How to Think Like a Computer Scientist

Logo Version

How to Think Like a Computer Scientist

Logo Version

Allen B. Downey
Guido Gay

Version 1.0
October 30, 2003

History:

March 6, 2003: Allen B. Downey, How to Think Like a Computer Scientist. Java Version, fourth edition.

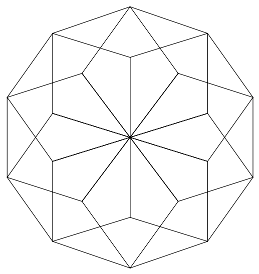
October 30, 2003: Allen B. Downey, Guido Gay, How to Think Like a Computer Scientist. Logo Version, first edition.

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of the book, which can be converted to other formats and printed.

The LaTeX source for this book is available from

`http://ibiblio.org/obp/thinkCS/`

Prefazione



Logo is a programming language mostly used as a teaching tool in primary education. If you know about its turtle graphics commands — such as `repeat 10 [repeat 5 [fd 30 lt 72] lt 36]` — you may be curious about its other capabilities.

Here you'll find a short introduction to Logo as a general purpose programming language.

This document adapts the first chapters of Allen Downey's Open Source book "How to Think Like a Computer Scientist. Java Version". Many thanks to Jeffrey Elkner and Chris Meyers too, for their nice Python version of the book. Allen Downey illustrates the philosopy of his project in the following Preface.

I would like to thank all LogoForum's members: I learned much from their lively discussions.

A sincere thank to Brian Harvey for his "Computer Science Logo Style", a very special book indeed, and to Stan Munson for his help.

Un bacio a Elsa; un abbraccio a Michele e Simone (che preferiscono il gioco del pallone).

Guido Gay
Milano, Italy
October, 2003

Preface

"As we enjoy great Advantages from the Inventions of others, we should be glad of an Opportunity to serve others by any Invention of ours, and this we should do freely and generously."

—Benjamin Franklin, quoted in *Benjamin Franklin* by Edmund S. Morgan.

Why I wrote this book

This is the fourth edition of a book I started writing in 1999, when I was teaching at Colby College. I had taught an introductory computer science class using the Java programming language, but I had not found a textbook I was happy with. For one thing, they were all too big! There was no way my students would read 800 pages of dense, technical material, even if I wanted them to. And I didn't want them to. Most of the material was too specific—details about Java and its libraries that would be obsolete by the end of the semester, and that obscured the material I really wanted to get to.

The other problem I found was that the introduction to object oriented programming was too abrupt. Many students who were otherwise doing well just hit a wall when we got to objects, whether we did it at the beginning, middle or end.

So I started writing. I wrote a chapter a day for 13 days, and on the 14th day I edited. Then I sent it to be photocopied and bound. When I handed it out on the first day of class, I told the students that they would be expected to read one chapter a week. In other words, they would read it seven times slower than I wrote it.

The philosophy behind it

Here are some of the ideas that made the book the way it is:

- Vocabulary is important. Students need to be able to talk about programs and understand what I am saying. I tried to introduce the minimum number of terms, to define them carefully when they are first used, and to organize them in glossaries at the end of each chapter. In my class, I include vocabulary questions on quizzes and exams, and require students to use appropriate terms in short-answer responses.

- In order to write a program, students have to understand the algorithm, know the programming language, and they have to be able to debug. I think too many books neglect debugging. This book includes an appendix on debugging and an appendix on program development (which can help avoid debugging). I recommend that students read this material early and come back to it often.

- Some concepts take time to sink in. Some of the more difficult ideas in the book, like recursion, appear several times. By coming back to these ideas, I am trying to give students a chance to review and reinforce or, if they missed it the first time, a chance to catch up.

- I try to use the minimum amount of Java to get the maximum amount of programming power. The purpose of this book is to teach programming and some introductory ideas from computer science, not Java. I left out some language features, like the `switch` statement, that are unnecessary, and avoided most of the libraries, especially the ones like the AWT that have been changing quickly or are likely to be replaced.

The minimalism of my approach has some advantages. Each chapter is about ten pages, not including the exercises. In my classes I ask students to read each chapter before we discuss it, and I have found that they are willing to do that, and that their comprehension is good. Their preparation makes class time available for discussion of the more abstract material, in-class exercises, and additional topics that aren't in the book.

But minimalism has some disadvantages. There is not much here that is intrinsically fun. Most of my examples demonstrate the most basic use of a language feature, and many of the exercises involve string manipulation and mathematical ideas. I think some of them are fun, but many of the things that excite students about computer science, like graphics, sound and network applications, are given short shrift.

The problem is that many of the more exciting features involve lots of details and not much concept. Pedagogically, that means a lot of effort for not much payoff. So there is a tradeoff between the material that students enjoy and the material that is most intellectually rich. I leave it to individual teachers to find the balance that is best for their classes. To help, the book includes appendices that cover graphics, keyboard input and file input.

Object-oriented programming

Some books introduce objects immediately; others warm up with a more procedural style and develop object-oriented style more gradually. This book is probably the extreme of the "objects late" approach.

Many of Java's object-oriented features are motivated by problems with previous languages, and their implementations are influenced by this history. Some of these features are hard to explain if students aren't familiar with the problems they solve.

It wasn't my intention to postpone object-oriented programming. On the contrary, I got to it as quickly as I could, limited by my intention to introduce concepts one at a time, as clearly as possible, in a way that allows students to practice each idea in isolation before adding the next. It just happens that it takes 13 steps.

Data structures

In Fall 2000 I taught the second course in the introductory sequence, called Data Structures, and wrote additional chapters covering lists, stacks, queues, trees, and hashtables.

Each chapter presents the interface for a data structure, one or more algorithms that use it, and at least one implementation. In most cases there is also an implementation in the `java.utils` package, so teachers can decide on a case-by-case basis whether to discuss the implementation, and whether students will build an implementation as an exercise. For the most part I present data structures and interfaces that are consistent with the implementation in `java.utils`.

The Computer Science AP Exam

During Summer 2001 I worked with teachers at the Maine School of Science and Mathematics on a version of the book that would help students prepare for the Computer Science Advanced Placement Exam, which used C++ at the time. The translation went quickly because, as it turned out, the material I covered was almost identical to the AP Syllabus.

Naturally, when the College Board announced that the AP Exam would switch to Java, I made plans to update the Java version of the book. Looking at the proposed AP Syllabus, I saw that their subset of Java was all but identical to the subset I had chosen.

During January 2003, I worked on the Fourth Edition of the book, making these changes:

- I added a new chapter covering Huffman codes.
- I revised several sections that I had found problematic, including the transition to object-oriented programming and the discussion of heaps.
- I improved the appendices on debugging and program development.
- I added a few sections to improve coverage of the AP syllabus.

- I collected the exercises, quizzes, and exam questions I had used in my classes and put them at the end of the appropriate chapters. I also made up some problems that are intended to help with AP Exam preparation.

Free books!

Since the beginning, this book and its descendents have been available under the GNU Free Documentation License. Readers are free to download the book in a variety of formats and print it or read it on screen. Teachers are free to send the book to a short-run printer and make as many copies as they need. And, maybe most importantly, anyone is free to customize the book for their needs. You can download the LaTeX source code, and then add, remove, edit, or rearrange material, and make the book that is best for you or your class.

People have translated the book into other computer languages (including Python and Eiffel), and other natural languages (including Spanish, French and German). Many of these derivatives are also available under the GNU FDL.

This approach to publishing has a lot of advantages, but there is one drawback: my books have never been through a formal editing and proofreading process and, too often, it shows. Motivated by Open Source Software, I have adopted the philosophy of releasing the book early and updating it often. I do my best to minimize the number of errors, but I also depend on readers to help out.

The response has been great. I get messages almost every day from people who have read the book and liked it enough to take the trouble to send in a "bug report." Often I can correct an error and post an updated version almost immediately. I think of the book as a work in progress, improving a little whenever I have time to make a revision, or when readers take the time to send feedback.

Oh, the title

I get a lot of grief about the title of the book. Not everyone understands that it is—mostly—a joke. Reading this book will probably not make you think like a computer scientist. That takes time, experience, and probably a few more classes.

But there is a kernel of truth in the title: this book is not about Java, and it is only partly about programming. If it is successful, this book is about a way of thinking. Computer scientists have an approach to problem-solving, and a way of crafting solutions, that is unique, versatile and powerful. I hope that this book gives you a sense of what that approach is, and that at some point you will find yourself thinking like a computer scientist.

Allen Downey
Boston, Massachusetts
March 6, 2003

Contents

Chapter 1

The way of the program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, "The way of the program."

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

1.1 The Logo programming language

The programming language you will be learning is Logo. Logo is an example of a **high-level language**; other high-level languages you might have heard of are Perl, Python, and Java.

As the Logo language is not completely standardized the way Python or Perl are, in this book we'll refer to Berkeley Logo, the closest to a standard as far as Logo is concerned. The Berkeley Logo interpreter is an open source software, running under a number of different operating systems and can be downloaded from Brian Harvey's web site (`http://http.cs.berkeley.edu/~bh`).

As you might infer from the name "high-level language," there are also **low-level languages**, sometimes referred to as "machine languages" or "assembly languages.” Loosely speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.

But the advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications.

Two kinds of programs process high-level languages into low-level languages: **interpreters** and **compilers**. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.

A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation.

Logo is considered an interpreted language because Logo programs are executed by an interpreter. There are two ways to use the interpreter: command-line mode and script mode.

In command line mode, you type Logo programs and the interpreter prints the result:

```
# logo
Welcome to Berkeley Logo version 5.3
? print sum 1 1
2
```

The first line of this example is the command that starts the Logo interpreter. The next line is the greeting message from the interpreter. The third line starts with `?`, which is the prompt the interpreter uses to indicate that it is ready. We typed `print sum 1 1` on the command line, pressed the “Return” key and the interpreter replied `2`.

Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a **script**. For example, we used a text editor to create a file named `example.lgo` with the following contents:

```
print sum 1 1
```

By convention, files that contain Logo programs have names that end with `.lgo`.

To execute the program, we have to tell the interpreter the name of the script:

```
# logo example.lgo
2
```

In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.

Most of the examples in this book are executed on the command line. Working on the command line is convenient for program development and testing, because you can type programs and execute them immediately. Once you have a working program, you should store it in a script so you can execute or modify it in the future.

1.2 What is a program?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or (strangely enough) compiling a program.

The details look different in different languages, but a few basic instructions appear in just about every language:

input: Get data from the keyboard, a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

conditional execution: Check for certain conditions and execute the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look more or less like these. Thus, we can describe programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

That may be a little vague, but we will come back to this topic later when we talk about **algorithms**.

1.3 What is debugging?

Programming is a complex process, and because it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

1.3.1 Syntax errors

Logo can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without spewing error messages. Logo is not so forgiving. If there is a single syntax error anywhere in your program, Logo will print an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

1.3.2 Runtime errors

The second type of error is a **runtime error**, so called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

1.3.3 Semantic errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its **semantics**) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (A. Conan Doyle, *The Sign of Four*)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does *something* and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux." (*The Linux Users' Guide* Beta Version 1).

Later chapters will make more suggestions about debugging and other programming practices.

1.4 Formal and natural languages

Natural languages are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> **Programming languages are formal languages that have been designed to express computations.**

Formal languages tend to have strict rules about syntax. For example, $3+3=6$ is a syntactically correct mathematical statement, but `3=+6$` is not. H_2O is a syntactically correct chemical name, but ${}_2Zz$ is not.

Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with `3=+6$` is that `$` is not a legal token in mathematics (at least as far as we know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax error pertains to the structure of a statement—that is, the way the tokens are arranged. The statement `3=+6$` is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

> *As an exercise, create what appears to be a well-structured English sentence with unrecognizable tokens in it. Then write another sentence with all valid tokens but with invalid structure.*

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a shoe is and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax, and semantics—there are many differences:

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.5 The first program

Traditionally, the first program written in a new language is a simple greeting message, such as "Hello, World!". In Logo, it looks like this:

```
print [Hello, World!]
```

This Logo istruction displays a value on the screen. In this case, the result is the sentence

```
Hello, World!
```

`[Hello, World!]` is a Logo list. The square brackets delimit the list but are not printed.

Some people judge the quality of a programming language by the simplicity of a simple greeting program. By this standard, Logo does as well as possible.

1.6 Glossary

problem solving: The process of formulating a problem, finding a solution, and expressing the solution.

high-level language: A programming language like Logo that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to execute; also called "machine language" or "assembly language."

portability: A property of a program that can run on more than one kind of computer.

interpret: To execute a program in a high-level language by translating it one line at a time.

compile: To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

source code: A program in a high-level language before being compiled.

object code: The output of the compiler after it translates the program.

executable: Another name for object code that is ready to be executed.

script: A program stored in a file (usually one that will be interpreted).

program: A set of instructions that specifies a computation.

algorithm: A general process for solving a category of problems.

bug: An error in a program.

debugging: The process of finding and removing any of the three kinds of programming errors.

syntax: The structure of a program.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to interpret).

runtime error: An error that does not occur until the program has started to execute but that prevents the program from continuing.

exception: Another name for a runtime error.

semantic error: An error in a program that makes it do something other than what the programmer intended.

semantics: The meaning of a program.

natural language: Any one of the languages that people speak that evolved naturally.

formal language: Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

token: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

parse: To examine a program and analyze the syntactic structure.

Chapter 2

Values and variables

2.1 Evaluation

Logo displays `2` on the screen in response to `print sum 1 1` on the command line. This instruction doesn't do much, but it's useful to illustrate how Logo evaluates instructions.

`print` and `sum` are primitive procedures, known by Logo in the beginning. A **procedure** is like a recipe that allows Logo to carry out a specific task.

Every procedure accepts a given number of inputs. Inputs can be numbers or other kinds of information. `print` accepts one input, `sum` two.

Commands are procedures that have an effect —`print` is a command that displays its input on the screen— while **functions** are procedures that output a **value**, returning it to another procedure. `sum` is a Logo function that adds two numbers.

A Logo **instruction** consists of the name of a command, followed by as many expressions as necessary to provide its inputs. A Logo **expression** is either an explicitly provided value such as a number or else the name of a function, followed by as many expressions as necessary to provide its inputs. `sum 1 1` is an expression, as an explicitly provided value like `2`.

We can now describe the steps Logo takes to evaluate the instruction `print sum 1 1`:

1. Logo first reads the word `print`. Logo knows that `print` is the name of a command that requires one input, so it keeps on reading.

2. Logo reads the word `sum`, the name of a function that requires two inputs, so it keeps on reading.

3. Logo reads the following two numbers and invokes the expression `sum 1 1` that returns the value `2` to the command `print`.

4. At this point Logo invokes `print 2` which displays `2` on the screen.

Using the value returned by one procedure as the input to another procedure is called **composition** of functions.

2.2 Words and numbers

A Logo **word** is an ordered collection of characters, delimited by spaces, square brackets, parentheses, braces and arithmetic operators.

When Logo is evaluating instructions it always interprets unquoted words as names of procedures. In order to convince Logo to read a word simply as itself, we must type a quotation mark (`"`) in front of it:

```
? print hello
I don't know how to hello
? print "hello
hello
```

In the first instruction Logo thinks that `hello` is the name of a procedure and therefore prints an `error message`. In the second the word is `quoted` and it is evaluated as itself, as the input of the command `print`.

If a word includes a a space, we should use the following idiomatic form:

```
? print "|Hello, World!|
Hello, World!
```

Numbers are special kinds of words, that happen to contain only digits, the decimal point and the character `e`.

The following are valid logo numbers:

```
1 1.9 0.01 0.05 .7 4. 1.1e1 0.2e2 .6e3
```

This can be seen by using a function called `numberp` that returns `true` if the input is in fact a number:

```
? print numberp 1
true
? print numberp 1.1e1
true
? print numberp "ER
false
```

`numberp` is a member of a class of functions that return the values `true` or `false`, also called **predicates**. `wordp` is another predicate, that tests if its input is a word.

Logo tries very hard to make things easy for beginners, so numbers of different kind —integers and decimals— can be freely mixed in arithmetic expressions.

```
? print sum 1.1 3
4.1
? print difference 3.25 4
-0.75
? print product 0.1 5
0.5
? print quotient 1 3
0.333333333333333
```

Numbers are always evaluated as themselves, that is their value after evaluation is what it was before evaluation. On the other hand, we can quote numbers, if we wish.

```
? print "1.1
1.1
? print sum 2 "3
5
? print product 0.1 "5e0
0.5
```

2.3 Lists and arrays

Lists are ordered collections of elements. In the following example, the list's elements are words.

```
? print [This is a flat list]
This is a flat list
? show [This is a flat list]
[This is a flat list]
```

`show` is a command that displays its input on the screen. It differs from `print` because it displays the delimiting characters of lists —square brackets.

Keep in mind that square brackets serve two purposes at once: they delimit a list —without actually being part of it— while quoting it, so that Logo's evaluator interprets the list as representing itself, without invoking the procedures it names.

```
? print [print "hello]
print "hello
? print "hello
hello
```

As with numbers and words, we can test the type `list` with a specific predicate:

```
? print listp [a b c 1]
true
? print listp [a [b 1] d]
true
? print listp [a {b 1} d]
true
```

Lists elements can be words, numbers, other lists or, as in the third example, arrays. If a list contains only words or numbers it is called a `sentence`.

As lists, **arrays** are ordered collections of elements. Differently from lists, arrays have a fixed length that should be declared beforehand, while lists can grow and shrink at will.

```
? print {This is an array}
{This is an array}
? print {1 {a b} 3 4}
{1 {a b} 3 4}
```

Braces delimit an array and quote it. Array elements can be words, numbers, other arrays or lists.

As with other data types, we can test arrays with a specific predicate:

```
? print arrayp {a b c 1}
true
? print arrayp {a [b 1] d}
true
? print arrayp {a {b 1} d}
true
```

The function `array` outputs an array with a given size, with members empty lists.

```
? print array 4
{[] [] [] []}
```

As we'll see in later chapters, arrays aren't often used in Logo, although sometimes they are the best solution to specific problems, particularly those requiring extensive shuffling of elements.

2.4 Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is identified by a name that refers to a value.

The command `make` creates new variables and gives them values:

```
? make "name  "Guido
? make "n 17
? make "hello [Hello, World!]
? make "age {12 13 11 13 14 10}
```

This example makes four assignments. The first assigns `Guido` as the value of a new variable named `name`. The second gives the integer `17` to `n`, the third assign the list `Hello, World!` to `hello` and the fourth assigns the array `12 13 11 13 14 10` as the value of a new variable named `age`.

The print command also works with variables.

```
? print thing "name
Guido
? print thing "n
17
? print thing "hello
Hello, World!
? print thing "age
{12 13 11 13 14 10}
```

`Thing` is a function. It takes one input, which must be a word that's the name of a variable, and outputs the value of the variable.

Examining the value of a variable is such a common task that Logo allows this abbreviation, where the character `:` stands for `thing "`.

```
? print :name
Guido
? print :n
17
? print :hello
Hello, World!
? print :age
{12 13 11 13 14 10}
```

2.5 Variable names

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case doesn't matter. `Bruce` and `bruce` are the same variables.

2.6 Operators and operands

Operators are special symbols that represent computations like addition and multiplication and can be used instead of functions such as `sum` or `product`. The values the operator uses are called **operands**.

The following are all legal Logo expressions whose meaning is more or less clear:

```
20+32   :hour-1    :minute/60      (5+9)*(15-7)
```

The symbols `+`, `-`, and `/`, and the use of parenthesis for grouping, mean in Logo what they mean in mathematics. The asterisk (`*`) is the symbol for multiplication.

There is no exponentiation operator in Logo and the function `power` should be used instead:

```
? print power 3 2
9
```

When a variable name appears in the place of an operand, it is replaced with its value before the function is performed.

2.7 Order of operators

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Logo allows the same precedence rules for its mathematical operators that mathematics does:

- Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4. You can also use parentheses to make an expression easier to read, as in `(:minute * 100) / 60`, even though it doesn't change the result.
- Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So `2*3-1` yields 5 rather then 4.
- Operators with the same precedence are evaluated from left to right.

2.8 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and they are marked with the `;` symbol:

```
; compute the percentage of the hour that has elapsed
make "percentage  (59 * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
make "percentage (59 * 100) / 60     ; elapsed time
```

Everything from the `;` to the end of the line is ignored—it has no effect on the program. The message is intended for the programmer or for future programmers who might use this code.

2.9 Glossary

procedure: a set of instructions that allows Logo to carry out a specific task (a computation).

primitive procedure: a procedure known by Logo by design.

command: a procedure that has an effect, such as print or forward.

function: a procedure that outputs a value, returning it to another procedure.

value: A number, word, list or array.

instruction: the name of a command, followed by as many expressions as necessary to provide its inputs.

expression: an explicitly provided value or else the name of a function, followed by as many expressions as necessary to provide its inputs.

word: an ordered collection of characters

predicate: a function that returns true or false

list: an ordered collection of elements

array: an ordered collection of elements

variable: It is identified by a name associated to a value.

assignment: A command that assigns a value to a variable.

operator: A special symbol that represents a simple computation like addition or multiplication.

operand: One of the values on which an operator operates.

rules of precedence: The set of rules governing the order in which arithmetic expressions involving multiple operators and operands are evaluated.

comment: Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

Chapter 3

Procedures

3.1 Manipulating words

Sometimes it's useful to take apart words or put them together.

The functions `first` and `butfirst` are the building blocks for many of the more interesting procedures we will see in later chapters.

```
? print first "abcd
a
? print first 231
2
? print butfirst "abcd
bcd
? print butfirst 1.023
.023
? print butfirst butfirst "abcd
cd
? print first butfirst "abcd
b
```

As the names imply, `first` returns the first character of its input, while `butfirst` returns all the characters of its input but the first.

The function `item` takes two inputs, the first a positive interger and the second a word. The function returns the *n*th character of the word if the first input is *n*.

```
? print item 2 "abcd
b
? print item 3 231
1
```

We can combine words with the function `word` that accepts two words as inputs and outputs a new word by concatenating the characters in the input words.

```
? print word "abcd "efg
abcdefg
? print word "abcd first 123
abcd1
```

We can also combine words in a sentence, with `sentence`, a function that takes two words as inputs and outputs a flat list.

```
? print sentence "Berkeley "Logo
Berkeley Logo
? print (sentence "Berkeley "Logo "is "a "nice "program)
Berkeley Logo is a nice program
? show (sentence 1 2 3 4 5 6 7 8 9 10)
[1 2 3 4 5 6 7 8 9 10]
```

A point is worth mentioning. Logo primitive procedures accept a default number of inputs —two in the case of `sentence`— but sometimes we can change the default number using parentheses.

```
? (print  "Berkeley "Logo)
Berkeley Logo
? (show 1 2 3 4 5 6 7 8 9 10)
1 2 3 4 5 6 7 8 9 10
? print (product 2 3 4)
24
? print (sum 20 30 10 40)
100
? show (word "U "C "B "Logo)
UCBLogo
```

Finally, `count` is a function that counts the characters in a word.

```
? print count "abcdefg
7
? print count word first "abcd butfirst 1234567890
10
```

3.2 Type conversion

Logo tries very hard to automatically convert values as needed.

Words formed only by digits, the decimal point and the character `e` arranged in specific sequences, can be used in mathematical expressions.

On the other side, as we've seen, numbers can be the input to functions that operate on words.

Still there are two explicit type converting funtions, namely `arraytolist` and `listtoarray`, that convert lists into arrays and viceversa.

```
? show arraytolist {1 2 3}
[1 2 3]
? show listtoarray [a b c]
{a b c}
```

3.3 Math functions

In mathematics, you have probably seen functions like `sin` and `log`, and you have learned to evaluate expressions like `sin(pi/2)` and `log(1/x)`. First, you evaluate the expression in parentheses (the input of the funtion or its argument). For example, `pi/2` is approximately 1.571, and `1/x` is 0.1 (if `x` happens to be 10.0).

Then, you evaluate the function itself, either by looking it up in a table or by performing various computations. The `sin` of 1.571 is 1, and the `log` of 0.1 is -1 (assuming that `log` indicates the logarithm base 10).

This process can be applied repeatedly to evaluate more complicated expressions like `log(1/sin(pi/2))`. First, you evaluate the argument of the innermost function, then evaluate the function, and so on.

Logo provides most of the familiar mathematical functions.

```
? make "decibel log10 17
? make "angle  45
? make "height sin :angle
```

The first statement sets `decibel` to the logarithm of 17, base `10`. There is also a function called `ln` that takes logarithm base `e`.

The third statement finds the sine of the value of the variable `angle`. `sin` and the other trigonometric functions take arguments in degrees. If you know your geometry, you can verify the result by comparing it to the square root of two divided by two:

```
? print sin 45
0.707106781186547
? print quotient 1 sqrt 2
0.707106781186548
```

As we have already seen, functions can be composed, as in the following example:

```
? make "pi 3.14159
? make "x cos sum :angle quotient :pi 2
```

This instruction takes the value of `pi`, divides it by 2, and adds the result to the value of `angle`. The sum is then passed as an argument to the `cos` function.

3.4 Adding new procedures

So far, we have only been using the procedures that come with Logo, but it is also possible to add new procedures. Creating new procedures to solve your particular problems is one of the most useful things about a general-purpose programming language.

In the context of programming, a **procedure** is a named sequence of instructions that performs a desired operation. This operation is specified in a **procedure**

definition. The procedures we have been using so far have been defined for us, and these definitions have been hidden. This is a good thing, because it allows us to use the procedures without worrying about the details of their definitions.

The syntax for a procedure definition is:

```
define "name [[List of parameters] [instruction instruction ...]]
```

You can make up any names you want for the procedures you create, except that you can't use a number. The list of parameters specifies what information, if any, you have to provide in order to use the new procedure.

The first couple of procedures we are going to write have no parameters, so the syntax looks like this:

```
define "newLine [[][print "]]
```

This procedure is named `newLine`. The empty square brackets indicate that it has no parameters. It contains only a single instruction, which displays a newline character.

The syntax for calling the new procedure is the same as the syntax for built-in procedures:

```
? newLine

?
```

Notice the extra space between the two command lines. What if we wanted more space between the lines? We could write a new procedure named `threeLines` that prints three new lines:

```
define "threeLines [[][newLine newLine newLine]]
? threeLines



?
```

You should notice a few things about this program:

1. You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.

2. You can have one procedure call another procedure; in this case `threeLines` calls `newLine`.

So far, it may not be clear why it is worth the trouble to create all of these new procedures. Actually, there are a lot of reasons, but this example demonstrates two:

- Creating a new procedure gives you an opportunity to name a group of instructions. Procedures can simplify a program by hiding a complex computation behind a single name.

- Creating a new procedure can make a program smaller by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to call `threeLines` three times.

3.5 Definitions and use

Pulling together the code fragments from the preceding section, the whole program looks like this:

```
define "newLine [[][print "]]
define "threeLines [[][newLine newLine newLine]]
```

This program contains two procedure definitions: `newLine` and `threeLines`. `threeLines` is the `top-level procedure`, that is the procedure we invoke on the command line to run the program.

```
? threeLines



?
```

As you might expect, you have to create a procedure before you can execute it. In other words, the procedure definition has to be executed before the first time the procedure is called.

3.6 Procedures with inputs

Some of the built-in procedures you have used require one or more inputs. For example, if you want to find the sine of a number, you have to indicate what the number is. The value taken by a a function's input when the function is invoked is called the actual argument. Thus, when invoked, `sin` takes a numeric value as its argument.

Some procedures take more than one argument. For example, `power` takes two arguments, the base and the exponent. Inside the procedure, the values that are passed get assigned to local variables called **parameters**.

Here is an example of a user-defined procedure that takes a parameter:

```
define "printTwice [[bruce][print :bruce print :bruce]]
```

This procedure takes a single argument and assigns it to a parameter named `bruce`. The value of the parameter (at this point we have no idea what it will be) is printed twice. The name `bruce` was chosen to suggest that the name you give a parameter is up to you, but in general, you want to choose something more illustrative than `bruce`.

The procedure `printTwice` works with different types of inputs:

```
? printTwice "john
john
john
? printTwice 3.1459
3.1459
3.1459
? printTwice [one two]
one two
one two
?
```

In the first procedure call, the argument is a word. In the second, it's a number. In the third, it's a list.

The same rules of composition that apply to built-in procedures also apply to user-defined procedures, so we can use any kind of expression as an argument for `printTwice`:

```
? printTwice word "Ucb "logo
Ucblogo
Ucblogo
? printTwice product 3 2
6
6
```

We can also use the value of a variable as an argument:

```
? make "name  "Michele
? printTwice :name
Michele
Michele
```

Notice something very important here. The name of the variable (`name`), whose value we pass as an argument, has nothing to do with the name of the parameter (`bruce`). It doesn't matter what the value was called back home (in the caller); here in `printTwice`, we call everybody `bruce`.

3.7 The `to` command

Procedures in Logo can also be defined with the special form `to`.

Consider the new procedure `square`:

```
define "square [[n][print product :n :n]]
```

and see what happens when we invoke `pops`:

```
? square 3
9
? pops
to square :n
print product :n :n
end
```

We defined `square` in the usual way but when we invoked `pops` — a command that prints the definitions of all procedures — we got an equivalent definition based on the special command `to`.

The first name after `to` is the name of the procedure (it should not be quoted); `n` is the name of the parameter and should be preceded by `:`; the second line is the body of the procedure which terminates with the keyword `end`, written by itself on a final line.

Using `define` or `to` is mainly a matter of taste. Following most Logo authors, in the rest of this manual we'll be using `to` to define new procedures.

3.8 Local variables and parameters

When you create a **local variable**, it only exists inside the procedure, and you cannot use it outside. For example:

```
to catTwice :part1 :part2
   local "cat
   make "cat word  :part1  :part2
   printTwice :cat
end
```

when invoked gives this result:

```
? catTwice "E "R
ER
ER
```

This procedure takes two arguments, creating two local variables named `part1` and `part2`; the command `local` creates a new (local) variable named `cat`; `make` assigns the value of the expression `word :part1 :part2` to `cat`; `printTwice` prints the result twice.

Notice that when we are invoking `to` from the command line, Logo reminds us that we are defining a new procedure beginning the lines following the first with a `>`.

When `catTwice` terminates, the variable `cat` is destroyed, as are the two parameters `part1` and `part2`. If we try to print them, we get an error:

```
? print :cat
cat has no value
? print :part1
part1 has no value
? print :part2
part2 has no value
```

3.9 Glossary

procedure: A named sequence of instructions that performs some useful operation. Procedures may or may not take parameters and may or may not produce a result.

procedure definition: an instruction that creates a new procedure, specifying its name, parameters, and the instructions it executes.

procedure invocation: to carry out a procedure, to do what the procedure says.

procedure call: to carry out procedure, to do what the procedure says.

argument: A value provided to a procedure when the procedure is called. This value is assigned to the corresponding parameter in the procedure.

parameter: A name used inside a procedure to refer to the value passed as an argument.

flow of execution: The order in which instructions are executed during a program run.

local variable: A variable defined inside a procedure. A local variable can only be used inside its procedure invocation.

Chapter 4

Conditionals and recursion

4.1 The remainder function

The remainder function works on integers and yields the remainder when the first operand is divided by the second.

```
? print remainder 7 3
1
```

So 7 divided by 3 is 2 with 1 left over.

The remainder function turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if `remainder :x :y` is zero, then `x` is divisible by `y`.

4.2 Boolean expressions

A **boolean expression** is an expression that is either true or false. In Logo, an expression that is true has the value `true`, and an expression that is false has the value `false`.

The predicate `equalp` compares two values and produces a boolean expression:

```
? print equalp 5  5
true
? print equalp 5  6
false
```

In the first command, the two arguments are equal, so the expression evaluates to true; in the second statement, 5 is not equal to 6, so we get false.

The `equalp` function is one of the three numerical comparison predicates; the other two are:

```
? print lessp 5 6
true
? print greaterp 5 6
false
```

We have already seen a number of predicates, such as those that check the type of a piece of information (`wordp, numberp, listp, arrayp`); we'll introduce the others as needed in the following chapters.

4.3 Logical functions

There are three **logical functions**: `and`, `or`, and `not`. The semantics (meaning) of these functions is similar to their meaning in English.

`and` accepts two arguments; each argument must be either the word `true` or the word `false`; it returns `true` if both its inputs are `true`, `false` if either of them is `false`.

`or` accepts two arguments; each argument must be either the word `true` or the word `false`; it returns `true` if at least one input is `true`, `false` if both inputs are `false`.

Both `and` and `or`, when enclosed in parentheses, accept more than two inputs.

`not` accepts one argument that must be either the word `true` or the word `false`; it returns `true` if its input is `false`, `false` if it's `true`.

```
make "x 3
make "y 6
? print and lessp :x 6 greaterp :x 2
true
? print or lessp :x 6 greaterp :x 4
true
? print not lessp :x 2
true
? print (and lessp :x 6 greaterp :x 2 greaterp :y 5 lessp :y 8)
true
? print and equalp remainder :y 3 0 equalp remainder :y 2 0
true
```

For example, the last instruction tests if `y` is divisible by 2 *and* 3.

4.4 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional procedures give us this ability. The simplest form is the `if` command:

```
? make "x 3
? if greaterp :x  0 [print "|x is positive|]
x is positive
```

`if` is a command with two arguments. The first must be a boolean expression, called the condition. The second is a list containing logo instructions. If the first input returns a value `true`, then the instructions in the following list are evaluated. If not, nothing happens.

4.5 Alternative execution

`ifelse` is a command such that, depending on the value of a boolean expression, two different alternatives are executed.

The command looks like this:

```
? make "x 8
? ifelse equalp remainder :x 2 0 [print "even][print "odd]
even
```

If the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays a message to that effect. If the condition is false, the second list is evaluated. The alternatives are called **branches**, because they are branches in the flow of execution.

4.6 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. In Logo one would normally express a computation like that using a combination of `if` with `stop`:

```
to printPositiveNegative :x
   if greaterp :x  0 [(print :x "is "positive) stop]
   if equalp :x  0 [print "zero stop]
   if lessp :x  0 [(print :x "is "negative) stop]
end
```

`stop` is a command that ends the execution of the procedure in which it appears. In the example, each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the procedure prints a message and then the command `stop` ends the procedure's execution.

```
? printPositiveNegative 4
4 is positive
? printPositiveNegative 0
zero
? printPositiveNegative -4
-4 is negative
```

4.7 Nested conditionals

One conditional can also be nested within another. We could have written the last example as follows:

```
to printPositiveNegative :x
   ifelse greaterp :x  0 [(print :x "is "positive)][
   ifelse equalp :x  0 [print "zero][(print :x "is "negative)]]
end
```

The outer conditional contains two branches. The first branch contains a simple print instruction. The second branch contains another `ifelse` instruction, which has two branches of its own. Those two branches are both print instructions, although they could have been conditional instructions as well.

Nested conditionals become difficult to read very quickly, and in general it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional instructions. For example, we can rewrite the following code using a single conditional:

```
to example :x
   if greaterp :x  0 [
      if lessp :x  10 [print "|0<x<10|]]
end
```

The `print` instruction is executed only if we make it past both the conditionals, so we can use the `and` logical function:

```
to example :x
   if and greaterp :x 0 lessp :x 10 [print "|0<x<10|]
end
```

4.8 Recursion

We mentioned that it is legal for one procedure to invoke another, and you have seen several examples of that. We neglected to mention that it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do. For example, look at the following command:

```
to countdown :n
   if equalp :n 0 [print "Blastoff! stop]
   print :n
   countdown difference :n 1
end
```

`countdown` expects the parameter, `n`, to be a positive integer. If `n` is 0, it prints the word “Blastoff!” and then stops. Otherwise, it prints `n` and then calls a function named `countdown`—itself—passing `difference n 1` as an argument.

What happens if we call this function like this:

```
? countdown 3
```

The execution of `countdown` begins with `n=3`, and since `n` is not 0, it prints the value 3, and then calls itself...

> The execution of `countdown` begins with `n=2`, and since `n` is not 0, it prints the value 2, and then calls itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is not 0, it prints the value 1, and then calls itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is 0, it prints the word, "Blastoff!"; it then invokes the command `stop` that stops the execution of the procedure.
> >
> > The `countdown` that got `n=1` has no more instructions to execute and therefore stops.
>
> The `countdown` that got `n=2` has no more instructions to execute and therefore stops.

The `countdown` that got `n=3` has no more instructions to execute and therefore stops.

So, this is what we get:

```
3
2
1
Blastoff!
```

The process of a function calling itself is **recursion**, and such functions are said to be recursive.

Invoking `trace`, Logo shows us the sequence of procedure invocations we have described verbally:

```
? trace "countdown
? countdown 3
( countdown 3 )
3
 ( countdown 2 )
2
  ( countdown 1 )
1
   ( countdown 0 )
Blastoff!
   countdown stops
  countdown stops
 countdown stops
countdown stops
?
```

The condition in the first line of `countdown` is called the **base case**.

4.9 Infinite recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not considered a good idea.

If we invoke `countdown` with a decimal number, we will never reach the base case, that is `n=0`.

```
? countdown 2.3
2.3
1.3
0.3
-0.7
-1.7
-2.7
-3.7
-4.7
```

In this example we interrupted `countdown` after a few iterations, otherwise it would have kept on printing negative numbers for a long time!

4.10 Keyboard input

The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time.

Logo provides several built-in procedures that get input from the keyboard. The simplest is called `readlist`. When this function is called, the program stops and waits for the user to type something. When the user presses Return or the Enter key, the program resumes and `readlist` returns what the user typed as a `list`:

```
? make "input readlist
This is a list
? show :input
[This is a list]
```

Before calling `readlist`, it is a good idea to print a message telling the user what to input. This message is called a **prompt**:

```
? type [What's your name? ] make "input readlist
What's your name? Arthur, King of the Britons!
? print :input
Arthur, King of the Britons!
```

4.11 Glossary

boolean expression: An expression that returns either true or false.

condition: The boolean expression in a conditional procedure that determines which branch is executed.

recursion: The process of calling the procedure that is currently executing.

base case: A branch of the conditional statement in a recursive procedure that does not result in a recursive call.

infinite recursion: A procedure that calls itself recursively without ever reaching the base case. Eventually, an infinite recursion causes a runtime error.

prompt: A visual cue that tells the user to input data.

Chapter 5

Functions

5.1 Return values

Some of the built-in procedures we have used, such as the math functions or functions operating on words, return values.

```
? print exp 1
2.71828182845905
? print first "ER
E
```

But so far, none of the procedures we have written has returned a value.

In this chapter, we are going to write procedures that return values. The first example is `area`, which returns the area of a circle with the given radius:

```
to area :radius
  local "temp
  make "temp product 3.14159 power :radius 2
  output :temp
end
```

`output` is a command that ends the execution of the procedure in which it appears; the procedure returns the argument of `output` to another procedure, as in the following example.

```
? print area 2
12.56636
```

`output's` argument can be arbitrarily complicated, so we could have written this function more concisely:

```
to area :radius
  output product 3.14159 power :radius 2
end
```

On the other hand, **temporary variables** like `temp` often make debugging easier.

Sometimes it is useful to have multiple `output` commands, one in each branch of a conditional:

```
to absoluteValue :x
  ifelse lessp :x  0 [output minus :x] [output :x]
end
```

Since these `output` commands are in an alternative conditional, only one will be executed. As soon as one is executed, the function terminates without executing any subsequent instructions.

5.2 Program development

At this point, you should be able to look at complete functions and tell what they do. As you write larger functions, you might start to have more difficulty, especially with runtime and semantic errors.

To deal with increasingly complex programs, we are going to suggest a technique called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the Pythagorean theorem, the distance is:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1}$$

The first step is to consider what a `distance` function should look like in Logo. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value.

Already we can write an outline of the function:

```
to distance :x1 :y1 :x2 :y2
  output 0.0
end
```

Obviously, this version of the function doesn't compute distances; it always returns zero. But it is syntactically correct, and it will run, which means that we can test it before we make it more complicated.

To test the new function, we call it with sample values:

```
? print distance 1 2 4 6
0.0
```

We chose these values so that the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactically correct, and we can start adding lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be—in the last line we added.

A logical first step in the computation is to find the differences $x_2 - x_1$ and $y_2 - y_1$. We will store those values in temporary variables named `dx` and `dy` and print them.

```
to distance :x1 :y1 :x2 :y2
  local "dx
  local "dy
  make "dx difference :x2 :x1
  make "dy difference :y2  :y1
  (print "|dx is | :dx)
  (print "|dy is | :dy)
  output 0.0
end
```

If the function is working, the outputs should be 3 and 4. If so, we know that the function is getting the right parameters and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of `dx` and `dy`:

```
to distance :x1 :y1 :x2 :y2
  local "dx
  local "dy
  local "dsquared
  make "dx difference :x2 :x1
  make "dy difference :y2  :y1
  make "dsquared sum power :dx 2 power :dy 2
  (print "|dsquared is | :dsquared)
  output 0.0
end
```

Notice that we removed the `print` instructions we wrote in the previous step. Code like that is called **scaffolding** because it is helpful for building the program but is not part of the final product.

Again, we would run the program at this stage and check the output (which should be 25).

Finally, we can use the `sqrt` function to compute and return the result:

```
to distance :x1 :y1 :x2 :y2
  local "dx
  local "dy
  local "dsquared
  local "result
  make "dx difference :x2 :x1
  make "dy difference :y2  :y1
  make "dsquared sum power :dx 2 power :dy 2
  make "result sqrt :dsquared
  output :result
end
```

If that works correctly, you are done. Otherwise, you might want to print the value of `result` before the return statement.

When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, the incremental development process can save you a lot of debugging time.

The key aspects of the process are:

1. Start with a working program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
2. Use temporary variables to hold intermediate values so you can output and check them.
3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple instructions into compound expressions, but only if it does not make the program difficult to read.

In the preceding example, we could consolidate istructions in the following way:

```
to distance :x1 :y1 :x2 :y2
  (output sqrt
   sum (power difference :x2 :x1 2) (power difference :y2 :y1 2))
end
```

Notice that we can always use parentheses when we feel it helps clarifying complex expressions.

5.3 Predicates

Functions can return boolean values, which is often convenient for hiding complicated tests inside functions. As you should remember, we call those functions `predicates`. For example:

```
to divisiblep :x :y
  output equalp remainder :x :y  0
end
```

The name of this function is `divisiblep`. It is common to give predicates names that end with `p` or with a `?`. `divisiblep` returns either `true` or `false` to indicate whether the `x` is or is not divisible by `y`.

This session shows the new function in action:

```
? print divisiblep 6 4
false
? print divisiblep 6 3
true
```

5.4 More recursion

So far, you have only learned a small subset of Logo, but you might be interested to know that this subset is a *complete* programming language, which means that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features you have learned so far (actually, you would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all).

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. If you take a course on the Theory of Computation, you will have a chance to see the proof.

To give you an idea of what you can do with the tools you have learned so far, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

frabjuous: An adjective used to describe something that is frabjuous.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the mathematical function factorial, you might get something like this:

$$0! = 1$$
$$n! = n(n-1)!$$

This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n-1$.

So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, 3! equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a Logo program to evaluate it. The first step is to decide what the parameters are for this function. With little effort, you should conclude that `factorial` takes a single parameter:

```
to factorial :n
end
```

If the argument happens to be 0, all we have to do is return 1:

```
to factorial :n
  if equalp :n 0 [output 1]
end
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n-1$ and then multiply it by n:

```
to factorial :n
  if equalp :n 0 [output 1]
  output product :n factorial difference :n 1
end
```

The flow of execution for this program is similar to the flow of `countdown`. If we invoke the instruction `print factorial 3`:

Since 3 is not 0, we calculate the factorial of `n-1`...

> Since 2 is not 0, we calculate the factorial of `n-1`...
>
> > Since 1 is not 0, we calculate the factorial of `n-1`...
> >
> > > Since 0 *is* 0, we return 1 without making any more recursive calls.
> >
> > The return value (1) is multiplied by n, which is 1, and the result is returned.
>
> The return value (1) is multiplied by n, which is 2, and the result is returned.

The return value (2) is multiplied by n, which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

We may also describe the flow of execution as the progressive composition of a number of arithmetical expressions:

```
(product 3 (factorial 2 ))
(product 3 (product 2 (factorial 1 )))
(product 3 (product 2 (product 1 (factorial 0))))
(product 3 (product 2 (product 1 1)))
(product 3 (product 2 1))
(product 3 2)
6
```

Finally, here is what the Logo trace looks like for this sequence of function calls:

```
? print factorial 3
( factorial 3 )
 ( factorial 2 )
  ( factorial 1 )
```

```
   ( factorial 0 )
   factorial stops
  factorial stops
 factorial stops
factorial stops
6
```

5.5 Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become labyrinthine. An alternative is what we call the "leap of faith." When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the appropriate value.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `cos` or `exp`, you don't examine the implementations of those functions. You just assume that they work because the people who wrote the built-in libraries were good programmers.

The same is true when you call one of your own functions. For example, we wrote a function called `divisiblep` that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct—by testing and examining the code—we can use the function without looking at the code again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (yields the correct result) and then ask yourself, "Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?" In this case, it is clear that you can, by multiplying by n.

Of course, it's a bit strange to assume that the function works correctly when you haven't finished writing it, but that's why it's called a leap of faith!

5.6 One more example

After `factorial`, the most common example of a recursively defined mathematical function is `fibonacci`, which has the following definition:

$$\begin{aligned}
&fibonacci(0) = 0\\
&fibonacci(1) = 1\\
&fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);
\end{aligned}$$

Translated into Logo, it looks like this:

```
to fibonacci :n
  if equalp :n 0 [output 0]
  if equalp :n 1 [output 1]
  output sum fibonacci difference :n 1 fibonacci difference :n 2
end
```

If you try to follow the flow of execution here, even for fairly small values of n, your head explodes. See how complex the trace of `fibonacci 5` already is:

```
? show fibonacci 4
( fibonacci 4 )
 ( fibonacci 3 )
  ( fibonacci 2 )
   ( fibonacci 1 )
   fibonacci outputs 1
   ( fibonacci 0 )
   fibonacci outputs 0
  fibonacci outputs 1
  ( fibonacci 1 )
  fibonacci outputs 1
 fibonacci outputs 2
 ( fibonacci 2 )
  ( fibonacci 1 )
  fibonacci outputs 1
  ( fibonacci 0 )
  fibonacci outputs 0
 fibonacci outputs 1
fibonacci outputs 3
3
```

But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

5.7 Checking types

If we call `factorial` giving it 1.5 as an argument, the function never stops.

```
to factorial :n
  if equalp :n 0 [output 1]
  output product :n factorial difference :n 1
end
```

The problem is that the values of `n` *miss* the base case —`equalp :n 0`.

In the first recursive call, the value of `n` is 0.5. In the next, it is -0.5. From there, it gets smaller and smaller, but it will never be 0.

We have two choices. We can try to generalize the `factorial` function to work with floating-point numbers, or we can make `factorial` check the type of its parameter. The first option is called the gamma function and it's a little beyond the scope of this book. So we'll go for the second.

While we're at it, we also make sure the parameter is positive:

```
to factorial :n
  if not numberp :n [
     print [Factorial is only defined for numbers]
     output "false]
  if lessp :n 0 [
     print [Factorial is only defined for positive integers]
     output "false]
  if not equalp :n int :n [
     print [Factorial is only defined for integers]
     output "false]
  if equalp :n 0 [output 1]
  output product :n factorial difference :n 1
end
```

Now we have four base cases. The first catches non numeric words. The second catches negative numbers. The third catches nonintegers, using the `int` function, that returns the integer part of a number. In each case, the program prints an error message and returns `false`:

```
? show factorial "guido
Factorial is only defined for numbers
false
? show factorial 1.1
Factorial is only defined for integers
false
? show factorial -1
Factorial is only defined for positive integers
false
?
```

If we get past the three checks, then we know that n is a positive integer, and we can prove that the recursion terminates.

This program demonstrates a pattern sometimes called a **guardian**. The first three conditionals act as guardians, protecting the code that follows from values that might cause an error. The guardians make it possible to prove the correctness of the code.

5.8 Glossary

return value: The value provided as the result of a function call.

temporary variable: A variable used to store an intermediate value in a complex calculation.

incremental development: A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

scaffolding: Code that is used during program development but is not part of the final version.

guardian: A condition that checks for and handles circumstances that might cause an error.

Chapter 6

Iteration

6.1 Multiple assignment

As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
? make "bruce  5
? print :bruce
5
? make "bruce  7
? print :bruce
7
```

The two `print` commands display first `5` and then `7`, because the first time `bruce` is printed, his value is 5, and the second time, his value is 7.

Although multiple assignment is frequently helpful, you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

6.2 The `while` command

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

We have seen a program, `countdown`, that uses recursion to perform repetition, which is also called **iteration**. Because iteration is so common, Logo provides several language features to make it easier. The first feature we are going to look at is the `while` command.

Here is what `countdown` looks like with a `while` statement:

```
to countdown :n
  while [greaterp :n 0] [print :n make "n difference :n 1]
  print "Blastoff!
end
```

Since we removed the recursive call, this function is not recursive.

You can almost read the `while` command as if it were English. It means, "While `n` is greater than 0, continue displaying the value of `n` and then reducing the value of `n` by 1. When you get to 0, display the word `Blastoff!`"

More formally, here is the flow of execution for a `while` command:

1. Evaluate the first expression list —that must evaluate to `true` or `false`— called the `condition`.
2. If the condition is false, exit the `while` command and continue execution at the next instruction.
3. If the condition is true, execute each of the instructions in the second list, the `body`, and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the instructions inside the loop are never executed.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of `countdown`, we can prove that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop, so eventually we have to get to 0 if `n` initially is an interger. In other cases, it is not so easy to tell:

```
to sequence :n
   while [not equalp :n 1] [print :n
      ifelse equalp remainder :n 2 0 [
         make "n quotient :n 2][make "n sum  product :n 3 1]]
end
```

The condition for this loop is `not equalp :n 1`, so the loop will continue until `n` is `1`, which will make the condition false.

Each time through the loop, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by 2. If it is odd, the value is replaced by `n*3+1`. For example, if the starting value (the argument passed to `sequence`) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program terminates. For some particular values of n, we can prove termination. For example, if the starting value is a power of two, then the value of n will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

Particular values aside, the interesting question is whether we can prove that this program terminates for *all* values of n. So far, no one has been able to prove it *or* disprove it!

6.3 Tables

One of the things loops are good for is generating tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, “This is great! We can use the computers to generate the tables, so there will be no errors.” That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.

Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium used to perform floating-point division.

Although a log table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and their logarithms in the right column:

```
to lntable
   local "x
   make "x 1
   while [lessp :x 10] [
      type form :x 4 0
      print form ln :x 12 8
      make "x sum :x 1
      ]
end
```

The function form accepts three inputs: a number, its width and the number of digits after the decimal point. It is useful for making columns of text line up, as in the output of the previous program:

```
   1  0.00000000
   2  0.69314718
   3  1.09861229
   4  1.38629436
   5  1.60943791
   6  1.79175947
   7  1.94591015
   8  2.07944154
   9  2.19722458
```

If these values seem odd, remember that the `ln` function uses base `e`. Since powers of two are so important in computer science, we often want to find logarithms with respect to base 2. To do that, we can use the following formula:

$$\log_2 x = \frac{log_e x}{log_e 2} \tag{6.1}$$

We can change the preceding function accordingly:

```
to lntable
   local "x
   make "x 1
   while [lessp :x 10] [
      type form :x 4 0
      print form quotient ln :x ln 2 12 8
      make "x sum :x 1
      ]
end
```

yielding:

```
   1  0.00000000
   2  1.00000000
   3  1.58496250
   4  2.00000000
   5  2.32192809
   6  2.58496250
   7  2.80735492
   8  3.00000000
   9  3.16992500
```

We can see that 1, 2, 4, and 8 are powers of two because their logarithms base 2 are round numbers.

6.4 Two-dimensional tables

A two-dimensional table is a table where you read the value at the intersection of a row and a column. A multiplication table is a good example. Let's say you want to print a multiplication table for the values from 1 to 6.

A good way to start is to write a procedure that prints the multiples of 2, all on one line:

```
to printMultiples2
localmake "i 1
while [lessp :i 7] [
  (type product 2 :i tab)
  make "i sum :i 1]
print "
end
```

The first command, `localmake`, assigns the value `1` to the local variable `i`, being effectively the combination of the commands `local` and `make`.

As the loop executes, the value of `i` increases from 1 to 6. When `i` is 7, the loop terminates. Each time through the loop, it displays the value of `2*i`, followed by a tabulation, the output of the funtion `tab`.

We defined `tab` in the following way:

```
to tab
output char 9
end
```

The `char` function accepts an integer between 0 and 255 as input and outputs the character represented in the ASCII code by that number. Therefore, the `tab` function outputs the tabulation character.

In `printMultiples2`, after the loop completes, the `print` command starts a new line.

The effect of the procedure is:

```
2       4       6       8       10      12
```

So far, so good. The next step is to **generalize**.

6.5 Generalization

Generalization means taking something specific, such as printing the multiples of 2, and making it more general, such as printing the multiples of any integer.

This procedure generalizes the previous procedure:

```
to printMultiples :n
localmake "i 1
while [lessp :i 7] [
  (type product :n :i tab)
  make "i sum :i 1]
print "
end
```

To generalize, all we had to do was replace the value 2 with the parameter `n`.

If we call this procedure with the argument 2, we get the same effect as before. With the argument 3, the result is:

```
3       6       9       12      15      18
```

With the argument 4:

```
4       8       12      16      20      24
```

By now you can probably guess how to print a multiplication table—by calling `printMultiples` repeatedly with different arguments. In fact, we can define another procedure:

```
to printMultTable
localmake "i 1
while [lessp :i 7] [
  printMultiples :i
  make "i sum :i 1]
end
```

Notice how similar this loop is to the one inside `printMultiples`.

The output of this program is a multiplication table:

```
? printMultTable
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

6.6 Local variables

You might be wondering how we can use the same variable, `i`, in both `printMultiples` and `printMultTable`. Doesn't it cause problems when one of the procedures changes the value of the variable?

The answer is no, because the `i` in `printMultiples` and the `i` in `printMultTable` are *not* the same variable, because we have declared them `local` in each procedure definition.

The value of `i` in `printMultTable` goes from 1 to 6. Each time through the loop, `printMultTable` calls `printMultiples` with the current value of `i` as an argument. That value gets assigned to the parameter `n`.

Inside `printMultiples`, the value of `i` goes from 1 to 6. Changing this variable has no effect on the value of `i` in `printMultTable`.

It is common and perfectly legal to have different local variables with the same name. In particular, names like `i` and `j` are used frequently as loop variables. If you avoid using them in one procedure just because you used them somewhere else, you will probably make the program harder to read.

6.7 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the six-by-six table. You could add a parameter to `printMultTable`:

```
to printMultTable :high
localmake "i 1
while [lessp :i sum :high 1] [
  printMultiples :i
  make "i sum :i 1]
end
```

We replaced the value 6 with the parameter `high`. If we call `printMultTable` with the argument 7, it displays:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
7       14      21      28      35      42
```

This is fine, except that we probably want the table to be square—with the same number of rows and columns. To do that, we add another parameter to `printMultiples` to specify how many columns the table should have.

Just to be annoying, we call this parameter `high`, demonstrating that different functions can have parameters with the same name (just like local variables). Here's the whole program:

```
to printMultiples :n :high
localmake "i 1
while [lessp :i sum :high 1] [
  (type product :n :i tab)
  make "i sum :i 1]
print "
end

to printMultTable :high
localmake "i 1
while [lessp :i sum :high 1] [
  printMultiples :i :high
  make "i sum :i 1]
end

to tab
output char 9
end
```

Notice that when we added a new parameter, we had to change the first line of the function (the function heading), and we also had to change the place where the function is called in `printMultTable`.

As expected, this program generates a square seven-by-seven table:

```
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
```

When you generalize a function appropriately, you often get a program with capabilities you didn't plan. For example, you might notice that, because $ab = ba$, all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of `printMultTable`. Change

```
  printMultiples :i :high
```

to

```
  printMultiples :i :i
```

and you get

```
1
2       4
3       6       9
4       8       12      16
5       10      15      20      25
6       12      18      24      30      36
7       14      21      28      35      42      49
```

6.8 Glossary

multiple assignment: Making more than one assignment to the same variable during the execution of a program.

iteration: Repeated execution of a set of instructions using either a recursive function call or a loop.

loop: An instruction or group of instructions that execute repeatedly until a terminating condition is satisfied.

infinite loop: A loop in which the terminating condition is never satisfied.

loop variable: A variable used as part of the terminating condition of a loop.

encapsulate: To divide a large complex program into components (like functions) and isolate the components from each other (by using local variables, for example).

generalize: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

development plan: A process for developing a program. In this chapter, we demonstrated a style of development based on developing code to do simple, specific things and then encapsulating and generalizing.

Chapter 7

Words

7.1 A compound data type

Words are examples of **compound data types**. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful.

7.2 Selecting characters

The `item` function —seen in chapter 3— selects a single character from a word.

```
? print item 1 "banana
b
```

The expression `item 1 "banana` selects character number 1 from the word `banana`. The first letter of `banana` is `b`.

On the other hand, in recursive functions we would probably use `first` and `butfirst`, that in combination allow us to select every character of a word.

7.3 Words length

The `count` function returns the number of characters in a word:

```
? print count "banana
6
```

To get the last letter of a word, you may use the `count` function:

```
? print item count "banana "banana
a
```

On the other hand, Logo provides two primitives to get the first and last character of a word:

```
? print last "banana
a
? print first "banana
b
? print butfirst "banana
anana
? print butlast "banana
banan
```

The last two examples extract all the word's characters except for the first or the last, using the functions `butfirst` and `butlast`.

7.4 Traversal, the `foreach` loop and recursion

A lot of computations involve processing a word one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to encode a traversal is with a `while` statement:

```
to traversal :w
  localmake "n count :w
  localmake "index  1
  while [lessequalp :index :n ][
    print item :index :w
    make "index  sum :index 1
  ]
end
```

This loop traverses the word and displays each letter on a line by itself. The loop condition is `lessequalp :index count :w`, so when `index` is equal to the length of the string plus one, the expression returns false, and the body of the loop is not executed.

`lessequalp` is not a primitive predicate, but can be easily defined:

```
to lessequalp :x :y
  output or lessp :x :y equalp :x :y
end
```

Using an index to traverse a set of values is so common that Logo provides an alternative, simpler syntax—the `foreach` loop:

```
to traversal :w
  foreach :w [print ?]
end
```

Each time through the loop, a character in the word, indicated by the `?` symbol, is printed. The loop continues until no characters are left.

The `foreach` command has a number of different syntactic forms. A full explanation is premature at this point, here we only show you another useful usage:

```
? foreach "abc [print #]
1
2
3
```

The # symbol represents the position in the input word of the character currently printed.

On the other hand, we could have defined the following recursive procedure:

```
to traversal :w
  if equalp :w " [stop]
  print first :w
  traversal butfirst :w
end
```

Which solution is better is largely a matter of taste. Moreover, at a deeper level, both `while` and `foreach` are defined in terms of recursive operations, being just syntactic forms that make programs easier to understand.

7.5 Word comparison

The comparison function `equalp` works on words. To see if two words are equal:

```
? show equalp "anna "clara
false
? show equalp "uno "uno
true
```

Another comparison function is `beforep`, that compares words in alphabetical order (in ASCII collating order):

```
? show beforep "can "cat
true
? show beforep "11 "2
true
? show not beforep "cans "can
true
? show beforep "CANS "can
false
```

You should be aware, though, that Logo by default does not handle upper and lowercase letters in the same way as the ASCII standard, in which all the uppercase letters come before all the lowercase letters. To duplicate the ASCII collating sequence you should set the variable `caseignoredp` to `false`:

```
? make "caseignoredp "false
? show beforep "CANS "can
true
```

7.6 Words are immutable

Words are **immutable**, which means you can't change an existing word. The best you can do is create a new word that is a variation of the original:

```
? make "greeting  "|Hello, world!|
? make "newGreeting word "J butfirst :greeting
? print :newGreeting
Jello, world!
```

The solution here is to concatenate a new first letter onto a part of `greeting`. This operation has no effect on the original word.

7.7 Finding the position of a character

What does the following function do?

```
to findIndex :w :ch
  localmake "n count :w
  localmake "index  1
  while [lessequalp :index :n] [
    if equalp :ch item :index :w [output :index]
    make "index  sum :index 1
  ]
  output "false
end
```

In a sense, `findIndex` is the opposite of the `item` function. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character first appears. If the character is not found, the function returns `false`.

This pattern of computation is sometimes called a "eureka" traversal because as soon as we find what we are looking for, we can cry "Eureka!" and stop looking.

We can write this function using recursion, like this:

```
to findIndex :w :ch [:index 1]
  if equalp :w " [output "false]
  if equalp :ch first :w [output :index]
  output (findIndex butfirst :w :ch sum :index 1)
end
```

This function has a number of novel features. In the definition, we use an optional parameter `index` with default value `1`. As the number of default parameters of the function is 2, when we invoke it in its body, we have to enclose it in parentheses. The `index` parameter acts as a counter, incrementing each time `findIndex` is invoked.

We can examine the flow of execution of `findIndex` using the command `trace`:

```
? show findIndex "guido "i
( findIndex "guido "i )
 ( findIndex "uido "i 2 )
  ( findIndex "ido "i 3 )
  findIndex outputs 3
 findIndex outputs 3
findIndex outputs 3
3
```

Finally, `findIndex` can be defined using `foreach`:

```
to findIndex :w :ch
  foreach :w [if equalp ? :ch [output #]
 output "false
end
```

7.8 Looping and counting

The following function is a variation of `findIndex` and counts the number of times a character appears in a word:

```
to countChar :w :ch
  localmake "n 0
  foreach :w [if equalp ? :ch [make "n sum :n 1]]
  output :n
end
```

This program demonstrates another pattern of computation called a **counter**. The variable `n` is initialized to 0 and then incremented each time `ch` is found. When the loop exits, `n` contains the result—the total number of characters `ch`.

A counter can be easily expressed as a recursive function:

```
to countChar :w :ch [:n 0]
  if equalp :w " [output :n]
  if equalp first :w :ch [output (countChar butfirst :w :ch sum :n 1)]
  output (countChar butfirst :w :ch :n)
end
```

7.9 Other functions

In this section we illustrate a few other functions useful to assemble and dissect words.

To convert characters from upper to lower case and viceversa we can use `lowercase` and `uppercase`; to reverse them `reverse`.

```
? print lowercase "|Hello, World!|
hello, world!
? print uppercase "|Hello, World!|
HELLO, WORLD!
? print reverse "|Hello, World!|
!dlroW ,olleH
```

`memberp` is a predicate that tests if a character is part (a member) of a word; `member` is a function that outputs the portion of a word from the first instance of the character for which `memberp` would be true to the end of the word:

```
? print memberp "e "Michele
true
? print member "e "Michele
ele
? print memberp "1 "234199
true
? print member "1 "234199
199
```

`substringp` is a predicate that outputs `true` if its first argument is a substring of its second argument.

```
? print substringp "ea "|Logo is easy!|
true
? print substringp "ch "MicheleMichele
true
```

With `substringp` we can easily write a function —let's call it `findString`— that finds the position of the first substring —from left to right— inside a word:

```
? show findString "LogoLogoLogo "oL
[4 5]
? make "caseignoredp "false
? show findString "LogoLogoLogo "ol
false
```

A basic implementation of `findString` is the following:

```
to findString :w :st
if not substringp :st :w [output "false]
localmake "rightindex findr :w :st
output sentence difference sum :rightindex 1 count :st  :rightindex
end

to findr :w :st :pos
if substringp :st :w [output (findr butlast :w :st difference :pos 1)]
output :pos
end
```

`findString` checks if the string `st` is in fact a part of the word `w`. If so, the procedure calculates the position of the end of the first occurrence of `st` inside `w`. It then outputs a list composed of two numeric elements: the position of the beginning of the first occurrence of `st` inside `w` (the left index); the position of the end of the first occurrence of `st` inside `w` (the right index).

`findr` outputs the right index: by now it should be easy to follow its flow of execution.

7.10 Rot13

Rot13 is a simple substitution cipher, that is an algorithm to translate each letter in a word into another letter, forming a coded message. Rot13 is used for `Usenet` news, not really to prevent anyone from reading a message but to require a deliberate choice to read what someone may find offensive.

```
? print rot13 "|This is a sequence of characters|
Guvf vf n frdhrapr bs punenpgref
? print rot13 "|Guvf vf n frdhrapr bs punenpgref|
This is a sequence of characters
```

As you can see, we can use the same algorithm to code and decode the message. Specifically, every letter in the message is rotated to the 13th next letter in the alphabet. Decrypting turns out to be the same function —rotating to the 13th next letter. For example, `e` becomes `r` and `r` rotates forward to `e`.

Our two implementations of Rot13 allows for uppercase and lowercase characters.

The first follows the usual recursive pattern:

```
to rot13 :w
  localmake "caseignoredp "false
  output rot13word :w
end

to rot13word :w
  if equalp :w " [output "]
  output word rot13char first :w  rot13word butfirst :w
end

to rot13char :c [
  :cl "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz][
  :co "NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm]
  if emptyp :cl [output :c]
  if equalp :c first :cl [output first :co]
  output (rot13char :c butfirst :cl butfirst :co)
end
```

The function `rot13` just sets `caseignoredp` locally to `false` and outputs the value returned by `rot13word`.

The function `rot13word` recursively combines the rotated characters using the `word` function.

The real job is done by `rot13char` that accepts a character as input. `rot13char` defines two optional parameters: `cl` is a word containing 52 distinct uppercase and lowercase characters, `co` is a word containing the coded characters in the same order. If the input character is equal to `cl`'s first character, `rot13char` outputs the first coded character. If the input character doesn't match `cl`'s last character, the procedure outputs its input. Otherwise `rot13char` is called again with one character less in `cl` and `co`.

It may be interesting to compare this solution with the following approach:

```
to rot13 :w
 output rot13word :w
end

to rot13word :w
  if equalp :w " [output "]
  output word rot13char first :w rot13word butfirst :w
end

to rot13char :c
  if and greaterp ascii :c 64 lessp ascii :c 91 [
  output char sum remainder difference ascii :c 52 26  65]
  if and greaterp ascii :c 96 lessp ascii :c 123 [
  output char sum remainder difference ascii :c 84 26 97]
  output :c
end
```

The new version of `rot13` simply outpus the value returned by `rot13word`.

The function `rot13word`, as before, recursively combines the rotated characters using the `word` function.

The real job is still done by `rot13char` that accepts a character as input.

To understand its working, we need to learn about two new functions.

`ascii` outputs an integer, between 0 and 255, that represents the input character in the ASCII code. Conversely, `char` outputs a character corresponding to the ASCII code in the input.

A few examples will clarify these definitions:

```
? print char 97
a
? print char 122
z
? print ascii "a
97
? print ascii "z
122
```

In `rot13char` first we test if a character is uppercase —ASCII code between 65 and 90— or lowercase —ASCII code between 97 to 122. We then apply an appropriate rotation formula to the ASCII code. Finally `rot13char` outputs the character corresponding to this transformed ASCII code.

The two formulas look formidable but an example should convince you that they just apply a sequence of arithmetic operations that rotate a given character to the 13th next letter in the alphabet.

Consider the transformation of character `c` into `p` and viceversa.

From `99` —the value of the ASCII code of the letter `c`— we subtract `84`, obtaining `15`. We then calculate the remainder of the division of `15` by `26`, equal to `15`. Finally we add this value to `97`, the ASCII code for `a`, getting `112`, the ASCII code for `p`. What you should notice is that this sequence of operation is equivalent to adding `13` to the ASCII code value of `c`.

Likewise, from the ASCII code of `p` we subtract `84`, obtaining `28`. We then add `2` —the remainder of the division of `28` by `26`— to `97` getting `99`, the value of the ASCII code of the letter `c`.

What can we say about these two implementations of Rot13?

The first is simpler but unfortunately much slower than the second. On a slow Pentium, we can encode an 8000 characters word in about 13 seconds using the second solution, while the first takes about 45 seconds.

7.11 Glossary

compound data type: A data type in which the values are made up of components, or elements, that are themselves values.

traverse: To iterate through the elements of an ordered collection of elements, performing a similar operation on each.

index: A variable or value used to select a member of an ordered collection of elements, such as a character from a word.

mutable: A compound data types whose elements can be assigned new values.

counter: A variable used to count something, usually initialized to zero and then incremented.

increment: To increase the value of a variable by one.

decrement: To decrease the value of a variable by one.

Chapter 8

Lists

A **list** is an ordered collection of elements, where each element is identified by an index. Lists are similar to words, which are ordered collections of characters, except that the elements of a list can be words, arrays or other lists. Lists and words—and other things that behave like ordered collections of elements—are called **sequences**.

8.1 List values

There are several ways to create a new list; the simplest is to enclose the elements in square brackets (`[` and `]`):

```
[10 20 30 40]
[spam bungee swallow]
```

The first example is a list of four integers. The second is a list of three words. The elements of a list don't have to be the same type.

Keep in mind that square brackets serve two purposes at once: they delimit a list while quoting it, so that Logo's evaluator interprets the list as representing itself, without invoking the procedures it names.

The following list contains a word, two numbers, and another list:

```
[hello 2.0 5 [10 20]]
```

A list within another list is said to be **nested**.

Lists that contain consecutive numbers are common, so Logo provides a simple way to create them:

```
? show iseq 3 7
[3 4 5 6 7]
? show rseq 3 7 9
```

```
[3 3.5 4 4.5 5 5.5 6 6.5 7]
? show rseq 2 30 15
[2 4 6 8 10 12 14 16 18 20 22 24 26 28 30]
```

The `iseq` function takes two arguments and returns a list containing all the integers from the first to the second argument.

The `rseq` function outputs a list of equally spaced rational numbers. Their number is equal to its third argument.

Finally, there is a special list that contains no elements. It is called the empty list, and it is denoted `[]`.

With all these ways to create lists, it would be disappointing if we couldn't assign list values to variables or pass lists as parameters to functions. We can.

```
? make "vocabulary  [ameliorate castigate defenestrate]
? make "numbers  [2 3 5 7 13 17]
? make "empty  []
? (show :vocabulary :numbers :empty)
[ameliorate castigate defenestrate] [2 3 5 7 13 17] []
```

8.2 Accessing elements

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a word —`first`, `butfirst`, `last`, `butlast` and `item`. Remember that the indices start at 1:

```
? show first :numbers
2
? show last :numbers
17
? show butfirst :numbers
[3 5 7 13 17]
? show butlast :vocabulary
[ameliorate castigate]
? show item 3 :numbers
5
```

If you try to read an element that does not exist, you get a runtime error:

```
? show item 7 :numbers
item doesn't like 7 as input
```

8.3 List length

The function `count` returns the length of a list. It is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list.

```
to traversal :w
  localmake "index  1
  localmake "n count :w
  while [lessequalp :index :n][
    print item :index :w
    make "index  sum :index 1
  ]
end

to lessequalp :x :y
output or lessp :x :y equalp :x :y
end
```

This loop traverses the list and displays each element on a line by itself. The loop condition is `lessequalp :index count :w`, so when `index` is equal to the length of the list plus one, the expression returns false, and the body of the loop is not executed.

As an example, we can call:

```
? traversal [Guido Michele Elsa Mario Gloria]
Guido
Michele
Elsa
Mario
Gloria
```

Although a list can contain another list, the nested list still counts as a single element. Therefore, the length of this list is four:

```
[spam! 1 [Brie Roquefort] [1 2 3]]
```

8.4 List membership

`memberp` is a boolean operator that tests membership in a sequence. We used it in the preceding chapter with words, but it also works with lists and other sequences:

```
? make "a [guido elsa [1 2 3] 32]
? show memberp "guido :a
true
? show memberp 33 :a
false
```

Since `guido` is a member of the `a` list, the `memberp` predicate returns true. Since `33` is not in the list, `memberp` returns false.

We can use the `not` in combination with `memberp` to test whether an element is not a member of a list:

```
? show not memberp 33 :a
true
```

member is a function that outputs the portion of a list from the first instance of the member for which memberp would be true to the end of the list:

```
? make "a [anna guido giovanni guido silvio]
? show memberp "guido :a
true
? show member "guido :a
[guido giovanni guido silvio]
```

Finally, to reverse the order of the elements in a list (or in a word), we can use reverse.

```
? show reverse :a
[silvio guido giovanni guido anna]
```

8.5 Lists and foreach loops

The foreach loop works with lists too. The following is an example of its use:

```
to traversal :w
  foreach :w [print ?]
end
```

Each time through the loop, an element in the list, indicated by the ? symbol, is printed. The loop continues until no elements are left. For example:

```
? traversal :a
guido
elsa
1 2 3
32
```

As we have seen in the preceding chapter, foreach can be used with the # symbol, which represents the position in the input list of the element currently printed.

```
? foreach :a [print #]
1
2
3
4
```

Any list expression can be used in a foreach loop:

```
? foreach iseq 1 20 [if equalp remainder ? 2 0 [print ?]]
2
4
6
8
10
12
14
```

```
16
18
20
? foreach [banana apple orange] [print (word "|I like to eat | ? "s)]
I like to eat bananas
I like to eat apples
I like to eat oranges
```

The first example prints all the even numbers between one and twenty. The second example expresses enthusiasm for various fruits.

8.6 Sentence

`sentence` allows us not only to combine words in a sentence but also the elements of two or more lists. For example:

```
? show (sentence "This "is "a "sentence)
[This is a sentence]
? show sentence [This is a list] [This too]
[This is a list This too]
? show sentence [This is [a nested] list] [This [too] !]
[This is [a nested] list This [too] !]
```

In the first example we combine words, in the second flat lists —that is lists whose elements are words— while in the third one two nested lists are combined.

`sentence` is mostly used as a combiner in recursive procedures. In the preceding chapter we showed you how to encode a word with Rot13. Here we'll extend the program to lists, as in the following example:

```
? show rot13 [Logo lists are useful]
[Ybtb yvfgf ner hfrshy]
? show rot13 [Ybtb yvfgf ner hfrshy]
[Logo lists are useful]
```

The extension is really very simple, as we can draw on the previous developed procedures. We only have to modify the top level procedure to process a list instead of a word and create `rot13list`:

```
to rot13 :l
 output rot13list :l
end

to rot13list :l
  if equalp :l [] [output []]
  output sentence rot13word first :l rot13list butfirst :l
end

to rot13word :w
  if equalp :w " [output "]
  output word rot13char first :w rot13word butfirst :w
```

```
end

to rot13char :c
  if and greaterp ascii :c 64 lessp ascii :c 91 [
  output char sum remainder difference ascii :c 52 26  65]
  if and greaterp ascii :c 96 lessp ascii :c 123 [
  output char sum remainder difference ascii :c 84 26 97]
  output :c
end
```

To understand `rot13list` we have to remember that `rot13word` outputs coded words, that are assembled recursively by `rot13list` using `sentence` in combination with `first` and `butfirst`. The flow of execution of `rot13list` is the following:

```
? show rot13list [Logo lists are useful]
( rot13list [Logo lists are useful] )
 ( rot13list [lists are useful] )
  ( rot13list [are useful] )
   ( rot13list [useful] )
    ( rot13list [] )
    rot13list outputs []
   rot13list outputs [hfrshy]
  rot13list outputs [ner hfrshy]
 rot13list outputs [yvfgf ner hfrshy]
rot13list outputs [Ybtb yvfgf ner hfrshy]
[Ybtb yvfgf ner hfrshy]
```

8.7 An extended example

Most computer programs do the same thing every time they execute, so they are said to be **deterministic**. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to generate random numbers and use them to determine the outcome of the program. Logo provides a built-in function that generates **pseudorandom** numbers, which are not truly random in the mathematical sense, but for our purposes they will do.

`random` returns an integer between 0 and its argument —called the upperbound—, which must be a whole number. Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

```
? foreach iseq 1 8 [print random 10]
4
8
2
```

```
5
0
8
4
2
```

8.7.1 List of random integers

The first step is to generate a list of random values. `randomList` takes two integer parameters — the length of the list we are creating and the upperbound in the random function. The function returns a list of random numbers.

`randomList` can be implemented in many ways: we'll present a recursive approach, one that uses `foreach` and another based upon `repeat`.

```
to randomList :n :u
  if equalp :n 0 [output []]
  output sentence random :u randomList difference :n 1 :u
end
```

Given the list length —`n`— and the upperbound —`u`— the procedure `randomList` keeps on calling itself decrementing `n`. When `n=0` it then returns a list of `n` random numbers between `0` and the upperbound `u` (excluded). It should be at this point easy to follow the flow of execution:

```
? show randomlist 5 10
( randomlist 5 10 )
 ( randomList 4 10 )
  ( randomList 3 10 )
   ( randomList 2 10 )
    ( randomList 1 10 )
     ( randomList 0 10 )
     randomList outputs []
    randomList outputs [0]
   randomList outputs [7 0]
  randomList outputs [0 7 0]
 randomList outputs [5 0 7 0]
randomlist outputs [0 5 0 7 0]
[0 5 0 7 0]
```

Our second implementation uses `foreach` in combination with `iseq` and a new command named `queue`.

```
to randomList :n :u
  localmake "list []
  foreach iseq 1  :n [queue "list random :u]
  output :list
end
```

`queue` adds an element to a list that is the value of a variable. The variable's initial value should be the empty list. New members are added at the end of the list. An example should clarify this definition:

```
? make "a []
? queue "a 1
? show :a
[1]
? queue "a 2
? show :a
[1 2]
```

In `randomList` we create a local variable `list` with initial value the empty list. Then for each number between `1` and `n` we add a random number to the beginning of the list. Finally the procedure outputs `list`.

Our third implementation is a simplification of the second. Instead of using `foreach iseq 1 :n` we can use `repeat :n`.

```
to randomList :n :u
 localmake "list []
 repeat :n [queue "list random :u]
 output :list
end
```

`repeat` repeatedly executes its second argument a number of times equal to its first argument.

As an aside, we should mention that `repcount` is a function that outputs the repetition count of `repeat`, as in the following example:

```
? repeat 5 [print repcount]
1
2
3
4
5
```

The integers generated by `random` are supposed to be distributed uniformly, which means that every value is equally likely. In a random list of, say, integers between 0 and 9, there should be approximately the same number of 0's, of 1's and so forth.

We can test this theory by writing a program to count the number of distinct values generated.

8.7.2 Counting

A good approach to problems like this is to look for subproblems that fit a computational pattern you have seen before.

In this case, we want to traverse a list of numbers and count the number of times an element is equal to a given integer. In our example we will consider a 250 element list of random integers between 0 and 9.

```
? show randomList 250 10
[6 5 7 4 4 4 9 8 8 5 5 8 0 4 7 3 7 3 4 6 1 6 3 8 2 4 4 2 7 9 9
 2 1 1 2 4 6 4 0 9 4 1 1 7 7 2 7 8 1 8 3 1 2 9 4 7 5 5 3 5 7 6
 9 5 5 6 1 6 0 9 6 5 3 6 8 9 9 6 3 0 4 2 7 4 3 6 4 0 8 5 3 0 1
 3 2 8 4 2 8 2 4 0 7 4 7 1 3 8 0 9 3 0 8 8 8 3 8 8 5 6 2 0 8 2
 7 8 0 9 8 6 5 9 0 4 1 7 9 2 5 0 1 3 9 5 0 0 6 2 4 5 6 0 0 8 0
 6 0 2 4 8 3 1 8 3 4 7 2 9 1 0 8 5 3 7 7 7 6 1 4 4 2 6 8 4 8 7
 1 1 7 1 0 0 4 9 9 5 6 2 9 3 7 9 6 3 1 4 1 2 8 6 4 8 8 0 1 6 8
 8 1 5 7 8 8 9 3 7 4 0 3 8 7 1 7 9 2 9 6 4 6 0 8 1 5 9 0 3 5 1
 1 1]
```

In the first place we'll count the occurrence of one number, say 1. Our first program is an adaption of the now well known traversal pattern.

```
to countOne :l
  if equalp :l [] [output 0]
  if equalp first :l 1 [output sum 1 countOne butfirst :l]
  output countOne butfirst :l
end
```

As usual the procedure traverses the list one element at a time. If the element's value is one, it adds one to a running sum, otherwise nothing is done. The procedure outputs the value of the sum when there are no more elements in the list.

`countOne` can be easily generalized to `countInteger`, a function that counts the number of an arbitrary integer in a list.

```
to countInteger :l :i
  if equalp :l [] [output 0]
  if equalp first :l :i [output sum 1 countInteger butfirst :l :i]
  output countInteger butfirst :l :i
end
```

Invoking this function 10 times with our list of numbers between 0 an 9, we get:

```
? repeat 10 [show sentence
    difference repcount 1 countInteger :list difference repcount 1]
[0 26]
[1 27]
[2 20]
[3 22]
[4 29]
[5 20]
[6 24]
[7 25]
[8 34]
[9 23]
```

So we have an answer: the calculated frequencies are fairly close to `25`, the expected value.

Although this solution works, it is not as efficient as it could be. Every time it calls `countInteger`, it traverses the entire list. As the number of distinct integers increases, that gets to be a lot of traversals.

Moreover, we have to know beforehand the range of distinct values —`0` to `9` in our example— while we would like to compute frequencies of every possible collection of integers.

We'll examine a better single pass solution in the next chapter, where we introduce a new data type, arrays.

8.8 Glossary

list: A collection of elements, where each element is identified by an index.

index: An integer value that indicates an element of a list.

element: One of the values in a list (or other sequence).

sequence: Any of the data types that consist of an ordered collection of elements, with each element identified by an index.

nested list: A list that is an element of another list.

list traversal: The sequential accessing of each element in a list.

Chapter 9

Arrays

9.1 Mutability and arrays

So far, you have seen two compound types: words, which are made up of characters; and lists, which are made up of elements that can be words, arrays or lists. One important difference we didn't note is that the elements of a list can be modified but the characters in a word cannot. In other words, words are **immutable** and lists are **mutable**.

We didn't examine list mutation because it can lead to unexpected effects if used without an adeguate understanding of how lists are represented in Logo.

Arrays are a compound type that allow safe mutation.

Like lists, **arrays** are ordered collections of elements. Differently from lists, arrays have a fixed length that should be declared beforehand, while lists can grow and shrink at will.

Arrays are delimited by braces; their elements can be words, numbers, other arrays or lists.

```
? print {This is an array}
{This is an array}
? print {1 {a b} 3 4}
{1 {a b} 3 4}
```

The function `array` outputs an array with a given size, with members empty lists.

```
? print array 4
{[] [] [] []}
```

Array members can be selected with `item` and changed with `setitem`.

```
? make "age {12 13 12 12 13 13 14 14 15 12 12 12 12 13}
? show :age
{12 13 12 12 13 13 14 14 15 12 12 12 12 13}
? show item 9 :age
15
? setitem 9 :age 16
? show :age
{12 13 12 12 13 13 14 14 16 12 12 12 12 13}
```

In the example, `setitem` replaces the 9th member of `age` with a new value, changing it from `15` to `16`.

Arrays are indexed starting from `1` by default, although sometimes we may want to start from `0` (or some other number, positive or negative):

```
? make "a (array 4 0)
? show item 0 :a
[]
? make "a {a b c}@0
? show item 0 :a
a
```

9.2 Frequencies revisited

Consider the list of `250` integers between 0 and 9 we used in the preceding chapter. Although the program we wrote works, it is not as general as it could be. With the new command `setitem` we can compute the frequencies more efficiently:

```
? show freqInteger [1 1 1 2 2 2 2 2 4 4 4 3 3 3 3 6]
[[1 3] [2 5] [3 4] [4 3] [5 0] [6 1]]
? show freqInteger [-1 1 1 1 0 0 0 -1 -1 -1 3 3 3 3 3 3 3]
[[-1 4] [0 3] [1 3] [2 0] [3 7]]
? show freqInteger :list
[[0 26] [1 27] [2 20] [3 22] [4 29] [5 20] [6 24] [7 25] [8 34] [9 23]]
```

`freqInteger` returns a list whose elements are lists made up of two elements: an integer value and its frequency in the input list.

Before showing you how `freqInteger` works, we have to define a new helper procedure —`maxmin`— that finds the minimum and the maximum values in a list of numbers.

```
to maxmin :l [max first :l] [min first :l]
  if emptyp :l [output sentence  :max :min]
  if lessp first :l :min [output (maxmin butfirst :l :max first :l)]
  if greaterp first :l :max [output (maxmin butfirst :l first :l :min)]
  output (maxmin butfirst :l :max :min)
end
```

`maxmin` traverses the input list, checking if the first element of the input list is less than `min` or if it is greater than `max`. In the first case it makes `min` equal to the lesser value, in the second `max` equal to the greater value. When the input list is empty, it returns `max` and `min`, that now are effectively the maximum and minimum values of the original input list.

`maxmin`'s flow of execution can be followed using `trace`, as in the following example:

```
? show maxmin [2 1 4 3]
( maxmin [2 1 4 3] )
 ( maxmin [1 4 3] 2 2 )
  ( maxmin [4 3] 2 1 )
   ( maxmin [3] 4 1 )
    ( maxmin [] 4 1 )
    maxmin outputs [4 1]
   maxmin outputs [4 1]
  maxmin outputs [4 1]
 maxmin outputs [4 1]
maxmin outputs [4 1]
[4 1]
```

We could also implement `maxmin` using `item` and a `foreach` loop.

```
to maxmin :l
  localmake "min first :l
  localmake "max first :l
  foreach butfirst :l [
     if lessp  ?  :min [make "min  ? ]
     if greaterp  ?  :max  [make "max  ? ]
  ]
  output list :max :min
end
```

We first count the input list's elements, then set `max` and `min` to the value of the input list's first element. The `foreach` loop is the core of the procedure: from 2 to `n`, if the current element is less than the current value of `min`, we set `min` equal to that element; likewise for `max`. At the end of the loop, the variables `max` and `min` hold the maximum and minimum.

We can now show `freqInteger`'s definition:

```
to freqInteger :l
  localmake "maxmin maxmin :l
  localmake "max first :maxmin
  localmake "min last :maxmin
  localmake "a (array difference sum :max 1 :min :min)
  foreach iseq :min :max [setitem ? :a 0]
  foreach :l [setitem ? :a sum item ? :a 1]
  foreach iseq :min :max [setitem ? :a list ? item ? :a]
  output arraytolist :a
end
```

The first three lines just create the local variables `max` and `min`. In the next three lines we create an array of zeros, whose length is equal to the number of integers between min and max. Then, for each element of the input list, we set the value of the array's element, whose index is the value of the list's element, to its previous value plus one. Finally we output a list whose elements are lists with two elements: an integer value and its frequency in the input list.

This procedure is still very limited, as it works only with lists of integers. In the next chapter we'll see a more general solution that uses a different composite data type, property lists.

9.3 Sorting

Many programming languages, like Perl, have builtin routines to order the elements of sequence data types like lists or arrays.

Logo doesn't give us this convenience and we have to write our own sort procedures.

9.3.1 Selection sort

Selection sort is a (relatively) simple algorithm, that unfortunately is one of the slowest.

```
? show selectionsortn [4 3 2 1 -1]
[-1 1 2 3 4]
```

To understand selection sort, it is useful to start with a related but simpler problem: find the smallest number in a list and swap it with the first element.

```
? show selectionsortnfirst [4 3 2 1 -1]
[-1 3 2 1 4]
```

In the preceding example the procedure exchanges the first and the last element, while the middle ones remain unchanged.

```
to selectionsortnfirst :l
  localmake "a listtoarray :l
  localmake "n count :a
  localmake "n1 difference :n 1
  local [i j min temp]
  make "i 1
  make "min  1
  repeat difference :n :i [
    make "j sum :i repcount
    if lessp item :j :a item :min :a [make "min  :j]
  ]
  make "temp  item :i :a
  setitem :i :a item :min :a
  setitem :min :a  :temp
  output arraytolist :a
end
```

In the first six lines the procedure sets a number of local variables. The `repeat` loop is the core of the procedure: at its end the variable `min` holds the index of the array's minimum element. The next three lines just swap the two array elements —the first element and the element that holds the minimum value— as we have seen in the example. Finally the procedure returns the modified list.

We can now apply this approach to the whole list: we just have to repeat the process for all of the list's elements but the last, using a second `repeat` loop.

```
to selectionsortn :l
  localmake "a listtoarray :l
  localmake "n count :a
  localmake "n1 difference :n 1
  local [i j min temp]
  repeat :n1  [
    make "i repcount
    make "min  :i
    repeat difference :n :i [
      make "j sum :i repcount
      if lessp item :j :a item :min :a [make "min  :j]
    ]
    make "temp  item :i :a
    setitem :i :a item :min :a
    setitem :min :a  :temp
  ]
  output arraytolist :a
end
```

In what follows we can see how the procedure swaps `a`'s elements, when we invoke `selectionsortn [4 3 2 1 -1]`.

```
{4 3 2 1 -1}
{-1 3 2 1 4}
{-1 1 3 2 4}
{-1 1 2 3 4}
```

9.3.2 Sorting numbers and words

In the preceding example we sorted a list of numbers. A simple change in the comparison predicate allows us to extend the procedure to alphanumeric lists.

```
? show selectionsort [19 3 2 1]
[1 2 3 19]
? show (selectionsort [19 3 2 1] "c)
[1 19 2 3]
? show (selectionsort [guido elsa michele] "c)
[elsa guido michele]
```

In the first example we order a list numerically, while in the second the same list is sorted following the ASCII conventions. In the third example we sort a list of words in ASCII collating order.

```
to selectionsort :l [t "n]
  output ifelse equalp :t "n [selectionsortn :l][selectionsortc :l]
end

to selectionsortn :l
  localmake "a listtoarray :l
  localmake "n count :a
  localmake "n1 difference :n 1
  local [i j min temp]
  repeat :n1  [
    make "i repcount
    make "min  :i
    repeat difference :n :i [
      make "j sum :i repcount
      if lessp item :j :a item :min :a [make "min  :j]
    ]
    make "temp  item :i :a
    setitem :i :a item :min :a
    setitem :min :a  :temp
  ]
  output arraytolist :a
end

to selectionsortc :l
  localmake "a listtoarray :l
  localmake "n count :a
  localmake "n1 difference :n 1
  local [i j min temp]
  repeat :n1  [
    make "i repcount
    make "min  :i
    repeat difference :n :i [
      make "j sum :i repcount
      if beforep item :j :a item :min :a [make "min  :j]
    ]
    make "temp  item :i :a
    setitem :i :a item :min :a
    setitem :min :a  :temp
  ]
  output arraytolist :a
end
```

As you can see, `selectionsortc` differs from `selectionsortn` only in the comparison predicate: `beforep` instead of `lessp`.

`selectionsort` just calls one of the two procedures depending on the value of `t`. The use of `ifelse` is new, though. This procedure can be used in fact as a command or as a function, as in the preceding example.

9.4 Glossary

array: an ordered collection of elements, where each element is identified by an index.

immutable type: A type in which the elements cannot be modified. Assignments to elements of immutable types cause an error.

mutable type: A data type in which the elements can be modified. All mutable types are compound types. Lists and arrays are mutable data types; words are not.

deterministic: A program that does the same thing each time it is called.

pseudorandom: A sequence of numbers that appear to be random but that are actually the result of a deterministic computation.

Chapter 10

Property lists

The compound types we learned about—strings, lists, and arrays—use integers as indexes. If you try to use any other type as an index, you get an error.

Property lists provide a simple yet powerful way to handle key-value pairs. They are similar to other compound types except that they can use any words as indexes. As an example, we will create a dictionary to translate English words into Italian.

One way to create a dictionary is to start with the empty property list and add elements.

```
pprop "dict "one "uno
pprop "dict "two "due
pprop "dict "three "tre
pprop "dict "four "quattro
```

The four assignments add new elements to an empty property list named `dict`. The command `pprop` accepts three arguments: two words —the name of the property list and the name of the key– and a value, that can be a word, a list or an array.

We can print a property list using `plist`:

```
? show plist "dict
[four quattro three tre two due one uno]
```

`plist` outputs a list, whose odd-numbered elements are property names while the even-numbered ones are the property values.

In a property list the order of the key-value pairs is not significant, and we should use `gprop` to retrieve a certain key's value:

```
? show gprop "dict "four
quattro
```

In the example, the key `four` of the property list named `dict` yields the value `quattro`.

`remprop` removes a key-value pair from a property list. For example, we can remove an entry from the previously defined property list:

```
? remprop "dict "three
? show plist "dict
[four quattro two due one uno]
```

We can also test if a name is associated to a non empty property list. `guido` is associated to an empty property list and so `plistp` returns false while `plist` returns the empty list, because in Logo names are by default associated to an empty property list.

```
? show plistp "dict
true
? show plistp "guido
false
? show plist "guido
[]
```

10.1 Other functions

Given these tools, it's easy to define new functions operating on property lists.

For example, `pcount` returns the number of key-value pairs in a property list:

```
to pcount :p
  output quotient count plist :p 2
end
```

Consider the following property list, that associates city names to a list containing their geographic coordinates:

```
pprop "city "Auburn [32.37  -85.29]
pprop "city "Albany [42.65  -73.75]
pprop "city "Adelphi [39.00  -76.97]
pprop "city "Aiken [33.54  -81.73]
pprop "city "Akron [41.08  -81.52]
pprop "city "Alexandria [38.80  -77.05]
pprop "city "Allentown  [40.37  -75.29]
pprop "city "Abilene [32.27  -99.44]
```

Invoking `pcount` we get:

```
? show pcount "city
8
```

`pkeys` extracts a list containing all the property list's keys:

```
to pkeys :pl
output pkeys1 plist :pl
end

to pkeys1 :l
if emptyp :l [output []]
output sentence first :l pkeys1 butfirst butfirst :l
end
```

`pkeys` outputs the value returned by `pkeys1`, which is a pretty straightforward recursive procedure applied to a list. Calling `pkeys` we get:

```
? show pkeys "city
[Auburn Albany Adelphi Aiken Akron Alexandria Allentown Abilene]
```

If we prefer to print the keys in alphabetical order we can use the previously defined `selectionsort` sorting procedure:

```
? show (selectionsort pkeys "city "c)
[Abilene Adelphi Aiken Akron Albany Alexandria Allentown Auburn]
```

If we want to print the key-value pairs in order, we can use a `foreach` loop:

```
? foreach (selectionsort pkeys "city "c) [show list ? gprop "city ?]
[Abilene [32.27 -99.44]]
[Adelphi [39.00 -76.97]]
[Aiken [33.54 -81.73]]
[Akron [41.08 -81.52]]
[Albany [42.65 -73.75]]
[Alexandria [38.80 -77.05]]
[Allentown [40.37 -75.29]]
[Auburn [32.37 -85.29]]
```

We can also easily test if a key exists, using `pkeyp`.

```
to pkeyp :key :pl
output ifelse equalp gprop :pl :key [] ["false] ["true]
end
```

For example, `Aiken` is a key of the property list `city`, while `Augusta` isn't.

```
? show pkeyp "Aiken "city
true
? show pkeyp "Augusta "city
false
```

10.2 Hints

If you played around with the `fibonacci` function from chapter five, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases very quickly. On a slow pentium, `fibonacci 10` finishes instantly, `fibonacci 20` takes about five seconds and `fibonacci 30` takes more than ten minutes.

To understand why, consider the trace of `fibonacci 20`. `fibonacci 2` is called 4181 times, `fibonacci 3` 2584 and so on. This is an inefficient solution to the problem, and it gets far worse as the argument gets bigger.

A good solution is to keep track of values that have already been computed by storing them in a property list. A previously computed value that is stored for later use is called a **hint**. Here is an implementation of `fibonacci` using hints:

```
to fibonacci :n
erpl "previous
local "newvalue
localmake "out fibonacci1 :n
erpl "previous
output :out
end

to fibonacci1 :n
  if equalp :n 0 [output 0]
  if equalp :n 1 [output 1]
  if not equalp gprop "previous :n [] [output gprop "previous :n]
  make "newvalue sum fibonacci1 difference :n 1 fibonacci1 difference :n 2
  pprop "previous :n :newvalue
  output :newvalue
end
```

The property list named `previous` keeps track of the Fibonacci numbers we already know. In the beginnig and the end, `erpl` erases the property list, because property list are always global.

Whenever `fibonacci1` is called, it checks the property list to determine if it contains the result. If it's there, the function can output it immediately without making any more recursive calls. If not, it has to compute the new value. The new value is added to the property list before the function outputs the new value.

Using this version of `fibonacci`, our machine can compute `fibonacci 73` in an eyeblink (equal to 806,515,533,049,393). But when we try to compute `fibonacci 74`, Logo gives us a wrong answer. Instead of 1,304,969,544,928,657 we get 1.30496954492866e+15.

The problem is that this number is too big to fit into a Logo integer and therefore we get an approximate answer, which in this case is not what we want.

10.3 Ebg13 ntnva

Back again to `rot13`, this time using a property list for the coded characters.

In the top level procedure we create a property list named `table`, containing 52 keys — equal to the upper and lowercase letters of the alphabet — associated to the coded characters. For example, the key `A` is linked to the value `N`.

`rot13list` and rot13word are recursive procedure that progressively combine the coded characters into words and sentences.

`rot13char` outputs a coded character if the original one is a key of the property list `table`; otherwise it outputs the original letter.

```
to rot13 :l
make "caseignoredp "false
erpl "table
pprop "table "A "N pprop "table "B "O
pprop "table "C "P pprop "table "D "Q
pprop "table "E "R pprop "table "F "S
pprop "table "G "T pprop "table "H "U
pprop "table "I "V pprop "table "J "W
pprop "table "K "X pprop "table "L "Y
pprop "table "M "Z pprop "table "N "A
pprop "table "O "B pprop "table "P "C
pprop "table "Q "D pprop "table "R "E
pprop "table "S "F pprop "table "T "G
pprop "table "U "H pprop "table "V "I
pprop "table "W "J pprop "table "X "K
pprop "table "Y "L pprop "table "Z "M
pprop "table "a "n pprop "table "b "o
pprop "table "c "p pprop "table "d "q
pprop "table "e "r pprop "table "f "s
pprop "table "g "t pprop "table "h "u
pprop "table "i "v pprop "table "j "w
pprop "table "k "x pprop "table "l "y
pprop "table "m "z pprop "table "n "a
pprop "table "o "b pprop "table "p "c
pprop "table "q "d pprop "table "r "e
pprop "table "s "f pprop "table "t "g
pprop "table "u "h pprop "table "v "i
pprop "table "w "j pprop "table "x "k
pprop "table "y "l pprop "table "z "m
localmake "rotl rot13list :l
make "caseignoredp "true
erpl "table
output :rotl
end

to rot13list :l
if equalp :l [] [output []]
output sentence rot13word first :l rot13list butfirst :l
end

to rot13word :w
if equalp :w " [output "]
output word rot13char first :w rot13word butfirst :w
end

to rot13char :c
```

```
ifelse equalp gprop "table :c [] [output :c][output gprop "table :c]
end
```

10.4 Counting words

In the preceding chapter we wrote a function that counted the number of occurrences of integers in a list. A more general version of this problem is to calculate the frequencies of possibly non-numeric words in a list.

Property lists provide an elegant way to generate a frequency distribution of words.

```
to freq :l
erpl "freq
local "f
freqiter :l
make "f plist "freq
erpl "freq
output :f
end

to freqiter :l
if equalp :l [] [stop]
(pprop "freq first :l
  ifelse equalp gprop "freq first :l [] [1] [sum gprop "freq first :l 1]
)
freqiter butfirst :l
end
```

We start with an empty property list named `freq`. In `freqiter`, for each element in the input list, we increment the current count. At the end, the property list contains pairs of letters and their frequencies. Notice the `ifelse` condition: if a property list key doesn't exist `gprop` returns the empty list: in that case we set the value to `0`, otherwise we add one to the previous value.

Consider, as an example, a list whose elements are `m` (for "male") and `f` (for "female"):

```
make "l [m m m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m m m m m m m f f f f f f f f f f f f f
 f f f f f f f f f f f f f f f f f f f f f f f f f f m m
 m m m m m m m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m f m m m m m m m m m m m f f f f f f f
 f f f f f f m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m m m m m m f f f f f f f f f f f f f f
 f f f f f f f f f f f f f f f f f f f f f f f f f m m m
 m m m m m m m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m m m m m m m m m m m m m m m m m m m m
 m m m m m m m m m m m m m m m m m m]
```

The list's length is 462, 92 females and 370 males.

```
? show count :l
462
? show freq :l
[f 92 m 370]
```

10.5 Glossary

property list: A collection of key-value pairs that maps from keys to values. The keys can be words, the values can be words, lists or arrays.

key: A value that is used to look up an entry in a property list.

key-value pair: One of the items in a property list.

hint: Temporary storage of a precomputed value to avoid redundant computation.

overflow: A numerical result that is too large to be represented in a numerical format.

Chapter 11

Files

While a program is running, its data is in memory. When the program ends, or the computer shuts down, data in memory disappears. To store data permanently, you have to put it in a **file**. Files are usually stored on a hard drive, floppy drive, or CD-ROM.

When there are a large number of files, they are often organized into **directories** (also called "folders"). Each file is identified by a unique name, or a combination of a file name and a directory name.

By reading and writing files, programs can exchange information with each other and generate printable formats like PDF.

Working with files is a lot like working with books. To use a book, you have to open it. When you're done, you have to close it. While the book is open, you can either write in it or read from it. In either case, you know where you are in the book. Most of the time, you read the whole book in its natural order, but you can also skip around.

All of this applies to files as well. To open a file, you specify its name and indicate whether you want to read or write.

In the following example we want to write something in a newly created file.

```
? openwrite "out.txt
? setwrite "out.txt
? print [This will be printed in file out.txt]
? setwrite []
? close "out.txt
```

The `openwrite` command takes one argument, the name of the file, and opens the file for writing.

If there is no file named `out.txt`, it will be created. If there already is one, it will be replaced by the file we are writing.

To put data in the file we have to set the write stream to that file, so that `print` and friends will write on that file instead of the screen. We use `setwrite` followed by the named of the already opened file name.

In the example, `print` then writes the list on the file. We then revert the write stream to the screen Closing the file with `close` tells the system that we are done writing and makes the file available for reading.

Now we can open the file again, this time for reading, and read the contents as a list into a variable named `line`:

```
? openread "out.txt
? setread "out.txt
? make "line readlist
? setread []
? close "out.txt
? show :line
[This will be printed in file out.txt]
? show count :a
7
```

If we try to open a file that doesn't exist, we get an error:

```
? openread "out.dat
File system error: I can't open that file
```

If we use `readword` we get, not surprisingly, a word:

```
? openread "out.txt
? setread "out.txt
? make "line readword
? setread []
? close "out.txt
? show :line
This will be printed in file out.txt
? show count :line
36
```

`readchars` can be used to read a given number of characters :

```
? openread "out.txt
? setread "out.txt
? make "fivechars readchars 5
? setread []
? close "out.txt
? print :fivechars
This
? show count :fivechars
5
```

If not enough characters are left in the file, `readchars` returns the remaining characters. When we get to the end of the file, `readchars` returns an empty list:

```
? openread "out.txt
? setread "out.txt
? ignore readchars 200
? show readchars 1
[]
? setread []
? close "out.txt
```

The following function copies a file, reading and writing up to fifty characters at a time. The first argument is the name of the original file; the second is the name of the new file:

```
to copyFile :in :out
  openread :in
  setread :in
  openwrite :out
  setwrite :out
  local "text
  while [not eofp] [
    make "text readchars 50
    type :text
  ]
  close :in
  close :out
  setread []
  setwrite []
end
```

The `eofp` predicate is new. It outputs TRUE if there are no more characters to be read in the read stream file, FALSE otherwise.

11.1 Text files

A **text file** is a file that contains printable characters and whitespace, organized into lines separated by newline characters.

The `readrawline` procedure reads all the characters up to and including the next newline character, returning a word.

The following is an example of a line-processing program. `filterFile` makes a copy of `in`, omitting any lines that begin with `;`:

```
to filterFile :in :out [:c "|;|]
  openread :in
  setread :in
  openwrite :out
  setwrite :out
  local "text
  while [not eofp] [
    make "text readrawline
    if not equalp first :text :c [print :text]
  ]
  close :in
  close :out
  setread []
  setwrite []
end
```

`readlist` is a very useful command as it creates a list for each line read, but it should be used with care, as it can lead to unexpected consequences.

Consider for example a file named `list.txt` made up of three lines:

```
[1 2
3
]
```

A single `readlist`, instead 6f reading just the first line, will read the whole file, creating the following list:

```
[1 2 3]
```

The explanation is simple: when `readlist` reads a square bracket it tries to complete the list, reading until it finds a closing square bracket.

You should also notice that `readlist` (and `readword`) process backslash, vertical bar, and tilde characters in the read stream: the output list will not contain these characters but they will have had their usual effect.

11.2 Directories

When you create a new file, the new file goes in the default working directory (how you set it is system dependent). Similarly, when you open a file for reading, Logo looks for it in the default working directory.

If you want to read a file somewhere else, you have to specify the **path** to the file, which is the name of the directory (or folder) where the file is located:

```
? setprefix "/usr/share/dict
? openread "words
? setread "words
? show readword
aback
```

This example opens a file named `words` that resides in a directory named `dict`, which resides in `share`, which resides in `usr`, which resides in the top-level directory of the system, called `/`.

You cannot use `/` as part of a filename; it is reserved as a delimiter between directory and filenames.

The file `/usr/share/dict/words` contains a list of words in alphabetical order, of which the first is `aback`.

11.3 Save, Load and Dribble

`save` is a command that saves in a file the procedures and variables defined in a Logo session. Conversely, `load` loads from an external file procedures and variables saved beforehand.

```
? make "a [1 2 3]
? show contents
[[] [a] []]
? save "session.lgo
? erall
? show contents
[[] [] []]
? load "session.lgo
? show contents
[[] [a] []]
```

In the preceding example, we first define a variable named `a`, than save the whole workspace in a file named `session.lgo`, erase all contents and load `session.lgo` restoring the previous workspace.

We could instead use `dribble` followed by a file name to save all the instructions typed afterwards on the command line.

11.4 Frequencies

We have a file that records the shoe size and sex of a sample of college students. This file is made up of 2007 lines, where the first field is either `m` or `f` and the second a whole number less than 50:

```
m 38
m 38
m 39
m 40
f 37
f 37
f 38
```

Shoe sizes are recorded in European measures, where the number `41` is about equal to `8` in American units. Using `freqfile` we get the following output:

```
? show freqfile "students.txt
[f:43 1 f:42 3 f:41 13 f:40 115 f:39 227 f:38 359 f:37 327
f:36 148 f:35 35 f:34 2 m:49 1 m:47 1 m:46 18 m:45 68 m:44 123
m:43 178 m:42 212 m:41 107 m:40 53 m:39 12 m:38 4]
```

`freqfile` is an adaption of `freq`, defined in the preceding chapter; it accepts a file name as input and returns a list. The list's odd items are the distinct combinations of the codes contained in the file, divided for convenience by `:`, while the even items are their frequencies. For example, in `students.txt` there are 212 male students that wear shoes sized 42: therefore we get the key-value pair `m:42 212`.

```
to freqfile :file
; file: one or more columns divided by one or more spaces
; last line ends with a return;
; no empty lines
openread :file
setread :file
erpl "freq
local [l f row column]
freqiterfile
setread []
close :file
make "f plist "freq
erpl "freq
output :f
end

to freqiterfile
if eofp [stop]
make "l butlast list2word readlist
pprop "freq :l ifelse equalp gprop "freq :l [] [1] [sum gprop "freq :l 1]
freqiterfile
end

to list2word :l
if equalp count :l 1 [output word first :l ":]
output list2worditer :l
end

to list2worditer :l
if emptyp :l [output "]
output (word first :l ": list2worditer butfirst :l)
end
```

The top level procedure `freqfile` opens and closes the file, erases a property list named `freq` and outputs a list contaning the key-values pairs of the property list.

`freqiterfile` iteratively reads into a list the file's lines, converting them into words using `list2word`; for each line in the input file it increments the current count. In the end, the property list `freq` contains pairs of codes and their frequencies.

It's worth noticing a couple of points.

Firstly, the input routine is quite inflexible: if we want to use a different file format we should modify accordingly `freqiterfile` and/or `list2word`. Moreover, you should be aware that our format doesn't allow for empty lines or a truncated last line, that is a line not ending with a return character.

Secondly, frequencies can be calculated for one, two or more fields. For example, using the following file:

```
a 1 x
a 1 y
a 2 y
a 1 x
b 1 x
b 1 x
b 1 y
a 2 x
a 1 y
b 2 x
```

we get:

```
[b:2:x 1 a:2:x 1 b:1:y 1 b:1:x 2 a:2:y 1 a:1:y 2 a:1:x 2]
```

11.4.1 A better presentation

From the output list of `freqfile "students.txt` we can print the following table:

```
? table freqfile "students.txt

        f       m       Total
34      2       0       2
35      35      0       35
36      148     0       148
37      327     0       327
38      359     4       363
39      227     12      239
40      115     53      168
41      13      107     120
42      3       212     215
43      1       178     179
44      0       123     123
45      0       68      68
46      0       18      18
47      0       1       1
49      0       1       1
Total   1230    777     2007
```

You'll probably agree that this presentation is better than `freqfile`'s raw output.

We won't illustrate `table` here. You'll find it — together with the student's data and `htmltable`, a procedure that creates an html page containing the tabulated data — in Appendix A.

11.5 Glossary

file: A named entity, usually stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of characters.

directory: A named collection of files, also called a folder.

path: A sequence of directory names that specifies the exact location of a file.

text file: A file that contains printable characters organized into lines separated by newline characters.

Appendix A

Logo examples

A.1 Freq

A.1.1 student's data

```
m 38 m 38 m 38 m 38 m 39 m 39 m 39 m 39 m 39 m 39 m 39 m 39 m 39 m 39 m 39
m 39 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40
m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40
m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40
m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 40 m 41 m 41 m 41 m 41 m 41 m 41
m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41
m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41
m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41
m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41
m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41
m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41
m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 41 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42
m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 42 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
```

```
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43
m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 43 m 44 m 44 m 44 m 44
m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44
m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44
m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44
m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44
m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44
m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44
m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44
m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 44 m 45
m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45
m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45
m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45
m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 45
m 45 m 45 m 45 m 45 m 45 m 45 m 45 m 46 m 46 m 46 m 46 m 46 m 46 m 46 m 46
m 46 m 46 m 46 m 46 m 46 m 46 m 46 m 46 m 46 m 46 m 47 m 49 f 34 f 34 f 35
f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35
f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35 f 35
f 35 f 35 f 35 f 35 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36
f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36
f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36
f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36
f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36
f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36
f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36
f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36
f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36
f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36 f 36
f 36 f 36 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
```

```
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37
f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 37 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38
f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 38 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39 f 39
f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40
f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40
f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40
f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40
f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40
f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40
```

```
f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40
f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 40 f 41 f 41 f 41 f 41 f 41
f 41 f 41 f 41 f 41 f 41 f 41 f 41 f 41 f 42 f 42 f 42 f 43
```

A.1.2 converting student's data

```
to convert.file :infile :outfile
openread :infile
setread :infile
openwrite :outfile
setwrite :outfile
convert.file.iter
close :infile
setread []
close :outfile
setwrite []
end

to convert.file.iter
if eofp [stop]
convert.line
convert.file.iter
end

to convert.line
print.line readlist
end

to print.line :l
if emptyp :l [stop]
(print first :l first butfirst :l)
print.line butfirst butfirst :l
end
```

A.1.3 freqfile

```
to freqfile :file
; file: one or more columns divided by one or more spaces
; last line ends with a return
; no empty lines
openread :file
setread :file
erpl "freq
local [l f row column]
freqiterfile
setread []
close :file
make "f plist "freq
erpl "freq
```

```
output :f
end

to freqiterfile
if eofp [stop]
make "l butlast list2word readlist
pprop "freq :l ifelse equalp gprop "freq :l [] [1] [sum gprop "freq :l 1]
freqiterfile
end
```

A.1.4 table

```
to table :f
local [br bc rtot ctot gtot i j]
erpl "freq
makefreq :f
make "br brows  :f
make "bc bcolumns  :f
make "rtot rtotals :f
make "ctot ctotals :f
make "gtot apply "sum :ctot
type tab
foreach :bc [type se ? tab]
print "Total
foreach :br [
  make "i ?
  make "j #
  type se :i tab
  foreach :bc [
 type se ifelse emptyp gprop "freq (word ? ": :i) [0] [gprop "freq (word ? ": :i)]  tab
  ]
  print item :j :rtot
]
type se "Total tab
foreach :bc [type se item # :ctot tab]
print :gtot
erpl "freq
end

to tab
output char 9
end

to makefreq :f
if emptyp :f [stop]
pprop "freq first :f first butfirst :f
makefreq butfirst butfirst :f
end
```

```
to split :w
localmake "l []
localmake "s "
localmake "cc 0
foreach :w  [
ifelse equalp ? ":  [make "cc 1][
   ifelse equalp :cc 1 [push "l :s make "cc 0 make "s ?][make "s word :s ?]]
]
output sentence reverse :l :s
end


to bcolumns :f
if emptyp :f [output []]
output (shellsort remdup sentence first split first :f bcolumns butfirst butfirst :f "c)
end

to brows :f
if emptyp :f [output []]
output (shellsort remdup sentence first butfirst split first :f brows butfirst butfirst :f "c)
end


to list2word :l
if equalp count :l 1 [output word first :l ":]
output list2worditer :l
end

to list2worditer :l
if emptyp :l [output "]
output (word first :l ": list2worditer butfirst :l)
end



to shellsort :l [:t "n]
localmake "a (listtoarray :l 0)
localmake "n  count :l
localmake "gap int quotient count :l 2
localmake "inv  0
local [i j exchange]
output ifelse equalp :t "n [arraytolist shellsort1][arraytolist shellsort2]
end

to shellsort1
if lessp :gap 1 [output :a]
    repeat difference :n :gap  [
                make "i difference repcount 1
        make "j sum :i :gap

        if (greaterp item :i :a  item :j :a) [
            make "exchange item :j :a
            setitem :j :a item :i :a
            setitem :i :a :exchange
            make "inv  1
        ]

    ]
    ifelse greaterp :inv 0 [make "inv 0] [make "gap  int quotient :gap  2]
output shellsort1
end

to shellsort2
if lessp :gap 1 [output :a]
    repeat difference :n :gap  [
                make "i difference repcount 1
                make "j sum :i :gap

                if (beforep item :j :a  item :i :a) [
                    make "exchange item :j :a
                    setitem :j :a item :i :a
                    setitem :i :a :exchange
                    make "inv  1
                ]

    ]
    ifelse greaterp :inv 0 [make "inv 0] [make "gap  int quotient :gap  2]
output shellsort2
end

to rtotals :f
local "i
localmake "rtot []
localmake "tot 0
foreach :br [
  make "i ?
  foreach :bc [
 make "tot sum :tot ifelse emptyp gprop "freq (word ? ": :i) [0] [gprop "freq (word ? ": :i)]
  ]
  push "rtot :tot
  make "tot 0
]
output reverse :rtot
end

to ctotals :f
local "i
localmake "ctot []
localmake "tot 0
foreach :bc [
  make "i ?
  foreach :br [
make "tot sum :tot ifelse emptyp gprop "freq (word :i ": ?) [0] [gprop "freq (word :i ": ?)]
  ]
  push "ctot :tot
  make "tot 0
]
output reverse :ctot
end
```

A.1.5 htmltable

```
to htmltable :f :file
openwrite :file
setwrite :file
local [br bc rtot ctot gtot i j]
erpl "freq
makefreq :f
make "br brows  :f
make "bc bcolumns  :f
make "rtot rtotals :f
make "ctot ctotals :f
make "gtot apply "sum :ctot

print "|<html><head><title>htmltable</title></head><body>|
print "|<table border="0">|

print "|<tr>|
type "|<td></td>|
foreach :bc [type (se "<td><b> ? "</b></td>)]
print "|<td><b>Total</b></td>|
print "|</tr>|

foreach :br [
  make "i ?
  make "j #
  type (se "<tr> "<td><b> :i "</b></td>)
  foreach :bc [
type (se "<td> ifelse emptyp gprop "freq (word ? ": :i) [0] [gprop "freq (word ? ": :i)] "</td>)
  ]
  print (se "<td> item :j :rtot "</td> "</tr>)
]
type (se "<tr> "<td><b> "Total "</b></td>)

foreach :bc [type (se "<td> item # :ctot "</td>)]

print (se "<td> :gtot "</td> "</tr>)
print "|</table>|
print "|</body></html>|

setwrite []
close :file
```

```
erpl "freq
end
```

Appendix B

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft," which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

B.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The

B.2 Verbatim Copying

that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in Section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

B.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

B.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of Sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

B.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in Section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements," and any sections entitled "Dedications." You must delete all sections entitled "Endorsements."

B.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

B.11 Addendum: How to Use This License for Your Documents

Index