0],&lcase,&ucase)
```

In this instance the function match() is used. This function resembles find(), but it always starts at the beginning of the line, or if the scanning facility is being used, from the position of the imaginary pointer which may be moved up by tab() or move(). Additionally it returns the position after, rather than before, the string which is found. Obviously the position before would be uninteresting since we know where we are starting. We now give a range, from the first position through the end of what match() finds for us, and then concatenate it with the rest of the line (indicated by line[10:0]) after it has had all its small letters made capitals by using the map() function. The first argument is the string with which we are concerned, and all letters of the second argument which occur in the string are switched to the corresponding letters of the third argument, and returned as the value of the function. The values of the keywords representing the small and big letters are really sets, but are automatically switched to strings for this purpose by the Icon system, just as in selecting the filename it automatically changes the integers included in the filename to strings. These automatic changes of data type go on all the time in Icon, and save the programmer considerable effort. The opening and closing of the files can be handled in exactly the same way as before. Two questions arise with regard to these little programs. First, since they are so small why not simply put them into the main procedure? The answer is that procedures

often get modified and reused, and it is better to start off with them as separate procedures, and use the main procedure only for traffic control. It is tempting to short-circuit by throwing everything into the main procedure, but this is a bad habit which should be avoided. Second, is it in order to use programs like this which contain no error checking? The answer is yes, provided that only you yourself use them, and you plan to use them promptly. It is a good idea to delete them after use, although you may want to save some procedures that look as if they could be reused. It would be inadvisable to allow someone else to use such a program. For example if there was already a file with the same name as the output file, this program would destroy that file in order to write the new one. It is easy to check to see if a file exists by attempting to open it and immediately close it. If that succeeds, then the file exists and you may warn the user that the file will be destroyed if the program is allowed to proceed. It is essential to put in such precautions against a careless user, since the risk of losing important information is a serious one. It also points out the need to back up files by making copies and placing them in some other location so that in the event of an accident all is not lost.

13.2 Other Icon Features

Icon is a language rich in features, and those described in this book represent only a part of the capabilities of the language which are described in full in the Griswolds' book mentioned in Chapter 1. The features discussed so far make possible programs of considerable complexity. Here some additional features will be discussed which are less likely to be used regularly but may prove useful from time to time.

13.3 Co-Expressions

Certain expressions known as generators produce a sequence of results rather than a single result. For example, 1 to 10 produces the first ten natural numbers in sequence. We have used this generator with the every loop to cause a loop to function a fixed number of times. It is possible to feed such a sequence into a variable, and produce the next number in the sequence whenever we wish. For example, consider the following fragment of code which would produce a "menu," and store the user's response in the variable answer:

```
write("Would you like to:")
write("l. Initialize")
write("2. Update")
write("3. Report")
writes("Enter number selected:")
answer := read()
```

If, as the program develops, we change the order of the items in the menu, or insert new ones, we shall constantly be changing the numbers. We could avoid that by rewriting the program fragment as follows:

```
a := create 1 to 10
write("Would you like to:")
write(@a,". Initialize")
write(@a,". Update")
write(@a,". Report")
writes("Enter number selected:”)
answer := read()
```

The first line stores the result sequence of 1 to 10 in the variable a, and every time we write that variable preceded by the at-sign, the next result of the sequence is produced. This is known as a co-

expression. In this way, the numbering will always be correct, provided it does not exceed 10, however much we change around the items, or insert new ones.

Additionally, we can reset the sequence to the beginning by prefixing a circumflex to the variable containing the result sequence.

```
b := ^a
```

stores in b the whole sequence anew from the beginning. The keyword ¤t contains the current co-expression, and may be useful if you are using several.

13.4 File Handling

We have already learned to use the function open() which gets a file ready to use if it already exists, and creates it first if it does not. Files should always be closed with close() as soon as they are no longer needed. A file can be deleted from within a program by using the function remove(). The sole argument to this function is the *name* of the file (not a file variable.) The file should always be closed before it is deleted. This function should be used with caution, since it is possible to delete files later using the operating system.

Another function can be used to give a file a new name. This is rename() which takes two arguments, separated as usual by commas, the first of which is the current name of the file, and the second is the new name.

As an example of how we might use these two functions, let us revert to those twenty files we spoke about earlier in this chapter in which we changed "colour" to "color". After concluding the changes, and perhaps arranging for some check that all is as it should be, we might delete the old files containing "colour", and then give the new file the original name:

```
every n := 1 to 20 do {
  remove("file" || n)
  rename("file" || n || ".new","file" || n)
  }
```

Appendix A: Character Sets

Computer character sets are an inheritance from telegraphy (Greek: *writing from a distance)* and a little history will aid in their understanding. In the sixteenth century the Italian physician and mathematician Gerolamo Cardano (1501-1576) suggested that messages could be sent by lighting beacons on five towers. If we designate an unlighted tower by 0 and a lighted tower by 1, then we can have a code such as:

00001 (beacon 5 lighted) = A
00010 (beacon 4 lighted) = B
00011 (beacons 4 & 5 lighted) = C

There are 32 possible combinations (2^5) and this allows for the entire alphabet plus a few extra characters. Cardano had in effect discovered the *bit* or binary digit, i.e., a unit of information which can have one of two states (lighted/unlighted, 0/1, strong current/weak current, and so on.) By combining these *bits*, messages can be encoded. Systems of lanterns using Cardano's principle became common in the nineteenth century. The five-bit Baudot Code named after the French engineer and inventor Jean-Maurice-Émile Baudot (1845-1903) was widely used in the transmission of messages by electrical means. Some characters were used to give information or instructions to the teletype at the other end of the line (to return the carriage or to indicate the end of the message, for example). These had no graphic representation and came to be known as *control characters.* In 1966 the American Standard Code for Information Interchange known as *ASCII* was adopted. This uses seven bits for information and hence has $2^7 = 128$ possible combinations.

There is an extra bit used for checking the accuracy of the transmission, and this makes a total of eight bits. In the ASCII code the first thirty-two characters (from 0 to 31) are control characters; for example, number 12 was used to instruct the machine to eject a page, and so was known as *formfeed* or *FF*. Now *L* in the ASCII code is character 76 which in eight binary digits is 01001100. Character 12 in binary digits is 00001100. To send character 12 the machine simply changed the second digit of character 76 from 1 to 0, and hence this is called control-L. (The "second digit" is normally called bit 6, starting from zero and numbering from the right—which betrays the Semitic origin of our numerical notation system!) The operator indicates this to the machine by pressing *L* while holding down the control key. (Lowercase letters bear a similar relationship. They change the third digit to a 1—01101100.) It is interesting to observe that the highest character 127 is also a control character called *rubout* or *delete.* This is on account of the paper tape which was widely used for input. Once a tape has holes in it (indicating 1's) it is impossible to remove them. But it was possible to backspace and then make all seven digits 1's by punching all of them, since repunching a hole leaves it unchanged. Hence the character 127 (01111111 in binary) was simply ignored. In adapting to computers, many of the control characters lost their original meaning and are used arbitrarily for a number of purposes (especially in word processing) although some of them such as *control-C* for *end of text* remain in many applications. Such decisions are made by the author of the particular program, and there is little consistency between different programs. A control character is often indicated by placing a wedge (caret) or circumflex (^, ∧ or ˆ) in front of the normal printing character.

In Icon this notation may be used to represent control characters if preceded by a backslash. The backslash indicates the special nature of the wedge. Thus in a string \^L represents *control-L.* Since the additional bit which was used for checking the correctness of the transmission may not be required in computer usage, this extra bit can also be used to represent characters, and the extended ASCII set of 256 characters contains additional symbols such as letters with accents used in foreign languages and

mathematical characters. There are other character sets, for example EBCDIC which also has 256 characters but in a quite different order from ASCII, and some special character sets for scientific applications. ASCII is now widely used however, and a text using ASCII only is referred to as “plain text.” Capital letters were traditionally used in indicating control characters, because the obsolete FIELDATA character set used capitals (uppercase letters) only. Small letters may be used.

It is worthwhile mentioning one matter which is the result of historical developments. The typewriter moved to a new line by two distinct movements. First, the platen moved sufficiently to feed in enough paper for one line. Then the carriage shifted to the beginning of the line. For this purpose the character set had two control characters: one to return the carriage and one to perform a linefeed. On computer screens, which have no carriage, only one character is needed. This is therefore dubbed "newline" and just the linefeed character is used. Icon represents this by \l (for “linefeed”) and \n (for “newline”). These are the same character (ASCII 10). Icon uses \r for the carriage return character (ASCII 13). This will cause the cursor to return to the beginning of the line.

Appendix B: Some Hints on Running Icon Programs

Icon for Apple OS X

The Apple OS X operating system is based on a fully-operational UNIX kernel called “Darwin”, and features a *terminal* application which is very helpful in writing, compiling, and executing Icon programs. An image of the terminal appears in the “Dock”, the bar of icons along the edge of the screen, and is invoked by clicking on it.

You will also want to install the Emacs editor that the Macintosh offers, which you will find in “Optional Installs” on the distribution disk. This is an excellent editor for writing programs. As soon as you save a file with the extension *.icn* Emacs will go into a mode specially designed for Icon. For an easy introduction to the main commands, download the following "cheat sheet". On the cheat sheet, C means that you hold down the control key, and press the following key; M means that you hit and release the esc key, and press the following key.

http://www.rgrjr.com/emacs/emacs_cheat.html

There is also a newer Emacs called Aquamacs. It works in a way more typical of the Mac than the old Emacs, but is not well suited to the terminal. Note that the mouse has no function in the terminal, apart from closing or minimizing the Terminal screen.

Now follow these steps, which are a little tedious, but at least you only have to do this once.

Step 1

Click on the Terminal icon, the oblong box containing >-. It will tell you when you last logged in, and show you a dollar sign ($) which is the prompt to enter a command. UNIX arranges directories (folders) in a tree structure. Type in pwd and you will learn that you are in your *home directory*, in my case /Users/alancorre. Your own username should appear of course. The command pwd is useful in case you get lost on your "tree," since it names for you the directory in which you are working. pwd stands for "print working directory" and came about in the days when there was no interactive computing with a screen. So let's think of "produce" rather than "print.

Step 2

We need to make an addition to your .profile file, which Apple sets up for you to initialize Darwin. This will ensure that you will be able to run Icon properly. At the prompt enter: emacs .profile You will see an Emacs screen. Press the down arrow on the keyboard to bypass preliminary information, then paste or type in the following, substituting your username for mine:

```
# Setting the path for MacPorts.
export PATH=/opt/local/bin:/opt/local/sbin:/Users/alancorre/icon.v950/bin:$PATH
```

Press the control key, followed immediately by x and c. You will be asked if you want to save the file. Type in yes. Emacs saves the file, and exits. At the prompt, key in logout and click on the red dot.

Step 3

Download the Icon version 9.5 software for the Intel Mac from:

http://www.cs.arizona.edu/icon/ftp/binaries/unix/mac-x64-v950.tgz

You must have OS X Version 10.6.4 or later for this software, issued in May, 2010.

The resulting zipped (compressed) file should appear on your screen in the Finder. Click on it, and your Mac will expand it for you. Drag the folder icon-v950 to the image of the Terminal in the Dock.

Step 4

It is not a good idea to write programs in your home directory, so on returning to the Terminal, key in mkdir iconprogs which will create a directory called iconprogs one level below your home directory. Give it another name if you like, just one word, no spaces. Then key in cd iconprogs and you will change to that directory. If you now key in pwd you will see that this working directory is empty—so start filling it up! To conclude you key in logout, click on the red dot, and take a deserved rest.

Icon for Windows

For Windows go to http://www.cs.arizona.edu/icon/v93w.htm and click on “5.1 ZIP file” which will download the Icon 9.3.2 installation package. This also works with Windows *Vista*. This piece of software is well designed and quite intuitive, so if you are familiar with Windows you should have no problem. You can use *Notebook* as an editor to write your programs, just be sure that the file is saved as text only.

Happy programming!

Index

ampersand 5, 81
any() 80
arithmetic mean 37
array 91
ASCII 26, 41, 119
asterisk 8
average 37
back slash 31, 79, 90, 104
backspace 86
Baudot Code 119
binary digit 119
binary operator 88
bit 119
caret 57, 119
case 93
center() 72
char() 26
character set 41, 80, 119
circumflex 57
close() 24
co-expression 117
command line 63
comment 32
control character 11, 41, 119
cset 80
data base 91
default 22, 93
delete 81
do 10
editor 17
effect 4
Emacs 17
end 8
end of text 119
every 14, 21
exchange operator 88
exclamation point 75
exponentiation 57
extension 19, 78
failure 4
FIFO 64
file 17
file handling 118
filename 19, 33
formfeed 119
generator 14, 21
global 13
graphics 21
hex number 41, 51
histogram 21
HTML 41
icont 9
indentation 8, 93
inherited failure 10
initial 13
integer() 4
keyword 5, 27
left() 29
LIFO 64
linefeed 41, 86
list 12
local variable 23
loop 2
main procedure 8, 69
many() 9, 15, 79
map() 116
match() 75, 116
matching 74
mean 37
median 38
menu 92, 117
mode 31
modulo 27
move() 27, 116
multi-way condition 93
newline 86, 120
not 25
null string 84
of 93
open() 24, 31, 52, 116
operating system 65, 91
or 25
parentheses 3, 38
password 91
pattern matching 74
Pearson's coefficient 71
pop() 86
procedure 8
program flow 8
push() 64, 86
question mark 7, 74
read() 11, 15, 64
real number 37
real() 37
record 91
remainder 27
remove() 118
rename() 118
repeat 66
result sequence 117
return 8
right() 26
rubout 119
scaling 26, 67
scanning facility 7, 54, 74
scattergram 66
set() 80
sort() 60, 65, 95
Spearman coefficient 66
square root 57
stack 64, 86

standard deviation 53
static 79, 82
stop() 12, 25
string 3, 7
suspend 32, 55
tab character 93
tab() 7, 15, 27
table 52, 74
tilde 57, 94
unary operator 88
underscore 3, 83, 114
until 25
up-arrow 57
upto() 7, 89
value 24
wedge 57, 119
while 10, 66
word frequency 59
word processing 115
write() 4
writes() 22
-:= 83
:=: 88
! 75
? 7
* 8
/ 79
& 27
&ascii 27
&cset 27
¤t 118
&digits 27
&lcase 11
&letters 11
&pos 54
&subject 61
&ucase 11
+:= 13
== 11
||:= 86

תושלב"ע