Preface

This, the second edition of this book, is intended to supplement Ken Rosen's *Discrete Mathematics and its Applications, Seventh Edition*, published by McGraw-Hill. It was developed with Maple Release 14, created by Waterloo Maple Inc. This is intended to be a guide as you explore concepts in discrete mathematics and to provide you with tools you can use to investigate further on your own. This text can significantly enhance a traditional course in discrete mathematics in several ways. First, it makes a plethora of examples readily available that you can interact with easily. Second, it makes the notion of algorithm, which is central in discrete mathematics, concrete by giving you the opportunity to actually implement algorithms rather than only analyze them in the abstract. Finally, and most significantly, it provides you greater freedom to make conjectures and experiment without getting bogged down in repetitive calculation.

The focus of this manual is on Maple code and does not attempt to explain discrete mathematics. It is expected that you are taking, or have taken, a course in discrete mathematics. Ideally you have access to Ken Rosen's *Discrete Mathematics and its Applications*. It is not assumed that you have any prior experience with Maple. The introductory chapter that follows this preface is designed to introduce you to Maple. Likewise, it is not assumed that you have any experience with computer programming languages. Part of the Introduction is devoted to the basic concepts and techniques of computer programming. Subsequent chapters gradually introduce increasingly sophisticated programming ideas. While this is not a textbook on computer programming, you will likely find yourself fairly comfortable with the basics of programming by the end.

With the exception of the Introduction, the structure of this book follows that of *Discrete Mathematics and its Applications*. For each section of each chapter in that text, this manual contains a corresponding section describing Maple commands and providing Maple procedures that are used to explore the mathematics topics in that section. Each chapter also contains solutions to some of the *Computer Projects* and *Computations and Explorations* exercises found at the end of the chapter of *Discrete Mathematics and its Applications*. You will also find a number of exercises at the conclusion of each chapter designed to suggest additional questions that you can explore using Maple.

This manual strikes a balance between describing existing Maple commands and creating new procedures that extend Maple's capabilities for exploring discrete mathematics. For example, Maple does not currently include the capability of calculating with pseudographs. Therefore, in Chapter 10, in addition to describing Maple's capabilities for modeling graphs, we also write procedures relating to pseudographs. Some readers may not be interested in the detailed descriptions of how new procedures and programs like these are created. However, even if you are not interested in the programming aspect, the procedures we create are still available to you to explore those topics.

Much has happened since the first edition of this text was written. Rosen's text has undergone four revisions and Maple is now in Release 15 (it was at Release 4 when the first edition was published). As a result, this manual has undergone extensive revision, and has also been substantially expanded and reorganized.

Acknowledgments
I am deeply indebted to the authors of the first edition for providing an excellent foundation on which to build. I am particularly grateful to Ken Rosen for his guidance throughout this project and for giving me the opportunity to be part of it.

Thanks go to the staff at McGraw-Hill Higher Education, in particular Bill Stenquist, Executive Editor for this project. I also thank Rose Kernan and her colleagues at RPK Editorial Services, Inc. I am also grateful to Nathan Linscheid, who checked the entire manuscript for accuracy.

I thank Darren McIntyre, Vice President of North American Sales for Maplesoft, for his support of this project.

I am grateful to Martin Erickson for his continued mentorship; to Constantin Rasinariu and Deborah Holdstein for their expert advice; and to Daniel Baack, Jason Beckfield, Elizabeth Davis-Berg, Michael Welsh, and Heather Minges Wols for their constant support and encouragement. Finally, I am always grateful to my parents for all they have done.

Daniel R. Jordan
djordan@colum.edu

Preface to the First Edition

This book is a supplement to Ken Rosen's text *Discrete Mathematics and its Applications, Third Edition*, published by McGraw-Hill. It is unique as an ancillary to a discrete mathematics text in that its entire focus is on the computational aspects of the subject. This focus has allowed us to cover extensively and comprehensively how computations in the different areas of discrete mathematics can be performed, as well as how results of these computations can be used in explorations. This book provides a new perspective and a set of tools for exploring concepts in discrete mathematics, complementing the traditional aspects of an introductory course. We hope the users of this book will enjoy working with it as much as the authors have enjoyed putting this book together.

This book was written by a team of people, including Stan Devitt, one of the principle authors of the Maple system and Eithne Murray who has developed code for certain Maple packages. Two other authors, Troy Vasiga, and James McCarron have mastered discrete mathematics and Maple through their studies at the University of Waterloo, a key center of discrete mathematics research and the birthplace of Waterloo Maple Inc.

To effectively use this book, a student should be taking, or have taken, a course in discrete mathematics. For maximum effectiveness, the text used should be Ken Rosen's *Discrete Mathematics and its Applications*, although this volume will be useful even if this is not the case. We assume that the student has access to Maple, Release 3 or later. We have included material based on Maple shareware and on Release 4 with explicit indication of where this is done. (Where to obtain Maple shareware is described in the Introduction.) We do not assume that the student has previously used Maple. In fact, working through the book can teach students Maple while they are learning discrete mathematics. Of course, the level of sophistication of students with respect to programming will determine their ability to write their own Maple routines. We make peripheral use of calculus in this book. Although all places where calculus is used can be omitted, students who have studied calculus will find this material of interest.

This volume contains a great deal of Maple code, much based on existing Maple functions. But substantial extensions to Maple can be found throughout the book; new Maple routines have been added in key places, extending the capabilities of what is currently part of Maple. An excellent example is new Maple code for displaying trees, providing functionality not currently part of the network package of Maple. All the Maple code in this book is available over the Internet; see the Introduction for details.

This volume contains an Introduction and ten chapters. The Introduction describes the philosophy and contents of the chapters and provides an introduction to the use of Maple, both for computation and for programming. This chapter is especially important to students who have not used Maple before. (More material on programming with Maple is found throughout the text, especially in Chapters 1 and 2.) Chapters 1 to 10 correspond to the respective chapters of *Discrete Mathematics and its Applications*. Each chapter contains a discussion of how to use Maple to carry out computation on the subject of that chapter. Each chapter also contains a discussion of the Computations and Explorations found at the end of the corresponding chapter of *Discrete Mathematics and its Applications*, along with a set of exercises and projects designed for further work.

Users of this book are encourage to provide feedback, either via the postal service or the Internet. We expect that students and faculty members using this book will develop material that they want to share with others. Consult the Introduction for details about how to download Maple software associated with this book and for information about how to upload your own Maple code and worksheets.

Acknowledgments
Thanks go to the staff of the College Division of McGraw-Hill for providing us with the flexibility and support to put together something new and innovative. In particular, thanks go to Jack Shira Senior Sponsoring Editor and Maggie Rogers, Senior Associate Editor, for their strong interest, enthusiasm, and frequent inquiries into the status of this volume, and to Denise Schanck, Publisher, for her overall support. Thanks also goes to the production department of McGraw-Hill for their able work.

We would also like to express thanks to the staff of Waterloo Maple Inc. for their support of this project. In particular, we would like to thank Benton Leong and Ha Quang Le for their suggestions. Furthermore, we offer our appreciation to Charlie Colbourn of the University of Waterloo for helping bring this working team together as well as for his contributions as one of the authors of Maple's networks package which is heavily used in parts of this book.

As always, one of the authors, Ken Rosen, would like to thank his management at AT&T Bell Laboratories, including Steve Nurenberg, Ashok Kuthyar, Hrair Aldermishian, and Jim Day, for providing the environment and the resources that have made this book possible. Another author, Troy Vasiga, would like to thank his wife for her encouragement and support during the preparation of this book.

Introduction

Modern mathematical computation software, such as Maple, allows us to carry out complicated computations quickly and easily. As a supplement to traditional exercises solved by hand, having computational tools available while learning discrete mathematics provides a new dimension to the learning experience. Specifically, Maple supports an *enquiry* and *experimental* approach to learning. This book is designed to connect the traditional approach to learning discrete mathematics with this experimental approach.

Using computational software, students can experiment directly with many objects that are important in discrete mathematics. These include sets, large integers, combinatorial objects, graphs, and trees. Furthermore, by using interactive computational software to do this, students can explore these examples more thoroughly, fostering a deeper understanding of concepts, applications, and problem

solving techniques.

This supplement has two main goals. The first is to help students learn how to carry out computations in discrete mathematics using Maple. The second is to be a guide and a model as students discover mathematics with the use of computational tools.

This book is intended for use by any student of discrete mathematics. No previous familiarity with Maple is required. Likewise, we do not assume any previous experience with computer programming. The fundamentals of Maple and the basic concepts of computer programming will be thoroughly explained as they are needed.

Structure of This Manual

This supplement begins with a brief introduction to Maple, its capabilities and its use. The material in this introductory chapter explains the philosophy behind working with Maple, how to use Maple to carry out computations, and its basic structure. This introduction continues by explaining the basic concepts and syntax for programming with Maple. This will provide those who are new to Maple and programming languages the background they will need in the rest of the book.

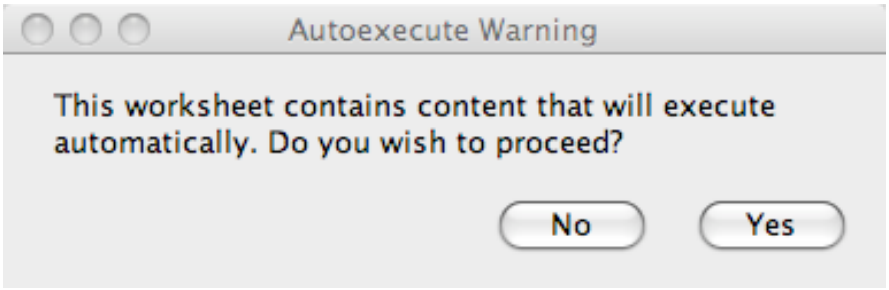
Following the introduction, the main body of this book contains thirteen chapters. Each chapter parallels a chapter of *Discrete Mathematics and Its Applications, Seventh Edition*, by Kenneth H. Rosen (henceforth referred to as the text or the textbook). Each chapter includes comprehensive coverage explaining how Maple can be used to explore the topics of the corresponding chapter of the text. This includes a discussion of relevant Maple commands, many new procedures written expressly for this book, and examples illustrating how to use Maple to explore topics in the text.

Additionally, we discuss selected *Computer Projects* and *Computations and Explorations* from the corresponding chapter of the text. We provide guidance, partial solutions, or complete solutions to these exercises. A similar philosophy governs the inclusion of these solutions as does the inclusion of answers to selected exercises in the back of most mathematics textbooks. You should attempt the problem on your own first. The solutions in this manual are intended to be referred to: after you have succeeded in solving a problem to see a (potentially) different approach; when you've stopped making progress on your own and need a slight boost to continue; or when you're trying to solve a similar problem.

Finally, each chapter concludes with a set of additional questions for you to explore. Some of these are straightforward computational exercises, while others are more accurately described as projects requiring substantial additional work, including programming.

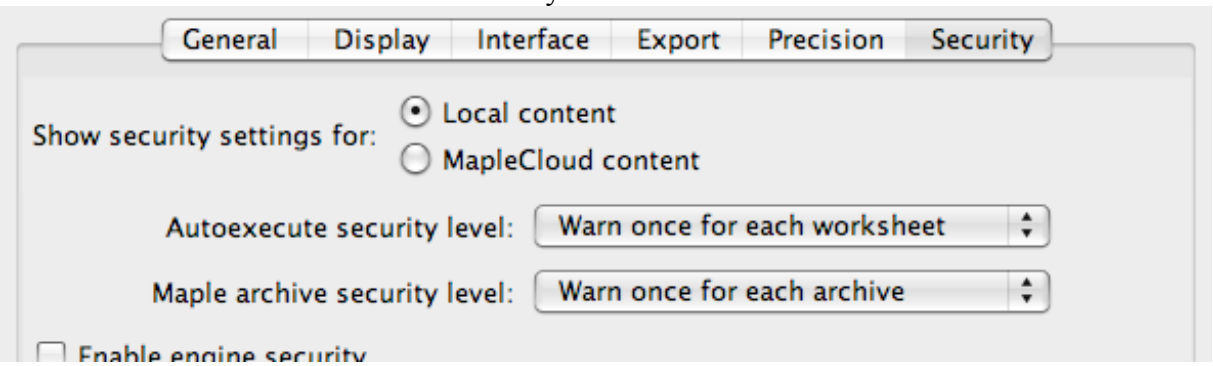
The chapters of this manual are available in two formats: as a pdf document and as a Maple Document. The pdf format contains all of the text and Maple commands and other information that you need. The Maple Document version of the chapter includes additional features, specifically active Maple code and links to Maple help pages. It is recommended that you primarily work with the Maple Document version of this manual, and use the pdf version for when you do not have access to Maple.

When you first open the Maple Document version of a chapter, a dialog box should pop up telling you that the worksheet contains content that will automatically execute (see image below). It asks whether you want to proceed. We recommend that you choose yes.



This way, the vital commands within the chapter, those that define variables and procedures, are executed for you right when you open the document. If you do not allow the automatic execution to happen, you may run into errors if you try to execute commands that require other statements to have been executed first.

If this dialog box does not pop up, check the security settings for your Maple session. In the **Tools** menu, select **Options** (or on a Mac, select **Preferences** from the **Maple** menu item). Within the options dialog, select the **Security** tab. Then, with "Local Content" selected, choose "Warn once for each worksheet" for the "Autoexecute security level."



The main benefit of the Maple document version of this book is that it is interactive. That is, you can execute the Maple commands demonstrated in the chapter. Even better, you can modify the example commands so that you can experiment right within the body of the chapter. Additionally, you have immediate access to Maple's help system. Within this manual, Maple commands appear in red. Most commands, and some terms that are not commands, are underlined indicating that they are links to a Maple help page.

This volume has been designed to help students achieve the main goals of a course in discrete mathematics. These goals, as described in the preface of the textbook, are the mastery of mathematical thinking, combinatorial analysis, discrete structures, algorithmic thinking, and applications and modeling. This supplement demonstrates how to use the interactive computational environment of Maple to enhance and accelerate the achievement of these goals.

Interactive Maple

Exploring discrete mathematics with Maple is like exploring a mathematical topic with an expert assistant at your side. As you investigate a topic you should always be asking questions. In many cases, the answer to your question can be found by experimenting. Maple, your highly trained mathematical assistant, can often carry out these directed experiments quickly and accurately, often with only a few simple instructions (commands).

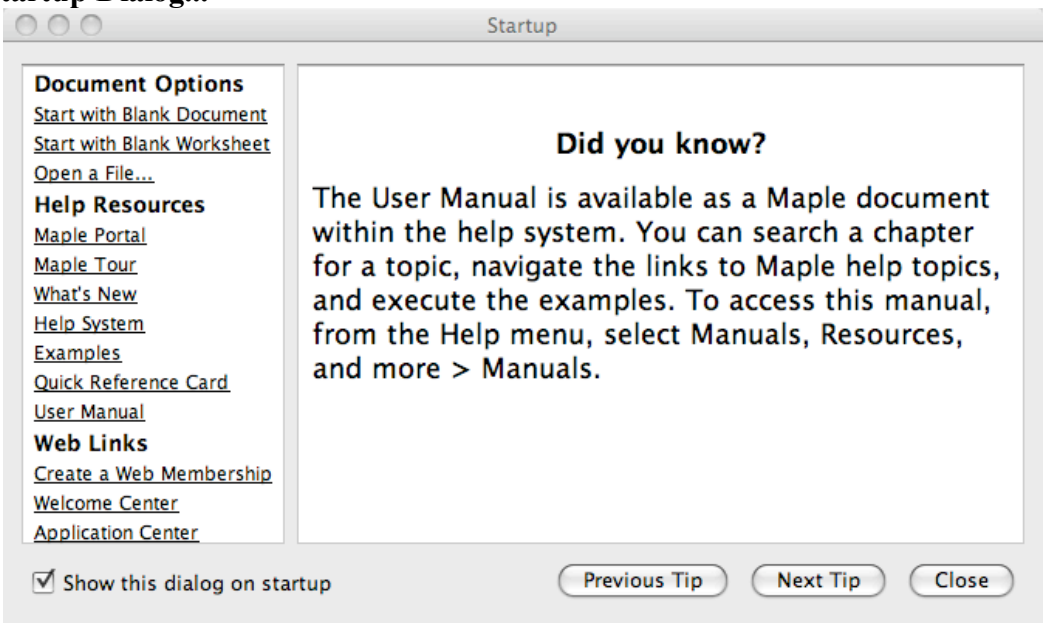
By hand, the magnitude and quantity of work required to investigate even one reasonable test case may be prohibitive. By delegating the details to Maple, your efforts can be much more focused on

choosing the right mathematical questions and on interpreting results. Moreover, with a system such as Maple, the types of objects you are investigating, and tools for manipulating them, already exist as part of the basic infrastructure provided by the system. This includes sets, lists, variables, polynomials, graphs, arbitrarily large integers, rational numbers, and most important, support for exact and fast computations.

The use of Maple is merely a means to the end of achieving the goals of a course in discrete mathematics. As with any tool, to use it effectively you must have some basic understanding of the tool and its capabilities. In this section we introduce Maple by working through a sample interactive session.

Starting Maple

A new Maple session begins when you start the Maple software. When you start Maple, you generally will see the Maple startup dialog, which will look something like the window shown below. If you do not see this window when you start Maple, go to the **Help** menu and select the option **Startup Dialog...**



The central portion of the startup dialog displays a tip. The left side of the window consists of several useful links. At the bottom of the list are three links to Maplesoft websites. Above the web sites are links to a variety of help resources.

At the top of the left hand pane are the document options. To open an existing file, perhaps a file that you created or one of the chapters of this manual, you click on "Open a File..." and a standard file selection dialog opens that allows you to select the file you want. Otherwise, if you want a blank file, you can choose between a blank Document or a Worksheet. These two options are very different from each other.

Documents versus Worksheets

This Introduction, and in fact all the chapters of this manual, were created as Maple Documents. A Maple Document can be thought of like a document in a word processing program. You can type text, change the font, insert images, and use other typical word processing tasks. But you also have a powerful mathematical engine at your fingertips.

A Maple Worksheet is more focused on executing Maple commands. When you start a Maple

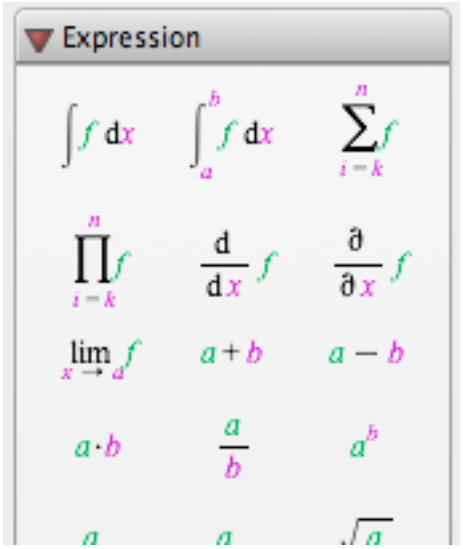
Worksheet, you immediately see a command prompt. The goal of this manual is to help you explore and learn discrete mathematics, so the execution of Maple commands is the focus. For this reason, you should generally use a Worksheet rather than a Document. So go ahead and click on "Start with Blank Worksheet" now.

Note, though, that the difference between a Worksheet and a Document is really only a matter of focus. Text blocks can be added to a Worksheet and command prompts can be added to a Document. In the toolbar at the top of the Maple window you should see the two icons shown below.

These icons should be about a third of the way along the toolbar, which begins with icons for a new file, open, save, and print. If you do not see them, click on the **View** menu and make sure that a checkmark appears next to **Toolbar**. When you're working in Maple, whether in a Worksheet or a Document, clicking on the capital T icon (or selecting **Text** from the **Insert** menu) will insert a text block immediately after your cursor. Clicking on the icon that looks like a left bracket followed by a greater than symbol (or selecting one of the **Execution Group** options in the **Insert** menu) will create a command prompt.

Maple Notation versus 2-D Math Mode

Along the side of the main window you will see palettes. Shown below is part of the Expression palette.



Palettes can be used to insert mathematical formulae and symbols within text blocks. They can also be used to make expressions in 2-D math mode. By default, for both Documents and Worksheets, math is input and output in 2-D math notation. However, this manual uses the original "Maple notation" for entering Maple commands. The 2-D notation allows you to use, for example, the integral symbol as part of an expression that you have Maple evaluate. With 2-D notation, you can also access context menus by right-clicking on an expression to instruct Maple to perform certain operations on the expression. These are nice features, but there are several reasons we chose to use the traditional Maple notation for command input.

1. It is much easier to communicate about the original Maple notation, which uses only characters found on the keyboard and does not rely on finding the appropriate element on one of the many palettes (which may be hidden or moved). This applies both to this manual and to communication between you and your instructor or fellow students. As an example, contrast

```
> f := x -> x^3 + 4*sqrt(x) - 7;
```

$$f := x \to x^3 + 4\sqrt{x} - 7 \tag{0.1}$$

with

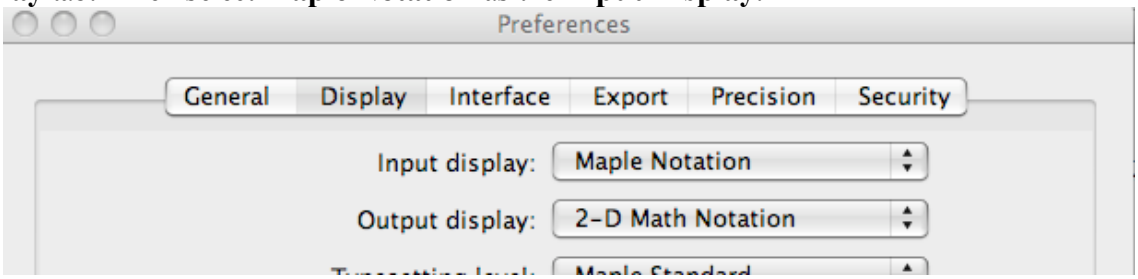
```
> f := x -> x^3 + 4*sqrt(x) - 7;
```

$$f := x \to x^3 + 4\sqrt{x} - 7 \tag{0.2}$$

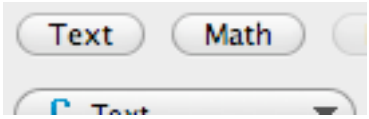
Do not worry about what this command does right now. The point is that the second of the two commands, written in the original Maple notation, is much easier to see how to replicate than the 2-D notation in the first example. The original notation is also much easier to discuss via email or a bulletin board or on a learning management system because it is nothing but text.

2. Traditional Maple notation makes arguments and options explicit, while in 2-D notation, applying options may be done in context menus which are not apparent from what is displayed.
3. Programming will be an important part of this manual and code for procedures obeys the same syntactical rules as the traditional Maple notation.

If you want to use the original Maple notation, as we recommend, you can change the default setting as follows. In the **Tools** menu (the **Maple** menu on a Mac), select **Options** and click on the **Display** tab. Then select **Maple Notation** as the **Input Display**.



You can also switch between the two styles of command input within a single document by selecting **Text** or **Math** in the **Context Bar**. If the **Context Bar** is not displayed, you can enable it from the **View** menu.



With your cursor on a Maple command line, clicking **Text** selects the traditional Maple notation while **Math** puts it in 2-D Math mode.

In Maple notation, complete commands end in semicolons (`;`). This is so that more than one command may be given on the same line and that one command may span multiple lines. (In 2-D input mode, semicolons are not required).

Executing Commands

To execute a command, make sure that the cursor is somewhere on the line containing the command and press the Enter or Return key to execute the command and display the result. It's time to execute your first Maple command. Let's start simple and add two plus three. To do this, make sure your cursor is on a command line and type `2+3;` and then press Enter or Return.

```
> 2+3;
```

$$5 \tag{0.3}$$

Here are a few more commands. Try entering them on your computer.

```
> add(i^2,i=1..10);
```

$$385 \tag{0.4}$$

```
> int((x-1)^3,x);
```

$$\frac{1}{4}(x-1)^4 \tag{0.5}$$

```
> expand(%);
```

$$\frac{1}{4}x^4 - x^3 + \frac{3}{2}x^2 - x + \frac{1}{4} \tag{0.6}$$

The percent symbol (%) is used to refer to the result of the most recently executed statement. The percent does not always refer to the value immediately above it, as commands can be executed out of order.

Also, the line numbers that automatically appear next to the results can be used in commands to refer to specific results. To use a result, it is not enough to type (0.5) in a statement. Instead, you must select **Label...** from the **Insert** menu and enter the number of the label in the dialog box that opens. You can also use the shortcut Control+L (or Command+L on a Mac) to open the dialog. If equation numbers are not appearing when you execute commands, you should turn them on. In the **Tools** menu, select **Options** (or **Preferences** from the **Maple** menu on a Mac). On the **Display** tab, make sure that the **Show Equation Labels** box is checked.

Try entering the following command.

```
> subs(x=3,(0.5));
```

$$4 \tag{0.7}$$

A First Encounter with Maple

As already indicated, working with Maple is like working with an expert mathematical assistant. This requires a subtle change in the way you think about a problem. When working on an exercise by hand, your attention is focused on the details and quite often you can lose sight of the "big picture." Maple takes care of the details for you and frees you to focus on deciding what needs to be done next. This is not to say that the details are not important, nor does it imply that you should forgo learning how to solve the problems by hand.

Much of discrete mathematics is about understanding the relationships between objects or sets of objects and using mathematical models to capture some property of these objects. Understanding these relationships often requires that you view either the objects or the associated mathematical model in different ways.

Maple allows you to manipulate the mathematical models almost casually. For example, the polynomial $(x+(x+z)y)^3$ can be entered into Maple as

```
> (x+(x+z)*y)^3;
```

$$(13x + 12z)^3 \tag{0.8}$$

The result of executing this statement is displayed immediately. In this case, Maple simply echoes the polynomial as no special computations were requested.

The power of having a computational tool like Maple is that a wide range of standard operations become immediately available. For example, you can expand, differentiate, and integrate just by telling Maple to do so. Suppose you decided that it would be useful to see the full expansion of the polynomial above. All you need to do is issue the appropriate command to Maple. In this case, the

command you would want is the **expand** command, which tells Maple to expand the polynomial.

```
> expand(%);
```

$$2197\,x^3 + 6084\,x^2\,z + 5616\,x\,z^2 + 1728\,z^3 \qquad \textbf{(0.9)}$$

(Recall that Maple uses the percent sign (%) to refer to the output from the previous command.)

Perhaps you decide it would be useful to look at this as a polynomial in the variable x, with the y's and z's placed in the coefficients of x. Then you would use the **collect** command.

```
> collect(%,x);
```

$$2197\,x^3 + 6084\,x^2\,z + 5616\,x\,z^2 + 1728\,z^3 \qquad \textbf{(0.10)}$$

To return to a factored form, simply ask Maple to **factor** the previous result.

```
> factor(%);
```

$$(13\,x + 12\,z)^3 \qquad \textbf{(0.11)}$$

We used several commands above without explanation. Rest assured that in the body of this manual we will always provide detailed explanations of the usage and syntax of new commands when we first encounter them. The purpose of the last several paragraphs was not to introduce the commands, but to illustrate how easy it is to quickly move between different representations of the same object. Having these kinds of routine tasks performed quickly and accurately means that you are freer to experiment and explore.

A second very important benefit is that the particular computations that you choose to have Maple execute are performed accurately. Thus, the results you get from your experiments are much more likely to be feedback on the model you had chosen rather than nonsense arising from simple arithmetic errors.

Finally, the sheer computational power of Maple allows you to run much more extensive experiments and many more of them. This can be important when trying to establish or identify a relationship between a mathematical model and a collection of discrete objects.

It is worth making some comments about terminology and syntax. First, in this manual, we will use the term *command* to refer to **expand**, **collect**, **factor** and the like. Maple's help documents refers to them as commands or functions, but we will avoid the use of function because of its mathematical meaning. Maple commands will always appear in the red Maple notation font and will be underlined indicating that it links to the Maple help documents. On the other hand, programs that we write will be referred to as procedures.

Second, when you use a command on one or more objects, the objects are referred to as arguments. To execute a command, you type its name followed by a pair of parentheses. Inside the parentheses you list the arguments, separated by commas.

```
> max(9,2,12,14,7,11);
```

$$14 \qquad \textbf{(0.12)}$$

Even commands that do not need any arguments require the parentheses. For example, the **time** command returns the total amount of computer time that the current Maple session has used.

```
> time();
```

$$0.077 \qquad \textbf{(0.13)}$$

The Basics

This section and the next are devoted to introducing you to the most essential Maple commands and concepts that will be used throughout this manual. Some of this material will be repeated, often in more depth, in the first few chapters when the topics arise naturally in conjunction with the content of the textbook. This section is focused on basic commands and the next focuses on programming.

Help

The most important command is the help command. Maple includes extensive documentation on all of its commands, including examples of the how the command is used. There are two primary ways to access Maple's help documents. First, you can select **Maple Help** from the **Help** menu. The Maple Help window will open and from there you can browse the table of contents or search for the topic or command you're interested in.

The more typical way to access Maple's help pages is by entering a question mark (`?`) on a command line. For example, if you need to know the command for computing the square root of a number, you could enter the following.

```
> ?square root
```

The Help window will open to the help page for the `sqrt` command, which computes the principle square root of a number or an algebraic expression. Note that you do not need to know the name of the command you're looking for help on. Following the `?` with "square root" finds the `sqrt` command. You should try to make your query as simple as possible, though. Often, taking a guess at the name of the command works well. Also note that, unlike every other Maple command, a semicolon is not needed to terminate a use of `?`. And remember that when commands are discussed in this manual, they appear in red and underlined. These are links to the Maple help documents and, provided that you are using the Maple Document version of this manual, clicking on them will open the relevant help page.

Each help page on a Maple command provides several examples of how to use that command. Open the `sqrt` help page now and take a look. The examples should appear in Maple notation. If they are in 2-D math notation, you should ensure that, under the Help system's **View** menu, the item **Display Examples with 2D math** does not have a check mark next to it. (Note: the Help system has a **View** menu separate from Maple's **View** menu, so be sure that the Help window is active in order to change this option.)

Arithmetic

Maple uses the typical notation for arithmetic. For addition and subtraction, Maples uses `+` and `-` just as you would expect. The `-` symbol is used for negation as well. Multiplication and division are performed with `*` and `/`, and `^` is used to for exponentiation.

Maple also obeys the usual order of precedence for arithmetic operators, and parentheses serve as grouping symbols. However, brackets, braces, and angle brackets all have different meanings in Maple and cannot be used as grouping symbols in arithmetic expressions. So to compute the expression $7+2\cdot\left[5-\left(\frac{2}{3}\pi\right)^2\right]$, you would enter the following, using `Pi` for π and parentheses in place of the brackets.

```
> 7+2*(5-(2/3*Pi)^2);
```

$$17-\frac{8}{9}\pi^2 \qquad \textbf{(0.14)}$$

Notice that Maple automatically computes the value of the expression, but as an exact value in terms of π. If you prefer a floating-point approximation to the result, you can use the **evalf** command (for evaluate floating-point). This command takes one argument, an expression, and evaluates it with floating-point arithmetic.

```
> evalf(%);
```

$$8.227018307 \qquad (0.15)$$

By default, **evalf** computes with ten significant figures. If you prefer more or fewer significant digits, you can specify the number of digits to use as shown below.

```
> evalf[3]((0.14));
```

$$8.23 \qquad (0.16)$$

```
> evalf[15]((0.14));
```

$$8.22701831014281 \qquad (0.17)$$

Recall that the % symbol is used to refer to the previous computation and references to specific output lines can be inserted by clicking on the **Label** item in the **Insert** menu or with the shortcut **Ctrl+L** (**Command+L** on a Mac).

Maple considers integers, fractions, and floating-point numbers to be different and it works with them differently. In expression **(0.14)**, we used only integers and the constant **Pi**. If we had used a floating-point number, Maple would have computed with floating-point arithmetic.

```
> 2/3 + 3.1/2;
```

$$2.216666667 \qquad (0.18)$$

The presence of 3.1 caused Maple to evaluate the entire expression with floating-point arithmetic. You can use this fact to cause Maple to evaluate with floating-point arithmetic even when only integers are involved. Mathematically, there is no difference between 3 and 3.0. Maple sees them as very different, however.

```
> 3/5;
```

$$\frac{3}{5} \qquad (0.19)$$

```
> 3.0/5;
```

$$0.6000000000 \qquad (0.20)$$

In fact, the trailing 0 is not required.

```
> 3./5;
```

$$0.6000000000 \qquad (0.21)$$

This discussion illustrates the concept of a *type*. A computer can be much more efficient if it knows what kinds of things it will be working with. If the computer knows that one object is going to be a floating-point number while another is going to be a string, it will allocate memory differently for the two objects, for instance.

Types also allows programs such as Maple and programming languages to make use of *operator overloading*. This means that the symbol + means one thing when applied to two integers, something else when applied to floating-point numbers, and something completely different when applied to matrices. The concept of type is what makes it possible for Maple to figure out which version of + is called for at the time. Maple recognizes over 200 different pre-defined types and users are free to create more. We'll see much more of types as we go forward.

Names, Assignment, and Equality

In mathematics, we talk about variables as symbols that stand in for something else. In Maple, this role is filled by *names*. The simplest definition of a name in Maple is that a name must begin with a letter and may be followed by letters, digits, or the underscore character. The following are all valid Maple names: **evalf**, **name**, **x**, **a15**, **B_5x_**.

Names can be used as a variable in an algebraic expression as in the following.

```
> 3*x^2 + 5*x - 7;
```

$$3x^2 + 5x - 7 \qquad (0.22)$$

Names can also be used to store particular values using the assignment operator. The assignment operator consists of a colon followed by an equals sign (**:=**). To assign a value to a name, you begin with the name, followed by the assignment operator and then the expression that you want stored in the name. And, of course, conclude the statement with a semicolon. For example, to assign the value 12 to the name **y**, you type the statement below.

```
> y := 12;
```

$$y := 12 \qquad (0.23)$$

When a value or other object has been assigned to a name, then any time that name appears in a statement, it is "resolved" to the expression stored in it.

```
> y + 5*x;
```

$$12 + 5x \qquad (0.24)$$

When Maple encounters an assignment statement, it first evaluates the right hand side of the statement and then makes the assignment. You can use this fact to modify values as follows.

```
> y := 2*y + 1;
```

$$y := 25 \qquad (0.25)$$

In the statement above, the right hand side is evaluated first, meaning that the **y** on the right is resolved to its "old" value of 12. Maple then computes $2 \cdot 12 + 1 = 25$ and assigns the value 25 to the name **y**, overwriting the value stored earlier.

In Maple parlance, **y** is referred to an assigned name, while **x** is an unassigned name. An assigned name, that is a name that has been assigned a value, can be used as an unassigned name by enclosing it in right single quotes. For example,

```
> 2*'y' + 5*x;
```

$$2y + 5x \qquad (0.26)$$

Right single quotes are used by Maple to delay evaluation of whatever expression they enclose. One important use of this is to unassign a name, as shown below.

```
> y := 'y';
```

$$y := y \qquad (0.27)$$

After this statement, **y** is no longer assigned a value.

```
> y;
```

$$y \qquad (0.28)$$

It is important to note that Maple distinguishes between the right and left single quote.

Practically any expression can be assigned to a name, not just numbers. For example, we can assign the algebraic expression $2y + 5x$ to the name **f**.

```
> f := 2*y+5*x;
```

$$f := 2y + 5x \qquad (0.29)$$

Then every time **f** appears in a statement, it is resolved to this expression.

```
> sqrt(f);
```

$$\sqrt{2\,y + 5\,x} \tag{0.30}$$

Even an equation can be assigned to a name.

```
> eqn := F = (9/5)*C + 32;
```

$$eqn := F = \frac{9}{5}\,C + 32 \tag{0.31}$$

Observe in the last example the different uses of the assignment operator and the equals sign. Some programming languages use the equals sign for assignment, but Maple reserves the equals sign for mathematical equality. The previous statement assigns the name **eqn** to the mathematical equation $F = \frac{9}{5}C + 32$. Since Maple understands this to be an equation, we can, for instance, solve it.

Given an equation and a name appearing in the equation, the **solve** command solves the equation for the given name. So we can solve for C as follows.

```
> solve(eqn,C);
```

$$-\frac{160}{9} + \frac{5}{9}\,F \tag{0.32}$$

When you execute the statement above, Maple first resolves **eqn** to the equation $F = \frac{9}{5}C + 32$. It also attempts to resolve **C**, but **C** is an unassigned name. (If **C** were not unassigned, an error would result.) Once the arguments have been evaluated, Maple applies the **solve** command to them.

Basic Types

We mentioned above that Maple recognizes many different types of objects. In this subsection we will discuss some of the most fundamental types. We won't go into all the details of what it means to be a type in Maple. Our goal in this section is to introduce you to the kinds of objects you'll be using and the use of the **type** command.

We have already seen numeric types, such as integers, fractions, and floating-point numbers. Maple has a host of names for numeric types, such as **integer**, **fraction**, **float**, **posint**, **realcons**, and **imaginary**, to name a few.

You can test whether any Maple expression is of a specific type by using the **type** command. This command requires two arguments. The first is the expression you want to test and the second is the type. For example, to see whether or not 2 is a positive integer, a fraction, and a float, you enter the following statements.

```
> type(2,posint);
```

true **(0.33)**

```
> type(2,fraction);
```

false **(0.34)**

```
> type(2,float);
```

false **(0.35)**

In addition to numeric types, Maple has a **string** type for strings of characters. You form a string by enclosing any sequence of characters within a pair of double quotes. For example, Einstein wrote,

```
> quotation := "Pure mathematics is, in its way, the poetry of
  logical ideas.";
```

quotation := "Pure mathematics is, in its way, the poetry of logical ideas." **(0.36)**

Strings may be combined with the concatenation operator, `||`, or with the command **cat**, as demonstrated below

```
> cat(quotation," - Einstein");
```

"Pure mathematics is, in its way, the poetry of logical ideas. - Einstein" **(0.37)**

```
> "abc" || "def";
```

"abcdef" **(0.38)**

You can ask Maple what the type of an object is with the **whattype** command. Since many objects satisfy the definitions of multiple types, this command returns the object's most basic type.

```
> whattype(2);
```

integer **(0.39)**

We will see why types are important in the next section when we discuss basic programming concepts.

Expression Sequences

An expression sequence, **exprseq**, or simply sequence, is the fundamental Maple data structure. An expression sequence is formed using commas to separate expressions. For example, the following assigns the expression sequence 7,8,9,10,11 to the name **S**.

```
> S := 7,8,9,10,11;
```

S := 7, 8, 9, 10, 11 **(0.40)**

To lengthen a sequence, you use a comma.

```
> S := S,12;
```

S := 7, 8, 9, 10, 11, 12 **(0.41)**

Selection

The *selection operation* is used to access members of a sequence and subsequences. The first element of the sequence can be obtained by typing the name assigned to the sequence followed by a pair of brackets enclosing the number 1.

```
> S[1];
```

7 **(0.42)**

You would use 2 to obtain the second element, 3 the third, and so on.

```
> S[2];
```

8 **(0.43)**

```
> S[3];
```

9 **(0.44)**

You may also use negative integers to count from the right. So -1 refers to the last element of the sequence, -2 to the next to last, *etc*.

```
> S[-1];
```

12 **(0.45)**

```
> S[-2];
```

11 **(0.46)**

Any expression can be placed inside the brackets provided it evaluates to an integer that is not 0 and does not exceed the bounds of the sequence (for example, 7 or higher and -7 or lower, in this case).

```
> S[2*3-2^2];
```

$$8 \tag{0.47}$$

Ranges

A **range** is a Maple type consisting of two expressions connected by two periods. A common use of a range is in conjunction with the selection operation to extract a subsequence from a sequence. For example, to extract the subsequence consisting of the third through the fifth elements of S, you use the range `3..5` within brackets.

```
> S[3..5];
```

$$9, 10, 11 \tag{0.48}$$

```
> S[-5..-3];
```

$$8, 9, 10 \tag{0.49}$$

Note that the left side of the range must be less than or equal to the right side. However, either or both sides may be omitted. If both are omitted, it is interpreted as the entire sequence.

```
> S[..];
```

$$7, 8, 9, 10, 11, 12 \tag{0.50}$$

If only one side is given, it is interpreted either as the sequence from the given location onwards,

```
> S[3..];
```

$$9, 10, 11, 12 \tag{0.51}$$

or as the sequence up to the given location.

```
> S[..5];
```

$$7, 8, 9, 10, 11 \tag{0.52}$$

The seq command and final comments on sequences

The **seq** command is often used to create sequences using a formula. Here we'll only discuss the most typical way to use **seq**, which will be discussed more thoroughly in Section 2.4 of this manual. You should first choose a name, which is referred to as the index variable. The letter `i` is a typical choice.

In the simplest form, **seq** requires two arguments. The first argument is any expression, typically one that involves the index variable in its computation. The second argument is an equation with the index variable on the left hand side and a range on the right hand side, for example `i=3..7`. The result of executing the **seq** command is the sequence obtained by evaluating the first argument at each value of the index variable determined by the range in the second argument. For example, we can obtain the squares of the integers between 5 and 11 as follows.

```
> seq(i^2,i=5..11);
```

$$25, 36, 49, 64, 81, 100, 121 \tag{0.53}$$

Expression sequences are an important Maple data structure and form the basis for several types including lists and sets, to be discussed below. However, you should be aware that sequences generally cannot be given as the argument to a command or procedure. To understand why, suppose that **ACommand** were a command that accepted only one argument. If you try to execute this command with a sequence, such as **S**, as the argument, Maple first resolves the expression sequence.

```
> ACommand(S);
```

$$ACommand(7, 8, 9, 10, 11, 12) \tag{0.54}$$

It looks to Maple like you were passing six arguments to **ACommand** instead of one. That would

generate an error if **ACommand** were actually a command. In order to pass a collection of values to a procedure or command, they must be enclosed in a list or a set.

Finally, the empty expression sequence is referred to as **NULL**. It is not displayed by Maple.

```
> NULL;
```

Lists

In Maple, a list is an ordered sequence of expressions. The name of the type in Maple is **list**. You create a list by enclosing an expression sequence, *i.e.*, values separated by commas, in brackets.

```
> L := [6,7,8,9,10,11];
```

$$L := [6, 7, 8, 9, 10, 11] \quad \textbf{(0.55)}$$

Also note that the **seq** command can be used to create a list by enclosing it in brackets. Maple computes the sequence and then forms the list based on that sequence.

```
> L2 := [seq(4*i+3,i=1..10)];
```

$$L2 := [7, 11, 15, 19, 23, 27, 31, 35, 39, 43] \quad \textbf{(0.56)}$$

At first glance, it may appear that the only difference between a list and the expression sequence that defines it is the brackets. Conceptually, the actual difference is that a list can be thought of as a single object. So, unlike a sequence, a list can be passed as an argument to a procedure or command.

Selection

Selection works essentially the same with lists as with sequences. To access a single element of the list, you follow the name of the list with a pair of brackets containing the integer indicating the position of the element. Remember that the first element is 1 and that negative numbers count from the end.

```
> L2[3];
```

$$15 \quad \textbf{(0.57)}$$

```
> L2[-2];
```

$$39 \quad \textbf{(0.58)}$$

As with sequences, you can select a range as well. The difference is that when used with a list, the result will be a list.

```
> L2[3..-2];
```

$$[15, 19, 23, 27, 31, 35, 39] \quad \textbf{(0.59)}$$

The op command

It is sometimes necessary to extract the underlying sequence from a list. This can be done with the **op** (operands) command. The most basic form of the **op** command takes one argument, which can be any expression. For lists, this will return the sequence of elements in the list.

```
> op(L2);
```

$$7, 11, 15, 19, 23, 27, 31, 35, 39, 43 \quad \textbf{(0.60)}$$

The **op** command can also accept either an integer or a range as its first argument with the list as the second. In this case, it behaves similar to selection, except **op** returns a sequence while selection returns a list.

```
> op(5,L2);
```

$$23 \quad \textbf{(0.61)}$$

```
> op(..5,L2);
```

$$7, 11, 15, 19, 23 \tag{0.62}$$

With objects other than lists, **op** has a slightly different behavior. For example, for polynomials, **op** will extract the terms.

```
> op(3*x^2-5*x+7);
```

$$3x^2, -5x, 7 \tag{0.63}$$

We will discuss other uses of **op** in later chapters as they are needed.

Related to **op** is the **nops** (number of operands) command. For lists, **nops** returns the number of elements in the list.

```
> nops(L);
```

$$6 \tag{0.64}$$

```
> nops(L2);
```

$$10 \tag{0.65}$$

We saw that extending a sequence is just a matter of using a comma to continue the sequence. Adding elements to a list is a bit more complicated. Suppose you want to add 47 to the end of **L2**. To do this, you use **op** to extract the sequence of elements from **L2**. Then add 47 to the sequence with a comma. Turn the extended sequence back into a list by surrounding it with brackets. And reassign the result to the name **L2**.

```
> L2 := [op(L2),47];
```

$$L2 := [7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47] \tag{0.66}$$

Adding to the beginning of the list is done in essentially the same way.

```
> L2 := [3,op(L2)];
```

$$L2 := [3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47] \tag{0.67}$$

If you want to insert an element in the middle of a list, say after the 5th member of **L2**, you would use **op** with ranges in order to separate the front and back ends of the existing list as follows.

```
> L2 := [op(..5,L2),21,op(6..,L2)];
```

$$L2 := [3, 7, 11, 15, 19, 21, 23, 27, 31, 35, 39, 43, 47] \tag{0.68}$$

The map command

We will discuss one last command related to lists: the **map** command. This command requires two arguments. The first argument is the name of a command or procedure. The second argument is a list. (Technically, the second argument can be any expression, but we will typically use **map** with a list as the second argument.) The result is the list obtained by applying the command to each element of the list. For example, the statement below produces the list of square roots of the given list.

```
> map(sqrt,[3,6,9]);
```

$$[\sqrt{3}, \sqrt{6}, 3] \tag{0.69}$$

Note that when **map** is applied to a name that has been assigned to a list, the named list is not modified.

```
> map(sqrt,L);
```

$$[\sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3, \sqrt{10}, \sqrt{11}] \tag{0.70}$$

```
> L;
```

$$[6, 7, 8, 9, 10, 11] \tag{0.71}$$

If you want the original list to be updated, you should reassign it.

```
> L := map(sqrt,L);
```

$$L := \left[\sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3, \sqrt{10}, \sqrt{11}\right] \quad \textbf{(0.72)}$$

```
> L;
```

$$\left[\sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3, \sqrt{10}, \sqrt{11}\right] \quad \textbf{(0.73)}$$

Note that it is typical for Maple commands to *not* modify their arguments.

For commands that require more than one argument, **map** can accept optional arguments following the list that are then passed to the command. There are also variants such as **map2** and **zip**. The use of additional arguments and these other commands will be discussed as they are needed.

Sets

In mathematics, a set is an collection of objects that is unordered and without repetition. Maple's **set** type models the mathematical notion. To form a set, you enclose a sequence in braces.

```
> {1,2,3};
```

$$\{1, 2, 3\} \quad \textbf{(0.74)}$$

```
> S := {seq(i^2,i=1..10)};
```

$$S := \{1, 4, 9, 16, 25, 36, 49, 64, 81, 100\} \quad \textbf{(0.75)}$$

Note that repeated elements in a set are automatically removed by Maple.

```
> {1,2,3,2,1};
```

$$\{1, 2, 3\} \quad \textbf{(0.76)}$$

This is a useful feature that we'll make use of quite often to avoid redundancy.

The **op** command works on sets in the same way as lists to return the sequence that underlies the set, and the **nops** command applied to a set returns the number of elements.

```
> op(S);
```

$$1, 4, 9, 16, 25, 36, 49, 64, 81, 100 \quad \textbf{(0.77)}$$

```
> nops(S);
```

$$10 \quad \textbf{(0.78)}$$

Note that, while sets are technically unordered, Maple actually imposes an order on them. This is done to improve efficiency. Rest assured that the implementation of sets and the commands related to them is done in such a way as to ensure that they behave as mathematical sets should. The import of this is that selection works with sets in the same way as lists, as does the **op** command with an integer or range as the first argument.

```
> S[3..7];
```

$$\{9, 16, 25, 36, 49\} \quad \textbf{(0.79)}$$

```
> op(7,S);
```

$$49 \quad \textbf{(0.80)}$$

Sets will be explored in much greater detail in Chapter 2.

Printing

There are three main ways to have Maple display the result of a computation. First, of course, is to execute a command that displays a result.

```
> 2+3;
```

(0.81)

```
5
```
(0.81)

On the other hand, sometimes you may wish for a result to not be displayed. In this case, you end the statement with a colon instead of a semicolon.

```
> 2+3:
```

The second way to get Maple to display information is the **print** command. This is often used within a procedure to display the results of intermediate calculations before the final result is computed and displayed. The **print** command accepts as its input a sequence of expressions and displays them on a line. For example,

```
> print(L);
```

$$[\sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3, \sqrt{10}, \sqrt{11}] \quad \textbf{(0.82)}$$

```
> print(S,L2);
```

$$\{1, 4, 9, 16, 25, 36, 49, 64, 81, 100\}, [3, 7, 11, 15, 19, 21, 23, 27, 31, 35, 39, 43, 47] \quad \textbf{(0.83)}$$

The third way to have Maple display output is **printf**. (There are three related commands that work in similar ways with slightly different purposes, but we will only discuss **printf**.)

The purpose of **printf** is to print expressions using a particular format (hence the f) that you specify. The first argument to the command is a string that details the format in which the information is to be displayed. The remaining arguments are the information to be displayed. Here is an example of using **printf** to display a number and its square with each number displayed with room for at least 5 digits.

```
> printf("The square of %5a is %5a.\n",7,7^2);
The square of     7 is    49.
```

The **%** symbol is used to indicate the beginning of the formatting specification **%5a**. The 5, which is optional, specifies that the width of that field is to be at least 5 characters. This is a useful option for displaying a table or otherwise ensuring that displayed values are aligned. Finally, the **a**, for anything, tells Maple to display the corresponding object however it normally would. Other letters can be used that are specific to different types of objects to display, such as integers and strings. The **\n** at the end of the formatting string indicates that a new line should be inserted at that point. Following the formatting string, are the two values **7** and **7^2**. Notice that the first value is put in place of the first formatting specification (the first **%5a**) and the second goes in place of the second.

The **printf** command is very flexible with a great many options. The interested reader should refer to the Maple help page for further information. In this manual, we will use **printf** rarely and we will not discuss it further here.

Programming Preliminaries

This section is intended for those readers who have little or no previous exposure to programming. We will endeavor to provide you with enough information to get you started so that you can work productively with Maple. For further information, you are encouraged to consult the Maple manuals, which will provide you with further examples of the use of Maple's programming facilities.

All programming languages provide a few basic means for the construction of algorithms. On the most basic level, a computer program is a sequence of instructions that the computer executes one after the other. Programs become more sophisticated when you start changing the flow of execution. Maple provides the same sort of mechanisms for flow control as is found in traditional

programming languages such as C. While the syntax varies from one computer language to the next, there are two primary kinds of control structures used: branching and iteration.

Branching

We will first discuss the concept of branching and its implementation in Maple. Branching is a mechanism that allows you to choose between statements based on conditions that can only be determined during a program's execution. This is also called a selection or conditional statement.

if-then

As an example, suppose that you want to display a message based on whether a particular value is positive. First we'll assign a value to the name **z**.

```
> z := 5;
```

$$z := 5 \qquad \textbf{(0.84)}$$

The following code will display the message "That's positive" if the value stored in z is greater than zero.

```
> if z > 0 then
    print("That's positive");
  end if;
```

"That's positive" **(0.85)**

First note that, in order to begin a new line within a single command prompt, you press Shift+Enter (or Shift+Return on a Mac). The line breaks and extra spaces are not required, in fact Maple ignores them, but they often make programs easier to read and understand.

Typically, the condition in an if statement depends on a value input to a program or some intermediary calculation or a value that changes during the execution of a procedure. In these examples, think about the name **z** as storing some value that varies based on some other computations. For instance, **z** could be the value of some function at a particular point. Then the value of z would depend on which point was chosen.

Let's now dissect the code above. The if statement begins with the keyword **if**. This is immediately followed by a conditional expression, that is, an expression that Maple can evaluate to true or false. Conditional expressions may include expressions that include relational operators (**<**, **<=**, **=**, **>**, **>=**, **<>**), logical operators (**and**, **or**, **not**), or procedures and commands that return logical values (**true**, **false**, or **FAIL**). We will see many examples of conditional expressions in Chapter 1.

Following the conditional expression, you must include the keyword **then**. Following the **then** keyword is a statement sequence, one or more statements that are to be executed in the event that the conditional expression is true. In our example above, the **then** keyword was followed by the statement sequence consisting of a single statement, the **print** command.

Finally, the phrase **end if;** is used to indicate the end of the conditional statement.

When you execute the statement above, Maple evaluates the conditional expression **z > 0**. Since this is a true statement (because **z** happened to be 5), Maple executes the **print** command in the statement sequence following the **then** keyword. If the conditional expression had been found to be false, then Maple would not execute the **print** command.

Below is another example, in which the conditional statement is false.

```
> if z >= 10 then
```

```
    print("That has at least two digits");
  end if;
```

Notice that in this case, nothing is displayed. Once Maple determines that the condition is false, it skips past the statement sequence following the **then** keyword.

else

Often, you will want to take one action if a condition is true and a different action if a condition is false. The **else** keyword allows you to extend a conditional statement to contain a second statement sequence to be executed if the condition is false.

```
> if z < 0 then
    print("It's negative");
  else
    print("It's not negative");
  end if;
```

"It's not negative" **(0.86)**

The **then** keyword separates the conditional expression from the statement sequence that is executed when the condition is true. The **else** keyword indicates the beginning of the statement sequence that is to be executed when the condition is not true.

In the example, Maple first tests to see if **z < 0**. Since this is false, Maple skips to the **else** clause and executes the second **print** command.

elif

You can also extend the if statement to a multiway branching structure for when there are more than two options. To do this, you use the **elif** keyword to introduce additional conditions that may be true when the initial **if** condition fails.

```
> if z >= 10 then
    print("That has at least two digits");
  elif z > 0 then
    print("It's positive");
  elif z >= 0 then
    print("It's zero");
  else
    print("It's negative");
  end if;
```

"It's positive" **(0.87)**

The statement above works as follows. First, Maple checks the condition **z >= 10**. If this were true, it would execute the first **print** statement. But since it is false, Maple moves on to the first **elif** condition, **z > 0**. This condition is true, so Maple executes the **print("It's positive.");** command.

Look at the next condition: **z >= 0**. This condition is also true, but Maple does not execute the corresponding **print** command. In an **if-elif-else** structure, once the first conditional expression is found to be true, the statement sequence following that condition is executed and any conditions that follow the first are skipped. Likewise, the **else** statement sequence is only executed in case all the conditional expressions were false. In other words, only one of the statement sequences is ever executed.

That is why we can say that the number is 0 if the test **z >= 0** is true. This condition is only checked if all the previous conditions failed. Thus, if the **z >= 0** test is evaluated true, we know

that `z > 0` was false, and hence `z` is 0.

Iteration

The previous subsection showed how to use branching in Maple to execute different blocks of code depending on whether or not a specified condition was met. In this subsection, we look at ways to repeat a block of code. Iteration is the mechanism for doing a given task repeatedly and is typically accomplished by a loop structure.

for loops

The most basic type of iteration is the for loop. The most basic kind of for loop executes a statement for each integer in a particular range. The example below computes the squares of the integers from 3 to 5.

```
> for i from 3 to 5 do
    i^2;
  end do;
```

$$9$$
$$16$$
$$25 \qquad (0.88)$$

The statement begins with the `for` keyword, indicating the type of loop. After the `for` keyword is a variable name, called the loop variable. The letter `i` is a typical choice. Then the `from` keyword is followed by the initial value of the loop variable. The `to` keyword is followed by the maximum value for the loop. After that, the `do` keyword precedes the body of the loop, which is terminated by the `end do;` phrase.

The statement(s) in between `do` and `end do;` form the body of the loop. That is, those are the statements that are executed repeatedly. When Maple executes this loop, here is what happens. First, Maple assigns the starting value, specified by the `from` clause, to the loop variable `i`. Then the statement sequence is executed and the loop variable is squared. Once the statement sequence is completed, Maple increments the loop variable by 1 and checks to see whether its new value exceeds the value specified by the `to` clause. If not, the statement sequence is executed again with the new value of the loop variable before incrementing it again. Once the loop variable exceeds the maximum value, the loop terminates.

In other words, `i` is assigned to `3`, and `3^2` is computed. Then `i` is incremented to `4` and `4^2` is computed. Then `i` is incremented to `5` and `5^2` is computed. Then `i` is incremented to `6`, which exceeds the maximum so the loop ends.

There is actually an extra step that we didn't mention. Immediately after assigning the starting value to the loop variable, the loop variable is tested against the maximum value. This means that it is possible to create loops that never execute their statement sequence.

```
> for i from 7 to 2 do
    print("Execute");
  end do;
```

Maple also includes the option for a `by` clause if you'd like to increment the loop variable by a value other than 1.

```
> for i from 3 to 11 by 2 do
    i^2;
  end do;
```

$$9$$

```
                                25
                                49
                                81
                               121                                (0.89)
```

In this example, the **by 2** clause indicates that the loop variable is incremented by 2. The value 2 is referred to as the *step*.

The **by** clause also provides a way to loop from high to low by using a negative step.

```
> for i from 10 to 5 by -1 do
    i^2;
  end do;
                               100
                                81
                                64
                                49
                                36
                                25                                (0.90)
```

while loops

Somewhat more general than a for loop is the while loop, which is another method for iteration available in Maple. In a while loop, execution continues as long as a conditional statement remains true.

```
> i := 2:
  while i < 10^7 do
    print(i);
    i := i^2 + 2;
  end do:
                                2
                                6
                               38
                             1446
                          2090918                               (0.91)
```

Note that loops, like other statements, can be ended with a colon. For loops, this suppresses the automatic printing of all the statements in the loop. Then you can use the **print** command to print only the information that you want displayed.

Let's look at the previous example carefully. First, we assigned the value 2 to the name **i**. This initialization step is done automatically for us in a for loop but must be made explicit in a while loop. The **while** keyword indicates that the loop is a while loop and is immediately followed by the conditional statement that controls the loop. This can be any condition you want. The condition is followed by the **do** keyword. Between the **do** keyword and the **end do:** phrase is the statement sequence. The statement sequence is executed repeatedly until the condition is false.

In our example, there are two statements in the statement sequence. First, the current value of **i** is printed. Then, the value of **i** is changed to the result of squaring it and adding 2. This continues as long as the value of **i** is less than 10^7.

It is very important in a while loop to be sure that the body of the loop will eventually have the effect of making the controlling condition false. Think about what would happen if the second command in the body of the previous loop had been `i := i - 2;`. In that case, `i` would have started out equal to 2. Then it would become 0, then -2, then -4, then -6, etc, and it would never exceed 10^7, so it would never cease. This is called an infinite loop. It requires great care to avoid creating infinite loops.

Mixing for and while

The for loops and while loops we've demonstrated in this section are the most common kinds of loops. However, Maple has a feature that allows you to combine the for and while loop semantics into a single loop construction. Here is an example of how this is done.

```
> for i from 3 to 44 while i^2 < 50 do
    i, i^2;
  end do;
```

3, 9

4, 16

5, 25

6, 36

7, 49 **(0.92)**

In fact, in Maple, there is only one kind of loop, and the `for` clause and `while` clause are considered optional statements that control the looping behavior in specific ways.

Looping over a list or set

There is one additional clause that can appear in a loop, the `in` clause. Given a list or a set, the `in` clause allows you to define a loop that executes once for each element of the list or set. In the example below, the loop squares each element of the list.

```
> for i in [2,5,6,11,8] do
    printf("The square of %2a is %3a.\n",i,i^2);
  end do;
The square of  2 is   4.
The square of  5 is  25.
The square of  6 is  36.
The square of 11 is 121.
The square of  8 is  64.
```

The `for` clause indicates that the name of the loop variable is `i`. The `in` clause specifies that the loop variable should be assigned to each element of the given list in turn. The `while` clause can also appear in conjunction with the in clause. However, none of `from`, `by`, or `to` can appear when `in` is used.

Premature Loop Exit

Sometimes it is necessary to terminate a loop prematurely. This may be in order to prevent an error or because the logic of a particular problem dictates that it must. In these cases, the `break` keyword is used to transfer control out of a loop. Consider the example below. Note that `even` and `odd` are considered types in Maple and so can be used as the second argument to the `type` command. (The reader is encouraged to research the Collatz conjecture, which forms the basis of this example.)

```
> n := 27:
  count := 0:
```

```
while n <> 1 do
  if type(n,odd) then
    n := 3*n + 1;
  else
    n := n/2;
  end if;
  count := count + 1;
  if count > 50 then
    break;
  end if;
end do:
count;
```

$$51 \qquad \textbf{(0.93)}$$

In the above, the **break** statement is executed if the number of iterations of the loop exceeds a specified threshold. This example is meant to illustrate how you can use **break** to place limits on the number of iterations in a while loop.

Related to **break** is the **next** command. Instead of terminating a loop entirely, the **next** command causes the rest of the statements in the body of the loop to be skipped, but the loop continues. In a for loop, this means that it moves on to the next value of the loop variable. The example below computes the value of the rational expression $\frac{x^2+3}{x-1}$ for the integers between -3 and 3. A **next** statement is used to avoid a division by zero error.

```
> for x from -3 to 3 do
    if x = 1 then
      next;
    end if;
    x,(x^2+3)/(x-1);
  end do;
```

$$-3, -3$$
$$-2, -\frac{7}{3}$$
$$-1, -2$$
$$0, -3$$
$$2, 7$$
$$3, 6 \qquad \textbf{(0.94)}$$

Procedures

A Maple procedure is very much like a function in mathematics. It is an object that is capable of receiving data as input and producing output.

A Maple procedure is created using the **proc** keyword. Ordinarily, procedures are assigned to a name. Consider the simple example below.

```
> MySum := proc(a,b)
    a + b;
  end proc;
```

$$MySum := \textbf{proc}(a, b)\ a + b\ \textbf{end proc} \qquad \textbf{(0.95)}$$

This creates a procedure and assigns it to the name **MySum**. Note that assignment of procedures to names via the **:=** assignment operator is identical to assignment of any other Maple object. Also

note that the output of this assignment is merely a repetition of the procedure definition. In the future, we will terminate procedure definitions with a colon rather than a semicolon as it is not necessary to see the procedure definition repeated.

Following the assignment operator, the procedure definition begins with the keyword **proc**. Immediately following the proc keyword is a matched pair of parentheses enclosing a sequence of names. These names are called the *parameters* to the procedure. When the procedure is called, for instance as below,

```
> MySum(2,3);
```

5 **(0.96)**

the *arguments* **2** and **3** are assigned to the parameters **a** and **b**. Parameters are names used in the definition of the procedure to hold the place of input values, while arguments are the particular input values used in a particular execution of a procedure. (Note that this distinction is sometimes blurred and some programming languages use different terminology.) It is possible to define a procedure that requires no parameters, but even in this case the parentheses are required both in the procedure definition and when executing the procedure.

Following the parameter declaration is the statement sequence. This is the body of the procedure, consisting of all the commands that the procedure needs to perform to compute its output value. In this example only one command is used, the two parameters are added. Note that the output of a procedure is the result of the last statement that is executed. In the example above, 5 is output by the procedure because the final (and only) statement is the sum of **a** and **b**.

Declaring parameter types

Within the parameter declaration, it is common, and very useful, to declare the types of the parameters. Consider the following procedure.

```
> MySum := proc(a::integer,b::integer)
    a + b;
  end proc:
```

In this redefined **MySum** procedure, the double colons followed by the Maple type name **integer** tells Maple that we expect the parameters **a** and **b** to be integers. If we try to execute this procedure with non-integer arguments, Maple will prevent the statement sequence from being executed and will report an error.

```
> MySum(3,2.5);
Error, invalid input: MySum expects its 2nd argument, b, to
be of type integer, but received 2.5
```

Declaring the types of parameters is good programming practice and very useful, and we will typically include parameter types in the procedures we create in this manual.

Return and global and local variables

Consider the procedure below.

```
> Alg1 := proc(a::numeric)
    global n;
    local x;
    n := n + 1;
    if a < 1 then
      x := 1;
    else
      x := 2;
```

```
    end if;
    n := n + x + 10;
    return n;
  end proc:
```

This procedure introduces a few more concepts. The first line assigns the procedure to the name **Alg1** and specifies that it takes one **numeric** argument (**numeric** is a broadly defined type for numeric data).

Note the use of the **return** command in the final line of the procedure body. We've mentioned that the output of a procedure is, by default, the final computed value. **return** can be used, as it is here, to make explicit what is being output. It can also be used to cause a procedure to stop execution and immediately output a particular result. In this manual, we will usually use **return** statements, even when they are not required, so that it is clear what the output of a procedure is.

The second and third lines in **Alg1** use the **global** and **local** keywords followed by variable names. In any procedure, all of the names used in the procedure fall into one of three types: the parameters, the global variables, and the local variables.

A variable is called local when it is only accessible within the procedure. Local variables exist only within the procedure and have no meaning outside of it.

To see what this means, recall that the name **x** was used in the last subsection. When we last saw it, it had been assigned the value 4.

```
> x;
```

4 **(0.97)**

If we apply the **Alg1** procedure to a value smaller than 1, within the body of the procedure, **x** will be assigned the value 1.

```
> Alg1(.5);
```

295 **(0.98)**

But if we check the value of **x**,

```
> x;
```

4 **(0.99)**

it has remained 4. This is because the **x** inside the procedure is declared local. You can think about the local **x** in the procedure as different than and isolated from the **x** that stores 4.

Global variables are the opposite. A variable is global when it is accessible and has the same value both inside and outside the procedure. In our example, the name **n** is declared global. If you look at its value before and after the execution of the procedure, you will see that, unlike **x**, the value of **n** is changed.

```
> n;
```

295 **(0.100)**

```
> Alg1(3);
```

308 **(0.101)**

```
> n;
```

308 **(0.102)**

Use of global variables in procedures is generally discouraged in most programming languages. There are a variety of reasons to avoid global variables, not least of which is that they can cause unpredictable results, especially in larger projects. In this manual, we will have cause to use global

variables on occasion, but generally we will declare variables to be local.

Parameters to a Maple procedure are considered local automatically, in the sense that previous values of those names are not affected by the execution of the procedure. In fact, parameters are subject to the additional restriction that they cannot be modified during execution. In particular, they cannot be assigned to. For example, the procedure below causes an error to be generated when we execute it. It also illustrates how you declare more than one variable.

```
> Alg2 := proc(a::numeric)
    local x, y;
    if a > 0 then
      x := sqrt(a);
      y := a^2;
    else
      x := sqrt(-a);
      y := -a^2;
    end if;
    a := x + y;
    return (a);
  end proc:
> Alg2(5);
Error, (in Alg2) illegal use of a formal parameter
```

The "illegal use" described in the error message is the assignment `a := x + y;`.

A final example

We conclude this section with one final example.

```
> Alg3 := proc(a::numeric)
    # This procedure does nothing
    if a > 0 then
      return NULL;
    else
      return FAIL;
    end if;
    a := sqrt(a);
  end proc:
```

First, note the line that begins with a pound symbol (also called hash or sharp or number symbol). This is Maple's syntax for commenting code. Anything following a # in a line is ignored by Maple. You can use comments to provide explanation, within your procedure's code, of what it does and how it works. Commenting can also be useful in debugging procedures because it allows you to temporarily deactivate lines of code without deleting them. In this manual, explanation of code will generally be given within the exposition rather than as comments within the code itself.

Second, this example has two **return** statements. In the case that the argument is positive, the procedure returns the value **NULL**. Recall from earlier that **NULL** is Maple's name for the empty sequence. Returning **NULL** is how you can cause a Maple procedure to have no output.

```
> Alg3(5);
```

The other **return** statement returns the value **FAIL**. It is typical to have a procedure return **FAIL** to indicate that the procedure is unable to compute the desired output.

```
> Alg3(-2);
```

FAIL **(0.103)**

Finally, notice that the two **return** statements block the final statement, `a := sqrt(a);`, from

ever being encountered. Since the statement is illegal (it assigns to a parameter), that is why errors were not generated by the illegal assignment. (It also explains why the comment asserts that the procedure does nothing, since the square root is never applied.)

Functional Operators

We conclude this section with a brief discussion of functional operators. In Maple, a functional operator is a particular kind of procedure. Functional operators are often used to model simple mathematical functions.

The following defines a functional operator that models the function $f(x,y) = 2x^2 + 3xy + 5y^2$.

```
> f := (x,y) -> 2*x^2 + 3*x*y + 5*y^2;
```

$$f := (x,y) \to 2x^2 + 3xy + 5y^2 \qquad \textbf{(0.104)}$$

Let's look at the example above piece by piece. First is the name **f** followed by the assignment operator indicating that we're assigning the functional operator to the name **f**. Following the assignment operator is the list of parameters enclosed in parentheses. In the example above there are two parameters, **x** and **y**. (If a functional operator only requires one parameter, then the parentheses are optional.) After the parameter list is the "arrow", a hyphen followed by a greater than sign. The definition concludes with the formula or other Maple expression that defines the operator.

Applying a functional operator is the same as applying a procedure. To calculate $f(2,-3)$, you enter **f(2,-3)**.

```
> f(2,-3);
```

$$35 \qquad \textbf{(0.105)}$$

Functional operators are particularly useful in conjunction with other commands that expect Maple commands or procedures as one of the arguments. For example, with the **map** command. Recall that the first argument of the **map** command must be a procedure. If a list is given as the second argument, then **map** returns the list obtained by applying the procedure to each element of the list. The following demonstrates how to use a functional operator together with **map** in order to square all of the elements of a list.

```
> square := x -> x^2;
```

$$square := x \to x^2 \qquad \textbf{(0.106)}$$

```
> map(square,[-7,-3,-1,0,2,3,5]);
```

$$[49, 9, 1, 0, 4, 9, 25] \qquad \textbf{(0.107)}$$

The functional operator **square** could also have been created using **proc**, but for such a simple procedure, the functional operator definition is easier. Functional operators also allow you to include the definition of the procedure as the argument. For instance, to cube the elements of a list, you could do the following.

```
> map(x -> x^3,[-7,-3,-1,0,2,3,5]);
```

$$[-343, -27, -1, 0, 8, 27, 125] \qquad \textbf{(0.108)}$$

Functional operators will be discussed in more detail in Chapter 2.

Saving and Reading

When you are developing long or complex procedures, or even a collection of procedures, you will want to be able to save them in a file. There are two ways to do this in Maple. The more

straightforward approach is to select **Save** from the **File** menu. This saves your Maple Worksheet or Document just like most word processing software.

The other option is a bit more complicated but can be useful. Especially if you've been developing and testing procedures, your worksheet may include much more than you actually need to save. If you only have a few procedure definitions that you want to save, you can use the **save** command. To save the definition of a few procedures, say **Proc1**, **Proc2**, and **Proc3**, you type the command **save Proc1, Proc2, Proc3, "filename";**. The **save** keyword is followed by a comma-separated list of the procedure definitions you want to save ending with the name of the file in quotation marks. It is a good idea to include the path with the filename as well, for otherwise the actual location of the file on your computer's disk may be difficult to find, depending on your installation of Maple.

You can load the procedure definitions at a later time using the command **read "filename";**. This will make the procedures that you saved available in your next Maple session.

System Architecture and Packages

This section briefly explains the overall structure of the Maple system. It is intended to help you better understand how Maple works and why some things work the way that they do.

Maple uses an innovative system architecture to achieve ambitious design goals. The Maple kernel implements the basic interpreter of the Maple programming language, the interface with the host computer's operating system, and certain time critical services.

However, nearly all of Maple's mathematical power dwells in the extensive Maple library. Consisting of thousands of lines of code, the Maple library is written in the Maple programming language itself. The advantages of this design include: portability, extensibility, and openness.

Portability refers to the ability to quickly and easily modify a program to run on a different computer system or a different operating system. With Maple, only the relatively small kernel needs to be ported between systems while nothing needs to be done to the library.

Extensibility is the ability of users, like yourself, to add capabilities and features to Maple. You can even redefine existing Maple library routines to extend or modify their behavior. You will see extensibility throughout this manual as we guide you through the process of writing procedures for exploring discrete mathematics that Maple does not already include.

Openness, in Maple, means that you can examine the source code for many of the library functions, thereby gaining greater understanding of the algorithms used by Maple.

Because of its size, much of the Maple library is organized into packages. A package is a collection of Maple library routines that offer related functionality. Since these are not normally loaded when you start Maple, you must request the services localized in each package by telling Maple explicitly that you want to load them. For this purpose, Maple provides the **with** command.

For example, the **combinat** package provides routines related to combinatorics. To use procedures from the **combinat** package, you would first load the package by typing

```
> with(combinat);
```

[Chi, *bell*, binomial, *cartprod*, *character*, *choose*, *composition*, *conjpart*, *decodepart*, *encodepart*, *eulerian1*, *eulerian2*, *fibonacci*, *firstpart*, *graycode*, *inttovec*, *lastpart*, **(0.109)**

multinomial, *nextpart*, *numbcomb*, *numbcomp*, *numbpart*, *numbperm*, *partition*, *permute*, *powerset*, *prevpart*, *randcomb*, *randpart*, *randperm*, *setpartition*, stirling1, stirling2, *subsets*, *vectoint*]

All the names of the newly loaded library routines are listed. Of course, you can suppress this list by terminating the statement with a colon.

You can access commands in a package without loading the whole package by using the long form of the command name. For example, Maple includes a command, **Mean**, for computing the arithmetic mean, or average, of a list of numbers. This command is located within the **Statistics** package. We can use it as shown below.

```
> Statistics[Mean]([1,2,3,4,5,6,7]);
```

$$4. \qquad (0.110)$$

In the long form, the name of the package is followed by the name of the command in brackets. In order to use the short form of the command name, that is, just **Mean**, you would first have to load the package.

```
> with(Statistics):
> Mean([5,7,12,21]);
```

$$11.25000000 \qquad (0.111)$$

Within a procedure definition, you have the option of specifying packages that the procedure requires by means of the **uses** keyword, as in the example below.

```
> Avg3 := proc(a::numeric,b::numeric,c::numeric)
    local m;
    uses Statistics;
    m := Mean([a,b,c]);
    return m;
  end proc:
> Avg3(9,2,11);
```

$$7.333333333 \qquad (0.112)$$

The purpose of the **uses** statement is to ensure that commands used in the procedure are available. This way, if you were to try to execute this procedure without first loading the **Statistics** package via the **with** command, Maple would still be able to execute it. In this manual, when writing procedures, we will either use the long form of the name or include a **uses** statement in the procedure definition.

Maple Versions

The procedures and examples in this manual were developed and tested with Maple release 14.

On-Line Material

The files for this manual, including both the pdf and Maple Document versions of all chapters, are available at the website for the seventh edition of *Discrete Mathematics and Its Applications* by Kenneth Rosen: www.mhhe.com/rosen. This site includes many other kinds of supplementary material for students and instructors.

The website includes one file in particular to be aware of: RosenMaplePackages.mla. This file is a repository for Maple packages that include many of the commands developed in this manual. There are two ways in which you can make these packages available.

The recommended way is to install RosenMaplePackages.mla in a library directory. First, execute the following statement (shown here without output)

```
> libname;
```

The name **libname** is a system variable that sepcifies where on your computer Maple looks for library files, such as those that define packages. When you execute the statement, Maple will display the current value of the variable, which should be one or more directories on your computer.

Choose one of the directories listed and copy RosenMaplePackages.mla into that directory.

Then restart the Maple kernel by executing the command

```
> restart;
```

The alternative, in case you are not allowed to copy files into the Maple directories, is to add a directory to libname. First, copy RosenMaplePackages.mla into a directory on your computer.

Suppose that the full path of the directory in which you copied RosenMaplePackages.mla is:

/Users/myaccount/DiscreteMath/

Then you would execute the following statement, replacing the directory shown with your directory.

```
> libname := libname,"/Users/myaccount/DiscreteMath/";
```

We have not displayed the output because yours will be different from ours.

Note that if you must use the second method, you will need to repeat it every time you start a new Maple session. The first approach needs to be done only once, however.

Once you have completed one of the two methods, the following commands should execute without error.

```
> with(Chapter0):
> test();
```

"Congratulations, the package installed correctly!!" **(0.113)**

Each chapter has an associated package, called "Chapter" followed by the number of the chapter. Each package contains the procedures defined in the respective chapter that may be of use to you as you explore discrete mathematics. All of these packages are defined in the RosenMaplePackages.mla repository.

Exercises

Exercise 1. Compute $17 \cdot \left(126^{34} - 93\right)$.

Exercise 2. Form the **list** (in the Maple sense) of the first 100 positive numbers that are 3 greater than a multiple of 7, using the **seq** command.

Exercise 3. Insert the number 300 between the 42nd and 43rd entries in the list you created in the previous exercise.

Exercise 4. Use a for loop to print the string "Hello World!" 10 times.

Exercise 5. Use a while loop to print the string "Hello World!" 10 times.

Exercise 6. **even** is a Maple type and thus can be used with the **type** command. Loop over the list you created in Exercise 2, and within the loop, use an if-else statement to print "even" or "odd" for each element of the list.

Exercise 7. Use `map` and a functional operator to apply the formula $x^2 + 3x - 2$ to the list $[-5, -4.5, -4, \ldots, 4, 4.5, 5]$.

Exercise 8. Write a procedure (using `proc`) that has one parameter, a list, and outputs the list in reverse order. You should use only commands discussed in the Introduction.