Maple Programming Guide

L. Bernardin
P. Chin
P. DeMarco
K. O. Geddes
D. E. G. Hare
K. M. Heal
G. Labahn
J. P. May
J. McCarron
M. B. Monagan
D. Ohashi
S. M. Vorkoetter

Copyright © Maplesoft, a division of Waterloo Maple Inc.
2015

Maple Programming Guide

by L. Bernardin, P. Chin, P. DeMarco, K. O. Geddes, D. E. G. Hare, K. M. Heal, G. Labahn, J. P. May, J. McCarron, M. B. Monagan, D. Ohashi, and S. M. Vorkoetter

Copyright

Maplesoft, Maple, MapleNet, MaplePrimes, Maplet, Maple T.A., and OpenMaple are all trademarks of Waterloo Maple Inc.

© Maplesoft, a division of Waterloo Maple Inc. 1996-2015. All rights reserved.
No part of this book may be reproduced, stored in a retrieval system, or transcribed, in any form or by any means — electronic, mechanical, photocopying, recording, or otherwise. Information in this document is subject to change without notice and does not represent a commitment on the part of the vendor. The software described in this document is furnished under a license agreement and may be used or copied only in accordance with the agreement. It is against the law to copy the software on any medium except as specifically allowed in the agreement.

Adobe and Acrobat are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Java is a registered trademarks of Oracle and/or its affiliates.

MATLAB is a registered trademark of The MathWorks, Inc.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

NAG is a registered trademark of The Numerical Algorithms Group Ltd.

All other trademarks are the property of their respective owners.

This document was produced using Maple and DocBook.

ISBN 978-1-926902-46-3

Contents

Preface xix
1 Introduction to Programming in Maple 1
1.1 In This Chapter 1
1.2 The Maple Software 1
The User Interface 1
The Computation Engine 1
1.3 Maple Statements 2
Getting Help 2
Displaying a Text String 2
Performing an Arithmetic Operation 3
Assigning to a Name 3
Using Maple Library Commands 3
1.4 Procedures 4
Defining a Simple Procedure 4
Entering a Procedure Definition 4
Adding Comments to a Procedure 7
Calling a Procedure 8
Maple Library Commands, Built-In Commands, and User-Defined Procedures 8
Full Evaluation and Last Name Evaluation 9
Viewing Procedure Definitions and Maple Library Code 10
1.5 Interrupting Computations and Clearing the Internal Memory 10
Interrupting a Maple Computation 10
Clearing the Maple Internal Memory 11
1.6 Avoiding Common Problems 11
Unexpected End of Statement 12
Missing Operator 12
Invalid, Wrong Number or Type of Arguments 13
Unbalanced Parentheses 13
Assignment Versus Equality 13
1.7 Exercises 14
2 Maple Language Elements 15
2.1 In This Chapter 15
2.2 Character Set 15
2.3 Tokens 16
Reserved Words 16
Programming-Language Operators 17
Names 20
2.4 Natural Integers 21
2.5 Strings 22
Length of a String 22

Substrings 22
Searching a String 23
String Concatenation 23
Mutability of Strings 24
Special Characters in Strings 24
Parsing Strings 25
Converting Expressions to Strings 26
2.6 Using Special Characters 26
Token Separators 26
Blank Spaces, New Lines, Comments, and Continuation 27
Punctuation Marks 28
Escape Characters 31
2.7 Types and Operands 31
DAGs 31
Maple Types 32
Operands and op 34
2.8 Avoiding Common Problems 37
Attempting to Assign to a Protected Name 37
Invalid Left-Hand Assignment 38
Incorrect Syntax in Parse 38
White Space Characters within a Token 38
Incorrect Use of Double and Single Quotes 38
Avoid Using Maple Keywords as Names 39
2.9 Exercises 40
3 Maple Expressions 41
3.1 In This Chapter 41
3.2 Introduction 41
Expressions and Statements 41
Automatic Simplification and Evaluation 41
Syntax and Constructors 41
3.3 Names 42
Creating Names: Lexical Conventions 43
3.4 Unevaluated Expressions 46
Protecting Names and Options 47
Generic Expressions 48
Pass by Reference 48
Displaying the Original Command 49
Unassigning Names 49
Evaluation and Automatic Simplification 50
Example: Defining a Procedure that Is Returned Unevaluated 51
3.5 Numbers 53
Integers 53
Fractions 53

Floats 54
Complex Numbers 58
3.6 Indexed Expressions 62
3.7 Member Selection 67
3.8 Functions 68
Calls to Procedures 69
3.9 Arithmetic Expressions 70
Arithmetic Operators 70
Noncommutative Multiplication 82
Factorials 84
Forming Sums and Products 85
3.10 Boolean and Relational Expressions 86
Boolean Constants 86
Boolean Operators 86
Relational Operators 90
Efficient Boolean Iteration 93
3.11 Expressions for Data Structures 94
Sequences 94
Lists 96
Sets 98
Tables 99
Rectangular Tables 100
3.12 Set-Theoretic Expressions 101
Membership 101
Set Inclusion 101
Other Binary Operators for Sets 102
3.13 Other Expressions 104
Functional Operators 104
Composition 105
Neutral Operators 107
Ranges 107
The Concatenation Operator 109
The Double Colon Operator 111
Series 112
3.14 Attributes 115
3.15 Using Expressions 116
Evaluating and Simplifying Expressions 116
Substituting Subexpressions 117
Structured Types 120
3.16 Exercises 124
4 Basic Data Structures 125
4.1 In This Chapter 125
4.2 Introduction 125

4.3 Immutable Data Structures 125
Lists 126
Sets 132
4.4 Mutable Data Structures 137
Tables 137
Arrays 144
4.5 Other Data Structure Operations 153
Filtering Data Structure Elements 153
Converting Data Structures 154
4.6 Other Data Structures 155
Records 155
Stacks 157
Queues 159
4.7 Data Coercion 162
4.8 Data Structure Performance Comparisons 162
Indexing 163
Membership 163
Building a Collection of Data 164
4.9 Avoiding Common Problems 165
Passing Sequences into Functions 165
Incorrect Index Values 165
Do Not Treat Lists and Sets as Mutable 166
4.10 Exercises 167
5 Maple Statements 169
5.1 In This Chapter 169
5.2 Introduction 169
5.3 Statement Separators 169
5.4 Expression Statements 170
5.5 Assignments 170
Multiple Assignment 171
5.6 Flow Control 173
Sequencing 173
Branching 173
Loops 177
Looping Commands 182
Non-Local Flow Control 185
5.7 The use Statement 189
5.8 Other Statements 192
The quit Statement 192
The save Statement 193
The read Statement 193
5.9 Exercises 194
6 Procedures 195

6.1 Terminology 195
6.2 Defining and Executing Procedures 196
6.3 Parameter Declarations 197
Required Positional Parameters 197
Optional Ordered Parameters 198
Expected Ordered Parameters 200
Keyword Parameters 200
The End-of-Parameters Marker 202
Default Value Dependencies 203
Parameter Modifiers 204
Procedures without Declared Parameters 208
6.4 Return Type 209
6.5 The Procedure Body 209
Description 210
Options 210
Variables in Procedures 217
Non-Variable Name Bindings 220
The Statement Sequence 220
Referring to Parameters within the Procedure Body 221
6.6 How Procedures Are Executed 227
Binding of Arguments to Parameters 229
Statement Sequence Interpretation 232
6.7 Using Data Structures with Procedures 241
Passing Data Structures to Procedures 241
Returning Data Structures from Procedures 242
Example: Computing an Average 243
Example: Binary Search 244
Example: Plotting the Roots of a Polynomial 245
6.8 Writing Usable and Maintainable Procedures 249
Formatting Procedures for Readability 249
Commenting Your Code 250
6.9 Other Methods for Creating Procedures 252
Functional Operators: Mapping Notation 252
The unapply Function 253
Anonymous Procedures 255
6.10 Recursion 256
6.11 Procedures that Return Procedures 258
Example: Creating a Newton Iteration 258
Example: A Shift Operator 261
6.12 The Procedure Object 262
The procedure Type 262
Procedure Operands 263
6.13 Exercises 266

7 Numerical Programming in Maple 267
7.1 In This Chapter 267
7.2 Numeric Types in Maple 267
Integers 267
Rationals 269
Floating-Point Numbers 270
Hardware Floating-Point Numbers 272
Extended Numeric Types 273
Complex Numbers 274
Non-numeric Constants 275
7.3 More about Floating-Point Numbers in Maple 275
Representation of Floating-Point Numbers in Maple 276
Precision and Accuracy 277
Floating-Point Contagion 279
More on the Floating-Point Model 282
7.4 Maple Commands for Numerical Computing 283
The evalf Command 283
Numeric Solvers 287
The evalhf Command 287
Numerical Linear Algebra 290
7.5 Writing Efficient Numerical Programs 294
Writing Flexible Numerical Procedures 294
Example: Newton Iteration 296
Example: Jacobi Iteration 300
8 Programming with Modules 307
8.1 In This Chapter 307
8.2 Introduction 307
Encapsulation 307
Creating a Custom Maple Package 307
Creating Objects 307
Creating Generic Programs 308
8.3 A Simple Example 308
8.4 Syntax and Semantics 309
The Module Definition 309
The Module Body 309
Module Parameters 310
Named Modules 310
Declarations 312
Exported Local Variables 314
Module Options 318
Special Exports 319
Implicit Scoping Rules 324
Lexical Scoping Rules 324

Modules and Types 328
8.5 Records 329
Creating Records 329
Record Types 331
Using Records to Represent Quaternions 331
Object Inheritance 332
8.6 Modules and use Statements 334
Operator Rebinding 335
8.7 Interfaces and Implementations 339
Generic Programming as a Good Software Engineering Practice 339
Distinction between Local and Exported Variables 340
Interfaces 340
A Package for Manipulating Interfaces 341
The load Option 345
9 Object Oriented Programming 347
9.1 In This Chapter 347
9.2 Introduction To Object Oriented Programming 347
Terminology 347
Benefits of Object Oriented Programming 347
9.3 Objects in Maple 348
Creating a New Class of Objects 348
Creating More Objects 348
Objects and Types 349
9.4 Methods 349
Methods Can Access Object Locals 349
Method Names Should Be Declared static 349
Methods Are Passed the Objects They Manipulate 349
Calling Methods 349
Objects in Indexed Function Calls 350
Special Methods 350
9.5 Overloading Operators 350
Supported Operators 351
Implementing Operators 351
9.6 Overloading Built-in Routines 351
Overridable Built-in Routines 351
9.7 Examples 352
9.8 Avoiding Common Mistakes 357
Overloaded Operators and Built-in Routines Must Handle All Possibilities 357
Make Sure to Access the Correct Routine 359
Be Aware of NULL 359
Lexical Scoping Does Not Circumvent local 359
10 Input and Output 361
10.1 In This Chapter 361

10.2 Introduction 361
10.3 Input and Output in the Worksheet 363
Interfaces 363
Interactive Output 364
Interactive Input 366
Customization 366
10.4 Input and Output with Files 367
Introduction 367
Working with General Files 367
Importing and Exporting Numerical Data 372
Files Used by Maple 374
10.5 Reading and Writing Formatted Data 376
The scanf and printf Commands 376
Format Strings 377
Related Commands 378
10.6 Useful Utilities 378
The StringTools Package 378
Conversion Commands 379
10.7 2-D Math 379
Introduction 379
The Typesetting Package 380
Additional Tips 381
10.8 Exercises 381
11 Writing Packages 383
11.1 In This Chapter 383
11.2 What Is a Package 383
Packages in the Standard Library 383
Packages Are Modules 383
Package Exports 384
Using Packages Interactively 384
11.3 Writing Maple Packages By Using Modules 385
A Simple Example 385
Custom Libraries 387
11.4 A Larger Example 389
ModuleLoad 390
The Preprocessor and Structured Source Files 390
Subpackages 392
11.5 Example: A Shapes Package 393
Source Code Organization 393
Package Architecture 394
The Package API 395
The make Procedure 395
The area Procedure 396

The circumference Procedure 396
Shape Representation 396
Procedure Dispatching 396
Dispatching on Submodule Exports 397
Conditional Dispatching 398
Table-based Dispatching 398
Shape-specific Submodules 399
The point Submodule 399
The circle Submodule 400
A Internal Representation 403
A.1 Internal Functions 403
Evaluators 403
Algebraic Functions 404
Algebraic Service Functions 404
Data Structure Manipulation Functions 404
General Service Functions 404
A.2 Flow of Control 404
A.3 Internal Representations of Data Types 405
AND: Logical AND 405
ASSIGN: Assignment Statement 406
BINARY: Binary Object 406
BREAK: Break Statement 406
CATENATE: Name Concatenation 406
COMPLEX: Complex Value 407
CONTROL: Communications Control Structure 407
DCOLON: Type Specification or Test 407
DEBUG: Debug 407
EQUATION: Equation or Test for Equality 408
ERROR: Error Statement 408
EXPSEQ: Expression Sequence 408
FLOAT: Software Floating-Point Number 409
FOR: For/While Loop Statement 409
FOREIGN: Foreign Data 410
FUNCTION: Function Call 410
GARBAGE: Garbage 410
HFLOAT: Hardware Float 411
IF: If Statement 411
IMPLIES: Logical IMPLIES 411
INEQUAT: Not Equal or Test for Inequality 412
INTNEG: Negative Integer 412
INTPOS: Positive Integer 412
LESSEQ: Less Than or Equal 413
LESSTHAN: Less Than 413

LEXICAL: Lexically Scoped Variable within an Expression 413
LIST: List 414
LOCAL: Local Variable within an Expression 414
MEMBER: Module Member 414
MODDEF: Module Definition 414
MODULE: Module Instance 416
NAME: Identifier 417
NEXT: Next Statement 417
NOT: Logical NOT 417
OR: Logical OR 417
PARAM: Procedure Parameter in an Expression 417
POLY: Multivariate Polynomials with Integer Coefficients 419
POWER: Power 420
PROC: Procedure Definition 420
PROD: Product, Quotient, Power 422
RANGE: Range 422
RATIONAL: Rational 422
READ: Read Statement 422
RETURN: Return Statement 423
RTABLE: Rectangular Table 423
SAVE: Save Statement 424
SDPOLY: Sparse Distributed Multivariate Polynomial 425
SERIES: Series 425
SET: Set 425
STATSEQ: Statement Sequence 426
STOP: Quit Statement 426
STRING: Character String 426
SUM: Sum, Difference 426
TABLE: Table 427
TABLEREF: Table Reference 427
TRY: Try Statement 427
UNEVAL: Unevaluated Expression 428
USE: Use Statement 428
XOR: Logical Exclusive-Or 428
ZPPOLY: Polynomials with Integer Coefficients modulo n 429
A.4 Hashing in Maple 429
Basic Hash Tables 430
Dynamic Hash Tables 430
Cache Hash Tables 431
The Simplification Table 432
The Name Table 433
Remember Tables 433
Maple Language Arrays and Tables 434

Maple Language Rectangular Tables 434
Portability 435
Index 437

List of Figures

Figure 1.1: Maple Toolbar 11
Figure 2.1: Expression Tree 35
Figure 2.2: Expression DAG 36
Figure 2.3: Actual Expression DAG 37
Figure 3.1: expr DAG 80
Figure 3.2: subsop Example DAGs 118
Figure 11.1: Organization of Package Source Files 394
Figure 11.2: Design of Package 395

List of Tables

Table 2.1: Special Characters 15
Table 2.2: Reserved Keywords 16
Table 2.3: Binary Operators 17
Table 2.4: Unary Operators 18
Table 2.5: Element-wise Operators 19
Table 2.6: Token Separators 28
Table 2.7: Subtype 33
Table 3.1: Initially Known Names 44
Table 5.1: Operators That Can Be Rebound 192
Table 6.1: Procedure Operands 263
Table 7.1: Floating-Point Contagion Rules 281
Table 11.1: RandomnessTests 391
Table A.1: Maple Structures 403

Preface

Technical computation forms the heart of problem solving in mathematics, engineering, and science. To help you, Maple™ offers a vast repository of mathematical algorithms covering a wide range of applications.

At the core of Maple, the symbolic computation engine is second to none in terms of scalability and performance. Indeed, symbolics was the core focus when Maple was first conceived at the University of Waterloo in 1980 and to this day Maple continues to be the benchmark software for symbolic computing.

Together with a large repository of numeric functionality, including industry-standard libraries such as the Intel® Math Kernel Library (MKL), Automatically Tuned Linear Algebra Software (ATLAS), and the C Linear Algebra PACKage (CLAPACK), as well as a broad selection of routines from the Numerical Algorithms Group (NAG®) libraries, you can rely on Maple to support you a across many domains and applications. Using its unique hybrid technology, Maple integrates the symbolic and numeric worlds to solve diverse problems more efficiently and with higher accuracy.

The Maple user interface allows you to harness all this computational power by using context-sensitive menus, task templates, and interactive assistants. The first steps are intuitively easy to use and quickly lead you into the captivating, creative, and dynamic world of Maple.

As you get more proficient, you will want to explore more deeply and directly access all of the computational power available to you. You can accomplish this through the Maple programming language. Combining elements from procedural languages (such as Pascal), functional languages (such as Lisp) and object-oriented languages (such as Java™), Maple provides you with an exceptionally simple yet powerful language to write your own programs. High-level constructs such as map allow you to express in a single statement what would take ten lines of code in a language like C.

Maple allows you to quickly focus and reliably solve problems with easy access to over 5000 algorithms and functions developed over 30 years of cutting-edge research and development.

Maple's user community is now over two million people. Together we have built large collections of Maple worksheets and Maple programs, much of which is freely available on the web for you to reuse or learn from. The majority of the mathematical algorithms you find in Maple today are written in the Maple Programming Language. As a Maple user, you write programs using the same basic tools that the Maple developers themselves use. Moreover you can easily view most of the code in the Maple library and you can even extend the Maple system, tying your programs in with existing functionality.

This guide will lead you from your first steps in Maple programming to writing sophisticated routines and packages, allowing you to tackle problems in mathematics, engineering, and science effectively and efficiently. You will quickly progress towards proficiency in Maple programming, allowing you to harness the full power of Maple.

Have fun!

Audience

This guide provides information for users who are new to Maple programming, as well as experienced Maple programmers. Before reading this guide, you should be familiar with the following.

- The Maple help system
- How to use Maple interactively
- The **Maple User Manual**

Maple User Interfaces

You can access Maple functionality through several *user interfaces*. Maple interfaces accept user input, communicate with the Maple computational engine, and display solutions to mathematical problems.

The Standard Interface

The standard interface facilitates the performance of computations and lets you manipulate mathematical expressions. It also provides layout and document processing features that you can use to annotate your problem-solving process. The standard interface will be the focus of this guide.

To display the standard interface, double-click your **Maple** desktop icon (Windows® and Macintosh®) or run the **xmaple** command (UNIX®).

Other Maple Interfaces

- **MapleNet™** lets you publish your interactive Maple documents on the web. Users with an Internet connection can then view and manipulate your published documents in a web browser. MapleNet also provides a web service interface that allows connected applications to pass data to Maple, run a program, and retrieve results. It also lets you create custom JavaServer™ Pages (JSP) applications and Java applets. For more information about MapleNet, see *???*.
- **OpenMaple™** is the Maple application programming interface (API) that lets you build custom user interfaces or embed Maple in an existing application. OpenMaple can be

used with a variety of languages including C, C++, Java, Fortran, Visual Basic®, and C#. For more information about OpenMaple, see *???*.

- The **Maple command-line interface** is a console-based application that can be used for batch processing Maple command files. For more information, see *???*.
- **Maplet™** applications are custom interfaces that are created using the Maple programming language. For more information, see *???*.

For more information about the Maple user interfaces, refer to the *Maple User Manual* or the versions help page.

Programming in the Standard Interface

Most of the time, you will enter Maple code directly in a worksheet or document. The standard interface also provides other functionality for entering Maple code. For example, you can enter your code in a *startup code region* if you want to run certain commands or procedures automatically when a Maple document is opened. You can also enter your code in a *code edit region* if you want to keep a set of Maple commands or procedures in a confined region within your document. For more information, refer to the worksheet,documenting,startupcode and CodeEditRegion help pages.

You can also include your code in an external text file to be read by a worksheet or document, or batch processed. For more information, refer to the file help page.

Document Mode and Worksheet Mode

Two modes of interactive operation are available in the standard interface: *document mode* and *worksheet mode*.

In document mode, you enter mathematical expressions within document blocks; no Maple input prompt (>) or execution group boundaries are displayed in the document. You can use this mode to create professional reports that combine text and typeset math with plots, images, and other interactive components.

In worksheet mode, you enter mathematical expressions at input prompts, which are displayed at the start of each input line in a Maple document. When you type an expression and press **Enter**, the expression is evaluated and a new input prompt is displayed in the next line. In both modes, the default format for entering mathematical text is 2-D math notation.

Both modes are equally suitable for creating and running programs in Maple. Select the mode that suits your preferences and tasks. For more information about both modes, refer to the worksheet,help,documentsvsworksheets help page.

1-D and 2-D Math Notation

When programming in Maple, you must also consider whether to use 2-D math notation or 1-D math notation. In 2-D math notation, typeset mathematical text is displayed in black italicized characters.

$$\int \sin(x)\, dx$$

In 1-D math notation (or Maple input), mathematical text is displayed in a red fixed-width font that is not typeset.

```
> int(sin(x),x):
```

1-D math notation can be used in external text files to write Maple code that can be read by a worksheet or batch processed. You can enter individual statements in 1-D math notation or configure Maple to display mathematical input in 1-D math by default in all future Maple sessions.

> **Note:** While 2-D math is the recommended format for mathematical text and equations and can be used for short command sequences and procedures, it is generally not recommended for long programs and package definitions.

Most input in this guide is shown in 1-D math notation. To clearly distinguish commands and input, this guide uses a leading prompt character (>) and all input is entered in worksheet mode.

For more information on starting Maple, toggling between 1-D and 2-D math notation, and managing your files, refer to the *Maple User Manual* or enter ?managing at the Maple prompt.

Web Resources

- **Maplesoft Application Center**: The Application Center provides thousands of complete applications that you can download and use in Maple. For more information, visit **http://www.maplesoft.com/applications**.
- **MaplePrimes™**: MaplePrimes is an online forum where you can search for tips and techniques, read blogs, and discuss your work in Maple with an active community. For more information, visit **http://www.mapleprimes.com**.
- **Maplesoft Online Help**: Documentation included with Maple is also posted online. The web version offers the latest updates, Google™-based searching, and an easy way to provide feedback on help documentation. For more information, visit, **http://www.maplesoft.com/support/help**.

- **Teacher Resource Center**: The Teacher Resource Center provides course content, lecture notes, demonstrations, and other resources to help teachers incorporate Maple in their classrooms. For more information, visit **http://www.maplesoft.com/TeacherResource**.
- **Student Resource Center**: The Student Resource Center provides online forums, training videos, and other resources to help students with their work in Maple. For more information, visit **http://www.maplesoft.com/studentcenter**.

For additional resources, visit **http://www.maplesoft.com**.

Conventions

This guide uses the following typographical conventions.

- **bold** font - Maple command, package name, option name, dialog box, menu, or text field
- *italics* - new or important concept
- **Note** - additional information that is relevant to a concept or section
- **Important** - information that must be read and followed

Customer Feedback

Maplesoft welcomes your feedback. For suggestions and comments related to this and other manuals, contact doc@maplesoft.com.

1 Introduction to Programming in Maple

Maple provides an interactive problem-solving environment, complete with procedures for performing symbolic, numeric, and graphical computations. At the core of the Maple computer algebra system is a powerful programming language, upon which the Maple libraries of mathematical commands are built.

1.1 In This Chapter

- Components of the Maple software
- Maple statements
- Procedures and other essential elements of the Maple language

1.2 The Maple Software

The Maple software consists of two distinct parts.

- The user interface
- The computation engine

The User Interface

You can use the Maple *user interface* to enter, manipulate, and analyze mathematical expressions and commands. The user interface communicates with the Maple computation engine to solve mathematical problems and display their solutions.

For more information about the Maple user interface, refer to the *Maple User Manual*.

The Computation Engine

The Maple computation engine is the command processor, which consists of two parts: the kernel and math library.

The *kernel* is the core of the Maple computation engine. It contains the essential facilities required to run and interpret Maple programs, and manage data structures. In this guide, the kernel commands are referred to as *built-in* commands.

The Maple kernel also consists of *kernel extensions*, which are collections of external compiled libraries that are included in Maple to provide low-level programming functionality. These libraries include Basic Linear Algebra Subprograms (BLAS), GNU Multiple Precision (GMP), the NAG® C Library, and the C Linear Algebra PACKage (CLAPACK).

The math *library* contains most of the Maple commands. It includes functionality for numerous mathematical domains, including calculus, linear algebra, number theory, and combinatorics. Also, it contains commands for numerous other tasks, including importing

data into Maple, XML processing, graphics, and translating Maple code to other programming languages.

All library commands are implemented in the high-level Maple programming language, so they can be viewed and modified by users. By learning the Maple programming language, you can create custom programs and packages, and extend the Maple library.

1.3 Maple Statements

There are many types of valid statements. Examples include statements that request help on a particular topic, display a text string, perform an arithmetic operation, use a Maple library command, or define a procedure.

Statements in 1-D notation require a trailing semicolon (**;**) or colon (**:**). If you enter a statement with a trailing semicolon, for most statements, the result is displayed. If you enter a statement with a trailing colon, the result is computed but *not* displayed.

```
> 2 + 3;
```

$$5 \tag{1.1}$$

```
> 2 + 3:
```

For more information about statements in Maple, see *Maple Statements (page 169)*.

Getting Help

To view a help page for a particular topic, enter a question mark (**?**) followed by the corresponding topic name. For example, **?procedure** displays a help page that describes how to write a Maple procedure.

For more information about getting help in Maple, refer to the help and HelpGuide help pages.

This type of Maple statement does not have a trailing colon or semicolon.

Displaying a Text String

The following statement returns a *string*. The text that forms the string is enclosed in *double quotes*, and the result (the string itself) is displayed because the statement has a trailing semicolon.

```
> "Hello World";
```

$$\text{"Hello World"} \tag{1.2}$$

Normally, you would create a string as part of another statement, such as an assignment or an argument for a procedure.

For more information about strings in Maple, see *Maple Language Elements (page 15)*.

Performing an Arithmetic Operation

The arithmetic operators in Maple are + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation). A statement can be an arithmetic operation that contains any combination of these operators. The standard rules of precedence apply.

```
> (44*3+13)^2/116;
```

$$\frac{725}{4} \tag{1.3}$$

Maple computes this result as an exact rational number.

Assigning to a Name

By naming a calculated result or complicated expression, you can reference it. To assign to a name, use the assignment operator, **:=**.

```
> a := 103993/33102;
```

$$a := \frac{103993}{33102} \tag{1.4}$$

```
> 2 * a;
```

$$\frac{103993}{16551} \tag{1.5}$$

For more information about names and assignment, see *Maple Language Elements (page 15)*.

Using Maple Library Commands

After a value is assigned to a name, for example, the value assigned previously to **a**, you can use the name as if it were the assigned object. For example, you can use the Maple library command evalf to compute a floating-point (decimal) approximation to **103993/33102** divided by **2** by entering the following statement.

```
> evalf(a/2);
```

$$1.570796326 \tag{1.6}$$

You can use the Maple library of commands, introduced in *The Computation Engine (page 1)*, for many purposes. For example, you can find the derivative of an expression by using the diff command.

```
> diff(x^2 + x + 1/x, x);
```

$$2x + 1 - \frac{1}{x^2} \tag{1.7}$$

Note the difference between the names used in these two examples. In the first example, **a** is a variable with an assigned value. In the second example, **x** is a symbol with no assigned value. Maple can represent and compute with symbolic expressions.

For more information about the Maple library commands, refer to the *Maple User Manual* or the help system.

1.4 Procedures

This section introduces the concept of procedures in Maple. For more information about procedures, see *Procedures (page 195)*.

Defining a Simple Procedure

A Maple procedure (a type of program) is a group of statements that are processed together. The easiest way to create a Maple procedure is to enclose a sequence of commands, which can be used to perform a computation interactively, between the **proc(...)** and **end proc** statements.

Entering a Procedure Definition

The following procedure generates the string *"Hello World"*. Enter this procedure in a Maple session by entering its definition on one line.

```
> hello := proc() return "Hello World"; end proc;
```

$$hello := \mathbf{proc}()\ \mathbf{return}\ \text{"Hello World"}\ \mathbf{end\ proc} \tag{1.8}$$

You can also enter a procedure or any Maple statement on multiple lines. To move the cursor to the next line as you are entering a multiline statement, hold the **Shift** key and press **Enter** at the end of each line.

Note: This is necessary in the interactive worksheet environment only. If you enter code in a *code edit region*, you can simply type the text and press **Enter** to move the cursor to next line. For more information on code edit regions, refer to the CodeEditRegion help page. For more information about using **Shift**+**Enter**, see *Unexpected End of Statement (page 12)*.

You can indent lines in a procedure by using the spacebar. After you enter the last line, **end proc;**, press **Enter**.

```
> hello := proc()
            return "Hello World";
  end proc;
```

$$hello := \mathbf{proc}(\,)\ \mathbf{return}\ \text{"Hello World"}\ \mathbf{end\ proc} \tag{1.9}$$

To run this procedure, enter its name followed by a set of parentheses and a semicolon:

```
> hello();
```

$$\text{"Hello World"} \tag{1.10}$$

Procedures can also accept arguments. Consider the following example.

```
> half := proc(x)
            evalf(x/2);
  end proc;
```

$$half := \mathbf{proc}(x)\ \ evalf(1/2*x)\ \ \mathbf{end\ proc} \tag{1.11}$$

This procedure requires one input, **x**. The procedure computes the approximation of the value of **x** divided by **2**. When a return statement is not specified, a Maple procedure returns the result of the last statement that was run. Since evalf(x/2) is the last calculation performed in the procedure **half** (in fact, it is the only calculation), the result of that calculation is returned.

The procedure is named **half** by using the **:=** notation in the same way that you would assign any other object to a name. After you have named a procedure, you can use it as a command in the current Maple session. The syntax to run your procedure is the same syntax used to run a Maple library command: enter the procedure name followed by the input to the procedure enclosed in parentheses.

```
> half(2/3);
```

$$0.3333333333 \tag{1.12}$$

```
> half(a);
```

$$1.570796326 \tag{1.13}$$

```
> half(1) + half(2);
```

$$1.500000000 \tag{1.14}$$

The basic syntax for a procedure is given below.

```
proc( P )
        ...
end proc
```

The letter **P** indicates the parameters. The body of the procedure is between the **proc** and **end proc** keywords.

Consider the following two statements, which calculate the angle in a right triangle given the lengths of two sides.

```
> theta := arcsin(opposite/hypotenuse);
```

$$\theta := \arcsin\left(\frac{opposite}{hypotenuse}\right) \tag{1.15}$$

```
> evalf(180/Pi*theta);
```

$$57.29577950 \arcsin\left(\frac{opposite}{hypotenuse}\right) \tag{1.16}$$

The following example shows a procedure that corresponds to these statements. The procedure definition contains two input parameters for the length of two sides of a right triangle.

```
> GetAngle := proc( opposite, hypotenuse )
          local theta;
          theta := arcsin(opposite/hypotenuse);
          evalf(180/Pi*theta);
  end proc;
```

$$\begin{aligned} &GetAngle := \mathbf{proc}(opposite, hypotenuse) \\ &\quad \mathbf{local}\ \text{theta}; \\ &\quad \text{theta} := \arcsin(opposite / hypotenuse); \\ &\quad evalf(180 * \text{theta} / \text{Pi}) \\ &\mathbf{end\ proc} \end{aligned} \tag{1.17}$$

When you run the procedure definition, the output shown is the Maple interpretation of this procedure definition. Examine it carefully and note the following characteristics.

- The *name* of this procedure (program) is **GetAngle**. Note that Maple is case-sensitive, so **GetAngle** is distinct from **getangle**.
- The procedure definition starts with **proc(opposite, hypotenuse)**. The two names in parentheses indicate the parameters, or inputs, of the procedure.
- Semicolons or colons separate the individual commands of the procedure.
- The **local theta;** statement declares **theta** as a local variable. A local variable has meaning in the procedure definition only. Therefore, if you were to declare another variable called **theta** outside of the procedure, that variable would be different from the local variable **theta** declared in the procedure and you could use **theta** as a variable name outside of the procedure **GetAngle** without conflict.

For more information about local variables, see *Variables in Procedures (page 217)*.

- Pi is a predefined variable in Maple. Two predefined functions, evalf and arcsin, are used in the calculation.
- The **end proc** keywords and a colon or semicolon indicate the end of the procedure.
- As you enter the procedure, the commands of the procedure do not display output. The procedure definition is displayed as output only after you complete it with **end proc** and a semicolon.
- There is no explicit **return** statement, so the result of calling the procedure is the result of the last calculation.
- The procedure definition that displays in the output is equivalent to, but not identical to, the procedure definition you enter. When Maple parses the statement, the commands of the procedure may be simplified.

The procedure definition syntax is flexible. You can do the following:

- Enter each statement on one or more lines
- Enter multiple statements on one line, provided they are separated by colons or semicolons
- Place extra semicolons between statements
- Omit the semicolon (or colon) from the statement preceding **end proc**

To hide the output resulting from a complicated procedure definition, use a colon instead of a semicolon at the end of the definition.

Adding Comments to a Procedure

Consider the following example.

```
(* this procedure computes an interior angle of a right
    triangle given the length of the side opposite the angle, and
    the length of the hypotenuse.
*)
GetAngle := proc( opposite, hypotenuse )
        local theta;
        theta := arcsin(opposite/hypotenuse);
        # convert the angle from radians to degrees
        evalf(180/Pi*theta);
end proc:
```

You can include single line comments anywhere in the procedure. They begin with a pound character (**#**). You can also enter multiline comments between **(*** and ***)** symbols as shown in the example above.

Note: Multiline comments cannot be entered in 2-D math notation. As an alternative, in a Maple document, you can enter comments as text by adding a paragraph above or below the Maple statement.

Calling a Procedure

Running a procedure is referred to as an *invocation* or a *procedure call*. When you invoke a procedure, Maple runs the statements that form the procedure body one at a time. The result of the last computed statement within the procedure is returned as the value of the procedure call.

For example, to run the procedure **GetAngle**--that is, to cause the statements that form the procedure to be run in sequence--enter its name followed by parentheses enclosing the inputs, in this case, two numbers delimited (separated) by commas (**,**). End the statement with a semicolon.

```
> GetAngle(4,5);
```

$$53.13010234 \tag{1.18}$$

Only the result of the last calculation performed within the procedure **GetAngle** is returned--the result of evalf(180/Pi*theta). The assignment **theta:=arcsin(opposite/hypotenuse);** is performed, but the statement result is not displayed.

Maple Library Commands, Built-In Commands, and User-Defined Procedures

Maple comes with a large collection of commands and packages. Before writing custom procedures, refer to the Maple help system to find out which commands are available. You can easily include complex tasks in your user-defined procedures by using existing Maple commands instead of writing new code.

Maple commands are implemented in one of two formats: those written and compiled in an external language such as **C** and those written in the Maple programming language.

The commands that are compiled as part of the Maple kernel are referred to as *built-in* commands. These are widely used in computations, and are fundamental for implementing other Maple commands.

For more information about built-in kernel commands, see *The Computation Engine (page 1)* and *The builtin Option (page 211)*.

The commands in the Maple library are written in the Maple programming language. These commands exist as individual commands or as packages of commands. They are accessed and interpreted by the Maple system as required. The code for the library commands and the definitions of user-defined procedures can be viewed and modified. However, before

exploring library commands, it is important that you learn about evaluation rules to understand the code.

Full Evaluation and Last Name Evaluation

For most expressions assigned to a name, such as **e** defined with the following statement, you can obtain its value by entering its name.

```
> restart;
```

```
> e := 3;
```

$$e := 3 \tag{1.19}$$

```
> e;
```

$$3 \tag{1.20}$$

This is called *full evaluation*--each name in the expression is fully evaluated to the last assigned expression in any chain of assignments. The following statements further illustrate how full evaluation works.

```
> c := b;
```

$$c := b \tag{1.21}$$

```
> b := a;
```

$$b := a \tag{1.22}$$

```
> a := 1;
```

$$a := 1 \tag{1.23}$$

```
> c;
```

$$1 \tag{1.24}$$

This group of statements creates the chain of assignments. $c \rightarrow b \rightarrow a \rightarrow 1$, and **c** fully evaluates to **1**.

If you try this approach with a procedure, Maple displays only the *name* of the procedure instead of its value (the procedure definition). For example, in the previous section, **GetAngle** is defined as a procedure. If you try to view the body of procedure **GetAngle** by referring to it by name, the procedure definition is *not* displayed.

```
> GetAngle;
```

$$GetAngle \tag{1.25}$$

This model of evaluation is called *last name evaluation* and it hides the procedure details. There are several reasons for this approach relating to advanced evaluation topics. The most important concept to understand is that you will only see the name of a procedure when you reference it by itself or when it is returned unevaluated; you will not see the full procedure definition. To obtain the value of the name **GetAngle**, use the eval command, which forces full evaluation.

Last name evaluation applies to procedures, tables, and modules in Maple. For more information, refer to the last_name_eval help page.

```
> eval(GetAngle);
```

$$GetAngle \tag{1.26}$$

Viewing Procedure Definitions and Maple Library Code

You can learn about programming in Maple by studying the procedure definitions of Maple library commands. To *print* the body of Maple library commands, set the Maple interface variable **verboseproc** to **2**, and then use the print command.

For example, to view the procedure definition for the Maple *least common multiple* command, lcm, enter the following statements.

For more information about interface variables, refer to the interface help page.

```
> interface(verboseproc = 2):
> print(lcm);
```

$$\mathbf{proc}(a, b)\ \ldots\ \mathbf{end\ proc} \tag{1.27}$$

Because the built-in kernel commands are compiled in machine code, and not written in the Maple language, you cannot view their definitions. If you print the definition of a built-in procedure, you will see that the procedure has only an **option builtin** statement and no visible body.

```
> print(add);
```

$$\mathbf{proc}(\,)\ \mathbf{option}\ builtin = add;\ \ \mathbf{end\ proc} \tag{1.28}$$

1.5 Interrupting Computations and Clearing the Internal Memory

Interrupting a Maple Computation

To stop a computation, for example, a lengthy calculation or infinite loop, use one of the following three methods.

Note: Maple may not always respond immediately to an interrupt request if it is performing a complex computation. You may need to wait a few seconds before the computation stops.

- Click the stop icon in the toolbar (in worksheet versions).
- Click the interrupt icon in the toolbar (in worksheet versions). See **Figure 1.1**.



Figure 1.1: Maple Toolbar

Note: For more information on the toolbar icons, refer to the worksheet/reference/WorksheetToolbar help page.

- Hold the **Ctrl** key and press the **C** key (in UNIX and Windows command-line versions).
- Hold the **Command** key and press the period key (**.**) (in Macintosh command-line and worksheet versions).

To perform a *hard* interrupt, which stops the computation and exits the Maple session, in the Windows command-line interface, hold the **Ctrl** key and press the **Break** key.

Clearing the Maple Internal Memory

Clear the internal memory during a Maple session by entering the restart command or clicking the restart icon in the worksheet toolbar. When you enter this command, the Maple session returns to its startup state, that is, all identifiers (including variables and procedures) are reset to their initial values.

```
> restart;
```

For more information on clearing the Maple internal memory and the restart command, refer to the restart help page. For more information on the toolbar icons, refer to the worksheet/reference/WorksheetToolbar help page.

Maple tracks the use of permanent and temporary objects. Its internal garbage collection facility places memory that is no longer in use on free lists so it can be used again efficiently as needed. For more information on garbage collection and the gc command, see *???*.

1.6 Avoiding Common Problems

This section provides a list of common mistakes, examples, and hints that will help you understand and avoid common errors. Use this section to study the errors that you may encounter when entering the examples from this chapter in a Maple session.

Unexpected End of Statement

Most valid statements in Maple must end in either a colon or a semicolon. An error message is displayed if you press **Enter** in an input region that is incomplete.

Tip: You can use the parse command to find errors in statements, and the Maple debugger to find errors in programs. For more information on the debugger, see *???* or refer to the parse and debugger help pages.

If you press **Enter** to move the cursor to a new line when you are entering a procedure definition on multiple lines, the following error is displayed.

```
> p:=proc()
```

To prevent this error message from displaying as you enter a procedure definition, hold the **Shift** key and press **Enter** at the end of each line, instead of pressing only **Enter**.

```
> p := proc()
          "Hello World";
  end proc;
```

$$p := \mathbf{proc}()\ \text{"Hello World"}\ \mathbf{end\ proc} \tag{1.29}$$

In 1-D math notation, if you do not enter a trailing semicolon or colon, Maple inserts a semicolon and displays the following warning message.

```
> 1 + 2
```

Maple also inserts a semicolon after **end proc** in procedure definitions.

```
> p := proc()
          "Hello World";
  end proc
```

$$\mathbf{proc}()\ \text{"Hello World"}\ \mathbf{end\ proc} \tag{1.30}$$

Missing Operator

The most common error of this type is omitting the multiplication operator.

```
> 2 a + b;
```

You can avoid this error by entering an asterisk (*) to indicate multiplication.

```
> 2*a + b;
```

$$2\,a + b \tag{1.31}$$

Implicit multiplication, which can be used in 2-D math input, is not valid syntax in 1-D math input.

Invalid, Wrong Number or Type of Arguments

An error is displayed if the argument(s) to a Maple library command are incorrect or missing.

```
> evalf();
Error, invalid input: evalf expects 1 or 2 arguments, but received 0
> solve(y=3*x+4, 5);
Error, (in solve) a constant is invalid as a variable, 5
> cos(x, y);
Error, (in cos) expecting 1 argument, got 2
```

If such an error occurs, check the appropriate help page for the correct syntax. Enter ?topic_name at the Maple prompt.

The same type of error message is displayed if you call a user-defined procedure, such as **GetAngle**, with the wrong number of the arguments.

Unbalanced Parentheses

In complicated expressions or nested commands, it is easy to omit a closing parenthesis.

```
> {[1,0], [0,1};
```

In a valid statement, each **(**, {, and **[** requires a matching **)**, }, and **]**, respectively.

```
> {[1,0], [0,1]};
```

$$\{[0, 1], [1, 0]\} \tag{1.32}$$

Assignment Versus Equality

When you enter statements in a Maple session, it is important to understand the difference between equality (using =) and assignment (using **:=**).

The equal sign, =, is used in equality tests or to create *equations*. Creating an equation is a valid Maple statement.

```
> x = y^2+3;
```

$$x = y^2 + 3 \tag{1.33}$$

```
> solve(%,y);
> x;
```

$$x \tag{1.34}$$

In the example above, **%** is a special name that stores the value of the last statement. The solve command is used to isolate **y** in the equation defined in the first statement. The first statement is not an assignment; **x** remains a symbol with no assigned value.

You can use the assignment operator, **:=**, to assign **x** the value **y^2+3**. The assignment operator assigns the value of the right-hand side to the left-hand side. After an assignment is made, the left-hand side can be used in place of the value of the right-hand side. The left-hand side cannot be a number; it must be a *name*, indexed name, function call, or sequence of these values.

```
> x := y^2+3;
```

$$x := y^2 + 3 \tag{1.35}$$

```
> solve(x,y);
```

$$I\sqrt{3}, -I\sqrt{3} \tag{1.36}$$

```
> x;
```

$$y^2 + 3 \tag{1.37}$$

For more information about equations and Boolean testing, see *Boolean and Relational Expressions (page 86)* or refer to the evalb help page. For more information about names and assignment, see *Names (page 42)* and *Assignments (page 170)*.

1.7 Exercises

1. Assign the integers **12321**, **23432**, and **34543** to the names **a**, **b**, and **c**. Use these names to find the sum and difference of each pair of numbers.
2. Write two procedures. The first requires two inputs and finds their sum. The second requires two inputs and finds their product. Use these procedures to add and multiply pairs of numbers. How could you use these procedures to add and multiply three numbers?
3. Display your procedure definitions. Are they identical to the code you entered to write them?

2 Maple Language Elements

Before programming in Maple, it is important to learn the properties and roles of the basic elements of the Maple language. This chapter introduces some of the main concepts, which will be described in more detail later in this guide.

2.1 In This Chapter

- Basic elements of the Maple language: the character set and tokens
- Maple tokens: reserved words, operators, names, strings, and natural numbers; function types
- Using special characters
- Maple data types related to the tokens

2.2 Character Set

The Maple character set consists of letters, digits, and special characters. These include 26 lowercase letters, 26 uppercase letters, and 10 decimal digits.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

There are also 33 special characters, which are listed in **Table 2.1**. These characters, or combinations of these characters, have special meanings in the Maple language.

Table 2.1: Special Characters

Character	Meaning	Character	Meaning
	blank	(	left parenthesis
;	semicolon	)	right parenthesis
:	colon	[	left bracket
+	plus	]	right bracket
-	minus	{	left brace
*	asterisk	}	right brace
/	slash	`	left single quote (back quote)
^	caret	'	right single quote (apostrophe)
!	exclamation	"	double quote
=	equal	|	vertical bar
<	less than	&	ampersand
>	greater than	_	underscore

Character	Meaning	Character	Meaning
@	at sign	%	percent
$	dollar	\	backslash
.	period	#	pound sign (sharp)
,	comma	?	question mark
~	tilde		

These are the only characters used in the Maple language. However, all character types can be used in names and strings, including international characters. For more information on how to create names using international characters, see *Names (page 20)*.

Note: When you manipulate a string or determine the length of a string, non-ASCII and international characters may be counted as more than one byte.

Many string manipulation commands interpret multibyte characters as multiple characters.

```
> s := "\xC3\xBC";
```

$$s := \text{"ü"} \tag{2.1}$$

```
> convert(s, bytes);
```

$$[195, 188] \tag{2.2}$$

2.3 Tokens

The Maple language combines characters into tokens. The set of tokens consists of reserved words (also called *keywords*), programming-language operators, names, strings, and natural integers.

Reserved Words

Maple *keywords* are reserved words that have special meanings. Therefore, you cannot change them or use them as variables in procedures. The keywords are listed in **Table 2.2**. You can find information about specific keywords in later chapters of this guide or the help system.

For more information about reserved words in Maple, refer to the keyword help page.

Table 2.2: Reserved Keywords

Keywords	Purpose
break, **next**	loop control
if, **then**, **elif**, **else**	**if** statement
for, **from**, **in**, **by**, **to**, **while**, **do**	**for** and **while** loops

Keywords	Purpose
proc, **local**, **global**, **option**, **error**, **return**, **options**, **description**	procedures
export, **module**, **use**	modules
end	ends structures
assuming	assume facility
try, **catch**, **finally**	exception handling
read, **save**	**read** and **save** statements
quit, **done**, **stop**	ending Maple
union, **minus**, **intersect**, **subset**	set operators
and, **or**, **not**, **xor**	Boolean operators
implies	implication operator
mod	modulus operator

Programming-Language Operators

There are two main types of Maple language operators: *unary* and *binary*. Simply put, a *unary* operator acts on one operand, as in **-a**, where the operator **-** is applied to **a**. A binary operator acts on two operands, as in **a**+**b**, where + is the operator and the operands are **a** and **b**.

The Maple binary and unary operators, and their meanings, are listed in **Table 2.3** and **Table 2.4**. For more information about these operators, refer to the operators,binary and operators,unary help topics.

For information about the order of precedence of programming-language operators, refer to the operators/precedence help page.

Table 2.3: Binary Operators

Operator	Meaning	Operator	Meaning
+	addition	<	less than
-	subtraction	<=	less or equal
*	multiplication	>	greater than
/	division	>=	greater or equal
^	exponentiation	<>	not equal
$	sequence operator	=	equal or equation
@	composition	union	set union
@@	repeated composition	minus	set difference
&*string*	neutral operator	intersect	set intersection
,	expression separator	::	type declaration and pattern binding

Operator	Meaning	Operator	Meaning
||	concatenation	in	membership
.	non-commutative multiplication	and	logical and
->	arrow operator	or	logical or
..	ellipsis	xor	exclusive or
mod	modulo	implies	implication
:=	assignment	subset	subset

Table 2.4: Unary Operators

Operator	Meaning
+	unary plus (prefix)
-	unary minus (prefix)
!	factorial (postfix)
$	sequence operator (prefix)
not	logical not (prefix)
&string	neutral operator (prefix)
.	decimal point (prefix or postfix)

Most of the unary and binary operators can also be used in *element-wise* form with objects that have multiple elements. To perform an element-wise operation, add a trailing tilde (~) after an operator that has an element-wise form. An element-wise operation allows you to apply an operation to the elements of a list, set, table, Array, Matrix, or Vector. For example, compare Matrix multiplication with element-wise multiplication of paired entries in a Matrix.

```
> <1,2;3,4> . <2,2;2,2>;
```

$$\begin{bmatrix} 6 & 6 \\ 14 & 14 \end{bmatrix} \tag{2.3}$$

```
> <1,2;3,4> .~ <2,2;2,2>;
```

$$\begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \tag{2.4}$$

The Maple element-wise operators are listed in **Table 2.5**. For more information about these operators, refer to the operators,elementwise help page.

Table 2.5: Element-wise Operators

Element-wise Operator	Meaning	Element-wise Operator	Meaning
+~	addition or unary plus	<~	less than
-~	subtraction or unary minus	<=~	less or equal
*~	multiplication	>~	greater than
/~	division	>=~	greater or equal
^~	exponentiation	<>~	not equal
!~	factorial (unary postfix)	=~	equal or equation
@~	composition	union~	set union
@@~	repeated composition	minus~	set difference
&*name* ~	neutral operator	intersect~	set intersection
&*name* ~	neutral operator (unary prefix)	in~	membership
subset~	subset	or~	logical or
.~	non-commutative multiplication	and~	logical and
||~	concatenation	xor~	exclusive or
mod~	modulo	implies~	implication
funct~	element-wise	not~	logical not (unary prefix)

Also, three special *nullary* operators (also called *ditto* operators) can be used in interactive sessions. These are special Maple names that can be used to refer to previously computed non-NULL expressions.

`%`	`last expression`
`%%`	`second-last expression`
`%%%`	`third-last expression`

While they can be used for simple computations, the ditto operators should be avoided when writing programs. For results that need to be used in subsequent expressions, assign values to variables instead.

Note: In a worksheet, the ditto operators do *not* necessarily reference the results of the lines located above the execution groups in which they are used. They reference the results of the *most recently performed* computations in the Maple session, regardless of the execution group or document in which they are located. Also, in terms of evaluation, the ditto operators are treated differently than local variables in a procedure. They are fully evaluated, which may require more processing than one-level evaluation of local variables. For more information about local variables, see *Local Variables (page 218)*.

For more information about the ditto operators, refer to the ditto help page.

Names

A *name* in Maple is a sequence of one or more characters that uniquely identifies a command, file, variable, or other entity.

The simplest instance of a name consists of a letter followed by a sequence of letters, digits, and underscores.

```
> My_Name_1;
```

$$My\_Name\_1 \tag{2.5}$$

If you need to create a name that includes blank spaces or international characters, use left single quotes (`).

```
> `A quoted name`;
```

$$A\ quoted\ name \tag{2.6}$$

```
> `1. A silly name`;
```

$$1.\ A\ silly\ name \tag{2.7}$$

In general any name that can be formed without left single quotes is identical to the same name with quotes. For example, **x** and **`x`** refer to the same name **x**. Left single quotes are similar to double quotes in that double quotes are used to build strings while left single quotes are used to build names.

Note that the reverse is not true, some names can be formed with left single quotes that are not identical to expressions typed in without quotes. One example is the name **`2`**. By putting quotes around the 2 here, a name is formed instead of a number. Another example is a quoted keyword, like **`module`**. To test if an expression is of type module, check type(expr,`module`). Without the quotes, the Maple parser determines that this is the start of a module definition and the parser will flag a syntax error.

Characters in Maple are case-sensitive. Therefore, for example, the name **Apple** is different from the name **apple**.

```
> Apple := 4;
```

$$Apple := 4 \tag{2.8}$$

```
> apple := 5;
```

$$apple := 5 \tag{2.9}$$

```
> Apple + apple;
```

$$9 \tag{2.10}$$

Other Maple names are used for

- mathematical functions such as **sin** and **cos**
- Maple commands such as **expand** or **simplify**
- type names such as **integer** or **list**
- symbols, for example, **x** and **y** in the expression **x+y**
- variables, or names with assigned values

For example, in the first statement below, **y** is a name that does not have a value. In the second statement, the variable **x** has the value **3**.

```
> 2*y - 1;
```

$$2\,y - 1 \tag{2.11}$$

```
> x := 3; x^2 + 1;
```

$$x := 3$$

$$10 \tag{2.12}$$

You can create an *empty name*, which has no characters in its spelling.

```
> type( ``, 'name' );
```

$$\mathit{true} \tag{2.13}$$

Early versions of Maple did not have separate types for names and strings. As a result, many commands for string processing will also accept names and process their characters the same way. It is generally better to use strings for such processing as strings can never have assigned values.

For more more information about names, see *Names (page 42)*.

2.4 Natural Integers

A *natural integer* is a sequence of one or more decimal digits.

```
> 00003141592653589793238462643;
```

$$3141592653589793238462643 \tag{2.14}$$

For more information about integers, see *Integers (page 53)* and *Numeric Types in Maple (page 267)*.

2.5 Strings

A *string* is a sequence of characters that evaluates to itself. To create a string, enclose any sequence of characters in double quotes.

```
> "This is a string";
```

"This is a string" (2.15)

You *cannot* assign a value to a string.

```
> "hello" := 5;
```

In the following sections, strings and string operations are described. For information on the StringTools package, refer to the StringTools help page.

Length of a String

Use the length command to determine the number of bytes in a string.

```
> length("What is the length of this string?");
```

34 (2.16)

All of the characters between, but excluding, the double quotes are counted. Each blank space is counted as one character. Non-ASCII characters may be counted as more than one byte.

The maximum string length is system-dependent and ranges from about 268 million bytes on 32-bit systems to more than 34 billion bytes on 64-bit systems.

Substrings

You can extract a substring of a string by using a subscripted integer range (also called a *selection operation*).

```
> S := "This is a string";
```

$S :=$ "This is a string" (2.17)

```
> S[6];
```

"i" (2.18)

```
> S[6..9];
```

"is a" (2.19)

Negative numbers in the range count backwards from the end of the string. -2 is the second last character in the string. Either range endpoint can also be left off to indicate from the beginning, or to the end.

```
> S[-6..-1];
```

"string" (2.20)

```
> S[11..];
```

"string" (2.21)

Searching a String

To perform case-sensitive and case-insensitive string searching, use the SearchText and searchtext commands, respectively.

SearchText(*pattern*, *exprString*, *range*);
searchtext(*pattern*, *exprString*, *range*);

The SearchText command searches for exact matches of **pattern** in **exprString**. The searchtext command performs the same search, but it is case-insensitive. If **pattern** is found, Maple returns an integer indicating the position of the first character in **pattern** in **exprString**. If the pattern is not found in **exprString**, **0** is returned.

```
> SearchText("my s", "This is my string.");
```

9 (2.22)

```
> searchtext("My S", "This is my string.");
```

9 (2.23)

The optional **range** restricts the search to the specified range. It is equivalent to performing a search on a substring, and it is useful when the pattern occurs more than once in the string.

```
> SearchText("is", "This is my string.", 4..-1);
```

3 (2.24)

String Concatenation

Strings can be formed through concatenation by using the **cat** command.

cat(*sequence*)

Here, the **sequence** parameter can contain any number of expressions that are separated by commas.

The **cat** command is commonly used to concatenate strings with names and integers, and the result returned has the type (name or string) of the first argument to **cat**.

```
> i := 5;
```

$$i := 5 \tag{2.25}$$

```
> cat( "The value of i is ", i, "." );
```

$$\text{"The value of i is 5."} \tag{2.26}$$

```
> filename := cat( kernelopts(mapledir), kernelopts(dirsep), "lib"
  );
```

$$\text{"C:\Program Files\Maple 16\lib"} \tag{2.27}$$

Mutability of Strings

Strings are not mutable objects in Maple. This means that appending text to a string is not done in-place, but involves allocating new storage for the result and copying the original text, plus the appended text, into that new space. This is typically not an issue unless you are incrementally processing large amounts of text. In the latter case, the **StringBuffer** command may be useful.

```
> with(StringTools):
> s := StringBuffer();
```

$$s := \text{""} \tag{2.28}$$

```
> s:-append("The quick brown fox"):
> s:-newline():
> s:-append("jumped over the lazy dog"):
> s:-value();
```

$$\begin{array}{l}\text{"The quick brown fox}\\ \text{mped over the lazy dog"}\end{array} \tag{2.29}$$

For more information, refer to the StringBuffer help page.

Special Characters in Strings

To display the double quote character in a string, enter a backslash character (\) followed by a double quote (") where you want the double quote character to appear. For more information, refer to the backslash help page.

```
> "a\"b";
```

"a"b" (2.30)

Similarly, to display a backslash character as one of the characters in a string, enter two consecutive backslash characters, \\. You must escape the backslash in this manner because backslash is itself a special character. For more information, see *Blank Spaces, New Lines, Comments, and Continuation (page 27)*.

```
> "a\\b";
```

"a\b" (2.31)

The special backslash character mentioned above counts as only one character, as demonstrated by using the **length** command.

```
> length((2.31));
```

3 (2.32)

Doubling up backslashes is most notable when entering full path names in Maple. For this situation it is easier to use forward slash instead. Forward slash is recognized as a directory separator on all platforms including Windows.

Parsing Strings

The **parse** command accepts any Maple string and parses the string as if it had been entered or read from a file. This is especially useful when you want to interpret commands typed into text-components inside your Maple document.

```
parse( exprString, option );
```

Without specifying extra options, the string should contain exactly *one* Maple expression. The expression is parsed and returned unevaluated.

```
> parse("a+b");
```

$$a + b \quad (2.33)$$

```
> parse("a+b;");
```

$$a + b \quad (2.34)$$

If the string is syntactically incorrect, the parse command displays an error message of the form **incorrect syntax in parse: ... (number)**.

The **number** indicates the offset in characters, counted from the beginning of the string, at which the syntax error was detected.

```
> parse("a++b");
```

```
Error, incorrect syntax in parse: `+` unexpected (near 3rd character
 of parsed string)
```

If the option **statement** is specified, the string is parsed and evaluated, and then the result is returned.

```
> parse("sin(Pi)");
```

$$\sin(\pi) \tag{2.35}$$

```
> parse("sin(Pi)", 'statement');
```

$$0 \tag{2.36}$$

Partial statements or incomplete expressions cannot be parsed. Multiple statements or expressions can be interpreted via multiple calls to parse using the **lastread** and **offset** options.

For more information, refer to the parse help page.

Converting Expressions to Strings

To convert an expression to a string, use the convert command.

Maple can convert a variety of expressions. For more information about expressions, see *Maple Expressions (page 41)*. For more information about conversions in Maple, refer to the convert help page.

```
> convert(a, 'string');
```

$$\text{"a"} \tag{2.37}$$

```
> convert(a+b-c*d/e, 'string');
```

$$\text{"a+b-c*d/e"} \tag{2.38}$$

```
> convert(42, 'string');
```

$$\text{"42"} \tag{2.39}$$

2.6 Using Special Characters

Token Separators

You can separate tokens by using white space characters or punctuation marks. The separator indicates the end of one token and the beginning of the next.

Blank Spaces, New Lines, Comments, and Continuation

The *white space characters* are space, tab, return, and line-feed. This guide uses the term *new line* to refer to a return or line-feed since the Maple system does not distinguish between these characters. The term *blank* refers to a space or tab.

The white space characters separate tokens, but are *not* themselves tokens. White space characters cannot normally be used within a token.

```
> a: = b;
```

However, you can use white space characters between tokens.

```
> a * x + x*y;
```

$$a\,x + x\,y \tag{2.40}$$

White space characters can be part of a token in a name or string formed by enclosing a sequence of characters in left single quotes or double quotes respectively. For more information, see *White Space Characters within a Token (page 38)*.

Except in a string, all characters that follow a pound sign "#" on a line are part of a *comment*.

For information about adding comments in Maple procedures, see *Adding Comments to a Procedure (page 7)*.

```
> a := 1 + x + x^2;  #This is a comment
```

$$a := x^2 + x + 1 \tag{2.41}$$

Since white space and new line characters are functionally identical, you can continue *statements* from line to line, as described in *Entering a Procedure Definition (page 4)*.

```
> a:= 1 + x +
  x^2;
```

$$a := x^2 + x + 1 \tag{2.42}$$

Note: Press **Shift**+**Enter** to continue typing on the next line without evaluating the expression.

To enter a long *number* or *string* on multiple lines, use the backslash character (\) as a line continuation character.

Line continuation functions as follows: if a backslash \ immediately precedes a new line character, the Maple parser ignores both the backslash and the new line. If a backslash is in the middle of a line, Maple usually ignores it. For more information about the backslash character and exceptions to this rule, refer to the backslash help page.

You can use the backslash character to break up a long sequence of digits into groups of smaller sequences to enhance readability.

```
> "The input should be either a list of \
  variables or a set of variables";
```

```
> G:= 0.57721566490153286060\
  6512090082402\43104215933593992;
```

$$G := 0.57721566490153286060651209008240243104215933593992 \quad (2.43)$$

You can also enter long strings by using a continuation character. Maple automatically concatenates string constants that are on separate lines, so another way to enter a long string is to close one set of double quotes at the end of a line and enter a new double quote at the beginning of the next line.

```
> S:= "This is the start of a long string "
  "and this is part of the same string with no line in between";
```

$$S := \text{"This is the start of a long string and this is part of the same string with no line in between"} \quad (2.44)$$

Punctuation Marks

The punctuation marks that act as token separators are listed in **Table 2.6**.

Table 2.6: Token Separators

;	semicolon	(	left parenthesis
:	colon	)	right parenthesis
`	left single quote	[	left bracket
'	right single quote	]	right bracket
|	vertical bar	{	left brace
<	left angle bracket	}	right brace
>	right angle bracket	,	comma

Semicolon (;) and Colon (:)

Use the semicolon and the colon to separate statements. During an interactive session, a semicolon displays the result of the statement while a colon prevents the result of the statement from displaying.

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

$$f := x \rightarrow x^2 \tag{2.45}$$

```
> p:=plot(f(x), x=0..10):
```

Right Single Quotes (')

Enclosing an expression, or part of an expression, in right single quotes (or *apostrophes*) delays the evaluation of an expression (subexpression) by one level. This is often used to ensure that procedure options are passed correctly as unevaluated names even when they have a value. For more information, see *Unevaluated Expressions (page 46)*.

```
> 'sin'(Pi), sin(Pi);
```

$$\sin(\pi), 0 \tag{2.46}$$

```
> right := 42;
```

$$right := 42 \tag{2.47}$$

```
> limit(1/x, x=0, 'right');
```

$$\infty \tag{2.48}$$

Left Single Quotes (`)

To form a name, enclose an expression in left single quotes.

```
> `My Var` := 4;
```

$$My\ Var := 4 \tag{2.49}$$

Parentheses

The left and right parentheses group terms in an expression, arguments in a function call, and parameters in a procedure definition.

```
> (a+b)*c; cos(Pi);
  proc( x, y, z )
    x+y+z;
  end proc:
```

$$(a + b)\ c$$

$$-1 \tag{2.50}$$

The left and right parentheses are also used to select components from certain data structures (programmer indexing).

Square Brackets

Use the left and right square brackets to form indexed (subscripted) names and to select components from data structures such as Arrays, tables, and lists. For more information on selection, see *Indexed Expressions (page 62)*. For more information on mathematical indexing and programmer indexing, see *Basic Data Access (page 145)*.

```
> a[1]; L:=[2,3,5,7]; L[3];
```

$$a_1$$

$$L := [2, 3, 5, 7]$$

$$5 \tag{2.51}$$

Square Brackets and Braces

Use the left and right square brackets ([]) to form lists, and the left and right braces ({}) to form sets. For more information on sets and lists, see *Immutable Data Structures (page 125)*.

```
> L:=[2,3,5,2]; S:={2,3,5,2};
```

$$L := [2, 3, 5, 2]$$

$$S := \{2, 3, 5\} \tag{2.52}$$

Angle Brackets

The left and right angle brackets (<>) in conjunction with the the comma, semicolon, and/or vertical bar (|) can be used to create Matrices and Vectors. For more information, refer to the Matrix and MVshortcut help pages.

```
> <1,2,3; 4,5,6>;
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{2.53}$$

```
> <1,2,3| 4,5,6>;
```

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \tag{2.54}$$

Comma

Use the comma to form an expression sequence. Expression sequences are used to specify the arguments of a function call or the elements of a list or set.

```
> sin(Pi), 0, limit(cos(xi)/xi, xi=infinity);
```

$$0, 0, 0 \tag{2.55}$$

Escape Characters

An *escape character* indicates that the character that follows the escape character must be handled in a special manner. The escape characters in Maple are **?**, **!**, **#**, and \.

? - The question mark character, if it appears as the first non-blank character on a line, opens the Maple help system. The words following **?** on the same line determine the terms used to display a help page. Use either "," or "/" to separate the words that follow the question mark character. For more information, refer to the help help page.

! - The exclamation mark character, if it appears as the first non-blank character on a line, passes the remainder of the line as a command to the host operating system. For more information, refer to the system and escape help pages.

and **(***, ***)** - The pound sign character indicates that the characters that follow it on the line are a comment. The multiline comment characters, **(***, and ***)** indicate the beginning and end of a comment. For more information, see *Adding Comments to a Procedure (page 7)* or refer to the comment help page.

\ - The backslash character is used to continue lines, to group characters in a token, and introduce control characters. For more information, refer to the backslash help page.

2.7 Types and Operands

In most programming languages, data is divided into different classes of information called *data types*. In Maple, there is a logical or mathematical notion of *type* that is related to, but distinct from, the underlying data structure.

DAGs

All data in Maple is stored as a *directed acyclic graph* (DAG). An identifying tag for each DAG indicates what type of data it stores and how it is stored. Names, strings, lists, and positive and negative integers are examples of some DAG types. For a list of DAG types and how they are stored in memory, see *Internal Representation (page 403)*. The **op** command (short for *operand*) can often be used to determine the DAG type of the underlying data by returning the zeroth operand. This only applies to certain data structures where **op(0,e)**, where **e** is an expression, is defined as a special case. For more information, see the particular data structure help page.

```
> op(0, [1,2,3]);
```

$$list \tag{2.56}$$

```
> op(0, "some text");
```

$$string \tag{2.57}$$

```
> op(0, `some name`);
```

$$symbol \tag{2.58}$$

```
> op(0, 123456);
```

$$Integer \tag{2.59}$$

However, the correspondence is not exact. A notable exception to this is function calls where the zeroth operand is the function name.

```
> op(0, f(x));
```

$$f \tag{2.60}$$

```
> op(0, op(0, f(x)));
```

$$symbol \tag{2.61}$$

When names are assigned to data, they act as pointers in other languages. However, for most purposes, the Maple evaluation rules are such that you can think of them as variables. If you want to manipulate an assigned name rather than the data assigned to it, you can use right single quotes (also called **unevaluation quotes**), which are described in more detail in *Unevaluated Expressions (page 46)*.

Assigned names that are pointers do not require type declarations as in low-level programming languages. This means that a name a may be assigned an integer and then later assigned a list without discretion. This system of *weak typing* means that, when writing robust code, you must verify types since variables may be assigned any value.

Maple Types

The type facility in Maple is accessed by using the **type** command. It is a mathematical type facility; however, some basic types such as **integer** or **list** map directly to the type of DAG. Some types, such as **numeric**, encapsulate a group of many different kinds of structures and structured types, such as **name^integer**, can match a very specific value.

Type checking is important in Maple to decide whether an expression is valid input for procedure calls and Maple commands. You can use the the type command or the **::** operator for type checking. The operator form is primarily used to declare the type of a procedure parameter (see *Procedures (page 195)*). The type command has the following syntax.

```
type( expression, typeName );
```

If the **expression** is of type **typeName**, the **type** command returns a value of **true**. Otherwise, a value of **false** is returned.

```
> type([1,2,3], 'list');
```

true (2.62)

```
> type("string", 'list');
```

false (2.63)

```
> type(123456, 'integer');
```

true (2.64)

```
> type(f(x), 'function');
```

true (2.65)

The type of any integer is **integer**. The type command can also interpret many subtypes of integers, some of which are listed in **Table 2.7**.

Table 2.7: Subtype

Subtype	Meaning
integer[8]	64-bit sized integer
integer[4]	32-bit sized integer
negint	negative integer
posint	positive integer
nonnegint	non-negative integer
nonposint	non-positive integer
even	even integer
odd	odd integer
prime	prime number

For more information, refer to the type help page, which also contains a complete list of types in Maple.

The type facility can also interpret compound or *structured types* such as list(integer) for a list of integers or list({negint,prime}) for a list of negative or prime integers.

```
> type([-1, 2, 11], 'list({negint,prime})');
```

true (2.66)

```
> type([0, 2, 11], 'list({negint,prime})');
```

false (2.67)

For more information about structured types, see type,structure.

Operands and op

In addition to providing information about the underlying type, the op command can provide information about the other operands or parts of a data structure. Most data structures in Maple can be divided into components. For example, an equation of the form x=y+x can be divided as follows.

- the operator, **=**
- the left-hand side, **x**
- the right-hand side, **y+x**

To determine the operands and the number of operands in an expression, use the op and nops commands respectively. These commands have the following basic syntax.

```
op( i, expression );
```

```
nops( expression );
```

If the optional first argument **i** to the op command is a positive integer, the **i**th operand of **expression** is returned.

```
> eq := x=y+x:
> nops(eq);
```

$$2 \tag{2.68}$$

```
> op(0, eq);
```

$$`=` \tag{2.69}$$

```
> op(1, eq);
```

$$x \tag{2.70}$$

```
> op(2, eq);
```

$$y + x \tag{2.71}$$

```
> op(0, eq)(op(1,eq), op(2,eq));
```

$$x = y + x \tag{2.72}$$

The op command can also determine the operands of an expression at various levels of a structure with the following syntax.

```
op( [i1, i2, ...], expression );
```

This syntax is equivalent to and more efficient than a nested call to the op command.

```
> op([2,0], eq);
```

$$`+` \tag{2.73}$$

```
> op(0,op(2, eq));
```

$$`+` \tag{2.74}$$

This hierarchical structure of expressions explains the name DAG. The internal representation of $x = y + x$ looks like an infix expression tree. See **Figure 2.1**

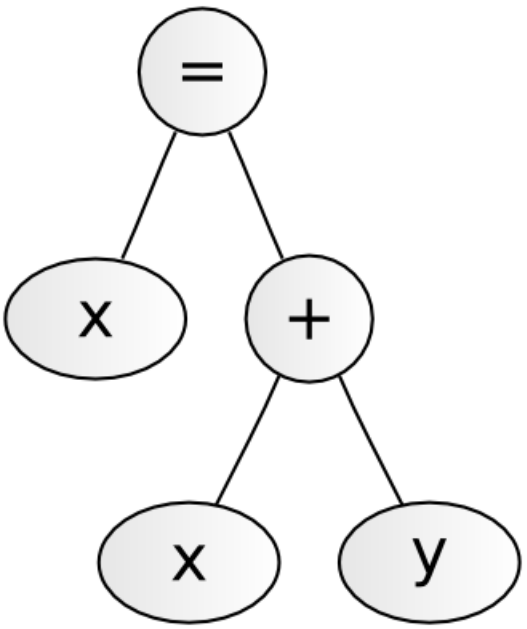


Figure 2.1: Expression Tree

For efficiency, Maple does not store multiple copies of identical objects, so the two x nodes in the tree can be represented in a picture like the one in **Figure 2.2**. In **Figure 2.1** you see two "x" nodes in the tree, implying a copy of each "x". **Figure 2.2** shows that the same instance of "x" is referred to in both places.

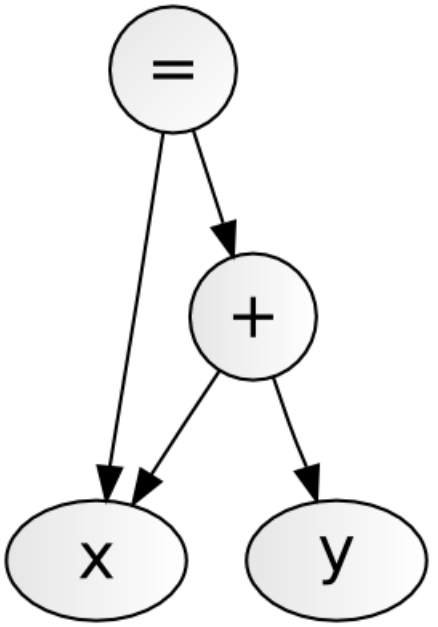


Figure 2.2: Expression DAG

The term *directed acyclic graph* simply refers to this variation of a tree where nodes may have multiple parents.

The tree form of an expression can be displayed using the dismantle command.

```
> dismantle(eq);


EQUATION(3)
   NAME(4): x
   SUM(5)
      NAME(4): y
      INTPOS(2): 1
      NAME(4): x
      INTPOS(2): 1
```

This model is not exactly what is used in practice but the principle of uniqueness, with respect to nodes, still applies. Maple uses a more sophisticated internal representation for sums as described in *Internal Representation (page 403)*. The real structure of the DAG shown in **Figure 2.3**.

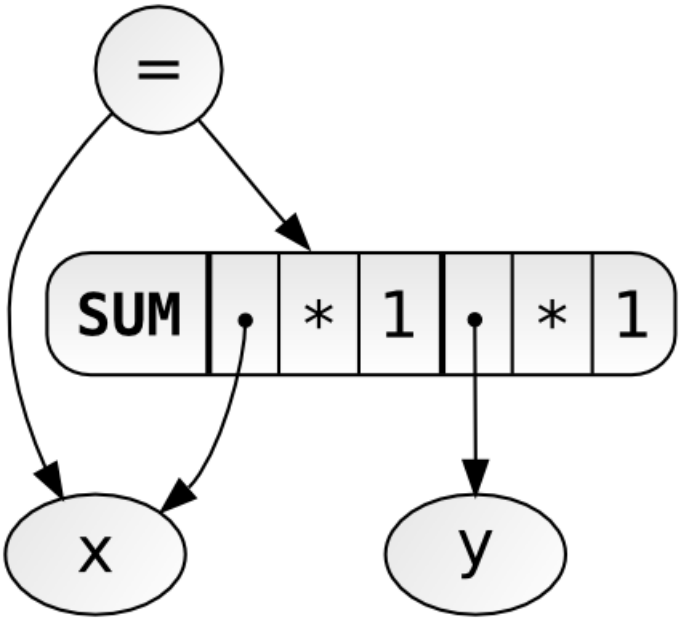


Figure 2.3: Actual Expression DAG

The next three chapters introduce many of the other types in Maple and describe how to create and use them in programs.

2.8 Avoiding Common Problems

This section provides you with a list of common mistakes, examples, and hints that will help you understand and avoid common errors. Use this section to study the errors that you may encounter when entering the examples from this chapter in a Maple session.

Attempting to Assign to a Protected Name

An exception is raised if you attempt to assign a value to a protected name. To resolve this error, specify a different name.

For more information about protected names, see *Protected Names (page 45)* or refer to the protect help page.

```
> int := 10;
```

```
Error, attempting to assign to `int` which is protected.  Try declaring
 `local int`; see ?protect for details.
```

Invalid Left-Hand Assignment

An exception is raised if you attempt to assign a value to a string.

For more information about strings, see *Strings (page 22)* or refer to the string help page.

```
> "my string" := 10;
```

Use only valid names on the left-hand side of an assignment statement.

Incorrect Syntax in Parse

The parse command accepts a string as its argument. An exception is raised if the string is syntactically incorrect.

For more information, refer to the parse help page.

```
> parse("a^2--b");
```

```
Error, incorrect syntax in parse: `-` unexpected (near 5th character
 of parsed string)
```

The error message indicates the character number (counted from the left double quote) where the error was detected. In this case, the **6**th character (the second minus sign) caused the error.

White Space Characters within a Token

An exception is normally raised if a white space character occurs in a token.

```
> evalb(2 < = 3);
```

The less-than-or-equal operator <= is a token in Maple. Therefore, it cannot contain a space.

```
> evalb(2 <= 3);
```

$$true \tag{2.75}$$

Incorrect Use of Double and Single Quotes

In Maple, double quotes form a string, left single quotes form a name, and right single quotes delay evaluation of an expression. Confusing a string with a name, or a name with delayed evaluation causes errors. Study the following examples to see the different uses of these quotes.

For more information about using quotes, see *Punctuation Marks (page 28)* or refer to the quotes help page.

To form a string, enclose the expression in double quotes.

```
> "2 + 3";
```

$$"2 + 3" \tag{2.76}$$

```
> type((2.76),'string');
```

$$true \tag{2.77}$$

To form a name, enclose the expression in left single quotes. Unlike a string, which is displayed with double quotes around it, names are usually printed without quotes. The name in this example only looks like an expression.

```
> `2 + 3`;
```

$$2 + 3 \tag{2.78}$$

```
> type((2.78),'name');
```

$$true \tag{2.79}$$

To delay the evaluation of an expression, enclose it in right single quotes.

```
> x := 2: y := 3: f := 'x + y';
```

$$f := x + y \tag{2.80}$$

```
> eval(f);
```

$$5 \tag{2.81}$$

Avoid Using Maple Keywords as Names

If you use a Maple keyword in a name, and do not enclose it in left single quotes, an exception is raised.

```
> 1 + end;
```

To resolve this issue, select a variable name that is not a Maple keyword.

Note: It is possible to use a Maple keyword as a name by enclosing it in left single quotes. For example,

```
> `end` := 2;
```

$$end := 2 \tag{2.82}$$

```
> 1 + `end`;
```

$$3 \tag{2.83}$$

However, this approach is not recommended, since it makes it very likely that errors will be introduced if, for example, you forget to place back-ticks around keywords. When possible, avoid using keywords as names.

2.9 Exercises

1. Using Maple operators, do the following:
   a Compute the sum of **5434** and **6342**.
   b Compute the product of **92** and **310**.
   c Compute the quotient of the result from a) divided by the result from b).
   d Create a list containing the numbers from a), b), and c).
   e Square each element of the list from d).
2. Create variables named "my quotient" and "my remainder". Use these variables and the **irem** command to find the integer quotient and remainder of **12345** divided by **234**. **Tip**: Because the **irem** command stores extra results by assigning to the third argument you will need to make sure the name is passed and not its assigned value. Do this by using uneval quotes (').
3. Compute **3^(3^98)** modulo **7**.
4. Concatenate the three strings "int", "(x^2,", and "x)". Parse the resulting string. Evaluate the parsed string.
5. Determine a random integer between 40 and 100 using the command rand(40..100). Concatenate this number with the string, "The student's grade is ". Extract the student's grade from the resulting string.
6. Assign the expressions **x^2** and **x*x** to the names **a** and **b**. Find the three operands of **a** and **b**. Compare the results with those returned by using the dismantle command, that is, dismantle(a) and dismantle(b). The dismantle command displays the internal data structure used.

3 Maple Expressions

This chapter introduces Maple expressions associated with *scalar* data structures.

3.1 In This Chapter

- Introduction: automatic simplification and evaluation; syntax and constructors
- Using names, strings, and numbers in expressions
- Unevaluated expressions
- Expression types: arithmetic, Boolean, relational, and set-theoretic expressions; expressions for data structures; indexed expressions; function and member selection expressions
- Assigning attributes
- Classifying, examining, and manipulating expressions

3.2 Introduction

Expressions and Statements

Maple language elements can be classified as either *expressions* or *statements*. An expression is a first-class data element in the Maple language. In other words, expressions can be stored in data structures, passed as arguments to procedures, and manipulated in various ways; they are often used to represent mathematical objects. Statements, on the other hand, are not first-class data elements; they generally describe non-mathematical programming constructs and are used to affect the state of Maple.

This chapter describes expressions associated with *scalar* data structures. For information about non-scalar data structures, see *Basic Data Structures (page 125)*.

For more information about Maple statements, see *Maple Statements (page 169)*.

Automatic Simplification and Evaluation

Maple uses two processes to compute expressions: *automatic simplification* and *evaluation*. Automatic simplification is a process that Maple applies to the input immediately; this process cannot be controlled. Expression evaluation occurs after an initial round of automatic simplification; this process can be controlled in certain ways. For each kind of expression described in this chapter, the rules for both automatic simplification and expression evaluation are described.

Syntax and Constructors

You can create most expressions by entering the appropriate syntax. However, some expressions, such as expressions that include tables or a series, can only be created by calling a

constructor. A constructor is a command that can be used as an alternative method of creating certain expressions.

For example, a sum that would normally be entered using the syntax for addition

```
> a + b + c + d;
```

$$a + b + c + d \tag{3.1}$$

can also be entered using the constructor `+`.

```
> `+`( a, b, c, d );
```

$$a + b + c + d \tag{3.2}$$

With some exceptions (for example, series, lists, sets, and procedures), the name of the constructor for an expression can be displayed by using the **op** command with its first argument equal to 0.

```
> op( 0, a + b + c + d );
```

$$`+` \tag{3.3}$$

The example above shows that the constructor for the expression a + b + c + d is the command assigned to the name `+`.

3.3 Names

Names have several purposes in Maple. They can be used to reference algebraic indeterminates, symbols, and variables in your code.

Names (page 20) provided a basic introduction to Maple names. The following section describes concepts related to names in more detail.

A Maple name can be either *global* or *local*, depending on its *scope*. In this chapter, only global names are used. A global name is created either by referring to it at the top level of your program or by declaring it to be global in either a procedure or module definition. For more information about scope, see *Variables in Procedures (page 217)*.

Two names are the same if they have the same spelling and scope. Maple keeps only one copy of any name in memory, so in a large expression that includes an indeterminate x, only one copy of the name x is kept in memory. Each occurrence of x in the expression refers to the same name x.

The polynomial

```
> x^3 - 3*x^2 + 3*x - 1;
```

$$x^3 - 3\,x^2 + 3\,x - 1 \tag{3.4}$$

contains three occurrences of the name x, but all three point to the same location in memory.

Maple is unique in that names can represent themselves. As a result, you can use names as algebraic indeterminates, for example, to construct polynomials or other algebraic expressions.

Names can also be used to represent variables in your code. When a name is *assigned* a value, that name is associated with another expression and evaluating the name results in its assigned value being returned. When a name is *unassigned*, evaluating the name results in the name itself.

In this example, the name a is assigned to the value 2.

```
> a := 2;
```

$$a := 2 \tag{3.5}$$

Before using a name on the left side of an assignment, the name has no assigned value.

```
> b;
```

$$b \tag{3.6}$$

When a value is assigned to a name, subsequent evaluation of the name results in its assigned value.

```
> a;
```

$$2 \tag{3.7}$$

For more information about assigning values, see *Assignments (page 170)*.

Creating Names: Lexical Conventions

When creating names in Maple, you must be aware of certain lexical conventions.

Environment Variables

Names beginning with an underscore character (_) are reserved for use by the Maple library. You should *not* create names that begin with an underscore.

As a special case, any name beginning with the four character sequence "_Env" is treated as an environment variable.

Environment variables are a special kind of variable in that an assignment to one within a procedure is automatically unassigned when the procedure has finished running. Therefore, environment variables only affect subprocedures called from that procedure, unless they are superseded locally.

The following predefined environment variables do not begin with **_Env**: **Testzero**, **UseHardwareFloats**, **Rounding**, **%**, **%%**, **%%%**, **Digits**, **_ans**, **index/newtable**, **mod**, **Order**, **printlevel**, **Normalizer**, **NumericEventHandlers**.

Environmental Variables Scope

Unlike a local variable, whose scope is restricted to the procedure containing the local variable itself, an environment variable can be referenced globally by all sub procedures called by or deeper than the current procedure, but the environment variable cannot be referenced by procedures above the current procedure.

For more information about environment variables, refer to the envvar help page. For more information about procedures, see *Procedures (page 195)* or refer to the procedure help page.

Constants

In addition to keywords, as described in *Reserved Words (page 16)*, Maple has several *predefined constants*.

You can display a sequence of all the names that represent *symbolic* constants in Maple by using the global variable **constants**.

```
> constants;
```

$$false, \gamma, \infty, true, Catalan, FAIL, \pi \tag{3.8}$$

```
> seq(i=evalf(i), i in constants);
```

$$\begin{aligned} &false = false, \gamma = 0.5772156649, \infty = Float(\infty), true \\ &= true, Catalan = 0.9159655942, FAIL = FAIL, \pi \\ &= 3.141592654 \end{aligned} \tag{3.9}$$

Maple also has several other special constants. **Table 3.1** lists some of them. For more information, refer to the initialconstants help page.

Table 3.1: Initially Known Names

Name	Meaning	Name	Meaning
lasterror	the most recent error	**constants**	initially known symbolic constants
libname	path of the Maple libraries	**Digits**	number of digits in floating-point computations
NULL	empty expression sequence	**FAIL**	cannot determine value
Order	truncation order for series	**printlevel**	control display of information

Name	Meaning	Name	Meaning
I	complex number	**undefined**	undefined numeric quantity

For more information about constants in Maple, refer to the constant help page.

Protected Names

A *protected name* has a predefined meaning; you cannot *directly* assign a value to a protected name. For example, the names of built-in commands such as **sin**; utility operations such as **degree**; commands such as **diff**; and type names such as **integer** and **list**, are protected names. An error occurs if you attempt to assign a value to any of these names.

```
> list := [1,2];
```

```
Error, attempting to assign to `list` which is protected.  Try
declaring `local list`; see ?protect for details.
```

The Maple system prevents these names from re-assignment. However, even though it is *not* recommended, it is possible to reassign values to these names by first unprotecting them as illustrated by the following statements.

Note: You can unassign values to Maple system names by entering a **restart** command or by ending your Maple session. In general, using the unprotect command to modify Maple system names is *not* recommended.

```
> unprotect(sin);
```

```
> sin := "a sin indeed";
```

$$\sin := \text{"a sin indeed"} \tag{3.10}$$

As a result, Maple components that rely on the sine function may not work as expected.

```
> plot( sin, 0..2*Pi, coords=polar );
```

To check whether a name is protected, use the type command.

```
> type(sin, protected);
```

$$\mathit{false} \tag{3.11}$$

```
> type(sine, protected);
```

$$\mathit{false} \tag{3.12}$$

To prevent values from being assigned to a name, use the **protect** command.

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

$$\mathit{mysqr} := x \rightarrow x^2 \tag{3.13}$$

```
> type(mysqr, protected);
```

$$false \tag{3.14}$$

```
> protect( mysqr );
> mysqr := 9;
```

```
Error, attempting to assign to `mysqr` which is protected.  Try
declaring `local mysqr`; see ?protect for details.
```

3.4 Unevaluated Expressions

In general, Maple evaluates all expressions immediately. In some situations, it is necessary to delay the evaluation of an expression. An expression enclosed in right single quotes is called an *unevaluated* expression. It takes the general form

```
'expr'
```

where *expr* is an arbitrary expression. All of the expressions contained within the right single quotes are not evaluated.

For example, the **sin** command normally performs the following computations.

```
> sin( 0.5 );
```

$$0.4794255386 \tag{3.15}$$

```
> sin( Pi / 2 );
```

$$1 \tag{3.16}$$

To prevent the evaluation of these computations, you can enclose the expressions in right single quotes (also called *unevaluation quotes*) as follows.

```
> 'sin( 0.5 )';
```

$$\sin(0.5) \tag{3.17}$$

```
> 'sin( Pi / 2 )';
```

$$\sin\left(\frac{1}{2}\,\pi\right) \tag{3.18}$$

You can enclose expressions of any length or complexity in unevaluation quotes.

```
> 'sin( 0.5 )^2 + cos( 0.5 )^2';
```

$$\sin(0.5)^2 + \cos(0.5)^2 \tag{3.19}$$

Also, you can enclose subexpressions in unevaluation quotes to prevent certain parts of an expression from evaluating.

```
> 'sin( 0.5 )'^2 + cos( 0.5 )^2;
```

$$\sin(0.5)^2 + 0.7701511530 \tag{3.20}$$

The sections below describe cases in which you may want to delay evaluation.

Protecting Names and Options

Unevaluation quotes can be used to prevent the evaluation of names.

```
> a := x^2 + x + 1;
```

$$a := x^2 + x + 1 \tag{3.21}$$

```
> a;
```

$$x^2 + x + 1 \tag{3.22}$$

```
> 'a';
```

$$a \tag{3.23}$$

This is important when you want to use a variable as a name, regardless of whether it has an assigned value.

Also, unevaluation quotes can be used to protect options. Names are often used as options to control the behavior of a command. If the name of that option has been used as a variable, the command that has been called uses the value of the variable and not the option name as expected. Unevaluation quotes can be used around option names to protect against this.

```
> periodic := 4;
```

$$periodic := 4 \tag{3.24}$$

```
> numtheory:-cfrac(3^(1/2),'periodic');
```

$$1 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{2 + \ldots}}}} \tag{3.25}$$

In the next example, an exception is raised because the name of a command option is not enclosed in unevaluation quotes.

```
> output := 2:
> CodeGeneration:-C( x^2, output = string );
Error, (in Translate) options [2 = string] not recognized
```

In this example, the best way to use the **output** option is to quote the name, thus preventing its evaluation in case the name **output** has an assigned value.

```
> CodeGeneration:-C( x^2, 'output' = 'string' );
```

```
"cg = x * x;
"
```
(3.26)

Tip: It is also recommended that you also use unevaluation quotes for the names of types and conversions. For more information, see *Structured Types (page 120)*.

For more information on types and conversions, refer to the type and convert help pages.

Generic Expressions

Expressions sometimes describe the operation to take place in a generic sense. For example, **B[i]** can be used in certain contexts with unevaluation quotes to denote a generic index into B. If unevaluation quotes are not used, Maple will try to look up the specific **i**th element of B.

```
> B := <1,2,3,4>;
```

$$B := \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \tag{3.27}$$

```
> sum(B[i], i = 1..4);
Error, bad index into Vector
> sum('B[i]', i = 1..4);
```

$$10 \tag{3.28}$$

Pass by Reference

Some commands accept a name as an argument, with the intent that it will be used to store a result. Unevaluation quotes ensure that the variable name (and not the value assigned to the variable) is used in the procedure.

```
> remainder := irem(45,3,'quotient'); quotient;
```

$$remainder := 0$$

$$15 \tag{3.29}$$

```
> remainder := irem(44,3,'quotient'); quotient;
```

$$remainder := 2$$

$$14 \tag{3.30}$$

If **quotient** is not enclosed in unevaluation quotes, the second call in the above example raises an exception because 15, the value of **quotient**, is not a valid third argument to the **irem** command.

Displaying the Original Command

For display purposes, it is sometimes useful to show the original command before a solution is computed.

```
> v := 'int(x*y^2, [x=0..1, y=0..1] )';
```

$$v := \int_0^1 \int_0^1 x\,y^2 \,dx\,dy \tag{3.31}$$

```
> v;
```

$$\frac{1}{6} \tag{3.32}$$

Unassigning Names

To reset the value of a name, assign the unevaluated name (its initial value) to the name. For example,

```
> x := 2+3;
```

$$x := 5 \tag{3.33}$$

```
> x := 'x';
```

$$x := x \tag{3.34}$$

Now, the value of **x** is reset to **x**.

Evaluation and Automatic Simplification

It is important to note the differences between computations that occur during the evaluation process and those that occur during the automatic simplification process. Unevaluation quotes do *not* prevent automatic simplifications from occurring. For example, basic numeric arithmetic is one form of automatic simplification. In the following expression, the unevaluation quotes do *not* prevent the numeric addition from occurring.

```
> '2 +3';
```

$$5 \tag{3.35}$$

In this example, Maple first simplifies the unevaluated sum '2 + 3' to the expression '5'. During the evaluation process, Maple "removes" the right single quotes and produces the numeric result **5**.

All unevaluated expressions are of the type **uneval**. You can use the **type** command to check whether an expression is an unevaluated expression.

```
> type( ''x'', 'uneval' );
```

$$\mathit{true} \tag{3.36}$$

In the example above, the first argument to the call to the **type** command is the name x, which is enclosed in two sets of unevaluation quotes. The result of evaluating the first argument is the unevaluated expression 'x' because the evaluation process removes one set of unevaluation quotes. The resulting expression is therefore of type **uneval**.

On the other hand, if you enclose the first argument to **type** in only one set of unevaluation quotes, the evaluation process removes the only set of unevaluation quotes, leaving the result as the name x, which is not an unevaluated expression.

```
> type( 'x', 'uneval' );
```

$$\mathit{false} \tag{3.37}$$

In other words, the **type** command accesses the name x, rather than the unevaluated expression 'x', since the **type** command accesses the result of its arguments that have been evaluated.

In the example above quotes were also used around the type name **uneval**. This provides a measure of protection just in case the variable name, **uneval** has an assigned value (which is unlikely because **uneval** is protected). During normal function evaluation, each argument, **x** and **uneval** is evaluated. With quotes, **'x'** becomes **x**, and **'uneval'** becomes **uneval** as seen by the **type** procedure. Without quotes, **x** would become the value of x (which may be the symbol **x** itself), and **uneval** would become the value of **uneval**, which is usually the symbol **uneval** itself. Unevaluation quotes make the displayed call robust against cases where the variable you are using unexpectedly has a value. It is rarely necessary to use this

level of caution in interactive use, but when you write programs, it is a good practice to include unevaluation quotes to make your code as robust as possible.

Another special case of unevaluation arises in function calls.

```
'f'(a)
```

Suppose **f** is not assigned to anything. Since evaluating **f** does not call a procedure, Maple returns the unevaluated function call **f(a)**.

```
> f(a);
```

$$f\left(x^2+x+1\right) \tag{3.38}$$

Similarly, using **uneval** quotes around a function evaluation will cause Maple to behave as if the named function had no value.

```
> ''sin''(Pi);
```

$$'\sin'(\pi) \tag{3.39}$$

```
> (3.39);
```

$$\sin(\pi) \tag{3.40}$$

```
> (3.40);
```

$$0 \tag{3.41}$$

You will find this facility useful when writing procedures that need to act on the whole original expression, not the evaluated result.

For more examples and information on unevaluated expressions, refer to the uneval help page.

Example: Defining a Procedure that Is Returned Unevaluated

You may need to use unevaluation quotes when you are defining a procedure that is returned unevaluated. This is necessary, for example, when you are defining a procedure that evaluates a numeric result for numeric inputs, but does not produce a numeric result otherwise. (The procedure may perform normalizations and apply symmetries, if appropriate.) It is important to write procedures using this method so that they can be plotted, optimized, or numerically integrated, for example.

Consider the following procedure.

```
> f := proc( x )
    if x > 2 then
      x
```

```
    else
      2
    end if
  end proc:
```

Using the wrong calling sequence in a call to plot results in an error.

```
> plot( f( x ), x = -10 .. 10 );
```

```
Error, (in f) cannot determine if this expression is true or false:
2 < x
```

The correct calling sequence would be either **plot('f'(x), x=-10..10)**, which puts uneval quotes around **f**, or **plot(f, -10..10)**, which avoids computing **f(x)** by omitting the variable altogether. Remember that arguments in a function call are evaluated first before the called procedure sees them.

Here, the precursor evaluation of **f(x)** tries to apply **f** to the unassigned symbol, **x**.

```
> f( x );
```

```
Error, (in f) cannot determine if this expression is true or false:
2 < x
```

The procedure could be rewritten so that it returns unevaluated whenever it encounters arguments that cannot be processed. This trick causes **f(x)** to evaluate to itself when non-numeric input is passed in.

```
> f := proc( x )
    if type( x, 'numeric' ) then
      if x > 0 then
        x
      else
        2
      end if
    else
      'procname( _passed )'
    end if
  end proc:
```

The unevaluated expression 'procname(_passed)' returns the full calling sequence unevaluated.

```
> f( x );
```

$$f(x) \tag{3.42}$$

The expression procname(_passed) must be enclosed in unevaluation quotes to prevent an infinite loop.

3.5 Numbers

Maple supports computation with exact numerical quantities, as well as approximate computation to arbitrarily high accuracy with floating-point numbers.

Integers

A *natural integer* is any sequence of one or more decimal digits.

```
> 12345;
```

$$12345 \tag{3.43}$$

The maximum number of digits is system-dependent. To determine the maximum number of digits, use the following command.

```
> kernelopts( 'maxdigits' );
```

$$38654705646 \tag{3.44}$$

A *signed integer* is formed by appending + or - before any natural integer.

```
> -42;
```

$$-42 \tag{3.45}$$

```
> +42;
```

$$42 \tag{3.46}$$

An *integer* is either a natural integer or a signed integer.

You can use the **length** command to determine the number of digits in an integer.

```
> 2^42;
```

$$4398046511104 \tag{3.47}$$

```
> length( 2^42 );
```

$$13 \tag{3.48}$$

Fractions

A rational number (fraction) is the quotient of two integers, where the denominator is always positive.

Use the division operator (forward slash) / to enter a fraction.

```
integer / natural
```

For example,

```
> 2 / 3;
```

$$\frac{2}{3} \tag{3.49}$$

You can enter a fraction in which the numerator and denominator have a common (integer) factor, but Maple automatically simplifies this to the lowest terms.

```
> 4 / 6;
```

$$\frac{2}{3} \tag{3.50}$$

In addition, Maple automatically moves a negative sign to the numerator.

```
> 2/(-3);
```

$$-\frac{2}{3} \tag{3.51}$$

Fractions are automatically simplified to an integer if the denominator is a divisor of the numerator.

```
> 6/3;
```

$$2 \tag{3.52}$$

You can use the **numer** and **denom** commands to extract the numerator and denominator, respectively, of a fraction.

```
> numer( 2/3 );
```

$$2 \tag{3.53}$$

```
> denom( 2/3 );
```

$$3 \tag{3.54}$$

Fractions can also be created by using the **Fraction** constructor with the numerator and denominator as arguments.

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

$$\frac{2}{3} \tag{3.55}$$

Floats

Maple supports computation with floating-point numbers to arbitrary precision.

A *float* can be input using a period for the decimal.

```
> 2.3;
```

$$2.3 \tag{3.56}$$

```
> 2.;
```

$$2. \tag{3.57}$$

```
> .7;
```

$$0.7 \tag{3.58}$$

```
> -.567;
```

$$-0.567 \tag{3.59}$$

Or, using exponent form using a suffix containing the letter "**e**" or "**E**" followed by an integer with no spaces between.

```
> 4e3;
```

$$4000. \tag{3.60}$$

```
> 2.3e6;
```

$$2.3\ 10^{6} \tag{3.61}$$

```
> .2E3;
```

$$200. \tag{3.62}$$

Observe that spaces are significant. The first example is a difference rather than a float in exponent form.

```
> .2e -3;
```

$$-2.8 \tag{3.63}$$

```
> .2e-3;
```

$$0.0002 \tag{3.64}$$

Also, the following is invalid.

```
> 3.e4;
```

Floats represent numbers of the form s*10^e, where the number s is called the *significand* or *mantissa* of the float, and the number e is called the *exponent*. The significand is a Maple integer. Therefore, it is restricted to values that have, at most, the number of digits indicated by the kernelopts('maxdigits') command.

```
> kernelopts( 'maxdigits' );
```

$$38654705646 \tag{3.65}$$

The maximum value of the exponent is a platform-dependent quantity whose value may be queried by using the **Maple_floats** command.

```
> Maple_floats( 'MAX_EXP' );
```

$$9223372036854775806 \tag{3.66}$$

Similarly, the minimum value of the exponent is given by the value

```
> Maple_floats( 'MIN_EXP' );
```

$$-9223372036854775806 \tag{3.67}$$

returned by the **Maple_floats** command. For more information, refer to the Maple_floats help page.

You can also create software floats by using the constructor **SFloat**. This constructor accepts the significand and exponent as arguments, and has the general form

```
SFloat( m, e )
```

```
> SFloat( 23, -1 );
```

$$2.3 \tag{3.68}$$

To extract the significand and exponent of a software float, use the **SFloatMantissa** and **SFloatExponent** commands.

```
> SFloatMantissa( 2.3 );
```

$$23 \tag{3.69}$$

```
> SFloatExponent( 2.3 );
```

$$-1 \tag{3.70}$$

The significand and exponent are also the operands of a software float.

```
> op( 2.3 );
```

$$23, -1 \tag{3.71}$$

Two software floats are equal if they represent the same number. However, equal floats by themselves do not need to be the same object in memory.

```
> evalb( 2.3 = 2.30 );
```

$$true \tag{3.72}$$

```
> addressof( 2.3 ); addressof( 2.30 );
```

$$1844679098624377 0078$$

$$1844679098624378 7774 \tag{3.73}$$

Observe that the significands (and therefore, also, the exponents) differ in this example.

```
> SFloatMantissa( 2.3 );
```

$$23 \tag{3.74}$$

```
> SFloatMantissa( 2.30 );
```

$$230 \tag{3.75}$$

Note that equal floats with different significands inside of two otherwise identical objects will require something stronger than **evalb** for comparison. **evalb** is the implicit comparison used when evaluating conditionals in **if** statements.

```
> evalb( 2.3 + x = 2.30 + x );
```

$$false \tag{3.76}$$

```
> evalb(<2.3,4.5> = <2.30,4.50>);
```

$$false \tag{3.77}$$

Testing the difference of the two expressions, or calling a command to do a deeper comparison may be necessary.

```
> evalb( (2.3 + x) - (2.30 + x) = 0 );
```

$$true \tag{3.78}$$

```
> EqualEntries(<2.3,4.5>, <2.30,4.50>);
```

$$true \tag{3.79}$$

The names of the constructor **SFloat** and accessors **SFloatMantissa** and **SFloatExponent** all begin with the letter **S**. The **S** stands for "software" because these floating-point numbers are implemented in software. Maple also supports the floating-point numbers supported by the underlying hardware, called *hardware floats* or *hfloats*. You can create a hardware float by using the hardware float constructor **HFloat**.

```
> HFloat( 24375, -3 );
```

$$24.3750000000000 \tag{3.80}$$

```
> h := HFloat( 24.375 );
```

$$h := 24.3750000000000 \tag{3.81}$$

```
> op( h );
```

$$243750000000000000, -16 \tag{3.82}$$

Note, however, that **hfloats** are binary floating-point numbers, rather than decimal floating-point numbers. That means that unlike the example above, there is often round-off error when decimal numbers are converted into **hfloats**. For more information, see *Hardware Floating-Point Numbers (page 272)*.

```
> op( HFloat(2.3) );
```

$$229999999999999982, -17 \tag{3.83}$$

The **SFloatMantissa** and **SFloatExponent** commands also accept hardware floats as input.

```
> SFloatMantissa( h );
```

$$243750000000000000 \tag{3.84}$$

```
> SFloatExponent( h );
```

$$-16 \tag{3.85}$$

For more information on floating-point numbers, see *Floating-Point Numbers (page 270)*.

Complex Numbers

Maple supports arithmetic with complex numbers of the form $a + bi$, where $i = \sqrt{-1}$ is the imaginary unit. In Maple, the imaginary unit is normally denoted by I; that is, the upper-case letter "I" is used rather than the lowercase "i". Therefore, the complex number with the real part equal to 2 and imaginary part equal to 3 is entered, naturally, as follows.

```
> 2 + 3*I;
```

$$2 + 3\,I \tag{3.86}$$

In general, a complex number has the form

```
re + im * I
```

where *re* and *im* are the real and imaginary parts of the complex number, respectively. If the expressions *re* and *im* are of type **extended_numeric**; the resulting complex number will be of type complex(extended_numeric). (It is not necessary that *re* and *im* are reals; they may be arbitrary algebraic expressions. However, in this case, the result of the syntax above will generally be an algebraic expression that will not be a complex numeric constant.)

You can also create complex numbers using the **Complex** constructor. It can be called using either one or two arguments. The single-argument form has the following syntax.

```
Complex( expr )
```

If the argument *expr* is of type **complex**, the **Complex** constructor returns the value of *expr*. Otherwise, if *expr* is of type **extended_numeric**, the **Complex** constructor returns expr * I.

```
> Complex( 2 ), Complex( 0 ), Complex( 0.0 );
```

$$2\,\mathrm{I},\ 0,\ 0.\,\mathrm{I} \tag{3.87}$$

```
> Complex( 2 + 3*I ), Complex( infinity ), Complex( undefined );
```

$$2 + 3\,\mathrm{I},\ \infty\,\mathrm{I},\ \mathit{undefined}\,\mathrm{I} \tag{3.88}$$

The two-argument form has the following syntax.

```
Complex( re, im )
```

The first argument is interpreted as the real part and the second argument is interpreted as the imaginary part, of the complex number constructed.

```
> Complex( 2, 3 ), Complex( 2.1, 3 ), Complex( 0, 0 );
```

$$2 + 3\,\mathrm{I},\ 2.1 + 3.\,\mathrm{I},\ 0 \tag{3.89}$$

Note that if either of the arguments is a float, the real and imaginary parts of the complex number created are *both* of type float.

A complex zero with floating-point real and imaginary components can have four sign combinations.

```
> z1 := 0.0 + 0.0*I; z2 := 0.0 - 0.0*I;
  z3 := -0.0 - 0.0*I; z4 := -0.0 + 0.0*I;
```

$$z1 := 0. + 0.\,\mathrm{I}$$

$$z2 := 0. - 0.\,\mathrm{I}$$

$$z3 := -0. - 0.\,\mathrm{I}$$

$$z4 := -0. + 0.\,\mathrm{I} \tag{3.90}$$

Similar to 0.0 = -0.0, numerically, these four complex zeros are numerically equivalent.

```
> evalb( z1 = z2 and z2 = z3 and z3 = z4 );
```

$$\mathit{true} \tag{3.91}$$

If the arguments *re* and *im* are not of type **extended_numeric**, the **Complex** constructor is returned unevaluated.

```
> Complex( u, v );
```

$$Complex(u, v) \tag{3.92}$$

Except if one of the arguments is complex, in which case, an exception is raised.

```
> Complex( 2 + 3*I, 1 );
```

```
Error, invalid arguments for Complex constructor
```

It is important to understand that there is a single complex infinity, which is a point on the Riemann sphere. It can be denoted in different ways:

```
> inf1 := infinity + infinity * I; inf2 := infinity - infinity * I;
  inf3 := -infinity - infinity * I; inf4 := -infinity + infinity *
   I;
```

$$inf1 := \infty + \infty\, I$$

$$inf2 := \infty - \infty\, I$$

$$inf3 := -\infty - \infty\, I$$

$$inf4 := -\infty + \infty\, I \tag{3.93}$$

However, all of these forms are numerically equivalent.

```
> evalb( inf1 = inf2 and inf2 = inf3 and inf3 = inf4 );
```

$$true \tag{3.94}$$

They are all treated as distinct from the positive and negative real infinities.

To select the real or imaginary part of a complex number, use the **Re** and **Im** commands, respectively.

```
> Re( 2.3 + sqrt(2)*I );
```

$$2.3 \tag{3.95}$$

```
> Im( 2.3 + sqrt(2)*I );
```

$$\sqrt{2} \tag{3.96}$$

Note that, for a symbolic expression of the form a + b*I, it is *not* assumed that **a** is the real part and **b** is the imaginary part. Therefore, the **Re** and **Im** commands are not unevaluated on such input.

```
> Re( a + b*I );
```

$$\Re(a + \mathrm{I}\, b) \tag{3.97}$$

```
> Im( a + b*I );
```

$$\Im(a + \mathrm{I}\, b) \tag{3.98}$$

However, the **evalc** command uses special rules for processing complex expressions, in which any unknown symbol is assumed to be real. Therefore, when the **evalc** is used, these expressions are returned as follows.

```
> evalc( Re( a + b*I ) );
```

$$a \tag{3.99}$$

```
> evalc( Im( a + b*I ) );
```

$$b \tag{3.100}$$

For more information, refer to the evalc help page.

You can change the default name used to input and display the imaginary unit by using the **interface** command.

```
> interface( 'imaginaryunit' = i );
```

$$I \tag{3.101}$$

(The previous value is returned.) After calling the command above, the name **i** is used to represent the imaginary unit.

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

$$2 + 3\,\mathrm{I} \tag{3.102}$$

When this command is used, the name **i** can no longer be used as a program variable. As an example, the following statements display error messages.

```
> i := 2;
Error, illegal use of an object as a name
> add( i^2, i = 1 .. 5 );
Error, illegal use of an object as a name
```

To restore the default imaginary unit, use the following command.

```
> interface( 'imaginaryunit' = I );
```

$$i \tag{3.103}$$

3.6 Indexed Expressions

Indexed expressions represent *selection operations*. The general form of an indexed expression is

```
expr [ index ]
```

where *expr* is an arbitrary expression and *index* represents a sequence of expressions. The following are examples of indexed expressions.

```
> 2[ 3, 4 ];
```

$$2_{3,\,4} \tag{3.104}$$

```
> a[];
```

$$a_{[\,]} \tag{3.105}$$

```
> a[ 1 ];
```

$$a_{1} \tag{3.106}$$

```
> a[ b ];
```

$$a_{b} \tag{3.107}$$

```
> a[ b, c ];
```

$$a_{b,\,c} \tag{3.108}$$

```
> map[ 2 ];
```

$$map_{2} \tag{3.109}$$

```
> [ 1, 2, 3 ][ 2 ..3 ][ 1 ];
```

$$2 \tag{3.110}$$

Note that the last example above contains a nested (or iterated) indexed expression.

The constructor for indexed expressions is the name **?[]**.

```
> `?[]`( S, [ a, b, c ] );
```

$$S_{a,\,b,\,c} \tag{3.111}$$

Note that the indices must be enclosed with square brackets in a list.

All or some of the elements of an index sequence can be extracted by using the **op** command. The **nops** command will tell you how many elements are in the index sequence.

```
> nops( a[ b, c, d ] );
```

$$3 \tag{3.112}$$

```
> op( a[ b, c, d] );
```

$$b, c, d \tag{3.113}$$

```
> op( 2, a[ b, c, d ] );
```

$$c \tag{3.114}$$

```
> op( 2..3, a[ b, c, d ] );
```

$$c, d \tag{3.115}$$

Indexed expressions are often used to perform selection operations. The behavior of a selection operation depends on the type of expression, *expr*, and the index sequence given.

If *expr* is itself a sequence of expressions, the index sequence must evaluate to a positive integer, an integral range, or the empty sequence. The following are all examples of valid ways to index a sequence.

```
> expr := (1,2,3,4);
```

$$expr := 1, 2, 3, 4 \tag{3.116}$$

```
> expr[ 3 ];
```

$$3 \tag{3.117}$$

```
> expr[ 1 .. 3 ];
```

$$1, 2, 3 \tag{3.118}$$

```
> expr[];
```

$$1, 2, 3, 4 \tag{3.119}$$

```
> expr[ 2 .. 1 ];
```

The result of evaluating an indexed sequence is a selection of the components of the sequence. The indexing sequence must represent a valid index or range of indices. Attempting to select an entry beyond the length of the sequence and will raise an error.

```
> expr[ 88 ];
Error, invalid subscript selector
```

Similarly, components of lists, sets, arrays, matrices, and vectors can be selected

```
> L := [1,2,3,4];
```

$$L := [1, 2, 3, 4] \tag{3.120}$$

```
> L[ 3 ];
```

$$3 \tag{3.121}$$

```
> L[ 1 .. 3 ];
```

$$[1, 2, 3] \tag{3.122}$$

```
> L[];
```

$$1, 2, 3, 4 \tag{3.123}$$

```
> M := <1,2,3;4,5,6>;
```

$$M := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{3.124}$$

```
> M[2,3];
```

$$6 \tag{3.125}$$

```
> M[1..2,1..2];
```

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} \tag{3.126}$$

```
> S := { red, blue, green, orange };
```

$$S := \{blue, green, orange, red\} \tag{3.127}$$

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

$$orange \tag{3.128}$$

Note that, because sets are sorted data structures, the order at construction time may not match the order stored internally. It is not predictable what color will be returned by the index used to specify the third entry above. (It may not be **green**.)

A negative number may be used as an index, which selects elements starting from the end of the list. Positive and negative indices mixed in a range return an empty selection.

```
> L[ -1 ];
```

$$4 \tag{3.129}$$

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

$$[2, 3] \tag{3.130}$$

```
> L[ -3 .. 1 ];
```

$$[\,] \tag{3.131}$$

Lists can be used as an index to pick out specific entries, such as the first and third entries of a list, or the four corners of a matrix.

```
> L[ [1,3] ];
```

$$[1, 3] \tag{3.132}$$

```
> M[[1,2],[1,3]];
```

$$\begin{bmatrix} 1 & 3 \\ 4 & 6 \end{bmatrix} \tag{3.133}$$

Indexing on arrays, matrices and vectors is very flexible. In the case of these data structures, round-brackets can also be used to index in a way that is useful to programming. For example, where **M[1]** will return the first row of the matrix, **M(1)** will return the first entry (regardless of the number of dimensions).

```
> M[1];
```

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \tag{3.134}$$

```
> M(1);
```

$$1 \tag{3.135}$$

This class of data structures are known as rectangular tables, or "rtables" for short. For more information on what ways they can be indexed, refer to the rtable_indexing help page.

If *expr* is a name with no assigned value, the result of evaluating the indexed expression is an indexed name. In this case, the index can be any sequence of expressions, and if desired, it is up to your program to define the meaning of the expression.

```
> aName[ x^2 - 3*x, "a string", anotherName[ 2, b ] ];
```

$$aName_{x^2 - 3\,x,\ \text{"a string"},\ anotherName_{2,\ b}} \tag{3.136}$$

A string may be indexed by a positive integer, a positive integral range, or a general sequence. The indexed string expression evaluates to itself, unless the indexing sequence is an integer or integral range, in which case, the result is a substring of the indexed string.

```
> "abcde"[ 3 ];
```

$$\text{"c"} \tag{3.137}$$

```
> "abcde"[ 2 .. 4 ];
```

$$\text{"bcd"} \tag{3.138}$$

```
> "abcde"[ u, v^2 - s*t ];
```

$$\text{"abcde"}_{u,\ -s\,t+v^2} \tag{3.139}$$

```
> "abcde"[];
```

$$\text{"abcde"}_{[\,]} \tag{3.140}$$

If *expr* evaluates to a table, and if the index given is found in the table the expression evaluates to the corresponding entry. Otherwise, the indexed expression evaluates to itself.

```
> t := table( [ a = 1, b = 2, (c,d) = 3 ] );
```

$$t := table([b = 2, (c, d) = 3, a = 1]) \tag{3.141}$$

```
> t[ a ];
```

$$1 \tag{3.142}$$

```
> t[ c, d ];
```

$$3 \tag{3.143}$$

```
> t[ u, v ];
```

$$t_{u,\ v} \tag{3.144}$$

If *expr* evaluates to a module, the index must evaluate to the name of an export of the module, and then the entire indexed expression evaluates to the value of expr:-index.

```
> m := module() export e, f := 2; end module:
> m[ e ];
```

$$e \tag{3.145}$$

```
> evalb( e = m[ e ] );
```

$$false \tag{3.146}$$

```
> m[ f ];
```

$$2 \tag{3.147}$$

For more information about modules, see *Programming with Modules (page 307)*.

3.7 Member Selection

The member selection operator **:-** is used to select exports of a module, and also to designate a symbol as a global symbol. Member selection expressions have one of the following general forms.

modexpr :- *expname*
:- *name*

The first form above is used to select a member of a module.

```
> m := module() export e, f:= 2; end module:
> m:-e;
```

$$e \tag{3.148}$$

```
> evalb( e = m:-e );
```

$$false \tag{3.149}$$

```
> m:-f;
```

$$2 \tag{3.150}$$

The first operand, *modexpr*, must evaluate to a module. The second operand, *expname*, must be a literal name; it is *not* evaluated. If *expname* is not a name, or is not the name of an export of the module *modexpr*, an exception is raised. The syntax **m:-e** is similar to **m[e]**, in that they both evaluate module **m**'s export **e**. The difference is that the index selection form will evaluate **e** before resolving the export.

In the second form, the operand *name* must be a literal name. The expression :-name then evaluates to the *global instance* of the name *name*.

The following example defines, and then immediately calls, a procedure which declares a local variable t. Since this local variable is never assigned, it evaluates to itself. The call to the **evalb** command then compares, on the left-hand side of the equation, the local name t to the global name t resulting from applying the member selection operator to t. The result is **false** because the global name t and the name t local to the procedure are different expressions.

```
> proc() local t; evalb( t = :-t ) end proc();
```

$$false \tag{3.151}$$

For more information on modules and member selection, see *Programming with Modules (page 307)*.

3.8 Functions

A *function* expression is a Maple expression of the form

```
expr( sequence )
```

that contains zero or more expressions in the sequence within the parentheses. It represents a function call.

```
> F();
```

$$F() \tag{3.152}$$

```
> F( x );
```

$$F(x) \tag{3.153}$$

```
> F( x, y );
```

$$F(x, y) \tag{3.154}$$

```
> sin( x + y );
```

$$\sin(x + y) \tag{3.155}$$

Typically, **expr** is the name of a procedure or mathematical function. It can be a general expression.

The zeroth operand of a function expression is **expr**.

```
> op( 0, F( x, y, z ) );
```

$$F \tag{3.156}$$

The other operands are the *arguments*,

```
> op( F( x, y, z ) );
```

$$x, y, z \tag{3.157}$$

and the number of operands is the number of arguments.

```
> nops( F( x, y, z ) );
```

$$3 \tag{3.158}$$

```
> nops( F() );
```

$$0 \tag{3.159}$$

Maple supports an algebra of operators, so that complicated expressions such as

```
> (f^2 + g@h - 2)( x );
```

$$f(x)^2 + g(h(x)) - 2 \tag{3.160}$$

can be formed. Note that Maple applies such "operator expressions" to the arguments. @ is the composition operator. For more information on composition of functions, see *Composition (page 105)*.

It is important to know that Maple computes numeric quantities as applicable operators with constant values. Therefore, for example, the expression

```
> 2( x );
```

$$2 \tag{3.161}$$

is computed as an application of the constant operator 2 to the argument x, which evaluates to 2. In fact, numeric "operators" can accept any number of arguments.

```
> 2( x, y, 3 );
```

$$2 \tag{3.162}$$

Note that an expression such as

```
> '2( 3 )';
```

$$2(3) \tag{3.163}$$

(in which unevaluation quotes are used to delay the evaluation process) appears to be a product. However, this expression is, in fact, a function expression. When permitted to evaluate fully, the result is the constant value of the operator.

```
> 2( 3 );
```

$$2 \tag{3.164}$$

Calls to Procedures

The most important kind of function expression to understand is the case in which the zeroth operands is a procedure or, more commonly, an expression (typically, as a name) that evaluates to a procedure.

```
p( arg1, arg2, ..., argN )
```

In this case, *p* is a procedure or an expression, such as a name, that evaluates to a procedure, and *arg1*, *arg2*, ..., *argN* are zero or more argument expressions.

For example, the name **sin** evaluates to a procedure that computes the mathematical **sin** function. A function expression of the form

```
sin( expr )
```

computes the **sin** of its argument **expr**. This is performed as follows: Maple evaluates the name **sin** and finds that it is assigned a procedure. The argument *expr* is evaluated to produce a result. That result is then passed to the procedure assigned to the name **sin** and the result computed by that procedure for the specific input is returned as the overall value of the function call sin(expr).

For information on defining functions and procedures, see *Functional Operators (page 104)* and *Procedures (page 195)*.

3.9 Arithmetic Expressions

Arithmetic Operators

The arithmetic operators in Maple include + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation). These operators are used to create rational expressions, such as polynomials.

```
> x^2 - 3*x + 1;
```

$$x^2 - 3\,x + 1 \tag{3.165}$$

Addition and Subtraction

The addition operator `+` and the subtraction operator `-` are typically used as binary infix operators, but may also be used as unary prefix operators to indicate a signed expression.

```
> a + b + 3;
```

$$a + b + 3 \tag{3.166}$$

```
> u - v;
```

$$u - v \tag{3.167}$$

```
> +7;
```

$$7 \tag{3.168}$$

```
> -42;
```

$$-42 \tag{3.169}$$

A sum resulting from the evaluation of either an addition or subtraction operation is an expression of type `+`.

```
> type( u - v, '`+`' );
```

$$true \tag{3.170}$$

The expression u-v has the operands u and -v; that is, it is a sum of the summands u and -v.

```
> op( u - v );
```

$$u, -v \tag{3.171}$$

Note that subtraction is not an associative operator.

```
> ( 1 - 2 ) - 3 <> 1 - ( 2 - 3 );
```

$$-4 \neq 2 \tag{3.172}$$

However, addition is both associative and commutative:

```
> b + a + c = a + b + c;
```

$$b + a + c = b + a + c \tag{3.173}$$

Although sums are formed by using the binary operator `+`, they are actually expressions of arbitrarily large arity (greater than unity). Since addition is associative, Maple "flattens" all sums of more than two summands during the automatic simplification process. Therefore, an expression of type `+` can have many operands.

```
> nops( a + b + c + d + e );
```

$$5 \tag{3.174}$$

You can use the name `+` as a constructor to form a sum.

```
> `+`( a, b, c );
```

$$b + a + c \tag{3.175}$$

Since Maple performs automatic simplification, the number of operands of a sum may not be apparent from the input.

```
> nops( a + 2 + b + 3 + c + 4 );
```

$$4 \tag{3.176}$$

In this example, Maple combines the integer terms in the sum.

```
> a + 2 + b + 3 + c + 4;
```

$$a + 9 + b + c \tag{3.177}$$

To see that this occurs during the automatic simplification process, enclose the input in unevaluation quotes to delay evaluation.

```
> 'a + 2 + b + 3 + c + 4';
```

$$a + 9 + b + c \tag{3.178}$$

In a sum such as

```
> '2 + 3';
```

$$5 \tag{3.179}$$

the addition is performed, as indicated, during the automatic simplification process. The same sum can be computed in another way:

```
> u := 3:
> '2 + u';
```

$$2 + u \tag{3.180}$$

In this example, the arithmetic is not performed because the value of the variable u does not replace the name u during the automatic simplification process. If the unevaluation quotes are removed to allow the full evaluation of the expression, numeric addition is performed.

```
> 2 + u;
```

$$5 \tag{3.181}$$

Since addition is commutative, the order of summands in an expression of type `+` is arbitrary. It is fixed within a Maple session, but may vary from one session to another. Therefore, you must not rely on the operands of a sum occurring in any specific order.

Operands of a sum are automatically simplified, recursively.

```
> '2/3 + sin( 5*Pi/6 - 2*Pi/3  )';
```

$$\frac{2}{3} + \sin\left(\frac{1}{6}\,\pi\right) \tag{3.182}$$

Since procedures are not called during the automatic simplification process, the example above does not fully simplify to the result

```
> 2/3 + sin( 5*Pi/6 - 2*Pi/3  );
```

$$\frac{7}{6} \tag{3.183}$$

during the automatic simplification process. However, the argument to the **sin** command is computed to the simpler form $\frac{1}{6}\pi$, just as it would if it had been entered by itself.

```
> '5*Pi/6 - 2*Pi/3';
```

$$\frac{1}{6}\pi \tag{3.184}$$

If any numeric literal in a sum is a float, all the numeric operands are converted to floats and their sum is computed as a float. For more information, see *Floating-Point Contagion (page 279)*.

```
> 'a + 2 + b + 3.7 + c + Pi';
```

$$a + 5.7 + b + c + \pi \tag{3.185}$$

Arithmetic computed during the automatic simplification process includes arithmetic with values of infinity, undefined values, and signed (floating-point) zeroes.

```
> '2.3 + undefined';
```

$$Float(undefined) \tag{3.186}$$

```
> '2.3 + infinity';
```

$$Float(\infty) \tag{3.187}$$

```
> '-0.0 + 0';
```

$$-0. \tag{3.188}$$

```
> 'infinity - infinity';
```

$$undefined \tag{3.189}$$

```
> 'infinity - Float(infinity)';
```

$$Float(undefined) \tag{3.190}$$

Sums of non-algebraic summands can be formed. A sum of lists of the same length returns the corresponding list of sums. This occurs during the automatic simplification process.

```
> '[ a, b, c ] + [ x, y, z ]';
```

$$[x + a, y + b, z + c] \tag{3.191}$$

Sums of arrays, matrices, and vectors occur during the regular evaluation process.

```
> <1,2;3,4> + <5,6;7,8>;
```

$$\begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix} \tag{3.192}$$

Attempting to add lists or matrices of different sizes results in an error.

```
> [ 1, 2 ] + [ 1, 2, 3 ];
```

```
Error, adding lists of different length
```

```
> <1,2;3,4> + <1,2>;
```

```
Error, (in `rtable/Sum`) invalid arguments
```

Since the addition of sets (which are not ordered) is not well-defined, a sum formed with a set is returned unevaluated.

```
> { 1, 2 } + { 3, 4 };
  { 1, 2 } + [ 3, 4 ];
```

$$\{1, 2\} + \{3, 4\}$$
$$\{1, 2\} + [3, 4] \tag{3.193}$$

Multiplication and Division

Products are formed by using the `*` and `/` operators. The result of evaluating either a multiplication or division operation is an expression of type `*`.

```
> type( a * b, '`*`' );
  type( a / b, '`*`' );
```

$$true$$
$$true \tag{3.194}$$

You can use the **dismantle** command to print a representation of the internal structure of any Maple expression.

```
> dismantle( a / b );
```

```
PROD(5)
   NAME(4): a
   INTPOS(2): 1
   NAME(4): b
   INTNEG(2): -1
```

The output shows that the quotient is actually stored as a product of two factors: one consisting of the expression a with a power of 1 and the other consisting of the expression b with a power of -1: $a^{(1)}\ b^{(-1)}$.

Similar to sums, products are commutative and associative. Also, products are flattened due to associativity, even though the `*` operator is binary. Automatic simplification is applied to products, so as with sums, numeric factors are automatically combined.

```
> '2 * 3 * x * y';
```

$$6\,x\,y \tag{3.195}$$

Also like sums, the order of factors in an expression of type `*` is arbitrary, and may vary between Maple sessions.

```
> 'y * x * 3 * 2';
```

$$6\,x\,y \tag{3.196}$$

The number of operands reflects the number of factors remaining after automatic simplification has taken place.

```
> nops( 2 * 3 * x * y );
```

$$3 \tag{3.197}$$

```
> op( 2 * 3 * x * y );
```

$$6, x, y \tag{3.198}$$

The name `*` can be used as a constructor to form products.

```
> `*`( a, b, c );
```

$$a\,b\,c \tag{3.199}$$

If any numeric constant in a product is a float, the result of gathering all of the constants into a single factor is a float.

```
> '3.1 * a / 2 / b * 4';
```

$$\frac{6.200000000\ a}{b} \tag{3.200}$$

```
> '2.3 * ( 5*Pi/6 - 2*Pi/3  )';
```

$$0.3833333333\ \pi \tag{3.201}$$

This effect does not extend into function calls.

```
> '2.3 * sin( 5*Pi/6 - 2*Pi/3  )';
```

$$2.3 \sin\left(\frac{1}{6}\,\pi\right) \tag{3.202}$$

You can multiply a list by a number and the product is applied to all of the list elements during the automatic simplification process.

```
> '2 * [ 2, 3 ]';
```

$$[4, 6] \tag{3.203}$$

Matrix multiplication is done with the `.` operator rather than `*`. Division is not defined for matrices.

```
> <1,2;3,4> . <5,6;7,8>;
```

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix} \tag{3.204}$$

```
> <1,2;3,4> . LinearAlgebra:-MatrixInverse( <5,6;7,8>);
```

$$\begin{bmatrix} 3 & -2 \\ 2 & -1 \end{bmatrix} \tag{3.205}$$

Multiplying or dividing two arrays of the same size will perform paired element-wise operations on the individual entries. The element-wise operators *~ and /~ can be used on both arrays and matrices to achieve the same result.

```
> Array([[1,2],[3,4]]) * Array([[5,6],[7,8]]);
```

$$\begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix} \tag{3.206}$$

```
> Array([[1,2],[3,4]]) / Array([[5,6],[7,8]]);
```

$$\begin{bmatrix} \frac{1}{5} & \frac{1}{3} \\ \frac{3}{7} & \frac{1}{2} \end{bmatrix} \tag{3.207}$$

```
> <1,2;3,4> /~ <5,6;7,8>;
```

$$\begin{bmatrix} \frac{1}{5} & \frac{1}{3} \\ \frac{3}{7} & \frac{1}{2} \end{bmatrix} \tag{3.208}$$

```
> <1,2;3,4> *~ <5,6;7,8>;
```

$$\begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix} \tag{3.209}$$

For more information on element-wise operators, see *Programming-Language Operators (page 17)*.

Exponentiation

Powers are formed by using the `^` operator.

```
> a^b;
```

$$a^b \tag{3.210}$$

It is strictly a binary operator; nested powers must be written with parentheses.

```
> (a^b)^c;
```

$$\left(a^b\right)^c \tag{3.211}$$

```
> a^(b^c);
```

$$a^{b^c} \tag{3.212}$$

The following input results in a syntax error.

```
> a^b^c;
```

Rational number powers are used to represent roots. Exact roots are left uncomputed, while floating-point roots are computed during the automatic simplification process.

```
> 4^(1/2);
```

$$\sqrt{4} \tag{3.213}$$

```
> '(2.1)^(1/3)';
```

$$1.280579165 \tag{3.214}$$

Expressions to a power of 0 are reduced to unity during the automatic simplification process. The type of the resulting 1 depends on the type of the zero power, unless the base of the expression is a float, in which case the result is a float.

```
> 'a ^ 0';
```

$$1 \tag{3.215}$$

```
> 'a ^ 0.0';
```

$$1.0 \tag{3.216}$$

```
> '(x^2 - 1 + 3)^0';
```

$$1 \tag{3.217}$$

There are some exceptions when infinity and undefined values are raised to a float zero power.

```
> 'Float( undefined ) ^ 0.0';
```

$$Float(undefined) + Float(undefined)\,\mathrm{I} \tag{3.218}$$

```
> 'Float( infinity ) ^ 0.0';
```

$$Float(undefined) \tag{3.219}$$

```
> 'Float( -infinity ) ^ (-0.0)';
```

$$Float(undefined) - 0.\,\mathrm{I} \tag{3.220}$$

Note the distinction between Float(-infinity) ^ (-0.0) and -Float(infinity) ^ (-0.0): the latter is first automatically simplified to - Float(undefined) and then to Float(undefined).

In Maple, the indeterminate form 0^0 with an exact base is interpreted as 1.

```
> 0^0;
```

$$1 \tag{3.221}$$

```
> 0.0 ^ 0;
```

$$1. \tag{3.222}$$

```
> 0 ^ 0.0;
```

$$Float(undefined) \tag{3.223}$$

Although a complex floating-point zero does not automatically simplify to a real zero, expressions raised to a complex zero are simplified automatically to an exact or floating-point unity.

```
> a^(0.0 + 0.0*I);
```

$$1 \tag{3.224}$$

Powering of matrices is done in the mathematical sense achieving repeated matrix products. Powering of arrays is done element-wise.

```
> <1,2;3,4> ^3;
```

$$\begin{bmatrix} 37 & 54 \\ 81 & 118 \end{bmatrix} \tag{3.225}$$

```
> Array([[1,2],[3,4]]) ^3;
```

$$\begin{bmatrix} 1 & 8 \\ 27 & 64 \end{bmatrix} \tag{3.226}$$

Rational Expressions

Using sums and products, more complicated expressions can be formed.

```
> expr := ( a + a*b ) / ( a*b - b );
```

$$expr := \frac{a\,b + a}{a\,b - b} \tag{3.227}$$

Conceptually, Maple creates the following structure.

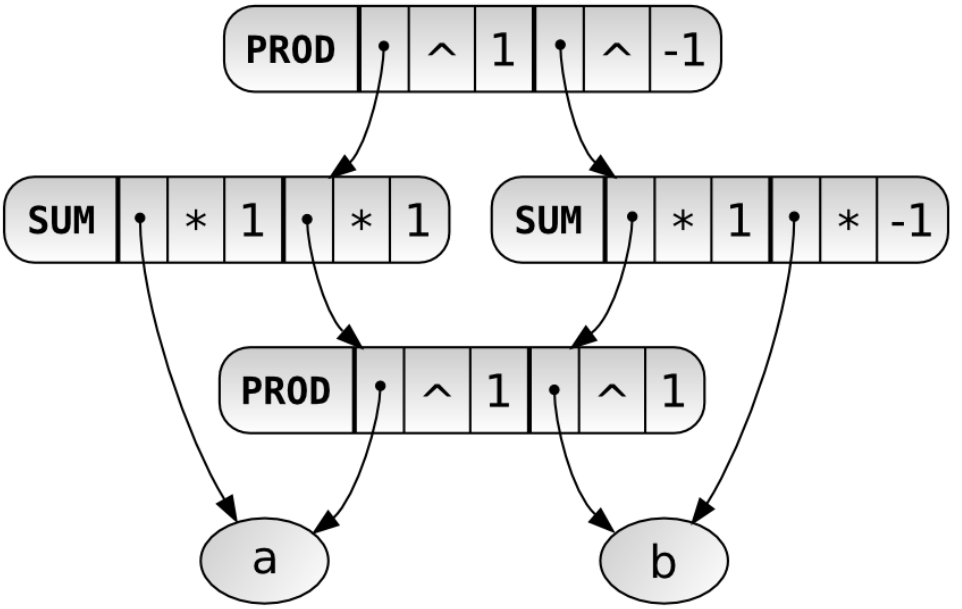


Figure 3.1: expr DAG

Here, *expr* is a product of two operands

```
> nops( expr );
```

$$2 \tag{3.228}$$

```
> op( expr );
```

$$a\,b + a,\ \frac{1}{a\,b - b} \tag{3.229}$$

and each operand is itself an expression with two operands.

```
> e1, e2 := op( expr );
```

$$e1,\ e2 := a\,b + a,\ \frac{1}{a\,b - b} \tag{3.230}$$

```
> nops( e1 ); nops( e2 );
```

$$2$$

$$2 \tag{3.231}$$

Maple does *not* automatically simplify the following expression. To perform such simplifications, use the **normal** command.

```
> expr := (x - 1)/(x^2 - 1);
```

$$expr := \frac{x-1}{x^2-1} \tag{3.232}$$

```
> normal( expr );
```

$$\frac{1}{x+1} \tag{3.233}$$

The normal command only performs normalization of rational expressions with rational coefficients.

```
> expr := ( (sin(t)^2 + cos(t)^2)*(x - 1)/(x^2 - 1));
```

$$expr := \frac{\left(\sin(t)^2+\cos(t)^2\right)(x-1)}{x^2-1} \tag{3.234}$$

```
> normal( expr );
```

$$\frac{\sin(t)^2+\cos(t)^2}{x+1} \tag{3.235}$$

Note: Use the **simplify** command to apply more powerful simplifications.

Maple also does not automatically expand the products of sums.

```
> (a + b) * (c + d);
```

$$(a+b)(c+d) \tag{3.236}$$

Use the **expand** command (or the **normal** command, with the **expanded** option) to perform such expansions.

```
> expr := (a + b) * (c + d);
```

$$expr := (a+b)(c+d) \tag{3.237}$$

```
> expand( expr );
```

$$a\,c + a\,d + b\,c + b\,d \tag{3.238}$$

```
> normal( expr, 'expanded' );
```

$$a\,c + a\,d + b\,c + b\,d \tag{3.239}$$

Similarly, you must use the **normal** command to simplify the following rational expression.

```
> expr2 := expand( expr ) / (a + b);
```

$$expr2 := \frac{a\,c + a\,d + b\,c + b\,d}{a + b} \tag{3.240}$$

```
> normal( expr2 );
```

$$c + d \tag{3.241}$$

Noncommutative Multiplication

Noncommutative multiplication is represented by the dot operator (.), which is used mainly in linear algebra computations for multiplication of matrices and vectors. It may also be used to represent the noncommutative product of other types of mathematical expressions.

```
A . B;
```

If **A** and **B** are of type **constant**, then A . B = A * B during the evaluation process (but not during the automatic simplification process). However, if one of **A** and **B** is a Matrix or a Vector, and the other is a Matrix, Vector, or constant, the product is interpreted as a matrix or vector product. If **A** or **B** is an Array (and the other is not a Matrix or Vector), then A . B is interpreted as *element-wise* multiplication. For arguments that are not of type Matrix, Vector, or constant, **A . B** remains unevaluated, but more importantly, it is not automatically simplified to or interpreted as being equal to **B . A**.

```
> 7 . 6;
```

$$42 \tag{3.242}$$

```
> '7 . 6';
```

$$7 \,.\, 6 \tag{3.243}$$

```
> A.B <> B.A;
```

$$A \,.\, B \neq B \,.\, A \tag{3.244}$$

```
> M:=<<1,0,2>|<0,1,2>|<0,0,2>>;
```

$$M := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 2 & 2 \end{bmatrix} \tag{3.245}$$

```
> V:=<10,0,0>;
```

$$V := \begin{bmatrix} 10 \\ 0 \\ 0 \end{bmatrix} \tag{3.246}$$

```
> M . V;
```

$$\begin{bmatrix} 10 \\ 0 \\ 20 \end{bmatrix} \tag{3.247}$$

```
> lambda . M . V;
```

$$\lambda \,.\, \begin{bmatrix} 10 \\ 0 \\ 20 \end{bmatrix} \tag{3.248}$$

```
> A := Array([[1,2],[3,4]]);
```

$$A := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{3.249}$$

```
> B := Array([[a,b,c],[d,e,f]]);
```

$$B := \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \tag{3.250}$$

```
> A . B;
```

$$\begin{bmatrix} a & 2\,b \\ 3\,d & 4\,e \end{bmatrix} \tag{3.251}$$

```
> 3 . B;
```

$$\begin{bmatrix} 3\,a & 3\,b & 3\,c \\ 3\,d & 3\,e & 3\,f \end{bmatrix} \tag{3.252}$$

The dot character has three meanings in Maple:

- as a decimal point in a floating-point number (for example, **2.3**),
- as part of a range (for example, **x..y**), or

- as the noncommutative multiplication operator. To distinguish between these three cases, Maple uses the following rule: any dot with spaces before and/or after it that is not part of a number is interpreted as the noncommutative multiplication operator.

For example, **2.3** is a number, **2 . 3** and **2 .3** return **6**, and **2. 3** displays an error.

```
> 2.3, 2 . 3, 2 .3;
```

$$2.3, 6, 6 \tag{3.253}$$

```
> 2. 3;
```

Factorials

The unary, postfix factorial operator ! is used to represent the mathematical factorial operation.

```
> 5!;
```

$$120 \tag{3.254}$$

Maple can compute large factorials quickly.

```
> length( 1000000! );
```

$$5565709 \tag{3.255}$$

If the argument of the **!** operator is symbolic, it is returned unevaluated.

```
> (a + b)!;
```

$$(a+b)! \tag{3.256}$$

The argument of the **!** operator is subject to automatic simplification, but factorials are not computed during the automatic simplification process.

```
> '(2+3)!';
```

$$5! \tag{3.257}$$

If the argument of the **!** operator is a float, the expression is computed using the **GAMMA** function.

```
> 2.3! = GAMMA( 3.3 );
```

$$2.683437382 = 2.683437382 \tag{3.258}$$

If the argument is a non-negative integer, Maple computes the factorial. If the argument is a negative integer, a numeric event is triggered.

```
> (-3)!;
```

```
Error, numeric exception: division by zero
```

However, if the argument is a negative integer float, the complex number Float(-infinity) - Float(infinity)*I is returned.

```
> (-3.0)!;
```

$$-Float(\infty) - Float(\infty)\,\mathrm{I} \tag{3.259}$$

For other arguments, the factorial operator is returned unevaluated after first evaluating its argument.

```
> sin( Pi / 6 )!;
```

$$\left(\frac{1}{2}\right)! \tag{3.260}$$

The command **factorial** is the same as the **!** operator.

```
> factorial( 5 );
```

$$120 \tag{3.261}$$

Forming Sums and Products

Since creating structures within loops may be inefficient, Maple provides commands for creating sums and products efficiently.

```
add( expression, i=m .. n);
mul( expression, i=m .. n);
```

where **i** is a name, **m** and **n** are numeric values, and **expression** is an expression that depends on **i**.

The **add** command is semantically equivalent to the following loop:

```
> S := 0;
  old := i;
  for i from m to n do
      S := S+expression;
  end do;
  i := old;
  S; # the result
```

The **add** command is more efficient since it does not build each of the many intermediate sums. The semantics of **mul** are similar with the exception that if $n < m$, the result is 1, rather than 0.

```
> mul(a+i, i=1..4);
```

$$(a+1)\,(a+2)\,(a+3)\,(a+4) \tag{3.262}$$

```
> add(a||i, i=0..3);
```

$$a0 + a1 + a2 + a3 \tag{3.263}$$

In the loop example shown above, each of the expressions $a0$, $a0 + a1$, and $a0 + a1 + a2$ are constructed, stored in memory, and then removed by the garbage collector. That overhead is part of what makes the loop less efficient than the **add** command in this case.

For more information on the **add** and **mul** commands, refer to the add help page. For more information on the concatenation operator, ||, see *The Concatenation Operator (page 109)*.

Note: The **add** and **mul** commands differ from sum and product in that the former are straightforward construction commands while the latter are commands for computing closed forms for symbolic sums and products.

3.10 Boolean and Relational Expressions

Boolean Constants

The Boolean constants in Maple are the names **true**, **false** and **FAIL**. These are otherwise ordinary names, but have a special meaning in a Boolean context.

When you call the Boolean evaluator **evalb**, the expression passed as its argument is interpreted as a Boolean-valued expression if possible, and evaluated as such.

Boolean Operators

Maple supports several operators for the Boolean combination of expressions: **not**, **and**, **or**, **xor**, and **implies**.

The not Operator

The **not** operator represents logical negation. It has the following general syntax.

```
not expr
```

When applied to a Boolean-valued expression, it returns a value according to the following table.

expr	**not *expr***
true	false
false	true

expr	**not** *expr*
FAIL	FAIL

For example,

```
> not true;
```

false (3.264)

```
> not false;
```

true (3.265)

```
> not FAIL;
```

FAIL (3.266)

The and Operator

The **and** operator represents logical conjunction. It is a binary operator of the form

```
expr1 and expr2
```

If both operands evaluate to a truth value, the entire expression is evaluated according to the following truth table.

expr1	*expr2*	*expr1* **and** *expr2*
true	true	true
true	false	false
true	FAIL	FAIL
false	true	false
false	false	false
false	FAIL	false
FAIL	true	FAIL
FAIL	false	false
FAIL	FAIL	FAIL

If a truth value cannot be determined, the expression is returned unevaluated.

```
> x and y;
```

x **and** y (3.267)

However, some automatic simplifications are applied to **and** expressions.

```
> true and x;
```

x (3.268)

The or Operator

The **or** operator represents logical disjunction. It is a binary operator of the form

```
expr1 or expr2
```

If both operands evaluate to a truth value, the entire expression is evaluated according to the following truth table.

expr1	***expr2***	***expr1* or *expr2***
true	true	true
true	false	true
true	FAIL	true
false	true	true
false	false	false
false	FAIL	FAIL
FAIL	true	true
FAIL	false	FAIL
FAIL	FAIL	FAIL

If a truth value cannot be determined, the expression is returned unevaluated.

```
> x or y;
```

$$x \text{ or } y \tag{3.269}$$

However, some automatic simplifications are applied to **or** expressions.

```
> false or x;
```

$$x \tag{3.270}$$

The xor Operator

The **xor** operator represents logical exclusive disjunction. It is a binary operator of the form

```
expr1 xor expr2
```

If both of its operands evaluate to truth values, the entire expression is evaluated according to the following truth table.

expr1	***expr2***	***expr1* xor *expr2***
true	true	false
true	false	true
true	FAIL	FAIL

expr1	*expr2*	*expr1* **xor** *expr2*
false	true	true
false	false	false
false	FAIL	FAIL
FAIL	true	FAIL
FAIL	false	FAIL
FAIL	FAIL	FAIL

The implies Operator

The **implies** operator represents logical implication. It is a binary operator of the form

```
expr1 implies expr2
```

If both of its operands evaluate to truth values, the entire expression is evaluated according to the following truth table.

expr1	*expr2*	*expr1* **implies** *expr2*
true	true	true
true	false	false
true	FAIL	FAIL
false	true	true
false	false	true
false	FAIL	true
FAIL	true	true
FAIL	false	FAIL
FAIL	FAIL	FAIL

If a truth value cannot be determined, the expression is returned unevaluated.

```
> x implies y;
```

$$x \Rightarrow y \tag{3.271}$$

Some automatic simplifications are applied to **implies** expressions.

```
> false implies x;
```

$$true \tag{3.272}$$

```
> x implies true;
```

$$true \tag{3.273}$$

Relational Operators

Relational operators are used to form comparisons to be evaluated in a Boolean context. The relational operators in Maple are =, <>, <, <=, and **in**. Each is a binary operator that accepts two operands. When evaluated in a Boolean context, each of these operators determines whether its two operands have a certain relationship.

An equation is formed by using the = operator.

```
> x = y;
```

$$x = y \tag{3.274}$$

This has the general form

```
expr1 = expr2
```

It represents an equation with **expr1** as the left-hand side and **expr2** as the right-hand side. When evaluated in a Boolean context, it returns a value of **true** if its operands are equal, and returns a value of **false** otherwise.

```
> evalb( 1 = 2 );
```

$$false \tag{3.275}$$

```
> evalb( 2 = 2 );
```

$$true \tag{3.276}$$

Note that comparing distinct unassigned names returns a value of **false**.

```
> evalb( x = y );
```

$$false \tag{3.277}$$

The names **x** and **y** are distinct and unequal names in Maple and, when they are unassigned, they are considered different expressions in a Boolean comparison. If the names **x** and **y** have assigned values, those values are first substituted into the comparison, and the equality computation is performed on the assigned values, rather than the names themselves.

In general, expressions are compared for equality according to their memory address. That is, two expressions are considered equal in a Boolean context if they have the same address in memory. However, for certain expressions, a more mathematical test for equality is used. For example, the floating-point numbers 2.0000 and 2.0 are considered numerically equal, even though they are distinct objects in memory.

```
> evalb( 2.0000 = 2.0 );
```

$$true \tag{3.278}$$

```
> addressof( 2.0000 );
```

$$18446790986243734750 \tag{3.279}$$

```
> addressof( 2.0 );
```

$$18446790986243734782 \tag{3.280}$$

In fact, when the floating-point number 2.0 is compared to the integer 2, they are considered equal.

```
> evalb( 2.0 = 2 );
```

$$\mathit{true} \tag{3.281}$$

Determining whether two procedures are semantically equivalent is an undecidable problem in Computer Science. However, procedures which are detectably equivalent by simple transformations are considered to be equal. For example, it is clear that the name of a procedure parameter is not normally important, so the following two simple procedures are considered equal, although they are distinct expressions in memory.

```
> evalb( proc(x) 2*x end proc = proc(y) 2*y end proc );
```

$$\mathit{true} \tag{3.282}$$

An inequation can be formed by using the <> operator. The general form is

```
expr1 <> expr2
```

This expression represents non-equality and returns a value of **true** if its operands are unequal, and **false** if its operands are equal.

```
> x <> y;
```

$$x \neq y \tag{3.283}$$

```
> evalb( 1 <> 2 );
```

$$\mathit{true} \tag{3.284}$$

```
> evalb( 2 <> 2 );
```

$$\mathit{false} \tag{3.285}$$

Testing for inequality is performed similarly to testing for equality. Comparing two distinct unassigned names using the <> operator computes the equality of the names. The expression

```
> evalb( x <> y );
```

$$\mathit{true} \tag{3.286}$$

returns a value of **true** because the names **x** and **y** are distinct as names.

A strict inequality is created by using the < operator. This has the general form

```
expr1 < expr2
```

and can also be constructed using the form

```
expr1 > expr2
```

For example,

```
> x < y;
```

$$x < y \tag{3.287}$$

You can also use the > operator.

```
> y > x;
```

$$x < y \tag{3.288}$$

Maple automatically converts this to the same expression as results from the first form.

When evaluated in a Boolean context, Maple performs the indicated mathematical comparison, or returns the inequality as unevaluated if the operands do not evaluate to comparable expressions. If the operands are comparable, the inequality evaluates to the value **true** if the first operand is less than, but not equal to, the second operand, and evaluates to **false** otherwise. If the operands are not comparable, the inequality evaluates to itself.

A non-strict inequality is formed using the <= operator. This has the general form

```
expr1 <= expr2
```

It can also be constructed using the form

```
expr1 >= expr2
```

For example,

```
> x <= y;
```

$$x \leq y \tag{3.289}$$

When evaluated in a Boolean context, and when the operands are comparable, it returns a value of either **true** or **false** according to whether the first operand is less than, or equal to, the second operand.

Membership is represented by the **in** operator. It is used in the general form

```
expr1 in expr2
```

When evaluated in a Boolean context, it evaluates to the value **true** if its first operand **expr1** is a member of its second operand **expr2**. If **expr1** does not belong to **expr2**, the expression

evaluates to **false**. Maple can determine a truth value if the second operand **expr2** is a container object; that is, either a set or list, or an unevaluated function call of the form SetOf(T), where **T** is a Maple type. An expression of the form

```
expr in SetOf( T )
```

where **T** is a Maple type is equivalent to the expression type(expr, T).

```
> evalb( 1 in { 1, 2, 3 } );
```

$$true \tag{3.290}$$

```
> evalb( 5 in { 1, 2, 3 } );
```

$$false \tag{3.291}$$

```
> evalb( x in X );
```

$$x \in X \tag{3.292}$$

```
> evalb( 2 in SetOf( integer ) );
```

$$true \tag{3.293}$$

```
> evalb( 2/3 in SetOf( integer ) );
```

$$false \tag{3.294}$$

Note the simplification applied to the statement with the **evalb** command in the following example.

```
> x in A union B;
```

$$x \in A \cup B \tag{3.295}$$

```
> evalb( x in A union B );
```

$$x \in A \textbf{ or } x \in B \tag{3.296}$$

If the second operand is not an explicit container object, the expression remains an unevaluated **in** expression. However, some automatic simplifications may be applied.

Efficient Boolean Iteration

In the same way the commands **add** and **mul** can be used to efficiently form + and * expressions, conjunctions and disjunctions can be evaluated efficiently using the andmap and ormap commands, which are similar to the map command described in *Maple Statements (page 169)*.

```
andmap( procedure, expression, ... )
```

```
ormap( procedure, expression, ... )
```

The following example considers type(element,name) for each element of the list. **ormap** determines whether this statement is true for at least one element of the list. **andmap** determines whether this statement is true for all the elements of the list.

```
> ormap(type, [1, "a", `a`, a()], name);
```

$$true \tag{3.297}$$

```
> andmap(type, [1, "a", `a`, a()], name);
```

$$false \tag{3.298}$$

The main difference between these commands and map is that andmap and ormap have short-circuit ("McCarthy") semantics, which means that an answer is returned as soon as it can be determined.

```
> andmap(proc(x) print(x); x<2 end proc, [1,2,3,4]);
```

$$1$$

$$2$$

$$false \tag{3.299}$$

3.11 Expressions for Data Structures

This section describes basic concepts related to data structure expressions. For more information on programming with data structures, see *Basic Data Structures (page 125)*.

Sequences

The most basic aggregate expression type in Maple is the *sequence*. Sequences are formed by using the `,` (comma) operator.

```
> a, 2/3, sin( x ), 5.1;
```

$$a, \frac{2}{3}, \sin(x), 5.1 \tag{3.300}$$

A sequence consists of zero or more other expressions, called *elements* or *members*. A sequence with exactly one member is automatically simplified to its unique member. The empty sequence, containing zero members, is the value of the name **NULL**, and may be written as ().

```
> evalb( () = NULL );
```

$$true \tag{3.301}$$

Sequences occur in many other data structures as a (principal) component, within which they acquire additional semantics. Some examples include lists, sets, and function calls.

Automatic simplification of sequences is affected by recursively simplifying the component expressions.

```
> '2 + 3, 1 - 7, 0^0, sin( Pi / 6 )';
```

$$5, -6, 1, \sin\left(\frac{1}{6}\,\pi\right) \tag{3.302}$$

Nested sequences are also flattened during the automatic simplification process.

```
> '( 1, 2 ), 3, ( 4, 5 )';
```

$$1, 2, 3, 4, 5 \tag{3.303}$$

Because sequences are used to pass multiple arguments to procedures, it is not normally possible to operate on a sequence *as such* (the **list** type described below is designed for exactly for that reason). For example, you cannot pass a (nontrivial) sequence to the **type** command to check its type. Therefore, there is no Maple type for sequences. However, the whattype command returns the name **exprseq** when it is passed either zero or more than one argument.

```
> whattype();
```

$$\mathit{exprseq} \tag{3.304}$$

```
> whattype( 1, 2 );
```

$$\mathit{exprseq} \tag{3.305}$$

Note that the name **exprseq** is not the name of any valid type in Maple.

Similarly, you cannot query the zeroth operand of a sequence. For example, the following results in an error.

```
> op( 0, ( 1, 2, 3 ) );
```

```
Error, invalid input: op expects 1 or 2 arguments, but received 4
```

This is because the sequence 0, (1, 2, 3) is flattened to the sequence 0, 1, 2, 3 during automatic simplification of the function call *before* the **op** command is actually called. Therefore, the **op** command is passed four arguments instead of only the two that are apparently intended.

There is no constructor for sequences, but there is a built-in command for creating sequences, called **seq**. The basic syntax of **seq** is below. It accepts many other types of arguments as well.

```
seq(expression, i = integer1..integer2)
```

```
> seq( i^2, i = 1 .. 5 );
```

$$1, 4, 9, 16, 25 \tag{3.306}$$

```
> seq( 2 .. 14 );
```

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 \tag{3.307}$$

```
> seq( i, i = 0.4 .. 1.1, 0.3 );
```

$$0.4, 0.7, 1.0 \tag{3.308}$$

For more information on the **seq** command, refer to the seq help page.

Another way to create sequences is to use the dollar sign ($) operator.

```
expression $ i = integer1 .. integer2
```

```
> i^2 $ i = 1 .. 5;
```

$$1, 4, 9, 16, 25 \tag{3.309}$$

The dollar sign operator is a binary operator that performs a similar function to the **seq** command, but behaves slightly differently: the $ operator evaluates the expression argument once before any substitutions, while the command does not evaluate until after each substitution of *i*.

```
> cat(a,x) $ x= 1..2;
```

$$ax, ax \tag{3.310}$$

```
> seq(cat(a,x), x= 1..2);
```

$$a1, a2 \tag{3.311}$$

In general, it is recommended that you use the **seq** command instead of the dollar sign operator.

Lists

Lists are created by enclosing a sequence of expressions between square brackets. Lists are essentially sequences, which are designated as a single unit for other operations.

```
[ sequence ]
```

```
> [ 1, 2, 3 ];
```

$$[1, 2, 3] \tag{3.312}$$

Unlike sequences, lists can form properly nested structures.

```
> [ 1, 2, [ 3, 4 ] ];
```

$$[1, 2, [3, 4]] \tag{3.313}$$

Use the **numelems** command to determine the number of members in the enclosed sequence. Note that lists can contain sublists. These are still counted as a single entry.

```
> numelems( [ 1, 2, 3 ] );
```

$$3 \tag{3.314}$$

```
> numelems( [ 1, 2, [ 3, 4 ] ] );
```

$$3 \tag{3.315}$$

To access the i-th operand of a list, use an index to the list expression.

```
> L := [ a, b, c, d ];
```

$$L := [a, b, c, d] \tag{3.316}$$

```
> L[ 3 ];
```

$$c \tag{3.317}$$

To access the sequence of all elements in a list, use the **op** command. Converting back and forth between lists and sequences can be a common operation, and is very efficient.

```
> Lseq := op(L);
```

$$Lseq := a, b, c, d \tag{3.318}$$

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

$$L2 := [a, b, c, d, a, b, c, d] \tag{3.319}$$

It is common to create a list by using the **seq** command to create the enclosed sequence.

```
> [ seq( i^2, i = 1 .. 5 ) ];
```

$$[1, 4, 9, 16, 25] \tag{3.320}$$

Lists are ordered; two lists with the same members in a different order are distinct.

```
> evalb( [ 1, 2, 3 ] = [ 2, 1, 3 ] );
```

$$false \tag{3.321}$$

Lists are immutable; you cannot change the elements of a list once it has been created. You can, however, create a new list using members of an existing list or lists.

In the next example, we create a new list with second entry **d**.

```
> L := [ a, b, c ];
```

$$L := [a, b, c] \tag{3.322}$$

```
> L2 := [ L[ 1 ], d, L[ 3 ] ];
```

$$L2 := [a, d, c] \tag{3.323}$$

You can also use the **subsop** command for this purpose.

```
> L3 := subsop( 2 = d, L );
```

$$L3 := [a, d, c] \tag{3.324}$$

```
> evalb( L2 = L3 );
```

$$true \tag{3.325}$$

The example above creates a new list using the original list **L** by substituting its second operand for the expression d. If you need to change elements frequently it is usually better to use an array. Arrays can be changed in-place avoiding the need for a copy. For more information on the **subsop** command, refer to the subsop help page.

For more information about lists, see *Lists (page 126)*.

Sets

Sets, similar to lists, are created from a sequence of expressions. However, sets use braces ({}) to enclose the sequence.

```
{ sequence }
```

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

$$\{1, 2, 3\} \tag{3.326}$$

In addition to the syntactical differences, sets differ from lists in that they are implicitly sorted and do not have duplicate entries. These two properties are enforced during the automatic simplification process.

```
> '{3, -1, 0}';
```

$$\{-1, 0, 3\} \tag{3.327}$$

```
> '{1, 1, 1, 1}';
```

$$\{1\} \tag{3.328}$$

In Maple 11 and earlier, the ordering of sets was unpredictable as it was based on the positions of the elements in memory. In Maple 12 and later, set ordering is deterministic, session independent, and based on properties of the contents. This just means that the same set will

now appear in the same order even after restarting Maple. For more information on the ordering of sets, refer to the set help page.

For more information on how to use sets in programming, see *Sets (page 132)*. More information on Maple expressions related to sets will be described later in this chapter.

Tables

Tables are mutable data structures that associate an index with an entry. Both the index and entry can be arbitrary expressions. The underlying structure is sparse (a hash table), and expands as more entries are inserted.

```
> T := table();
```

$$T := table([\,])\tag{3.329}$$

```
> T[color] := "red";
```

$$T_{color} := \text{"red"}\tag{3.330}$$

```
> T[color];
```

$$\text{"red"}\tag{3.331}$$

```
> T[1,2,3] := x^2+4;
```

$$T_{1,\,2,\,3} := x^2 + 4\tag{3.332}$$

Assigning values to indexed names is further described in *Indexed Expressions (page 62)*.

Tables can be initially populated by providing a list of equations as an argument to the **table** constructor.

```
> T := table([a=1, b=2, c=3, d=4]);
```

$$T := table([b = 2, d = 4, c = 3, a = 1])\tag{3.333}$$

```
> T[a] + T[c];
```

$$4\tag{3.334}$$

For names with tables assigned to them, last name evaluation rules apply. Last name evaluation is explained in more detail in *Evaluation Rules for Tables (page 141)*. The most visible effect of last name evaluation is that the name of the table is displayed by default rather than all of its entries.

```
> T;
```

$$T\tag{3.335}$$

```
> eval(T);
```

$$table([b = 2, d = 4, c = 3, a = 1]) \tag{3.336}$$

Rectangular Tables

Rectangular tables, or *rtables*, are mutable data structures that associate a numeric index sequence with an arbitrary entry. The bounds of the index are predefined and directly correspond to the amount of memory reserved to hold entries.

The same **rtable** data structure is used to implement arrays, matrices, and vectors.

```
> A := Array(0..5,i->2*i);
```

$$A := Array(0..5, \{1 = 2, 2 = 4, 3 = 6, 4 = 8, 5 = 10\}, datatype = anything, storage = rectangular, order = Fortran\_order) \tag{3.337}$$

```
> A[0];
```

$$0 \tag{3.338}$$

```
> A[5];
```

$$10 \tag{3.339}$$

```
> V := Vector([1,2,3]);
```

$$V := \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \tag{3.340}$$

```
> V[1];
```

$$1 \tag{3.341}$$

```
> M := Matrix(3,3,shape=identity);
```

$$M := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.342}$$

```
> M[2,2];
```

$$1 \tag{3.343}$$

Rectangular tables are very flexible and offer a rich set of features. For a more in-depth discussion of them, see *Arrays (page 144)*.

3.12 Set-Theoretic Expressions

Maple includes a full set of set-theoretic operators for membership relation, set inclusion, and other operations.

Membership

In Maple, the set membership relation is expressed by using the **in** operator. It has the following syntax.

```
a in b
```

where *a* and *b* can be arbitrary expressions.

Normally, a membership expression is returned unevaluated.

```
> a in b;
```

$$a \in b \tag{3.344}$$

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

$$1 \in \{1, 2, 3\} \tag{3.345}$$

However, when evaluated in a Boolean context, one of the values **true** and **false** is returned if the expression *b* evaluates to a set or list and Maple can determine whether the expression *a* belongs to the expression *b*. For more information on Boolean evaluation of membership expressions, see *Boolean and Relational Expressions (page 86)*.

Use the **rhs** and **lhs** commands to extract the right or left hand side of the an **in** operator.

```
> lhs( a in b );
```

$$a \tag{3.346}$$

```
> rhs( a in b );
```

$$b \tag{3.347}$$

Set Inclusion

Set inclusion (the subset relation) is represented in Maple by the binary **subset** operator. It has the following syntax.

```
a subset b
```

where *a* and *b* are arbitrary expressions that can evaluate to sets.

```
> a subset b;
```

$$a \subseteq b \tag{3.348}$$

```
> { 1, 2 } subset {2, 3, 5 };
```

$$false \tag{3.349}$$

```
> {} subset T;
```

$$true \tag{3.350}$$

```
> T subset {};
```

$$T \subseteq \{\,\} \tag{3.351}$$

If Maple can determine whether the expressed relation is true or false, the expression evaluates to **true** or **false**. Otherwise, the expression is returned unevaluated.

An unevaluated set inclusion expression has two operands *a* and *b*.

```
> nops( a subset b );
```

$$2 \tag{3.352}$$

```
> op( a subset b );
```

$$a, b \tag{3.353}$$

The individual operands can be accessed by using the **lhs** and **rhs** commands.

```
> lhs( a subset b );
```

$$a \tag{3.354}$$

```
> rhs( a subset b );
```

$$b \tag{3.355}$$

Other Binary Operators for Sets

You can create new sets from existing sets by using any of the binary set-theoretic operators.

The union of two sets is created by using the **union** operator, which has the following syntax.

```
a union b
```

where *a* and *b* are expressions that can evaluate to a set.

```
> a union b;
```

$$a \cup b \tag{3.356}$$

```
> { 1, 2 } union { 2, 3, 4 };
```

$$\{1, 2, 3, 4\} \tag{3.357}$$

```
> { 1, 2 } union T;
```

$$T \cup \{1, 2\} \tag{3.358}$$

The following expression displays an error message, since the second operand cannot evaluate to a set.

```
> { a, b, c } union "a string";
```

```
Error, invalid input: `union` received a string, which is not valid
for its 2nd argument
```

A union expression may be returned unevaluated, and the operands of an unevaluated union expression a union b are the expressions a and b.

```
> nops( a union b );
```

$$2 \tag{3.359}$$

```
> op( a union b );
```

$$a, b \tag{3.360}$$

Note that the union operation is commutative.

```
> a union b;
```

$$a \cup b \tag{3.361}$$

```
> b union a;
```

$$a \cup b \tag{3.362}$$

The union operation is also associative. A union of three or more operands returns an unevaluated function call.

```
> a union b union c;
```

$$union(a, b, c) \tag{3.363}$$

The union operation performs certain normalizations.

```
> a union a;
```

$$a \tag{3.364}$$

```
> {} union a;
```

$$a \tag{3.365}$$

Intersections of sets are represented using the **intersect** operator, which has the general syntax.

```
a intersect b
```

The operands *a* and *b* are expressions that can evaluate to a set.

```
> a intersect b;
```

$$a \cap b \tag{3.366}$$

```
> { 1, 2, 3 } intersect { 3, 4, 5 };
```

$$\{3\} \tag{3.367}$$

```
> {} intersect T;
```

$$\{\} \tag{3.368}$$

Note that although union and intersection are mutually distributive, neither distributes automatically over the other in a symbolic expression. However, the **expand** command can distribute intersections over unions.

```
> expand( a intersect (b union c) );
```

$$a \cap b \cup a \cap c \tag{3.369}$$

Maple takes the canonical form of a set-theoretic expression to be a union of intersections, so the **expand** command does not distribute symbolic unions over intersections.

```
> expand( a union (b intersect c) );
```

$$a \cup b \cap c \tag{3.370}$$

3.13 Other Expressions

Functional Operators

The operator **->** (arrow) can be used as a short-hand form to create procedures inline in commands which take procedures as arguments such as Array constructors and the map command.

```
( vars ) -> result
```

The following two procedures are identical except in how they are displayed:

```
> x -> x^2;
```

$$x \rightarrow x^2 \tag{3.371}$$

```
> proc(x) x^2 end proc;
```

$$\mathbf{proc}(x)\ x\text{^}2\ \mathbf{end\ proc} \tag{3.372}$$

as are these two:

```
> (x,y,z) -> sqrt(x^2+y^2+z^2);
```

$$(x, y, z) \to \sqrt{x^2 + y^2 + z^2} \tag{3.373}$$

```
> proc(x,y,z) sqrt(x^2+y^2+z^2) end proc;
```

$$\mathbf{proc}(x, y, z)\ \mathrm{sqrt}(x\text{^}2 + y\text{^}2 + z\text{^}2)\ \mathbf{end\ proc} \tag{3.374}$$

For more information on the arrow operator, refer to the operators,functional help page. For more information on procedures, see *Procedures (page 195)*.

Composition

Use the operators @ and @@ to represent the composition (of functions). The operator @ denotes the composition of functions and takes the general form

```
f @ g
```

where each of **f** and **g** can be an arbitrary expression.

```
> (f@g)(x);
```

$$f(g(x)) \tag{3.375}$$

Note that @ has lower precedence than function application, so that the parentheses surrounding f@g above are necessary:

```
> f@g(x);
```

$$f@\,g(x) \tag{3.376}$$

The @ operator performs numerous simplifications and normalizations, and is (left) associative.

```
> (exp @ ln)( s );
```

$$s \tag{3.377}$$

```
> a @ b @ c @ d;
```

$$a@\,b@\,c@\,d \tag{3.378}$$

Repeated composition is represented by the operator @@. It has the general form

```
f @@ n
```

This denotes the n-fold composition of a function f.

```
> expand( (f@@3)( x ) );
```

$$f(f(f(x))) \tag{3.379}$$

Note that the iterated composition is not automatically expanded in the example above. It is necessary to apply the **expand** command.

It is important to distinguish repeated composition of an expression from the arithmetic power. The former is represented in Maple using the @@ operator, while the latter is represented using the ^ operator.

```
> expand( (f@@2)( x ) );
```

$$f(f(x)) \tag{3.380}$$

```
> (f^2)( x );
```

$$f(x)^2 \tag{3.381}$$

The first example above denotes the 2-fold composition of f with itself, while the second denotes the arithmetic square of f. In particular, although the inverses of the circular functions are commonly denoted by a power-like notation in written mathematics, in Maple, for example, sin^(-1) denotes the reciprocal of the $\sin$ function, while sin@@(-1) denotes the arcsine (arcsin).

```
> sin@@(-1);
```

$$\arcsin \tag{3.382}$$

```
> (sin@arcsin)( x );
```

$$x \tag{3.383}$$

```
> sin^(-1);
```

$$\frac{1}{\sin} \tag{3.384}$$

```
> (sin^(-1))( x );
```

$$\frac{1}{\sin(x)} \tag{3.385}$$

Neutral Operators

Neutral operators are constructions that are treated as operators by Maple, but that have no predefined meaning so that they can be customized.

A neutral operator symbol is formed by the ampersand character (&) followed either by a valid Maple name not containing ?, or by a sequence of one or more special characters. For more information, refer to the neutral help page.

```
a &name b
```

```
> expr := a &your_operator_name_here b;
```

$$expr := a \ \&your\_operator\_name\_here \ b \tag{3.386}$$

A commonly used neutral operator is **&*** which is often used for representing a non-commutative multiplication. Unlike dot (.), it does not automatically combine scalar constants.

```
> 1 &* 2;
```

$$1 \ \&* \ 2 \tag{3.387}$$

```
> 1 . 2;
```

$$2 \tag{3.388}$$

Ranges

The **..** operator is used to construct ranges, and usually has the following syntax.

```
a .. b
```

in which the endpoints **a** and **b** can be arbitrary expressions.

It is important to distinguish between a range expression, such as 3 .. 7, with explicit numeric endpoints, and the corresponding sequence 3, 4, 5, 6, 7. The **seq** command can be used to produce the latter from the former.

Often, a range is used in an expression of the form i = a .. b, as an argument to a command (such as add), and denotes the range over which an index such as i is to vary.

```
> add( i^2, i = 1 .. 5 );
```

$$55 \tag{3.389}$$

A consecutive sequence of two *or more* dots (.) is parsed as a range operator. For example,

```
> 2 ......... 7;
```

$$2 \ldots 7 \tag{3.390}$$

If the left-hand endpoint of a range is a float ending with a decimal point, or if the right-hand endpoint is a float beginning with a decimal point, it is therefore necessary to separate the endpoint from the range operator with one or more space characters.

```
> 2....3;
```

$$2\,..3 \tag{3.391}$$

```
> 2. .. .3;
```

$$2.\,..0.3 \tag{3.392}$$

The number of operands of a range expression is equal to 2.

```
> nops( a .. b );
```

$$2 \tag{3.393}$$

The operands are the left and right-hand endpoints.

```
> op( a .. b );
```

$$a, b \tag{3.394}$$

Use the **lhs** and **rhs** commands to extract the individual operands of a range expression.

```
> lhs( a .. b );
```

$$a \tag{3.395}$$

```
> rhs( a .. b );
```

$$b \tag{3.396}$$

The type of a range expression is **range** or the equivalent form **`..`**.

```
> type( a .. b, 'range' );
```

$$true \tag{3.397}$$

```
> type( a .. b, '`..`' );
```

$$true \tag{3.398}$$

Ranges can be used to index complex data structures as well as strings and sequences.

```
> [ 1, 2, 3, 4, 5 ][ 2 .. 3 ];
```

$$[2, 3] \tag{3.399}$$

```
> { 1, 2, 3, 4, 5 }[ 2 .. 3 ];
```

$$\{2, 3\} \tag{3.400}$$

```
> "abcde"[ 2 .. 3 ];
```

"bc" (3.401)

```
> ( 1, 2, 3, 4, 5 )[ 2 .. 3 ];
```

2, 3 (3.402)

There is a special form of input syntax for ranges in which one or both endpoints is missing.

```
> ..;
```

( ) ..( ) (3.403)

In the example above, each endpoint is the empty sequence () (or NULL). It is valid to omit just one of the endpoints.

```
> a ..;
```

a ..( ) (3.404)

```
> .. b;
```

( ) ..*b* (3.405)

When used in this way to index a data structure, a missing endpoint denotes the end of the valid range of indices.

```
> [ 1, 2, 3, 4, 5 ][ 3 .. ];
```

[3, 4, 5] (3.406)

```
> [ 1, 2, 3, 4, 5 ][ .. 4 ];
```

[1, 2, 3, 4] (3.407)

```
> [ 1, 2, 3, 4, 5 ][ .. ];
```

[1, 2, 3, 4, 5] (3.408)

Note the distinction between the third example above and the following example

```
> [ 1, 2, 3, 4, 5 ][];
```

1, 2, 3, 4, 5 (3.409)

in which the index is empty.

The Concatenation Operator

The operator || denotes the concatenation of names and strings. It takes the general form

```
a || b
```

in which the first operand **a** can be either a name or a string, and the second operand **b** can be a name, a string, an integer, an integral range, a character range, or an expression sequence of names, strings, and integers. If the second operand *b* is another kind of expression, an unevaluated || expression is returned.

```
> "foo" || "bar";
```

"foobar" (3.410)

```
> foo || bar;
```

foobar (3.411)

```
> foo || "bar";
```

foobar (3.412)

```
> "foo" || bar;
```

"foobar" (3.413)

```
> x || 1;
```

x1 (3.414)

```
> x || (1..3);
```

x1, x2, x3 (3.415)

```
> "x" || (1,2,3);
```

"x1", "x2", "x3" (3.416)

```
> x || ("a" .. "f");
```

xa, xb, xc, xd, xe, xf (3.417)

```
> x || ("s", "t", "w" );
```

xs, xt, xw (3.418)

```
> f( y ) || t;
```

$(f(y))\,||\,t$ (3.419)

The type of the result, if not an unevaluated || expression, is determined by the type of the first operand. If the first operand *a* is a string, the type of the result (or results, in the case of a sequence) is a string. If the first operand is a name, the type of the result, or results, is a name.

The first operand of the || operator is not evaluated, but the second operand is.

```
> u := 2: v := 3:
> u || v;
```

$$u3 \tag{3.420}$$

The symbol `||`, which must be enclosed in left single quotes when not used as an infix operator, is a type name.

```
> type( f( y ) || t, '`||`' );
```

$$true \tag{3.421}$$

If a concatenation expression is returned unevaluated, it has two operands.

```
> nops( f( s ) || t );
```

$$2 \tag{3.422}$$

```
> op( f( s ) || t );
```

$$f(s), t \tag{3.423}$$

For most applications, the **cat** command is more appropriate, as it evaluates all of its arguments. For more information, refer to the cat help page.

The Double Colon Operator

The **::** (double colon) operator is used to represent a type test, and takes two operands in the following form.

```
expr :: t
```

where **expr** is an arbitrary expression, and **t** is a type expression. When evaluated in a Boolean context, it is equivalent to type(expr, t).

```
> evalb( [ 1, 2, 3 ] :: list );
```

$$true \tag{3.424}$$

```
> [ 1, 2 ] :: list and 2 > 3;
```

$$false \tag{3.425}$$

In addition to its use as a general Boolean expression, it is used to introduce type annotations on parameters and type assertions for local variables and procedure return types. For more information, see *Procedures (page 195)*.

Outside of a Boolean context, the **::** operator is essentially inert, and returns an expression of type **::** with two operands.

```
> type( a :: b, '`::`' );
```

$$true \tag{3.426}$$

```
> nops( a :: b );
```

$$2 \tag{3.427}$$

```
> op( a :: b );
```

$$a, b \tag{3.428}$$

You can use the **lhs** and **rhs** commands to access the operands of a **::** expression.

```
> lhs( a :: b );
```

$$a \tag{3.429}$$

```
> rhs( a :: b );
```

$$b \tag{3.430}$$

Series

Maple supports generalized power series expansions using a **series** data structure. This is a basic symbolic data structure that is used in many fundamental algorithms, such as the computation of symbolic limits and symbolic integration. There is no syntax for the input of series; a series structure is created by calling the **series** constructor, which has the general forms

```
series( expr, eqn, ord )
series( expr, name, ord )
```

where **expr** is an algebraic expression, **eqn** is an equation of the form

```
name = pt
```

where **name** is a name and **pt** is the point of series expansion. The optional argument *ord* specifies the order of truncation of the series. This is, by default, equal to the value of the environment variable **Order**, whose default value is 6. If the second form using *name* is used, the expansion point is taken to be 0.

```
> series( exp( x ), x );
```

$$1 + x + \frac{1}{2} x^2 + \frac{1}{6} x^3 + \frac{1}{24} x^4 + \frac{1}{120} x^5 + O(x^6) \tag{3.431}$$

```
> series( exp( x ), x, 10 );
```

$$1 + x + \frac{1}{2} x^2 + \frac{1}{6} x^3 + \frac{1}{24} x^4 + \frac{1}{120} x^5 + \frac{1}{720} x^6 + \frac{1}{5040} x^7 + \frac{1}{40320} x^8 + \frac{1}{362880} x^9 + O(x^{10}) \tag{3.432}$$

```
> series( exp( x ), x = 0 );
```

$$1 + x + \frac{1}{2} x^2 + \frac{1}{6} x^3 + \frac{1}{24} x^4 + \frac{1}{120} x^5 + O(x^6) \tag{3.433}$$

```
> series( exp( x ), x = 1 );
```

$$e + e\,(x-1) + \frac{1}{2} e\,(x-1)^2 + \frac{1}{6} e\,(x-1)^3 + \frac{1}{24} e\,(x-1)^4 + \frac{1}{120} e\,(x-1)^5 + O\left((x-1)^6\right) \tag{3.434}$$

In general, a truncated power series expansion to order *ord* of *expr*, about the point *pt* is computed. If the expansion point *pt* is infinity, then an asymptotic expansion is computed.

In general, the series expansion is not exact, so there will be an order term of the form

```
O( pt^ord )
```

present as the last term in the series. This is not always present, however. For example, a series expansion of a low-degree polynomial is exact.

```
> series( x^2 + x + 1, x );
```

$$1 + x + x^2 \tag{3.435}$$

The presence of an order term depends also on the truncation order.

```
> series( x^20 + x + 1, x );
```

$$1 + x + O(x^{20}) \tag{3.436}$$

```
> series( x^20 + x + 1, x, 30 );
```

$$1 + x + x^{20} \tag{3.437}$$

A series data structure prints very much like a polynomial, but it is a distinct data structure. In certain cases, a polynomial (sum of product) data structure is returned. This happens when the generalized series expansion requires fractional exponents.

```
> s := series( sqrt( sin( x ) ), x );
```

$$s := \sqrt{x} - \frac{1}{12} x^{5/2} + \frac{1}{1440} x^{9/2} + O\left(x^{13/2}\right) \tag{3.438}$$

```
> type( s, 'series' );
```

$$\mathit{false} \tag{3.439}$$

The operands of a series expression are as follows.

The 0th operand is an expression of the form $x - a$, where x is the variable of expansion, and a is the expansion point. Odd operands are the coefficients of the series, while positive even operands are the exponents. In general, for a series expression s, op(2*i-1,s) evaluates to the ith coefficient of s, while op(2*i,s) evaluates to the ith exponent.

```
> op( 0, series( F( x ), x = a ) );
```

$$x - a \tag{3.440}$$

```
> op( series( exp( x^2 ), x ) );
```

$$1, 0, 1, 2, \frac{1}{2}, 4, O(1), 6 \tag{3.441}$$

Note that the series data structure is sparse in the sense that terms with 0 coefficient are not part of the data structure.

A series structure can be converted to a polynomial by using the **convert** command with the name **polynom** as the second argument.

```
> s := series( exp( x ), x );
```

$$s := 1 + x + \frac{1}{2} x^2 + \frac{1}{6} x^3 + \frac{1}{24} x^4 + \frac{1}{120} x^5 + O\left(x^6\right) \tag{3.442}$$

```
> type( s, 'series' );
```

$$\mathit{true} \tag{3.443}$$

```
> p := convert( s, 'polynom' );
```

$$p := 1 + x + \frac{1}{2} x^2 + \frac{1}{6} x^3 + \frac{1}{24} x^4 + \frac{1}{120} x^5 \tag{3.444}$$

```
> type( p, 'polynom' );
```

$$\mathit{true} \tag{3.445}$$

3.14 Attributes

In addition to their operands, certain types of expressions can have other information associated with them in the form of *attributes*. As described earlier in this chapter, protected names are a type of attribute. If an expression has attributes, they can be examined by using the attributes command.

```
> attributes(sin);
```

$$protected, \_syslib \tag{3.446}$$

Attributes can be assigned to expressions of the following types: name, string, list, set, Array, Matrix, Vector, equation, procedure, unevaluated function call, or float using the **setattribute** command.

```
setattribute(expression, attributes)
```

The **setattribute** command returns a copy of the expression with the attributes assigned. If the expression is a symbol or string, it is modified in-place. For other data types, the original expression is left unchanged.

```
> x := 1.0;
```

$$x := 1.0 \tag{3.447}$$

```
> setattribute('x', "blue");
```

$$x \tag{3.448}$$

```
> attributes('x');
```

$$\text{"blue"} \tag{3.449}$$

```
> myname := "Johanessphere":
> setattribute(myname, "Great Name", "Not a Real Name");
```

$$\text{"Johanessphere"} \tag{3.450}$$

```
> attributes("Johanessphere");
```

$$\text{"Great Name", "Not a Real Name"} \tag{3.451}$$

```
> y := setattribute('f(z)',"common");
```

$$y := f(z) \tag{3.452}$$

```
> attributes(y);
```

$$\text{"common"} \tag{3.453}$$

```
> attributes('f(z)');
```

All Maple expressions are valid attributes, including expression sequences.

You can check whether an expression has attributes by using the **attributed** type. For more information, refer to the type,attributed help page.

```
> type(`just a name`, 'attributed');
```

$$false \tag{3.454}$$

```
> type(sin, 'attributed');
```

$$true \tag{3.455}$$

3.15 Using Expressions

Evaluating and Simplifying Expressions

Example 1

To understand how Maple evaluates and simplifies expressions, consider the following example.

```
> x := Pi/6:
> sin(x) + 2*cos(x)^2*sin(x) + 3;
```

$$\frac{17}{4} \tag{3.456}$$

Maple first reads and parses the input. As the input is parsed, Maple builds an expression tree to represent the value.

$$\sin(x) + 2\cos(x)^2 \sin(x) + 3$$

Maple simplifies the expression tree and then evaluates the result. The evaluation process substitutes values for variables and invokes any commands or procedures. In this case, **x** evaluates to $\frac{1}{6}\pi$. Therefore, with these substitutions, the expression is

$$\sin\left(\frac{\mathrm{Pi}}{6}\right) + 2\cos\left(\frac{\mathrm{Pi}}{6}\right)^2 \sin\left(\frac{\mathrm{Pi}}{6}\right) + 3$$

When the **sin** and **cos** commands are called, Maple obtains a new "expression tree,"

$$\frac{1}{2} + 2\left(\frac{1}{2}\sqrt{3}\right)^2 \frac{1}{2} + 3$$

Maple simplifies this result to obtain the fraction 17/4.

Example 2

Alternatively, consider the next example: evaluation occurs, but no simplification is possible.

```
> x := 1;
```

$$x := 1 \tag{3.457}$$

```
> sin(x) + 2*cos(x)^2*sin(x) + 3;
```

$$\sin(1) + 2\cos(1)^2 \sin(1) + 3 \tag{3.458}$$

Substituting Subexpressions

The simplest method of substitution in Maple is to use the subsop command. This is an operation on the expression tree. It creates a new expression by replacing an operand in the original expression with the given value.

subsop(*n=value*, *expr*);
subsop(*list=value*, *expr*);

```
> L := [a, b, [c,d,e]]:
> M := subsop(1=A, L):
> L, M;
```

$$[a, b, [c, d, e]], [A, b, [c, d, e]] \tag{3.459}$$

```
> subsop([3,1]=C, L);
```

$$[a, b, [C, d, e]] \tag{3.460}$$

Note that most operations in Maple do not alter expressions *in-place* but, in fact, create new expressions from old ones. For a list of exceptions, see *Mutable Data Structures (page 137)*.

Also, note that the **subsop** command acts on the expression tree by changing an arrow in the DAG, and not by changing the value of a node which would change all identical instances. That is, in the following example only one instance of a is changed, not all. See **Figure 3.2**.

```
> expr := (a+a*b)/(a*b-b);
```

$$expr := \frac{a\,b + a}{a\,b - b} \tag{3.461}$$

```
> subsop([1,1]=2*c, expr);
```

$$\frac{2\,c+a}{a\,b-b} \tag{3.462}$$

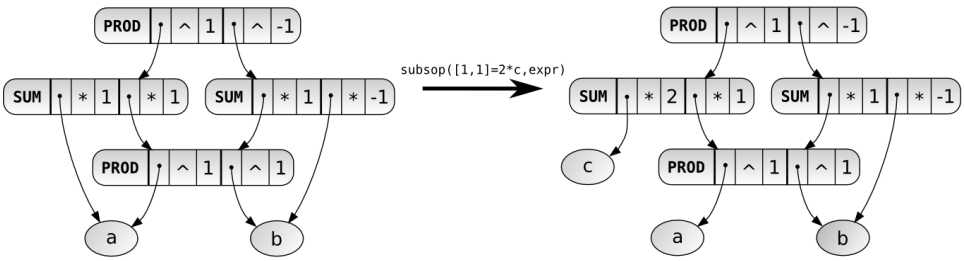


Figure 3.2: subsop Example DAGs

The **subsop** command is powerful, but generally useful only in very specific programming applications. The most generally useful command for substitution is the two-argument version of the **eval** command.

The **eval** command has the following syntax, where **s** is an equation, list, or set of equations.

```
eval( expr, s );
```

```
> expr := x^3 + 3*x + 1;
```

$$expr := x^3 + 3\,x + 1 \tag{3.463}$$

```
> eval( expr, x=y );
```

$$y^3 + 3\,y + 1 \tag{3.464}$$

```
> eval( expr, x=2 );
```

$$15 \tag{3.465}$$

```
> eval( sin(x) + x^2, x=0 );
```

$$0 \tag{3.466}$$

The **eval** command performs substitution on the expression considered as a DAG rather than a tree, so it can be quite efficient for large expressions with many repeated subexpressions.

An alternative to the **eval** command is the **subs** command, which performs *syntactic* substitution. It computes the expression as a *tree* and replaces subexpressions in an expression with a new value. The subexpressions must be operands, as identified by the **op** command.

Using the **subs** command is equivalent to performing a **subsop** operation for each occurrence of the subexpressions to be replaced.

The **subs** command has the following syntax, where *s* is an equation, list, or set of equations.

```
subs( s, expr );
```

```
> f := x*y^2;
```

$$f := x\,y^2 \tag{3.467}$$

```
> subs( {y=z, x=y, z=w}, f );
```

$$y\,z^2 \tag{3.468}$$

The other difference between the **eval** and **subs** commands is demonstrated in the following example.

```
> subs( x=0, cos(x) + x^2 );
```

$$\cos(0) \tag{3.469}$$

```
> eval( cos(x) + x^2, x=0 );
```

$$1 \tag{3.470}$$

In the preceding **subs** command, Maple substitutes **0** (zero) for **x** and simplifies the result. Maple simplifies **cos(0) + 0^2** to **cos(0)**. By using the **eval** command, Maple evaluates **cos(0)** to **1** (one).

During the substitution process, operands are compared in the expression tree of **expr** with the left-hand side of an equation.

```
> eval( a*b*c, a*b=d );
```

$$a\,b\,c \tag{3.471}$$

The substitution does not result in d*c because the operands of the product a*b*c are **a**, **b**, **c**. That is, the products a*b, b*c, and a*c do not appear specifically as operands in the expression a*b*c. The easiest way to make such substitutions is to solve the equation for one unknown and substitute for that unknown.

```
> eval( a*b*c, a=d/b );
```

$$d\,c \tag{3.472}$$

You cannot always use this method; for certain expressions, it may not produce the expected results. The algsubs command provides a more powerful substitution facility.

```
> algsubs( a*b=d, a*b*c );
```

$$d\,c \tag{3.473}$$

Two more useful substitution commands are subsindets and evalindets. These commands perform substitution on all subexpressions of a given type; the former uses the **subs** functionality and the latter uses the **eval** functionality.

subsindets(*expr*, *atype*, *transformer*, *rest*)
evalindets(*expr*, *atype*, *transformer*, *rest*)

```
> evalindets([1,2,3,4,5], prime, x->(x+1)^2);
```

$$[1, 9, 16, 4, 36] \tag{3.474}$$

```
> evalindets((x+1)^2+x^4-1, {`*`,`^`}, expand);
```

$$x^4 + x^2 + 2\,x \tag{3.475}$$

Structured Types

A simple type check may not always provide sufficient information. For example, the command

```
> type( x^2, `^` );
```

$$true \tag{3.476}$$

verifies that x^2 is an exponentiation, but does not indicate whether the exponent is, for example, an integer. To do so, you must use *structured* types. Consider the following example.

```
> type( x^2, 'name^integer' );
```

$$true \tag{3.477}$$

Because **x** is a name and **2** is an integer, the command returns a value of **true**.

To learn more about structured types, study the following examples. The square root of **x** does not have the structured type **name^integer**.

```
> type( x^(1/2), 'name^integer' );
```

$$false \tag{3.478}$$

The expression (x+1)^2 does not have type **name^integer**, because x+1 is not a name.

```
> type( (x+1)^2, 'name^integer' );
```

$$false \tag{3.479}$$

The type **anything** matches any expression.

```
> type( (x+1)^2, 'anything^integer' );
```

$$true \tag{3.480}$$

An expression matches a *set* of types if the expression matches one of the types in the set.

```
> type( 1, '{integer, name}' );
```

$$true \tag{3.481}$$

```
> type( x, '{integer, name}' );
```

$$true \tag{3.482}$$

The type **set(type)** matches a *set* of elements of type **type**.

```
> type( {1,2,3,4}, 'set(integer)' );
```

$$true \tag{3.483}$$

```
> type( {x,2,3,y}, 'set({integer, name})' );
```

$$true \tag{3.484}$$

Similarly, the type **list(type)** matches a *list* of elements of type **type**.

```
> type( [ 2..3, 5..7 ], 'list(range)' );
```

$$true \tag{3.485}$$

Note that e^2 is not of type **anything^2**.

```
> exp(2);
```

$$e^2 \tag{3.486}$$

```
> type( (3.486), 'anything^2' );
```

$$false \tag{3.487}$$

Because e^2 is the typeset version of **exp(2)**, it does not match the type **anything^2**.

```
> type( exp(2), 'exp'(integer) );
```

$$true \tag{3.488}$$

The next example illustrates why you should use unevaluation quotes (**'**) to delay evaluation when including Maple commands in **type** expressions.

```
> type( int(f(x), x), int(anything, anything) );
```

```
Error, testing against an invalid type
```

An error occurs because Maple evaluates **int(anything, anything)**.

```
> int(anything, anything);
```

$$\frac{1}{2}\,anything^2 \tag{3.489}$$

This is not a valid type. If you enclose the **int** command in unevaluation quotes, the type checking works as intended.

```
> type( int(f(x), x), 'int'(anything, anything) );
```

$$true \tag{3.490}$$

The type **specfunc(type, f)** matches the function **f** with zero or more arguments of type **type**.

```
> type( exp(x), 'specfunc(name, exp)' );
```

$$true \tag{3.491}$$

```
> type( f(), 'specfunc(name, f)' );
```

$$true \tag{3.492}$$

The type **function(type)** matches any function with zero or more arguments of type **type**.

```
> type( f(1,2,3), 'function(integer)' );
```

$$true \tag{3.493}$$

```
> type( f(1,x,Pi), 'function( {integer, name} )' );
```

$$true \tag{3.494}$$

In addition to testing the type of arguments, you can test the number of arguments. The type **anyfunc(t1, ..., tn)** matches any function with **n** arguments of the listed types in the correct order.

```
> type( f(1,x), 'anyfunc(integer, name)' );
```

$$true \tag{3.495}$$

```
> type( f(x,1), 'anyfunc(integer, name)' );
```

$$false \tag{3.496}$$

```
> type( f(x), 'anyfunc(integer, name)' );
```

$$\mathit{false} \tag{3.497}$$

Another useful variation is to use the **And**, **Or**, and **Not** type constructors to create Boolean combinations of types. Note that these are different from the **and**, **or**, and **not** logical operators.

```
> type(Pi, 'And( constant, numeric)');
```

$$\mathit{false} \tag{3.498}$$

Pi is of type **symbol**, not of type **numeric**.

```
> type(Pi, 'And( constant, Not(numeric))');
```

$$\mathit{true} \tag{3.499}$$

For more information on structured types, refer to the type,structured help page. For more information on how to define your own types, refer to the type,defn help page.

The indets command is useful for extracting a set of all the subexpressions of a given type.

```
indets( expr, atype)
```

```
> indets(z-exp(x^2-1)+1, 'name');
```

$$\{x, z\} \tag{3.500}$$

```
> indets(Pi+3^(1/2)+4.1, 'integer');
```

$$\{3\} \tag{3.501}$$

```
> indets(Pi+3^(1/2)+4.1, 'numeric');
```

$$\left\{3, \frac{1}{2}, 4.1\right\} \tag{3.502}$$

```
> indets(Pi+3^(1/2)+4.1, 'constant');
```

$$\left\{3, \frac{1}{2}, 4.1, \pi, \sqrt{3}, \pi + \sqrt{3} + 4.1\right\} \tag{3.503}$$

Note that the **indets** command analyzes the entire expression so that the base of the exponent $\sqrt{3}$ is recognized as an integer. If you want to select only subexpressions from the top level, use the command select described in *The select, remove, and selectremove Commands (page 183)*.

If you want to test whether that an expression has a subexpression of a given type, use the hastype command rather than the **indets** command since it avoids building a potentially large set of expressions.

```
> hastype([1,2,3.,5.,6.,7.], 'float');
```

$$true \tag{3.504}$$

3.16 Exercises

1. Find the numerator and denominator of the irreducible form of **4057114691** divided by **4404825097799**.
2. Construct floating-point numbers using the floating-point number constructor. Construct the number **917.3366** using a positive exponent, and then using a negative exponent. Construct a floating-point approximation of **1/3**.
3. Without using the **Digits** environmental variable, find the difference between π estimated to **20** digits and **10** digits.
4. Calculate the negative complex root of **-1369**, and then sum **3** and the root. Find the inverse of this complex sum. Find the inverse of **(a*b)/c+((a-d)/(b*e))*I)** in standard form, where **a**, **b**, **c**, **d**, and **e** are real.
5. The Fibonacci numbers are a sequence of numbers. The first two numbers in the sequence are zero (**0**) and one (**1**). For n greater than two, the nth number in the sequence is the sum of the two preceding numbers. Assign values to indexed names representing the first, second, and general Fibonacci numbers.
6. Using the time command, determine the time required to multiply two ten-by-ten matrices.
7. Use Maple to verify de Morgan's laws.
8. Contrast the behavior of functions and expressions by performing the following commands.
   a Define a function **f** equal to x^3. Define an expression **g** equal to x^3.
   b Evaluate **f** and **g** at **2**.
   c Evaluate **f** and **g** at **y**.
   d Assign the value **2** to **x**. Evaluate **f** and **g**.
9. Swap the values of two variables using one statement.
10. Sum the smallest **100** prime integers.

Hint: Use the ithprime or nextprime function.

4 Basic Data Structures

The appropriate use of data structures is an important part of writing efficient programs. Maple provides various data structures that can be used to help make your programs efficient.

4.1 In This Chapter

- Defining and manipulating sets, lists, tables, Arrays, records, stacks, and queues
- Converting between data structures
- Mathematical versus programmer indexing for Arrays
- Performance comparisons of data structures

4.2 Introduction

Maple provides various data structures that you can use for programming and interacting with Maple functions. This chapter focuses on the use of data structures in programming. However, the sections *Lists (page 126)* and *Sets (page 132)* may be useful for users who want to construct arguments for Maple functions.

Maple has many data structures that provide similar functionality, but certain data structures are better suited for certain types of operations. Therefore, when choosing which data structures to use, it is important to select a structure that performs well on the operations used in your code.

Many aspects affect the performance of data structures. However, in Maple, the provided data structures can be divided into two basic classes: *mutable* and *immutable*. The mutable data structures can be modified, that is, the values they store can change. The immutable data structures cannot be changed after they are created. Instead, copies of these structures can be made with different contents. This difference in behavior can have significant impact on the performance of code that uses these structures.

4.3 Immutable Data Structures

Immutable data structures can be useful when storing a fixed collection of elements. Also, because immutable structures are more compact than mutable data structures, especially when storing a small number of elements, they are more memory efficient.

Immutable structures are created with their contents fixed. This means that they cannot be modified in-place. When you change an immutable data structure, a new copy is created with the modified contents and stored as a distinct object in memory. Thus, immutable structures may not be the correct choice if you want to modify the contents of a structure.

In Maple, there are two basic immutable data structures: lists and sets.

Lists

A *list* stores an *ordered* sequence of expressions. The ordering of the elements in a list is fixed when the list is created. Lists, in contrast to sets, will maintain duplicate elements.

Creating Lists

The easiest way to create a list is to enclose a sequence of expressions in square brackets ([]). A sequence of expressions is a series of comma-separated expressions.

```
[ sequence ]
```

This creates a list that contains the elements of **sequence** in the specified order. In the case where **sequence** is empty, **[]** represents an empty list. Compare the results of these examples to those in the *Sets (page 132)* section.

```
> [x, y, y];
```

$$[x, y, y] \tag{4.1}$$

```
> [a, 1, b, 2];
```

$$[a, 1, b, 2] \tag{4.2}$$

```
> [y[1], x, x[1], y[1]];
```

$$[y_1, x, x_1, y_1] \tag{4.3}$$

The elements of a list can be any expressions, even other lists.

```
> L := [[1], [2, a], [X, Y, Z]];
```

$$L := [[1], [2, a], [X, Y, Z]] \tag{4.4}$$

In Maple, nested lists whose inner lists have the *same* number of elements have a special name, **listlist**.

```
> M := [[a,b], [1,2], [3, 4]];
```

$$M := [[a, b], [1, 2], [3, 4]] \tag{4.5}$$

```
> type(M, list);
```

$$true \tag{4.6}$$

```
> type(L, listlist);
```

$$false \tag{4.7}$$

```
> type(M, listlist);
```

$$true \tag{4.8}$$

Many Maple functions return sequences. Thus, enclosing a call to one of those functions in square brackets [] creates a list. For example, the seq command generates sequences.

```
> [ seq( x^j, j=1..3) ];
```

$$[x, x^2, x^3] \tag{4.9}$$

The op command can be used to extract the sequence of elements in a list.

```
> L := [1,2,3];
```

$$L := [1, 2, 3] \tag{4.10}$$

```
> op(L);
```

$$1, 2, 3 \tag{4.11}$$

Thus **op** can be used to create new lists based on existing lists. For example, you can create a new list with an additional element added to the start of the list.

```
> L2 := [ 0, op(L) ];
```

$$L2 := [0, 1, 2, 3] \tag{4.12}$$

A list with another element added to the end of the list can be created in a similar way.

```
> L3 := [ op(L2), 4 ];
```

$$L3 := [0, 1, 2, 3, 4] \tag{4.13}$$

Multiple lists can also be combined into a single list.

```
> L4 := [ op(L), op(L2), op(L3) ];
```

$$L4 := [1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 4] \tag{4.14}$$

Accessing Data Stored in a List

The selection operation, [], can be used to read an element from a list.

```
> L := [1,2,3];
```

$$L := [1, 2, 3] \tag{4.15}$$

```
> L[1];
```

$$1 \tag{4.16}$$

```
> L[2];
```

$$2 \tag{4.17}$$

```
> L[3];
```

$$3 \tag{4.18}$$

You can also specify a range in the selection operation to extract a sublist containing the elements that are indexed by that range.

```
> L := [ seq( i^2, i=1..10 ) ];
```

$$L := [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] \tag{4.19}$$

```
> L[3..6];
```

$$[9, 16, 25, 36] \tag{4.20}$$

```
> L[5..8];
```

$$[25, 36, 49, 64] \tag{4.21}$$

While it is possible to make an assignment to a list index, this operation can be inefficient since it creates a new list. In fact, assignment to a large list is not permitted in Maple and will produce an error. Assigning a list element is a common error, so if you find yourself wanting to do this, consider using a mutable data structure instead. For more information, see *Mutable Data Structures (page 137)*.

```
> L := [1,2,3]:
> L[1] := 3;
```

$$L_1 := 3 \tag{4.22}$$

```
> L;
```

$$[3, 2, 3] \tag{4.23}$$

L is now a new list with a different element at index 1. Thus, assigning to a single element of a list causes the entire list to be copied in the same way as using the **subsop** command. In fact, the previous example is equivalent to the following except in how the result is displayed.

```
> L := [1,2,3]:
> L := subsop(1=3, L);
```

$$L := [3, 2, 3] \tag{4.24}$$

If you attempt to assign to an index to a large list, an error will occur. Therefore, if you need to make a copy of a list with one changed element, it is recommended that you use the **subsop** command instead.

```
> LL := [ seq( i, i=1..200 ) ]:
> LL[1] := -1;
Error, assigning to a long list, please use Arrays
> subsop(1=-1, LL);
```

$$[-1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200] \tag{4.25}$$

Determining If an Element Is in a List

To test if an expression is contained in a list, use the member function.

```
> member( 1, [ 1,2,3 ] );
```

$$true \tag{4.26}$$

```
> member( 1, [ 2,3,4 ] );
```

$$false \tag{4.27}$$

You can also use the in operator.

```
> evalb( 1 in [1,2,3] );
```

$$true \tag{4.28}$$

```
> evalb( 1 in [2,3,4] );
```

$$false \tag{4.29}$$

Getting the Number of Elements in a List

To find the length of a list, use the numelems command.

```
> numelems( [ 1,2,3 ] );
```

$$3 \tag{4.30}$$

```
> numelems( [ 1,2,3,4,5 ] );
```

$$5 \tag{4.31}$$

```
> numelems( [ seq( i, i=1..127 ) ] );
```

$$127 \tag{4.32}$$

This can be useful for many tasks, for example, using lists in a loop. For more information on **selectremove**, see *Filtering Data Structure Elements (page 153)*.

```
> L := [seq( i, i=2..100)]:
> divisor := 2:
> while ( numelems( L ) > 0 )
  do
     divisible, L := selectremove( i->(i mod divisor = 0), L ):
     n := numelems( divisible );
     if ( n > 0 ) then
        printf( "%d integer%s whose smallest prime divisor is %d\n",

              n, `if`( n > 1, "s", "" ), divisor ):
     end if;
     divisor := nextprime( divisor );
  end do:
```

```
50 integers whose smallest prime divisor is 2
17 integers whose smallest prime divisor is 3
7 integers whose smallest prime divisor is 5
4 integers whose smallest prime divisor is 7
1 integer whose smallest prime divisor is 11
1 integer whose smallest prime divisor is 13
1 integer whose smallest prime divisor is 17
1 integer whose smallest prime divisor is 19
1 integer whose smallest prime divisor is 23
1 integer whose smallest prime divisor is 29
1 integer whose smallest prime divisor is 31
1 integer whose smallest prime divisor is 37
1 integer whose smallest prime divisor is 41
1 integer whose smallest prime divisor is 43
1 integer whose smallest prime divisor is 47
1 integer whose smallest prime divisor is 53
1 integer whose smallest prime divisor is 59
1 integer whose smallest prime divisor is 61
1 integer whose smallest prime divisor is 67
1 integer whose smallest prime divisor is 71
1 integer whose smallest prime divisor is 73
1 integer whose smallest prime divisor is 79
1 integer whose smallest prime divisor is 83
1 integer whose smallest prime divisor is 89
1 integer whose smallest prime divisor is 97
```

Sorting a List

The sort command can create a new list with sorted elements from any given list. By default, **sort** arranges elements in *ascending* order.

```
> sort( [ 4,2,3 ] );
```

$$[2, 3, 4] \tag{4.33}$$

The **sort** command can also accept a second argument that specifies the ordering to use when sorting the elements.

```
> sort( [4,2,3], `>` );
```

$$[4, 3, 2] \tag{4.34}$$

Applying a Function to the Contents of a List

It is often useful to be able to apply a function to all the elements of a list. The map command performs this operation in Maple.

```
> L := [ seq( Pi*i/4, i=0..3 ) ]:
> map( sin, L );
```

$$\left[0, \frac{1}{2}\sqrt{2}, 1, \frac{1}{2}\sqrt{2}\right] \tag{4.35}$$

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

$$\left[1, \frac{1}{2}\sqrt{2}, 0, -\frac{1}{2}\sqrt{2}\right] \tag{4.36}$$

Maple provides other operations that can work with the members of a list, such as add and mul.

```
> add( i, i in [ seq( j, j=1..100 ) ] );
```

$$5050 \tag{4.37}$$

```
> mul( i^2, i in [ 1,2,3,4,5,6,7,8,9,10 ] );
```

$$13168189440000 \tag{4.38}$$

Finally, a **for** loop can be combined with the **in** operator to loop over the contents of a list.

```
> for i in [1,2,3,4]
  do
      print( i^2 );
  end do;
```

$$1$$
$$4$$
$$9$$
$$16 \tag{4.39}$$

Sets

A set is an *unordered* sequence of *unique* expressions. When a set is created, Maple reorders the expressions to remove duplicate values and to make certain operations faster.

Creating Sets

The easiest way to create a set is to enclose a sequence of expressions in braces ({}).

```
{ sequence }
```

When Maple creates the set, it performs automatic simplification. This process creates a set that contains the elements of **sequence**; however, during the automatic simplification process, any duplicate elements are removed and the remaining elements are reordered.

Compare the results of these examples to those in the *Lists (page 126)* section.

```
> {x, y, y};
```

$$\{x, y\} \tag{4.40}$$

```
> {a, 1, b, 2};
```

$$\{1, 2, a, b\} \tag{4.41}$$

```
> {y[1],x,x[1],y[1]};
```

$$\{x, x_1, y_1\} \tag{4.42}$$

Similar to lists, sets can be created using functions such as **seq** that return sequences.

```
> { seq( i mod 3, i=1..10 ) };
```

$$\{0, 1, 2\} \tag{4.43}$$

Again, similar to lists, the **op** command can be used to extract the sequence of elements in a set.

```
> S := {1,2,3};
```

$$S := \{1, 2, 3\} \tag{4.44}$$

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

$$1, 2, 3 \tag{4.45}$$

However, unlike lists, Maple provides operations for set arithmetic, so for sets **op** is somewhat less important.

Set Arithmetic

Maple provides operators for mathematical set manipulations: union, minus, intersect, and subset. These operators allow you to perform set arithmetic in Maple.

```
> s := {x,y,z};
```

$$s := \{x, y, z\} \tag{4.46}$$

```
> t := {y,z,w};
```

$$t := \{w, y, z\} \tag{4.47}$$

```
> s union t;
```

$$\{w, x, y, z\} \tag{4.48}$$

```
> s minus t;
```

$$\{x\} \tag{4.49}$$

```
> s intersect t;
```

$$\{y, z\} \tag{4.50}$$

```
> s subset t;
```

$$false \tag{4.51}$$

```
> s subset {w,x,y,z};
```

$$true \tag{4.52}$$

Accessing Data Stored in a Set

The selection operation, [], can be used to read an element from a set. However, unlike lists, the order in which the elements are specified when creating the set may not correspond to the order they are accessed by indexing.

```
> S := {3,2,1}:
> S[1];
```

$$1 \tag{4.53}$$

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

$$2 \tag{4.54}$$

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

$$3 \tag{4.55}$$

Unlike lists, you cannot use the selection operation to create new sets.

```
> S[1] := 4;
Error, cannot reassign the entries in a set
```

You can specify a range in the selection operation to extract the elements indexed by the range.

```
> S2 := { seq( i^2, i=1..10 ) };
```

$$S2 := \{1, 4, 9, 16, 25, 36, 49, 64, 81, 100\} \tag{4.56}$$

```
> S2[3..6];
```

$$\{9, 16, 25, 36\} \tag{4.57}$$

```
> S2[5..8];
```

$$\{25, 36, 49, 64\} \tag{4.58}$$

Determining If an Element Is in a Set

To test if an element is contained in a set, use the **member** function.

```
> member( 1, {1,2,3} );
```

$$\mathit{true} \tag{4.59}$$

```
> member( 1, {2,3,4} );
```

$$\mathit{false} \tag{4.60}$$

You can also use the **in** operator.

```
> evalb( 1 in {1,2,3} );
```

$$\mathit{true} \tag{4.61}$$

```
> evalb( 1 in {2,3,4} );
```

$$\mathit{false} \tag{4.62}$$

Getting the Number of Elements in a Set

To find the number of elements in a set, use the **numelems** command.

```
> numelems( {1,2,3} );
```

$$3 \tag{4.63}$$

```
> numelems( {1,2,3,4,5} );
```

$$5 \tag{4.64}$$

```
> numelems( {seq( i, i=1..127 )} );
```

$$127 \tag{4.65}$$

In this example, the features of sets are used to test Collatz's conjecture on the first million integers. Collatz's conjecture states that given any integer, *i*, if the following function is applied repeatedly, the result will eventually be 1.

```
> collatz := proc( i )
      if ( i = 1 ) then
```

```
        1;
    elif ( type( i, even ) ) then
        i/2;
    else
        3*i+1;
    end if;
end proc:
```

Begin with a set **S** that consists of the integers from 1 to 1 million. Under repeated application of **collatz**, as numbers converge to 1, the set automatically removes duplicate values, until eventually there is only 1 element left. For more information on the use of **map**, see *Applying a Function to the Contents of a Set (page 136)*.

```
> S := {seq( i, i=1..1000000)}:
> while ( numelems( S ) > 1 )
  do
      S := map( collatz, S ):
  end do:
> S;
```

$$\{1\} \tag{4.66}$$

Applying a Function to the Contents of a Set

As with lists, it can be useful to apply a function to all of the elements of a set. The **map** command works on sets, as it does with lists.

```
> S := { seq( Pi*i/4, i=0..3 ) }:
> map( sin, S );
```

$$\left\{0, 1, \frac{1}{2}\sqrt{2}\right\} \tag{4.67}$$

```
> map( cos, S );
```

$$\left\{0, 1, -\frac{1}{2}\sqrt{2}, \frac{1}{2}\sqrt{2}\right\} \tag{4.68}$$

Notice that when applying a function to a set, the output is also a set, which means the elements are reordered and duplicate elements are removed.

Maple provides other operations that can work with the members of a list, such as **add** and **mul**.

```
> add( i, i in { seq( j, j=1..100 ) } );
```

$$5050 \tag{4.69}$$

```
> mul( i^2, i in { 1,2,3,4,5,6,7,8,9,10 } );
```

$$13168189440000 \tag{4.70}$$

Finally a **for** loop can be combined with the **in** operator to loop over the contents of a set. Note that the set has been reordered.

```
> for i in {1,4,3,2}
  do
      print( i^2 );
  end do;
```

$$1$$
$$4$$
$$9$$
$$16 \tag{4.71}$$

4.4 Mutable Data Structures

Mutable data structures are structures whose contents can be changed.

The most flexible mutable data structure provided by Maple is the table.

Tables

A *table* stores a collection of index/entry pairs. For a given index, the table contains a particular value, called an entry. Index/entry pairs can be created or removed, or the value associated with an index can be modified.

Creating Tables

A new table can be created by calling the table function.

```
> t := table();
```

$$t := table([\,]) \tag{4.72}$$

With no arguments, table creates a new empty table. To create a table that contains certain index/entry pairs, specify the pairs as a list of equations. The left-hand side of an equation is the index and the right-hand side is the entry.

```
> t := table( [ 1=2, a=b, f(x)=y ] );
```

$$t := table([1 = 2, f(x) = y, a = b]) \tag{4.73}$$

If the given list contains one or more expressions that are not equations, the list is treated as a list of entries and the indices are the positions of the entries in the list (1, 2, 3, ...).

```
> t := table( [ a, b, c=d ] );
```

$$t := table([1 = a, 2 = b, 3 = (c = d)]) \tag{4.74}$$

Note that **c=d** is treated as a entry and not an index/entry pair.

Tables are also created implicitly when you assign to an indexed name.

```
> t2[new] := 10;
```

$$t2_{new} := 10 \tag{4.75}$$

```
> eval(t2);
```

$$table([new = 10]) \tag{4.76}$$

Accessing Stored Values

Table indexing is performed using the selection operation, []. To extract data from a table, specify an index in square brackets. The corresponding entry is returned.

```
> t := table( [1=2,a=b,f(x)=y] );
```

$$t := table([1 = 2, f(x) = y, a = b]) \tag{4.77}$$

```
> t[1];
```

$$2 \tag{4.78}$$

```
> t[a];
```

$$b \tag{4.79}$$

```
> t[f(x)];
```

$$y \tag{4.80}$$

If the table does not contain a entry associated with the index, an unevaluated table reference is returned.

```
> t[2];
```

$$t_2 \tag{4.81}$$

The selection operation can also be used to add new index/entry pairs to the table.

```
> t[2] := 3;
```

$$t_2 := 3 \tag{4.82}$$

```
> t[c] := d;
```

$$t_c := d \tag{4.83}$$

```
> t[sin(x)] := 1;
```

$$t_{\sin(x)} := 1 \tag{4.84}$$

```
> t[2];
```

$$3 \tag{4.85}$$

```
> t[c];
```

$$d \tag{4.86}$$

```
> t[sin(x)];
```

$$1 \tag{4.87}$$

Removing an Element

The best way to remove an element from a table is to call the unassign function.

```
> t[1] := x;
```

$$t_1 := x \tag{4.88}$$

```
> t[sin(x)] := y;
```

$$t_{\sin(x)} := y \tag{4.89}$$

```
> unassign( 't[1]' );
> t[1];
```

$$t_1 \tag{4.90}$$

```
> unassign( 't[sin(x)]' );
> t[sin(x)];
```

$$t_{\sin(x)} \tag{4.91}$$

The selection operation can also be used to remove an index/entry pair from a table. By assigning the unevaluated table entry to its name, that element is removed from the table. This can be done by using unevaluation quotes (') or the evaln command.

```
> t[1] := x;
```

$$t_1 := x \tag{4.92}$$

```
> t[1] := 't[1]';
```

$$t_1 := t_1 \tag{4.93}$$

```
> t[1];
```

$$t_1 \tag{4.94}$$

```
> t[sin(x)] := y;
```

$$t_{\sin(x)} := y \tag{4.95}$$

```
> t[sin(x)] := evaln(t[sin(x)]);
```

$$t_{\sin(x)} := t_{\sin(x)} \tag{4.96}$$

```
> t[sin(x)];
```

$$t_{\sin(x)} \tag{4.97}$$

Getting the Number of Elements Stored in a Table

The **numelems** function returns the number of elements stored in a table.

```
> numelems( table( [1] ) );
```

$$1 \tag{4.98}$$

```
> numelems( table( [1,2,3,4,5] ) );
```

$$5 \tag{4.99}$$

```
> numelems( table( [seq( i, i=1..127)] ) );
```

$$127 \tag{4.100}$$

Checking If an Index Is Used

It is often useful to know if a particular index has a value in a table. Use the assigned function to check if a table index has an associated entry.

```
> t := table( [1=1] ):
> assigned( t[1] );
```

$$true \tag{4.101}$$

```
> assigned( t[2] );
```

$$false \tag{4.102}$$

Evaluation Rules for Tables

Tables, like procedures, use last name evaluation. If a name is assigned a table, the result of evaluating that name is the name and not the table assigned to the name. For more information about last name evaluation, refer to the last_name_eval help page.

```
> t := table([1,2,3,4]);
```

$$t := table([1 = 1, 2 = 2, 3 = 3, 4 = 4]) \tag{4.103}$$

```
> t;
```

$$t \tag{4.104}$$

To get the assigned value (the table), use the eval command.

```
> eval(t);
```

$$table([1 = 1, 2 = 2, 3 = 3, 4 = 4]) \tag{4.105}$$

Extracting Data

Tables are often used as simple containers for data. Sometimes, it is useful to have a list of the indices used in the table. Maple provides the indices function for this purpose.

```
> t := table( [a=1, b=2, c=3, d=4] );
```

$$t := table([c = 3, b = 2, d = 4, a = 1]) \tag{4.106}$$

```
> indices( t );
```

$$[c], [b], [d], [a] \tag{4.107}$$

You may not expect to see that **indices** returns a sequence of lists, where each list contains the index. This is because Maple allows sequences to be used as indices in tables.

```
> t2 := table( [ a=1, b=2, (a,b,c)=3 ] );
```

$$t2 := table([(a, b, c) = 3, b = 2, a = 1]) \tag{4.108}$$

```
> indices( t2 );
```

$$[a, b, c], [b], [a] \tag{4.109}$$

If the indices were not wrapped in a list, it would be impossible to determine if an index is a single expression or a sequence of expressions. Since using sequences as indices is uncom-

mon, **indices** accepts a **nolist** option, for which **indices** returns a simple sequence and does not wrap each index in a list.

```
> indices( t, 'nolist' );
```

$$c, b, d, a \tag{4.110}$$

Note that, with the **nolist** option, indices that are sequences are not returned properly.

```
> indices( t2, 'nolist' );
```

$$a, b, c, b, a \tag{4.111}$$

You can also use the **entries** function to get all the values stored in the table.

```
> entries( t );
```

$$[3], [2], [4], [1] \tag{4.112}$$

```
> entries( t, 'nolist' );
```

$$3, 2, 4, 1 \tag{4.113}$$

To extract the index/entry pairs as a sequence of equations, use the **pairs** option to either of the **indices** or **entries** commands.

```
> entries( t, 'pairs' );
```

$$c = 3, b = 2, d = 4, a = 1 \tag{4.114}$$

Copying Tables

If you assign a table to multiple names, all the names reference the same table. Thus, changes to the table using one name are visible from the other names.

```
> t := table( [a=1,b=2,c=3] );
```

$$t := table([c = 3, b = 2, a = 1]) \tag{4.115}$$

```
> t1 := eval( t );
```

$$t1 := table([c = 3, b = 2, a = 1]) \tag{4.116}$$

```
> t[d] := 4;
```

$$t_d := 4 \tag{4.117}$$

```
> eval( t );
```

$$table([c = 3, b = 2, d = 4, a = 1]) \tag{4.118}$$

```
> eval( t1 );
```

$$table([c = 3, b = 2, d = 4, a = 1]) \tag{4.119}$$

If you want to create a copy of a table, use the copy function so that the tables can be modified independently.

```
> t1 := copy( t );
```

$$t1 := table([c = 3, b = 2, d = 4, a = 1]) \tag{4.120}$$

```
> t[e] := 5;
```

$$t_e := 5 \tag{4.121}$$

```
> eval( t );
```

$$table([c = 3, e = 5, b = 2, d = 4, a = 1]) \tag{4.122}$$

```
> eval( t1 );
```

$$table([c = 3, b = 2, d = 4, a = 1]) \tag{4.123}$$

Applying a Function to the Contents of a Table

The **map** function works with tables as it does with lists and sets. When executing a **map** on a table, the mapped function is given the value associated with an index. In the returned table, the result is the entry associated with the index.

```
> t := table( [ x, x^2+2, x^3-x+1, 1/x^2 ] );
```

$$t := table\left(\left[1 = x, 2 = x^2 + 2, 3 = x^3 - x + 1, 4 = \frac{1}{x^2}\right]\right) \tag{4.124}$$

```
> map( diff, t, x );
```

$$table\left(\left[1 = 1, 2 = 2\,x, 3 = 3\,x^2 - 1, 4 = -\frac{2}{x^3}\right]\right) \tag{4.125}$$

You can use the **indices** and **entries** functions to produce a list that can be mapped over or used in a for-in loop. You can also use this technique to modify the original table.

```
> for i in entries(t,'pairs')
  do
     t[lhs(i)] := int( rhs(i), x );
  end do;
```

$$-\frac{1}{x} \tag{4.126}$$

```
> eval(t);
```

$$table\left(\left[1 = \frac{1}{2}x^2, 2 = \frac{1}{3}x^3 + 2x, 3 = \frac{1}{4}x^4 - \frac{1}{2}x^2 + x, 4 = -\frac{1}{x}\right]\right) \tag{4.127}$$

Arrays

In Maple, an Array stores data as an n-dimensional rectangular block (*rtable*), that is, an Array has 1 or more dimensions and each dimension has an range of integer indices. By specifying one integer from each range, an element of the Array can be indexed.

Because Arrays are mutable, the values stored in an Array can change.

Creating Arrays

To create an Array in Maple, use the Array command and specify the ranges for the dimensions. This creates a new Array with each entry initialized to 0. For Arrays, the ranges do not need to start at 1.

```
> Array( 1..3 ); # 1 dimensional Array
```

$$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \tag{4.128}$$

```
> Array( 1..3, 1..4 ); # 2 dimensional Array
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.129}$$

```
> Array( 1..3, 2..5, -1..1 ); # 3 dimensional Array
```

$$\begin{bmatrix} 1..3 \text{ x } 2..5 \text{ x } -1..1 \text{ Array} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{4.130}$$

When creating an Array, you can also specify a generator function to populate the Array with data. The generator function takes an index as an argument and returns a value for the corresponding entry.

```
> Array( 1..3, x->x+1 );
```

$$\begin{bmatrix} 2 & 3 & 4 \end{bmatrix} \tag{4.131}$$

```
> Array( 1..3, 1..4, (x,y)->(x+y) );
```

$$\begin{bmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix} \tag{4.132}$$

You can also provide the data for the Array by specifying the data as a list or nested lists.

```
> Array( [1,2,3] );
```

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \tag{4.133}$$

```
> Array( [[1,2],[3,4],[5,6]] );
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \tag{4.134}$$

Basic Data Access

Arrays are implemented in Maple as a type of rtable, a structure also used for Matrices and Vectors. This means that Arrays have two different indexing mechanisms: mathematical indexing and programmer indexing. Mathematical indexing is intended for use when the Array is viewed as a mathematical object. Programmer indexing provides functionality that is more convenient when using Arrays as a programming tool.

The basic indexing operator, [], provides mathematical indexing. Programmer indexing is accessed by using round brackets, (). For Arrays whose dimension ranges all start at 1, the two indices behave similarly.

```
> A := Array( 1..2, 1..3 ):
> A[1,1] := 1;
```

$$A_{1,1} := 1 \tag{4.135}$$

```
> A(2,1) := 2;
```

$$A(2,1) := \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix} \tag{4.136}$$

```
> A(1,1);
```

$$1 \tag{4.137}$$

```
> A[2,1];
```

$$2 \tag{4.138}$$

You may notice that the assignment that uses programmer indexing is displayed differently than the assignment that uses mathematical indexing. This is because the result of an assignment to a programmer indexed Array is the entire array. This can be important when working with large sub-Arrays.

When the ranges do not start at 1, mathematical and programmer indexing are different. Mathematical indexing requires that the indices match the specified ranges, but programming indexing always normalizes the ranges to start at 1.

```
> A := Array( 3..4, 5..6, (x,y)->x+y ):
> A[3,5];
```

$$8 \tag{4.139}$$

```
> A(3,5);
Error, index out of bounds
> A(1,1);
```

$$8 \tag{4.140}$$

This means that programmer indexing can always take advantage of negative indexing, which normally only works when the ranges start at 1. Negative indexing counts backwards from the end of the range.

```
> A[3,-1];
Error, Array index out of range
> A[3,6];
```

$$9 \tag{4.141}$$

```
> A(1,-1);
```

$$9 \tag{4.142}$$

Sub-Array Access

A sub-Array of an Array can be accessed by specifying a subrange in place of the indices.

```
> A := Array( 1..5, 1..5, (x,y)->x+y );
```

$$A := \begin{bmatrix} 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \\ 6 & 7 & 8 & 9 & 10 \end{bmatrix} \tag{4.143}$$

```
> A[1..2,1..3];
```

$$\begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} \tag{4.144}$$

```
> A(2..4,2..3);
```

$$\begin{bmatrix} 4 & 5 \\ 5 & 6 \\ 6 & 7 \end{bmatrix} \tag{4.145}$$

Sub-Array indexing can also be used to assign to the specified sub-Array.

```
> A[2..4,2..3] := Array( [[a,a],[a,a],[a,a]] );
```

$$A_{2..4,\,2..3} := \begin{bmatrix} a & a \\ a & a \\ a & a \end{bmatrix} \tag{4.146}$$

```
> A(4..5,4..5) := Array( [[b,b],[b,b]] );
```

$$A(4..5, 4..5) := \begin{bmatrix} 2 & 3 & 4 & 5 & 6 \\ 3 & a & a & 6 & 7 \\ 4 & a & a & 7 & 8 \\ 5 & a & a & b & b \\ 6 & 7 & 8 & b & b \end{bmatrix} \tag{4.147}$$

Note that the commands perform the same operation, but display the result differently. This is the consequence of an important difference in how the modification is performed. This

can be important when working with large sub-Arrays. Compare the time to perform the assignment in the following examples:

```
> N := 4000:
> A := Array( 1..N, 1..N, (x,y)->rand() ):
> B := Array( 1..N, 1..N ):
> t := time():
> B[1001..4000,1001..4000]:=A[1..3000,1..3000]:
> time()-t;
```

$$0.195 \tag{4.148}$$

```
> t := time():
> B(1001..4000,1001..4000):=A(1..3000,1..3000):
> time()-t;
```

$$0.059 \tag{4.149}$$

The difference in running time of these copies is due to the difference in the result of an assignment to an Array index. For mathematical indexing, a new 3000 by 3000 Array must be created as the result. With programmer indexing, the result is the Array being assigned to in its entirety - an object that already exists.

Automatic Resizing

One of the most important differences between mathematical and programmer indexing is automatic resizing. When reading from or writing to an entry using mathematical indexing, an index that is outside the bounds of the Array will raise an exception.

```
> A := Array( [[1,2,3],[4,5,6]] );
```

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{4.150}$$

```
> A[1,4];
Error, Array index out of range
> A[1,4] := a;
Error, Array index out of range
```

However, programmer indexing allows you to write to an entry that is outside the bounds of the current Array. Instead of raising an exception, the Array are automatically resized so that the element can be stored. Reading from an out-of-bounds index will still raise an exception.

```
> A(1,4) := a;
```

$$A(1,4) := \begin{bmatrix} 1 & 2 & 3 & a \\ 4 & 5 & 6 & 0 \end{bmatrix} \tag{4.151}$$

```
> A(3,5);
Error, index out of bounds
> A(3,5) := b;
```

$$A(3,5) := \begin{bmatrix} 1 & 2 & 3 & a & 0 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & b \end{bmatrix} \tag{4.152}$$

More Array Indexing

There are more features of, and differences between, mathematical and programmer indexing. For more information on Array indexing, refer to the rtable_indexing help page.

Getting the Number of Elements in an Array

The **numelems** function returns the number of elements defined by the bounds of an Array.

```
> numelems( Array( [1,2,3,4,5] ) );
```

$$5 \tag{4.153}$$

```
> numelems( Array( [[1,2,3],[4,5,6]] ) );
```

$$6 \tag{4.154}$$

Getting the Bounds of an Array

As Array bounds may not start at 1, it is important that procedures that accept Arrays be aware of this possibility. The **upperbound** and lowerbound functions can be used to get the bounds on the ranges of an Array.

```
> printer := proc( A )
    local lower, upper, i, j;
    lower := lowerbound( A );
    upper := upperbound( A );
    for i from lower[1] to upper[1]
    do
        for j from lower[2] to upper[2]
        do
            printf( "%a ", A[i,j] );
```

```
        end do;
        printf( "\n" );
    end do;
  end proc:
> printer( Array( [[1,2],[3,4]] ) ):
1 2
3 4
> printer( Array( 2..5, 5..7, (x,y)->(x+y) ) ):
7 8 9
8 9 10
9 10 11
10 11 12
```

Copying an Array

As with tables, having multiple variables referencing the same Array does not create new copies of the Array. You can use **copy** to copy the Array.

```
> A := Array( 1..2, 1..2 ):
> B := A;
```

$$B := \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{4.155}$$

```
> A[1,1] := 1:
> B;
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{4.156}$$

```
> B := copy(A):
> A[1,2] := 2:
> A;
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 0 \end{bmatrix} \tag{4.157}$$

```
> B;
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{4.158}$$

Testing If Two Arrays Are Equal

For Arrays, there are two notions of equality: do two references point to the same Array, or are they different Arrays that store the same values. To determine if two references refer to the same Array, use = and **evalb**. To test if two Arrays contain the same elements, use the **EqualEntries** command.

```
> CompareArray := proc( A, B )
    if A = B then
       print("two names for one array");
    elif EqualEntries(A,B) then
       print("same elements");
    else
       print("at least one element is different");
    end if;
  end proc:
```

```
> A := Array( [[1,2],[3,4]] );
```

$$A := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{4.159}$$

```
> AC := copy(A);
```

$$AC := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{4.160}$$

```
> CompareArray(A,AC);
```

"same elements" (4.161)

```
> AR :=  A;
```

$$AR := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{4.162}$$

```
> CompareArray(A,AR);
```

"two names for one array" (4.163)

```
> B := Array( [[1,2],[3,5]] );
```

$$B := \begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix} \tag{4.164}$$

```
> CompareArray(A,B);
```

"at least one element is different" (4.165)

There are some other advanced notions of equality such as whether or not arrays with undefined entries should be treated as having equal entries, and whether a Matrix and Array with identical entries should be considered the same. The **IsEqual** command in the **ArrayTools** package allows for different solutions for these two issues compared to **EqualEntries**. The **ArrayTools** package contains a variety of functions for working with Arrays. For more information, refer to the ArrayTools help page.

Applying a Function to the Contents of an Array

map can be used with an Array as you would expect

```
> map( x->(x/2), Array( [[1,2,3],[4,5,6]] ) );
```

$$\begin{bmatrix} \frac{1}{2} & 1 & \frac{3}{2} \\ 2 & \frac{5}{2} & 3 \end{bmatrix} \tag{4.166}$$

indices, **entries**, and the **in** operator work with Arrays, so you can use Arrays in **add**, **mul**, and **for** loops. **entries(A,pairs)** can also be used to obtain a list of index/value pairs in the same way that it does for tables.

```
> A := Array( [x,x^3,sin(x)] ):
> for entry in entries(A,'pairs')
  do
     A[lhs(entry)] := diff( rhs(entry), x ):
  end do:
> A;
```

$$\begin{bmatrix} 1 & 3x^2 & \cos(x) \end{bmatrix} \tag{4.167}$$

Better Performance with Numeric Arrays

When creating an Array, you can specify a datatype for the Array elements. The given datatype can be either a Maple type or a hardware datatype specifier: **integer[n]**, **float[n]**, **complex[n]**. **n** refers to the number of bytes of data for each element. For **integer[n]**, **n** can be 1, 2, 4, or 8. For **float[n]** or **complex[n]**, **n** can be 4 or 8. The datatype **integer[4]** uses 4-bytes, or 32-bits per integer, and **integer[8]** uses 8-bytes, or 64-bits. The 64-bit version has a wider range of signed values, but uses more memory.

When assigning values into the Array, Maple will raise an exception if the given value does not match the specified type.

```
> A := Array( [1,2,3,4], datatype=float[8] );
```

$$A := \begin{bmatrix} 1. & 2. & 3. & 4. \end{bmatrix} \tag{4.168}$$

```
> A[1];
```

$$1. \tag{4.169}$$

```
> A[1] := 1.5;
```

$$A_1 := 1.5 \tag{4.170}$$

```
> A[2] := x^2;
```

```
Error, unable to store 'x^2' when datatype=float[8]
```

If you are working with numeric values that can be stored in these hardware types, it can be much faster to use an Array with a hardware type. For more information on numerical programming in Maple, see *Numerical Programming in Maple (page 267)*.

Deprecated: array

The **array** data structure is an older implementation of Arrays. Its use has been deprecated; use Array instead.

4.5 Other Data Structure Operations

Filtering Data Structure Elements

The select, remove, and selectremove functions provide ways to filter the elements of data structures.

`select( f, x )`
`remove( f, x )`
`selectremove( f, x )`

The parameter **f** must be a Boolean-valued function. This function is applied to each of the elements of the data structure **x**. **select** returns the a data structure containing those elements for which **f** returns true. **remove** returns a data structure containing those elements for which **f** returns false. **selectremove** returns two structures, the first consisting of the elements for which **f** returned true and the second consisting of the elements for which **f** returns false.

The type of the return value of these functions matches the type of the argument **x**.

```
> x := [seq(i,i=1..10)];
```

$$x := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \tag{4.171}$$

```
> select( isprime, x );
```

$$[2, 3, 5, 7] \tag{4.172}$$

```
> remove( isprime, x );
```

$$[1, 4, 6, 8, 9, 10] \tag{4.173}$$

```
> selectremove( isprime, x );
```

$$[2, 3, 5, 7], [1, 4, 6, 8, 9, 10] \tag{4.174}$$

Calling **selectremove** is more efficient than calling **select** and **remove** separately.

Converting Data Structures

Maple provides the convert function, which allows various expressions to be converted from one form to another.

```
convert( x, t )
```

convert attempts to convert the expression **x** into the form **t**. In particular, Maple supports conversions between the **list**, **set**, **table**, and **Array** types.

```
> x := [1,2,3,4];
```

$$x := [1, 2, 3, 4] \tag{4.175}$$

```
> convert( x, 'set' );
```

$$\{1, 2, 3, 4\} \tag{4.176}$$

```
> convert( x, 'table' );
```

$$table([1 = 1, 2 = 2, 3 = 3, 4 = 4]) \tag{4.177}$$

```
> convert( x, 'Array' );
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \tag{4.178}$$

4.6 Other Data Structures

Records

In Maple, a record is a structured data type. It allows you to create a fixed-size structure with user-defined fields. You can use records to create customized structures that can make Maple code easier to read and write.

Create a Record

To create a new record, use the Record command. **Record** accepts a sequence of names as parameters. Each name becomes a field in the returned record.

```
> r := Record( 'expression', 'variable' );
```

$$r := Record(expression, variable) \tag{4.179}$$

```
> r:-expression := x^2;
```

$$expression := x^2 \tag{4.180}$$

```
> r:-variable := x;
```

$$variable := x \tag{4.181}$$

```
> int( r:-expression, r:-variable );
```

$$\frac{1}{3} x^3 \tag{4.182}$$

If **Record** is passed a single record as an argument, a copy of that record is returned.

```
> r2 := Record( eval(r,1) );
```

$$r2 := Record(expression = x^2, variable = x) \tag{4.183}$$

```
> r2:-expression := sin(x^2);
```

$$expression := \sin(x^2) \tag{4.184}$$

```
> int( r2:-expression, r2:-variable );
```

$$\frac{1}{2} \sqrt{2} \sqrt{\pi}\, \mathrm{FresnelS}\left(\frac{\sqrt{2}\, x}{\sqrt{\pi}} \right) \tag{4.185}$$

Note that you must call **eval** on **r** before passing it into **Record**. This is because records use last name evaluation rules, similar to tables.

Record Equality

As with Arrays, two references to Records are considered equal if they reference the same structure. Two different structures that have the same fields and values are not considered equal.

```
> r := Record( 'a'=1, 'b'=2, 'c'=3 ):
> rc := r:
> r2 := Record( 'a'=1, 'b'=2, 'c'=3 ):
> evalb( r = rc );
```

true (4.186)

```
> evalb( r = r2 );
```

false (4.187)

To compare two different records, you can use the verify command with the **record** argument. **verify/record** returns **true** if the two records have the same set of fields with equal values assigned to them.

```
> r3 := Record( 'a'=1, 'b'=2, 'c'=3, 'd'=4 ):
> r4 := Record( 'a'=1, 'b'=2, 'c'=4 ):
> verify( r, r2, 'record' );
```

true (4.188)

```
> verify( r, r3, 'record' );
```

false (4.189)

```
> verify( r, r4, 'record' );
```

false (4.190)

Packed Records

The **Record** constructor function can also be called with the indexed name **Record[packed]**, to produce a packed record.

Unlike a regular record, a packed record does not create a unique instance of each field name for each record instance. When working with thousands of similar records each with many fields, this can save a significant amount of memory.

Fields of packed records do not exhibit last name evaluation. That is, the expression **r:-a** always produces a value, even if that value is a procedure, table, Matrix, Vector, or another record.

Similarly, it is not possible for a packed record field to *not* have a value. The **assigned** function will always return true, and **unassign**ing a packed record field will set its value to **NULL** instead.

Stacks

A stack is an abstract data type that provides two main operations: push and pop. A push places a new value onto the top of the stack and pushes the existing elements down. A pop removes the element from the top of the stack, moving the elements below up. This creates a element access order referred to as *last in first out* (LIFO).

Stacks are useful for many operations. A typical use of a stack is to turn a recursive algorithm into an iterative one. Instead of recursing on elements, those elements get pushed onto a stack. When the current element has been handled, the element on top of the stack is removed and handled next. By using a stack, the recently discovered elements are handled before elements that were already in the stack, which is similar to how a recursive algorithm works.

Creating a Stack

In Maple, you can create a stack by calling stack:-new. If you do not specify any arguments, stack:-new creates an empty stack. Maple stacks are implemented on top of tables.

```
> s := stack:-new():
> stack:-push( 1, s );
```

$$1 \tag{4.191}$$

```
> stack:-push( 2, s );
```

$$2 \tag{4.192}$$

```
> stack:-pop( s );
```

$$2 \tag{4.193}$$

```
> stack:-pop( s );
```

$$1 \tag{4.194}$$

You can also pass values into stack:-new that populate the stack. These elements are pushed in the order specified.

```
> s := stack:-new(1,2,3,4,5):
> stack:-pop( s );
```

$$5 \tag{4.195}$$

```
> stack:-pop( s );
```

$$4 \tag{4.196}$$

Pushing and Popping

To push and pop elements onto the stack, use the stack:-push and stack:-pop functions.

```
> s := stack:-new():
> stack:-push( 1, s ):
> stack:-push( 2, s ):
> stack:-pop( s );
```

$$2 \tag{4.197}$$

```
> stack:-push( 3, s ):
> stack:-pop( s );
```

$$3 \tag{4.198}$$

```
> stack:-pop( s );
```

$$1 \tag{4.199}$$

More Stack Functions

To get the number of elements stored in the stack, call stack:-depth.

```
> s := stack:-new(a,b,c):
> while stack:-depth( s ) > 0
  do
     print( stack:-pop( s ) );
  end do;
```

$$c$$

$$b$$

$$a \tag{4.200}$$

To test if a stack is empty, call stack:-empty.

```
> s := stack:-new(c,b,a):
> while not stack:-empty( s )
  do
     print( stack:-pop( s ) );
  end do;
```

$$a$$

$$b$$

$$c \tag{4.201}$$

You can examine the element on the top of a stack, without removing it, by calling stack:-top.

```
> s := stack:-new(x,x^2,sin(x)):
> stack:-depth(s);
```

$$3 \tag{4.202}$$

```
> stack:-top(s);
```

$$\sin(x) \tag{4.203}$$

```
> stack:-pop(s);
```

$$\sin(x) \tag{4.204}$$

```
> stack:-depth(s);
```

$$2 \tag{4.205}$$

```
> stack:-top(s);
```

$$x^2 \tag{4.206}$$

```
> stack:-pop(s);
```

$$x^2 \tag{4.207}$$

```
> stack:-depth(s);
```

$$1 \tag{4.208}$$

```
> stack:-top(s);
```

$$x \tag{4.209}$$

Queues

The queue is an abstract data type similar to a stack; however, instead of the most recently added element being returned first, the oldest element in the queue is returned first. Elements in a queue are analogous to people waiting in a line. The main operations provided by a queue are **enqueue**, which adds an element to the queue, and **dequeue**, which removes an element from the queue. The access order used by a queue is called *first in first out*, or FIFO.

A queue is used when you want to handle elements in the order that they are discovered. A typical example of using a queue is a breadth-first search of a graph. You dequeue a node and then enqueue any unvisited nodes that are neighbors of the current node. By using a queue, the order in which the nodes are visited is breadth-first.

Create a Queue

To create a queue in Maple, use the queue:-new command.

```
> q := queue:-new():
> queue:-enqueue( q, 1 );
```

$$1 \tag{4.210}$$

```
> queue:-enqueue( q, 2 );
```

$$2 \tag{4.211}$$

```
> queue:-dequeue( q );
```

$$1 \tag{4.212}$$

```
> queue:-dequeue( q );
```

$$2 \tag{4.213}$$

You can also pass values into queue:-new to populate the new queue. The elements are enqueued in the order they are specified.

```
> q := queue:-new( 1,2,3 ):
> queue:-dequeue( q );
```

$$1 \tag{4.214}$$

```
> queue:-dequeue( q );
```

$$2 \tag{4.215}$$

```
> queue:-dequeue( q );
```

$$3 \tag{4.216}$$

Enqueue and Dequeue

You can insert a new element into a queue using queue:-enqueue and remove an element from the queue using queue:-dequeue.

```
> q := queue:-new():
> queue:-enqueue( q, 1 ):
```

```
> queue:-enqueue( q, 2 ):
> queue:-dequeue( q );
```

$$1 \tag{4.217}$$

```
> queue:-enqueue( q, 3 ):
> queue:-dequeue( q );
```

$$2 \tag{4.218}$$

```
> queue:-dequeue( q );
```

$$3 \tag{4.219}$$

More Queue Functions

You can get the number of elements stored in the queue by calling queue:-length.

```
> q := queue:-new(a,b,c):
> while queue:-length( q ) > 0
  do
     print( queue:-dequeue( q ) );
  end do;
```

$$a$$

$$b$$

$$c \tag{4.220}$$

You can test if a queue is empty by calling queue:-empty.

```
> q := queue:-new(c,b,a):
> while not queue:-empty( q )
  do
     print( queue:-dequeue( q ) );
  end do;
```

$$c$$

$$b$$

$$a \tag{4.221}$$

You can examine the front element of a queue, without removing it, by calling queue:-front.

```
> q := queue:-new(x,x^2,sin(x)):
```

```
> queue:-length(q);
```

$$3 \tag{4.222}$$

```
> queue:-front(q);
```

$$x \tag{4.223}$$

```
> queue:-dequeue(q);
```

$$x \tag{4.224}$$

```
> queue:-length(q);
```

$$2 \tag{4.225}$$

```
> queue:-front(q);
```

$$x^2 \tag{4.226}$$

```
> queue:-dequeue(q);
```

$$x^2 \tag{4.227}$$

```
> queue:-length(q);
```

$$1 \tag{4.228}$$

```
> queue:-front(q);
```

$$\sin(x) \tag{4.229}$$

4.7 Data Coercion

Data Coercion refers to the ability to take one data type and automatically convert it into a different data type. This is particularly useful for arguments passed into a procedure, where the expected data type for the procedure is explicitly declared. For more information on data coercion in Maple, see the coercion help page.

Maple provides two methods for enabling data coercion. For more information see *The coercion Modifiers (page 207)*.

4.8 Data Structure Performance Comparisons

Maple provides many different data structures, many of which can be used together to perform specific tasks. However, the different performance characteristics of the data structures means that some are better than others in certain situations.

Indexing

The time to perform an indexed look-up into a list, set, table, and Array are all constant time operations. This means that the time needed to find the element does not vary based on the number of elements stored in the structure. Time to perform a look-up into a list or set is relatively similar and is faster than Arrays, which is faster than a table.

Similarly, writing into a table or Array is also a constant time operation, with Array look-ups being slightly faster than table look-ups.

Membership

The **member** function determines if a particular element is stored in a structure. For lists, this requires a linear search of the data in the list. Therefore, the time is proportional to the total length of the list. A set is sorted, so searches of the list can be performed more quickly. Searching within a set takes time proportional to the log[2] of the number of elements in the set.

You can use a table for very fast membership testing. Use the table *key* as objects you want to test for, and anything you want for the value. You can then call the **assigned** command to test if the element exists in the table. A table index is a constant time operation, so this membership test is also constant time.

```
> N := 2*10^5:
> memtest := proc( D, N )
      local i;

      for i from 1 to N
      do
          member( i, D ):
      end do:
  end proc:
> L := [seq( i, i=1..N )]:
> time(memtest(L,N));
```

$$26.625 \tag{4.230}$$

```
> S := {seq( i, i=1..N )}:
> time(memtest(S,N));
```

$$0.352 \tag{4.231}$$

```
> t := table( [seq( i=1, i=1..N ) ] ):
```

```
> start := time():
  for i from 1 to N
  do
      assigned( t[i] ):
  end do:
  time()-start;
```

0.200 (4.232)

Note that to benchmark the **list** and **set** membership functions, the call to **member** is within a function. This is because of the Maple evaluation rules. If the call to the **member** command is at the top level, the list or set is fully evaluated, which requires inspecting each element of the list or set for each call to **member**. The overhead required for these full evaluations would distort the results.

For more information on the Maple evaluation rules, see *Unevaluated Expressions (page 46)*.

Building a Collection of Data

It is often necessary to build a collection of data when you do not know how many elements you are going to have. You should use a table, Array (using programmer indexing), stack, or queue. All of these mutable structures support adding elements in constant time. Using an immutable data structure is slower; the use of a list or a set is not recommended in this situation.

```
> N := 5*10^4:
```

```
> A := Array( [] ):
  start := time():
  for i from 1 to N
  do
      A( i ) := 1:
  end do:
  time()-start;
```

0.048 (4.233)

```
> t := table():
  start:=time():
  for i from 1 to N
  do
      t[i] := 1:
  end do:
  time()-start;
```

0.092 (4.234)

```
> l := []: # using a list is quite slow
  start := time():
  for i from 1 to N
  do
      l := [ op(l), i ]:
  end do:
  time()-start;
```

$$47.802 \tag{4.235}$$

4.9 Avoiding Common Problems

When working with data structures, there are a few common problems that you may encounter. This section describes some of these problems to help you avoid making these mistakes yourself.

Passing Sequences into Functions

When a sequence is passed into a procedure, each element of the sequence is treated as a separate argument. This can lead to errors if the procedure is unable to handle the multiple arguments, for example, with the **op** command.

```
> s := a,b,c;
```

$$s := a, b, c \tag{4.236}$$

```
> op( 2, s );
```

```
Error, invalid input: op expects 1 or 2 arguments, but received 4
```

Instead, wrap the sequence in a list.

```
> op( 2, [s] );
```

$$b \tag{4.237}$$

Incorrect Index Values

Be careful with the values used for indexing. Specifying values outside valid ranges will raise exceptions. In particular, in Maple, lists and sets start indexing at 1, not 0.

```
> L := [1,2,3,4,5,6,7,8];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8] \tag{4.238}$$

```
> L[0];
```

```
Error, invalid subscript selector
```

```
> L[9];
Error, invalid subscript selector
```

Further, when specifying the endpoints of a range, make sure that the left-hand side of the range specifies an element before the element specified by the right-hand side.

```
> L[6..3];
Error, invalid subscript selector
> L[6..-5];
Error, invalid subscript selector
```

The only exception to this is if the left-hand side of the range is n, then the right-hand side can be n-1 and the result of this range is an empty structure (list or set).

```
> L[6..5];
```

$$[\,] \tag{4.239}$$

Similar exceptions happen with using [] for selection from Arrays.

```
> A := Array( [5,6,7,8,9,10] );
```

$$A := \left[\begin{array}{cccccc} 5 & 6 & 7 & 8 & 9 & 10 \end{array}\right] \tag{4.240}$$

```
> A[7];
Error, Array index out of range
> A[5..3];
Error, inverted range in Array index
```

Array Indices Do Not Always Start at 1

In an Array, the lower bound of the indices may not be 1. If you write a procedure that accepts an Array, you should be prepared to handle Arrays that have been defined for a range of indices that does not start at 1. For more information on how to write procedures that can handle such Arrays, see *Getting the Bounds of an Array (page 149)*.

Do Not Treat Lists and Sets as Mutable

You can use commands such as **op** and **subsop** with lists and sets to create new structures. It is, therefore, possible to treat lists and sets like mutable structures. However, by doing so, you can add a significant amount of processing time to your computations. Make sure that you use actual mutable structures instead.

```
> N := 2*10^4:
```

```
> l := [seq( i=i, i=1..N)]:
> t := table( l ):
  start:=time():
  for i from N to 1 by -1
  do
      t[i] := evaln(t[i]):
  end do:
  time()-start;
```

0.032 (4.241)

```
> start := time():
  for i from N to 1 by -1
  do
      l := subsop( i=NULL, l );
  end do:
  time()-start;
```

20.277 (4.242)

4.10 Exercises

1. Define a set with elements that are the powers of **13** modulo **100** for exponents ranging from **1** to **1000**. Is **5** a member of the set? Why is it beneficial to use a set instead of a list?

   Hint: You can determine the set by using one statement if you use the seq command.

2. Generate the sums of **4** and the first **100** multiples of **3**. Determine the sums that are square-free composite numbers.

   Hint: The **numtheory** package has a function that you need to use.

3. Find floating-point approximations for the sum of the square root and cubic root of each of the first **15** powers of 2.

   Hint: Use map, seq, and zip.

4. Write a procedure that implements the sieve of Eratosthenes: Count the number of integers (less than or equal to a given integer) that are prime.

5 Maple Statements

5.1 In This Chapter

- Introduction
- Expression Statements
- Assignments
- Flow Control
- The use Statement
- Other Statements

5.2 Introduction

A *statement* is a single complete piece of code that Maple can execute. There are many types of statements in Maple, including expression statements, assignment statements, selection statements (if ... then), repetition statements (loops), and program instructions (quit, save, read).

A statement differs from an expression in that it is normally evaluated for effect, rather than for its value. Most statements that do *not* consist of a single expression are formed so as to have a side effect.

5.3 Statement Separators

Statements in Maple must be terminated with a semicolon (**;**) or a colon (**:**).

Statements can be run in Maple one at a time, or multiple statements can be run on one line. If multiple statements are run on one line, the statements must be separated by a statement separator, either a semicolon (**;**) or a colon (**:**).

At the top level, the output of a statement that ends with a colon is hidden.

```
> a:=2: a^2;
```

$$4 \tag{5.1}$$

Note: In the standard interface, for input in 2-D math, the semicolon at the end of a statement can be omitted.

5.4 Expression Statements

The simplest kind of statement in Maple is the expression statement. It consists of an arbitrary expression, whose evaluation constitutes the effect of the statement.

```
> Pi;
```

$$\pi \tag{5.2}$$

```
> sin( Pi - x );
```

$$\sin(x) \tag{5.3}$$

```
> int( sin( Pi - x ), x );
```

$$-\cos(x) \tag{5.4}$$

5.5 Assignments

Assignment statements allow you to associate a value or expression with a name. The assignment statement has the general form

```
lhs := rhs
```

Evaluating the assignment associates the value on the right-hand side of the assignment with the name on the left-hand side. After the assignment has taken effect, the result is the associated value when the assigned name is evaluated.

Here, the name **a** has no assigned value, so it evaluates to itself.

```
> a;
```

$$a \tag{5.5}$$

The following assignment statement associates the value 2 / 3 with the name **a**.

```
> a := 2 / 3;
```

$$a := \frac{2}{3} \tag{5.6}$$

Subsequent evaluation of the name **a** results in the assigned value 2 / 3.

```
> a;
```

$$\frac{2}{3} \tag{5.7}$$

```
> a + 1 / 3;
```

$$1 \tag{5.8}$$

Associate the symbolic expression Pi / 2 with the name **b** by executing the following assignment statement.

```
> b := Pi / 2;
```

$$b := \frac{1}{2}\,\pi \tag{5.9}$$

Subsequently, the assigned value of **b** is used whenever the name **b** appears in an expression.

```
> sin( b );
```

$$1 \tag{5.10}$$

In this expression, the assigned value Pi / 2 of the name **b** is substituted to yield the expression sin(Pi / 2), and then the value of the procedure **sin** at this expression is computed, resulting in the overall value 1 for the expression.

Multiple Assignment

You can perform several assignments in a single statement, known as a *multiple assignment*. This has the general form

```
(lhs1, lhs2, ..., lhsN) := (rhs1, rhs2, ..., rhsN)
```

The parentheses on the right- and left-hand sides of the assignment are not required, but are considered good practice.

For example, the multiple assignment statement

```
> (x, y, z) := ( sin( t ), cos( t ), tan( t ) );
```

$$x, y, z := \sin(t), \cos(t), \tan(t) \tag{5.11}$$

establishes assigned values for all three names **x**, **y**, and **z**.

```
> x;
```

$$\sin(t) \tag{5.12}$$

```
> y;
```

$$\cos(t) \tag{5.13}$$

```
> z;
```

$$\tan(t) \tag{5.14}$$

The number of components on each side of the assignment operator **:=** must be the same.

```
> (a, b, c) := (2, 3);
```

```
Error, ambiguous multiple assignment
```

A common idiom is to use a multiple (double) assignment to swap the values of two variables without introducing an additional temporary variable.

```
> (x, y) := (1, 2):
> x;
```

$$1 \tag{5.15}$$

```
> y;
```

$$2 \tag{5.16}$$

```
> (x, y) := (y, x):
> x;
```

$$2 \tag{5.17}$$

```
> y;
```

$$1 \tag{5.18}$$

Note that using the swap idiom with unassigned names will lead to an infinite recursion.

```
> (u, v) := (v, u);
```

$$u, v := v, u \tag{5.19}$$

Evaluating **u** or **v** (full evaluation) produces an error. If you evaluate one level at a time using eval(u, i), you can see what happens.

```
> u;
> v;
> seq( eval( u, i ), i = 1 .. 10 );
```

$$v, u, v, u, v, u, v, u, v, u \tag{5.20}$$

```
> seq( eval( v, i ), i = 1 .. 10 );
```

$$u, v, u, v, u, v, u, v, u, v \tag{5.21}$$

5.6 Flow Control

A number of Maple statements are used to direct the flow of control in a program; that is, the sequence in which the various statements of the program are run.

Sequencing

The simplest form of a Maple program is a sequence of zero or more statements, separated either by semicolons or colons. A sequence of statements is run in the order in which they are entered.

For example, running these three statements

```
> a := 2;
```

$$a := 2 \tag{5.22}$$

```
> b := 3;
```

$$b := 3 \tag{5.23}$$

```
> sin( a + b );
```

$$\sin(5) \tag{5.24}$$

executes the assignment to the name **a**, then the assignment to the name **b** is executed and, finally, the value of the expression sin(a + b) is computed.

The flow of control in a Maple program consisting of a sequence of statements moves from one statement to the next, in order.

Many Maple statements are *compound* statements that contain statement sequences as constituents.

Branching

The simplest form of flow control is a branching, or **if** statement. Basically, an **if** statement has the syntax

```
if condition then
    statseq
end if
```

in which **condition** is a Boolean-valued expression (that is, one which evaluates to one of the values **true**, **FAIL**, or **false**), and **statseq** is a (possibly empty) sequence of Maple statements, often called the *body* of the **if** statement.

The effect of an **if** statement is to divert the flow of control, under the right conditions, to the body of the statement. If the **condition** expression evaluates to **true**, the flow of control moves into the body of the **if** statement. Otherwise, if the **condition** expression evaluates to **FAIL** or **false**, Maple exits the **if** statement and the flow of control continues at the statement (if any) following the **if** statement.

```
> if 2 < 3 then
     print( "HELLO" )
  end if;
```

"HELLO" (5.25)

```
> if 2 > 3 then
     print( "GOODBYE" )
  end if;
```

More generally, an **if** statement has the syntax

```
if condition then
    consequent
else
    alternative
end if
```

Here, **consequent** and **alternative** are statement sequences. If the **condition** expression evaluates to **true**, the **consequent** branch of the **if** statement is executed. Otherwise, the **alternative** branch is executed.

```
> if 2 < 3 then
     print( "CONSEQUENT" )
  else
     print( "ALTERNATIVE" )
  end if;
```

"CONSEQUENT" (5.26)

```
> if 2 > 3 then
     print( "CONSEQUENT" )
  else
     print( "ALTERNATIVE" )
  end if;
```

"ALTERNATIVE" (5.27)

The most general form of an **if** statement can have several conditions, corresponding consequents, and an optional alternative branch. This general form has the syntax:

```
if condition1 then
    consequent1
elif condition2 then
    consequent2
 ....
else
    alternative
end if
```

in which there can be any number of branches preceded by **elif**. The effect of this general form of the **if** statement is to divert the flow of control into the *first* branch whose conditional expression evaluates to **true**. This means that the order of the **elif** branches can affect the behavior of the **if** statement.

The branch introduced by **else** is optional. If it is present, and none of the earlier condition expressions evaluates to **true**, then control flows into the **else** branch. If it is not present, and none of the earlier condition expressions evaluates to **true**, then the flow of execution continues with the first statement following the entire **if** statement.

```
> if 2 > 3 then
     print( "CONSEQUENT1" )
  elif 3 > 4 then
     print( "CONSEQUENT2" )
  elif 1 < 5 then
     print( "CONSEQUENT3" )
  elif 2 < 5 then
     print( "CONSEQUENT4" )
  else
     print( "ALTERNATIVE" )
  end if;
```

"CONSEQUENT3" (5.28)

```
> if 2 > 3 then
     print( "CONSEQUENT1" )
  elif 3 > 4 then
     print( "CONSEQUENT2" )
  elif 1 > 5 then
     print( "CONSEQUENT3" )
  elif 2 > 5 then
```

```
    print( "CONSEQUENT4" )
  else
    print( "ALTERNATIVE" )
  end if;
```

"ALTERNATIVE" (5.29)

The **else** branch, if present, must appear last.

An **if** statement can appear at the top level, as in the examples shown above, but is most commonly used within a procedure or module definition.

A typical use of the **if** statement is to control the flow of execution inside a procedure, depending on information coming from the arguments passed to it.

```
> p := proc( expr )
      if type( expr, 'numeric' ) then
          sin( 2 * expr )
      elif type( expr, { '`+`', '`*`' } ) then
          map( thisproc, _passed )
      else
          'procname'( _passed )
      end if
  end proc:
```

```
> p( 2 );
```

$$\sin(4) \tag{5.30}$$

```
> p( x );
```

$$p(x) \tag{5.31}$$

```
> p( x + 1 );
```

$$p(x) + \sin(2) \tag{5.32}$$

In this example, the procedure **p** uses the **type** command to examine its argument **expr**. If the argument is numeric, then it computes the value as sin(2 * expr). Otherwise, if the argument is either a sum or a product, the procedure maps itself over the operands of the expression. Otherwise, the procedure returns unevaluated.

The if Command

There is an operator form of branching that can be used within an expression. In this form, **if** is always called with three arguments. The **if** operator has the following syntax:

```
`if`( condition, consequent, alternative )
```

The first argument **condition** is a Boolean-valued expression. The second argument **consequent** is an expression to evaluate if the first argument evaluates to the value **true**. The third argument is an expression to evaluate if the first argument evaluates to either **false** or **FAIL**.

```
> `if`( 1 < 2, a, b );
```

$$a \tag{5.33}$$

```
> `if`( 1 > 2, a, b );
```

$$b \tag{5.34}$$

Note that the name **if** must be enclosed in name (left) quotes in this form.

The **if** command evaluates only one of its second and third arguments, determined based on the value of the first argument. The other argument is not evaluated.

The value of the **if** command (as opposed to the statement form) is that you can embed it within a larger expression.

```
> a := 2/3:
> sin( `if`( a > 0, Pi / 2, -Pi / 2 ) );
```

$$1 \tag{5.35}$$

However, the **if** command is much more limited than the **if** statement. The consequent and alternative must be single expressions, and there is nothing corresponding to the **elif** parts of the statement form.

Loops

To cause a statement, or sequence of statements, to be run more than once, use a loop statement. Maple has a general and flexible loop statement.

The simplest loop has the form do end do. This loop does not perform any tasks.

A loop statement has one of the following general forms.

```
for var from start to finish by increment while condition do
   statseq
```

```
end do
```

```
for var in container while condition do
   statseq
end do
```

The first line in each of these forms is called the *loop header* or, more formally, the *loop control clause*. The **statseq** part of the loop is a (possibly empty) sequence of statements, referred to as the *body* of the loop.

Each clause that occurs before the keyword **do** in the loop header is optional.

Since most of the examples below are infinite loops; you must interrupt the Maple computation to terminate the loop. For more information, see *Interrupting a Maple Computation (page 10)*.

```
> do end do;
```

```
> by -14 do end do;
```

```
> for i do end do;
```

```
> from 42 do end do;
```

```
> to 3 do end do;
```

```
> while true do end do;
```

If more than one of the optional clauses appears in the loop header, they may appear in any order.

While Loops

One simple kind of terminating loop is the *while loop*.

```
while condition do
   statseq
end do;
```

The loop header of a **while** loop involves only a single termination condition introduced by the keyword **while**. The loop repeats the statement sequence **statseq** until the Boolean-valued expression **condition** does not hold.

In this example, a loop counts the number of primes whose square is less than 1000.

```
> count := 0:
  p := 2:
  while p^2 < 1000 do
    count := 1 + count;
```

```
    p := nextprime( p )
  end do:
  count;
```

$$11 \tag{5.36}$$

This example uses the **nextprime** command, which returns the least prime greater than its argument. The name **count** is given the initial value 0, and the name **p**, which is used to store the current prime, is initially set to 2. The loop condition is the expression p^2 < 1000, appearing after the keyword **while** and before the keyword **do**. This condition is evaluated at the beginning of each iteration of the loop. If the condition evaluates to **true**, the body of the loop is executed. If the condition evaluates to **false** or **FAIL**, the code continues to execute at the next statement following the loop statement.

If the condition expression evaluates to a value other than **true**, **false** or **FAIL**, an exception is raised.

```
> while 3 do end do;
```

```
Error, invalid boolean expression: 3
```

```
> while u < v do end do;
```

```
Error, cannot determine if this expression is true or false: u < v
```

For more information on Boolean expressions, see *Boolean and Relational Expressions (page 86)*.

Counted Loops

You can use a loop to repeatedly execute a sequence of statements a fixed number of times. These loops use the **from** and **to** clauses.

```
> from 1 to 3 do print( "HELLO" ) end do;
```

$$\text{"HELLO"}$$
$$\text{"HELLO"}$$
$$\text{"HELLO"} \tag{5.37}$$

or equivalently

```
> to 3 do print( "HELLO" ) end do;
```

$$\text{"HELLO"}$$
$$\text{"HELLO"}$$
$$\text{"HELLO"} \tag{5.38}$$

If the **from** clause is omitted, the default value of 1 is used.

Inductive Loops

The most common kind of loop is an inductive loop which is similar to a counted loop, but uses an induction variable whose value changes at each iteration of the loop. This is a particular kind of *for loop* with the general form

```
for var from start to finish by increment do
   statseq
end do;
```

The default value for **start** is 1, for **finish** is infinity, and for **increment** is 1.

```
> for i to 3 do
    print( i )
  end do;
```

$$1$$

$$2$$

$$3 \tag{5.39}$$

This loop performs the following tasks:

Maple assigns **i** the (default) value **1** since a starting value was not specified.

Because **1** is less than **3**, Maple executes the statement in the body of the loop, in this case, printing the value of **i**.

Then **i** is incremented by **1** and tested again.

The loop executes until **i**>**3**. In this case, when the loop terminates, the final value of **i** is **4**.

```
> i;
```

$$4 \tag{5.40}$$

```
> for i from 7 to 2 by -2 do
    print( i )
  end do;
```

$$7$$

$$5$$

$$3 \tag{5.41}$$

Loop control parameters (**start**, **finish**, and **increment**) do not need to be integers.

```
> for i from 0.2 to 0.7 by 0.25 do
    print( i )
  end do;
```

$$0.2$$

$$0.45$$

$$0.70 \tag{5.42}$$

In addition to iterating over a numeric range, you can iterate over a range of characters. In this case, you must specify *both* the initial value **start** and the final value **finish** for the induction variable. Furthermore, the value of **increment** must be an integer.

```
> for i from "a" to "g" by 2 do
    print( i )
  end do;
```

$$\text{"a"}$$

$$\text{"c"}$$

$$\text{"e"}$$

$$\text{"g"} \tag{5.43}$$

Iterating over a Data Structure

An alternative form of the loop statement allows you to iterate over the operands of an expression (often, a data structure such as a set or list).

```
for var in expr do
   statseq
end do;
```

The induction variable **var** takes on, successively, the operands of the expression **expr**. There are a few exceptions. First, if **expr** is an expression sequence, it does not have operands as such, but the induction variable **var** iterates over the operands of the list [expr]. If **expr** is a table, the loop iterates over [entries](expr). (For more information on **entries**, see *Extracting Data (page 141)*.) The order in which these entries are visited is not specified and may vary from one session to another. Finally, if **expr** is an rtable, the loop iterates over the entries of **expr**, but the order of the iteration is not specified.

```
> for i in [ 1, 2, 3 ] do
    print( i )
  end do;
```

1

2

3 (5.44)

Note that there is a difference between the loop above and the seemingly equivalent loop

```
> for i from 1 to 3 do
    print( i )
  end do;
```

1

2

3 (5.45)

The difference is the value of the induction variable **i** at the end of the loop. To see this, evaluate the induction variable **i** immediately after running the loop to display its value.

```
> for i in [ 1, 2, 3 ] do end do: i;
```

3 (5.46)

```
> for i from 1 to 3 do end do: i;
```

4 (5.47)

Looping Commands

Maple provides commands to create some commonly used types of loops. These commands are generally meant to build expressions without creating many intermediate expressions.

The map Command

The map command applies a function to every element of an aggregate object. The simplest form of the map command is

```
map( f, x )
```

where **f** is a function and **x** is an expression. The **map** command replaces each operand **elem** of the expression **x** with f(elem).

For tables and Arrays (or other rtables), the function is applied to each entry.

```
> map( f, [a,b,c] );
```

$$[f(a), f(b), f(c)] \tag{5.48}$$

Given a list of integers, you can create a list of their absolute values and of their squares by using the **map** command.

```
> L := [ -1, 2, -3, -4, 5 ];
```

$$L := [-1, 2, -3, -4, 5] \tag{5.49}$$

```
> q:=map(abs, L);
```

$$q := [1, 2, 3, 4, 5] \tag{5.50}$$

```
> map(x->x^2, L);
```

$$[1, 4, 9, 16, 25] \tag{5.51}$$

The general syntax of the map command is

```
map( f, x, y1, ..., yn )
```

where **f** is a function, **x** is any expression, and **y1**, ..., **yn** are expressions. The action of **map** is to replace each operand of **x** such that the **i**th operand of **x** is replaced by **f(op(i,x), y1, ..., yn)**.

```
> map( f, a+b+c, x, y );
```

$$f(a, x, y) + f(b, x, y) + f(c, x, y) \tag{5.52}$$

```
> map( (x,y) -> x^2+y, L, 1);
```

$$[2, 5, 10, 17, 26] \tag{5.53}$$

For more information and examples, see *Basic Data Structures (page 125)*.

The select, remove, and selectremove Commands

The select, remove, and selectremove commands also operate on the operands of an expression. The select command returns the operands for which the specified Boolean-valued function returns **true**. The remove command returns the operands for which the specified Boolean-valued function returns **false**. The selectremove command returns two objects: the operands for which the specified Boolean-valued function returns **true** and the operands for which the specified Boolean-valued function returns **false**. The select, remove, and selectremove commands have the same syntax as the map command.

```
> X := -3*y^4*z - w*y*z^2 + 2*x*y^2 + 2*y^3 + 3*w*z;
```

$$X := -3\,y^4 z - w\,y\,z^2 + 2\,x\,y^2 + 2\,y^3 + 3\,w\,z \tag{5.54}$$

```
> select(has, X, z);
```

$$-3\,y^4\,z - w\,y\,z^2 + 3\,w\,z \tag{5.55}$$

```
> remove( x -> degree(x)>3, X );
```

$$2\,x\,y^2 + 2\,y^3 + 3\,w\,z \tag{5.56}$$

For more information on these commands, see *Other Data Structure Operations (page 153)* or refer to the select help page.

The zip Command

The zip command merges two lists or Arrays and then applies a binary function. The zip command has two forms

`zip(f, u, v)`
`zip(f, u, v, d)`

where **f** is a binary function, **u** and **v** are both lists or rtables, and **d** is any value. The **zip** command takes each pair of operands u[i], v[i], and creates a new list or vector from **f(u[i], v[i])**.

```
> zip( (x,y) -> x || y, [a,b,c,d,e,f], [1,2,3,4,5,6] );
```

$$[a1, b2, c3, d4, e5, f6] \tag{5.57}$$

If the lists or vectors are not the same length, the length of the result depends on whether you provide the argument **d**.

If you do not specify **d**, the length of the result is the same as the length of the smaller list or vector.

```
> zip( (x,y) -> x+y, [a,b,c,d,e,f], [1,2,3] );
```

$$[a+1, b+2, c+3] \tag{5.58}$$

If **d** is specified, the length of the result of the **zip** command is the same as the length of the longer list or vector. Maple replaces the missing value(s) with **d**.

```
> zip( (x,y) -> x+y, [a,b,c,d,e,f], [1,2,3], xi );
```

$$[a+1, b+2, c+3, d+\xi, e+\xi, f+\xi] \tag{5.59}$$

Non-Local Flow Control

There are a couple of statements that are generally used in procedures to control how execution of the procedure ends: return and error. For more information on these statements, see *Procedures (page 195)*.

The return Statement

The return statement causes an immediate return to the point where the current procedure was invoked.

In Command-line Maple, the return statement causes an error if it is run at the top level: *Error, return out of context*. In the Standard worksheet interface, return can be used at the top level in conjunction with DocumentTools:-RunWorksheet.

The error Statement and Exception Handling

The error statement raises an exception and interrupts the execution of the current statement. If the exception is not caught (see the following section), a message is printed indicating that an error occurred.

`error` *`string`*
`error` *`string`*`,` *`parameter1`*`,` *`parameter2`*`, ...`

In the first case, an error message is given as a **string**.

```
> error "my error";
```

```
Error, my error
```

In the second case, **string** contains several placeholders of the form **%n** or **%-n**, where **n** is a positive integer, to include the provided parameters in the message.

The placeholder **%n** is replaced by the **n**th parameter given. The placeholder **%-n** is replaced by the ordinal form of the **n**th parameter, which should evaluate to an integer. The special placeholder **%0** is replaced with the sequence of all parameters separated by commas and spaces.

```
> x := a+b: n := 10;
```

$$n := 10 \tag{5.60}$$

```
> error "my error in %1 of the %-2 kind", x, n;
```

```
Error, my error in a+b of the 10th kind
```

Trapping Errors

The try statement is a mechanism for executing procedure statements in a controlled environment so that if an error occurs, it does not immediately terminate the procedure. The try statement has the following syntax

```
try tryStatSeq
   catch catchStrings : catchStatSeq
   finally finalStatSeq
end try
```

This statement can include several **catch** clauses. The **finally** clause is optional.

If procedure execution enters a **try...catch** block, the **tryStatSeq** is executed. If *no* exceptions occur during the execution of **tryStatSeq**, the **finalStatSeq** in the **finally** clause (if present) is executed. Execution then continues with the statement after **end try**.

If an exception *occurs* during the execution of **tryStatSeq**, execution of **tryStatSeq** terminates immediately. The exception object corresponding to the exception is compared against each **catchString**. Any number of catch clauses can be provided, and each can have any number of **catchStrings** separated by commas. Alternatively, a catch clause does not need to have a catch string. Any given **catchString** (or a catch clause without one) can appear only once in a **try...end try** construct.

If a matching catch clause is found, or the catch clause contains no **catchStrings**, the **catchStatSeq** of that catch clause is executed, and the exception is considered to have been caught. If no matching catch clause is found, the exception is considered *not caught*, and is re-raised outside of the try block.

When Maple searches for a matching catch clause, the following definition of "matching" is used.

- Neither the exception object nor the **catchStrings** are evaluated (the exception object has already been evaluated by the error statement that produced it).
- The **catchStrings** are considered to be prefixes of the exception object's **msgString**. If a **catchString** has **n** characters, only the first **n** characters of the **msgString** need to match the **catchString**. This permits the definition of classes of exceptions.
- A catch clause without a **catchString** matches any exception.
- The "result" of a try statement (the value that **%** returns if it is evaluated immediately after execution of the try statement) is the result of the last statement executed in the try statement.

A **catchStatSeq** can contain an error statement with no arguments, which also re-raises the exception. When an exception is re-raised, a new exception object is created that records the current procedure name, and the message and parameters from the original exception.

Normally, the **finalStatSeq** of the finally clause, if there is one, is always executed before control leaves the try statement. This is true in the case that an exception occurs, independent of whether it is caught or whether another exception occurs in the **catch** clause.

This is true even if a **catchStatSeq** re-raises the exception, raises a new one, or executes a return, break, or next statement.

Under certain abnormal circumstances, the **finalStatSeq** is not executed:

- If an exception is caught in an interactive debugger session and you exit the debugger
- If one of the following untrappable exceptions occurs, the exception is not caught, and **finalStatSeq** is not executed:

1. A computation timed out. This exception can only be caught by the timelimit command, which raises a "time expired" exception that can be caught. For more information on the timelimit command, refer to the timelimit help page.
2. A computation has been interrupted. In other words, you pressed **Ctrl+C**, **Break**, or equivalent.
3. Internal system error. This exception indicates a bug in Maple itself.
4. ASSERT or local variable type assertion failure. Assertion failures cannot be trapped because they indicate a coding error, not an algorithmic failure.
5. Stack overflow. If a stack overflow occurs, there is generally not enough stack space to perform tasks such as running cleanup code.

If an exception occurs during the execution of a **catchStatSeq** or the **finalStatSeq**, it is treated in the same way as if it occurred outside the **try...end try** statement.

Example 1

A useful application of the try and error statements is to stop a large computation as quickly and cleanly as possible. For example, suppose that you are trying to compute an integral by using one of several methods, and in the middle of the first method, you determine that it will not succeed. You want to stop that method and try another one. The following code implements this example.

```
> try
    result := MethodA(f,x)
  catch "FAIL":
    result := MethodB(f,x)
  end try:
```

MethodA can stop its computation at any time by executing the statement **error "FAIL"**. The catch clause catches that exception, and proceeds to try **MethodB**. If any other error occurs during the execution of **MethodA**, or if an error occurs during the execution of **MethodB**, it is not caught.

Example 2

Another useful application of the try statement is to ensure that certain resources are made available when you are done with them, regardless of whether anything went wrong while you were using them.

Use the following code to access the Maple I/O facilities to read the lines of a file and process them in some way.

```
> f := fopen("myfile",READ,TEXT):
  try
    line := readline(f);
    while line < 0 do
       ProcessContentsOfLine(line);
       line := readline(f)
    end do
  finally
    fclose(f)
  end try:
```

In this example, if any exception occurs while reading or processing the lines of the file, it is not caught because there is no catch clause. However, fclose(f) is executed before execution leaves the try statement, regardless of whether there was an exception.

The next example uses both catch and finally clauses to write to a file instead of reading from one.

```
> f := fopen("myfile",WRITE,TEXT):
  try
    for i to 100 do
       fprintf(f,"Result %d is %q\n",i,ComputeSomething(i))
    end do
  catch:
    fprintf(f,"Something went wrong: %q\n",lastexception);
    error
  finally
    fclose(f)
  end try:
```

If an exception occurs, it is caught with the catch clause that has no **catchString**, and the exception object is written into the file. The exception is re-raised by executing the error statement with no **msgString**. In all cases, the file is closed by executing fclose(f) in the finally clause.

5.7 The use Statement

The **use** statement specifies local bindings of names, module exports, and operator overriding. It has the following syntax:

```
use exprseq in
   stateseq
end use
```

where **stateseq** is a sequence of statements and **exprseq** is a sequence of expressions.

The expressions can be any of the following.

- equation of the form **name = expression**
- module member selection **m:-e**, which is equivalent to the equation **e = m:-e**
- module expression **m**, which is equivalent to the equations **e = m:-e** for all exports **e** of **m**.

For more information about modules and member selection, see *Programming with Modules (page 307)*.

Running a use statement executes the body of the statement. Each occurrence of a name that appears on the left side of any of the binding equations is replaced by the right side of the corresponding equation.

For example,

```
> use f = sin, g = cos in
      f( x )^2 + g( x )^2
  end use;
```

$$\sin(x)^2 + \cos(x)^2 \tag{5.61}$$

The following example establishes local bindings for all of the exports of the **StringTools** package.

```
> use StringTools in
    s := Random( 10, 'lower' );
    Reverse( s )
  end use;
```

"iwdiamerzd" (5.62)

Among these are the names **Random** and **Reverse**. Without the **use** statement enclosing them, the two statements would have to be written using fully qualified names.

```
> s := StringTools:-Random( 10, 'lower' );
```

$$s := \text{"qknnyxmuvd"} \tag{5.63}$$

```
> StringTools:-Reverse( s );
```

"dvumxynnkq" (5.64)

You can employ the **use** statement to establish general name bindings.

```
> use a = 2, b = 3, c = 4 in
      a + b + c
  end use;
```

9 (5.65)

(This is useful when the names bound are used many times within the body of the **use** statement.)

The **use** statement is unique in Maple. It is the only Maple statement that is resolved during the automatic simplification process rather than during the evaluation process. To see this, consider the following simple procedure.

```
> p := proc( x, y )
    use a = x + y, b = x * y in
        a / b
    end use
  end proc;
```

$$p := \mathbf{proc}(x, y)\ \ (x + y) / (x^* y)\ \ \mathbf{end\ proc} \tag{5.66}$$

Note that there is no **use** statement in the procedure after it has been processed, and **a** and **b** in the body have been replaced by the values on the right-hand side of the binding equations. To see that this processing occurred during automatic simplification (of the procedure definition), enclose the procedure definition in unevaluation quotes.

```
> p := 'proc( x, y )
    use a = x + y, b = x * y in
        a / b
    end use
  end proc';
```

$$p := \mathbf{proc}(x, y)\ \ (x + y) / (x^* y)\ \ \mathbf{end\ proc} \tag{5.67}$$

use statements can be nested.

```
> use a = 2, b = 4 in
    use x = 3, y = 5 in
         a * x + b * y
    end use
  end use;
```

$$26 \tag{5.68}$$

If a name is bound in **use** statements at two different levels of nesting, the innermost binding visible at the level of an expression is used.

```
> use a = 2 in
    a^2;
    use a = 3 in
       a^2
    end use
  end use;
```

$$9 \tag{5.69}$$

In the following example, the inner binding of the value **3** to the name **a** takes precedence, so the value of the expression **a** + **b** (and therefore the entire statement) is the number **6**. The inner binding of **3** to **a** has an effect only within the body of the inner use statement. Once the execution has exited the inner use statement, the binding of **2** to **a** is restored.

```
> use a = 2, b = 3 in
      # here a is bound to 2 and b to 3
      use a = 3 in
          # here, b is still bound to 3, but a is bound to 3
          a + b
      end use;
      # binding of a to 2 is restored
      a + b
  end use;
```

$$5 \tag{5.70}$$

The **use** statement also allows you to rebind Maple operators to override their default behavior. The following is an example in which addition and multiplication are replaced by nonstandard meanings.

```
> use `+` = ((x,y) -> (x+y) mod 3), `*` = ((x,y) -> (x*y) mod 3) in
```

```
  1 + 2 * 4
end use;
```

$$0 \tag{5.71}$$

The following operators can have their default behavior overridden by a **use** statement.

Table 5.1: Operators That Can Be Rebound

@	@@	%	%%	%%%	.
+	*	-	/	mod	^
!	union	minus	intersect	subset	in
$	and	or	not	xor	implies
=	<>	<	<=	assuming	<|>
<,>	[]	{}	?()	?[]	~

Notes:

- The following operators cannot be rebound: concatenation operator (||), member selection operator (**:-**), type operator (**::**), range (**..**), comma (**,**), functional operator (**->**), and the assignment operator (**:=**). The relational operators **>** and **>=** can be rebound, but not independently of **<** and **<=**, respectively.
- All of the element-wise operators are processed through the element-wise operator (~).
- The operators **-** and **/** are treated as *unary* operators (that represent negation and inversion, respectively). Subtraction is represented internally in Maple by forming addition and negation: **a - b = a + (-b)**. Division is formed in a similar way. Therefore, it is not necessary to override the *binary* infix operators **-** and **/**.

Note also that an expression such as **a + b + c + d** is treated as though it were parenthesized as **((a + b) + c) + d**, so that each **+** operator is binary. For example,

```
> use `+` = F in
      a + b + c + d;
      a + ( ( b + c ) + d )
  end use;
```

$$F(a, F(F(b, c), d)) \tag{5.72}$$

5.8 Other Statements

The quit Statement

The Maple keywords **quit**, **done**, and **stop** perform the same task and, when entered in the command-line interface, cause the Maple process to terminate.

```
> quit
> done
> stop
```

Note: The **quit** statement cannot be used in the Maple standard interface. In the standard interface, use **File > Close Document** to end your Maple session.

quit, **stop**, and **done** are available as command names if quoted using name quotes. These forms allow you to exit Maple while passing an integer in the range 0 .. 255 as an argument to be returned to the calling process as exit status.

The save Statement

You can save Maple expressions to a file by using the **save** statement. It has the general form

```
save name1, name2, ..., nameN, file
```

The names **namei** are names (that have assigned values) to be saved to the file **file**.

Normally, the file name **file** is a string.

For example, make the following three assignments and run the subsequent **save** statement.

```
> a := proc( x ) sin( x / 2 ) end proc:
> b := 42:
> c := "some text":
> save a, b, c, "myfile.txt";
```

The file **myfile.txt** is created in the current directory (assuming adequate file permissions) containing the following Maple assignment statements.

```
a := proc (x) sin(1/2\*x) end proc;
b := 42;
c := "some text";
```

The read Statement

The **read** statement takes the following form.

```
read filename
```

where **filename** is a string.

```
> read "myfile.txt";
```

$$c := \text{"some text"}$$

$$b := 42$$

$$a := \mathbf{proc}(x)\ \ \sin(1/2*x)\ \ \mathbf{end\ proc} \tag{5.73}$$

The file named by **filename** must consist of valid Maple language statements. The statements in the file are executed as they are read, as though they were input into the Maple session in which the **read** statement was entered. Maple displays the results of executing each statement. However, the input statements are *not* echoed to the interface, by default. To change this, set the interface variable **echo** to a value of 2 or higher.

```
> interface( 'echo' = 2 ):
```

5.9 Exercises

1. Find the product of the square root of all prime numbers less than **100**.

   Hint: The function isprime determines the primality of an integer.

2. Find the sum of all odd composite numbers less than **150**.

3. Find the sum of the first **30** powers of **2**.

4. Write a looping structure that finds the four substrings (of a string assigned to the name **MyString**) containing only lowercase letters, uppercase letters, decimal digits, and special characters.

   Hint: You can use relational operators to compare characters.

5. Write a procedure, **SPLIT**, that, upon input of a product **f** and a variable **x**, returns a list of two values. The first item in the list should be the product of the factors in **f** that are independent of **x**, and the second item should be the product of the factors that contain an **x**.

   Hint: Use the has, select, remove, and selectremove commands.

6 Procedures

A Maple procedure is a sequence of parameter declarations, variable declarations, and statements that encapsulates a computation. Once defined, a procedure can be used to perform the same computation repeatedly for different argument values, from different places in a program, or both. A procedure in Maple corresponds to a function in languages such as C or Java, a procedure or function in Pascal, or a subroutine in FORTRAN and modern versions of BASIC.

Chapter 1 gave a brief introduction to procedures. This chapter describes the syntax and semantics of procedures in detail, and discusses how to best make use of procedures in your programs.

6.1 Terminology

Several terms are used frequently when discussing procedures in Maple and other programming languages. Some of these terms are sometimes used interchangeably, but the distinctions between them are important:

Procedure - In Maple, a procedure is an object that can be invoked by a function call, be passed arguments, perform some operations, and return a result. A procedure definition begins with the keyword proc, and ends with end proc.

Function Call - A function call, of the form name(arguments), evaluates the **arguments** and then invokes a procedure if **name** has a value that is a procedure. The value of the function call is then the value returned by the procedure. If **name** has no value, then the value of the function call is just name(evaluatedArguments).

Argument - An argument is one of one or more values explicitly included in a function call. Note that a default value is not an argument.

Parameter or *Formal Parameter* - A parameter is a name that is declared in a procedure definition to receive the value of an argument. The parameter name is used to refer to that value within the body of the procedure.

Actual Parameter - An actual parameter is neither an argument nor a (formal) parameter. The term refers to the value that a formal parameter takes during the execution of a procedure. This value can come from an argument or a default value. The term is defined here for completeness; it is not further used in this chapter. Instead we will refer to *the value of the parameter*.

6.2 Defining and Executing Procedures

A Maple procedure definition has the following general syntax:

```
proc( parameterDeclarations ) :: returnType;
     description shortDescription;
     option optionSequence;
     local localVariableDeclarations;
     global globalVariableDeclarations;
     statementSequence
end proc
```

A procedure definition is considered to be an expression in Maple, the evaluation of which produces the procedure itself. The resulting procedure is usually assigned to a name, but it can also be used in other ways such as passing it as an argument to another procedure, or invoking it immediately.

The following is a simple procedure definition. It contains two formal parameters, **x** and **y**, and one statement in the procedure body. There is no description, there are no options, and the procedure does not make use of any local or global variables. In order to be able to use the procedure later, we'll assign it to a name:

```
> SumOfSquares := proc( x, y )
      x^2 + y^2
  end proc;
```

$$SumOfSquares := \mathbf{proc}(x, y)\ \ x\text{^}2 + y\text{^}2\ \ \mathbf{end\ proc} \tag{6.1}$$

This procedure computes the sum of the squares of its two arguments. The procedure can be called with any two arguments and Maple will attempt to compute the sum of their squares. Like any computation in Maple, the result can be symbolic. If you want to restrict the types of arguments that are permitted, it is possible to specify the **type** for each argument in the parameter declarations, as described in the next section.

You can invoke (or execute) a procedure by using it in a *function call*:

```
procedureName( argumentSequence )
```

The **procedureName** is usually the name that the procedure was assigned to, although it can also be an actual procedure definition, or another expression that evaluates to a procedure.

The **argumentSequence** is a sequence of expressions that will be evaluated, and then substituted for the corresponding parameters before the execution of the statements comprising the body of the procedure. Note that the arguments are evaluated only once before the exe-

cution of the procedure begins. They are *not* evaluated again during execution of the procedure.

The value returned by the procedure is the result of the last statement executed within the procedure. In the following function call, Maple executes the statements in the body of the procedure **SumOfSquares**, replacing the formal parameters **x** and **y** with the arguments a and 3. The result of the last (and in this case, only) statement in the procedure is the returned value:

```
> SumOfSquares(a,3);
```

$$a^2 + 9 \tag{6.2}$$

For more information about return values, see *Returning Values from a Procedure (page 234)*.

6.3 Parameter Declarations

In the procedure definition, **parameterDeclarations** is a sequence of parameter declarations. Procedure parameter declarations can range from very simple to very sophisticated. In its simplest form, a parameter declaration is just the parameter's name. When you call the procedure, you can pass any value as an argument for such a parameter, and if you pass no value at all, the parameter will have no value.

You can extend a parameter declaration by adding a type specification and/or a default value. A type specification ensures that, when the procedure is called, the value of the parameter within the procedure will be of the indicated type, and a default value ensures that a parameter will always have a value even if no corresponding argument was passed.

Maple procedures can also have *keyword parameters*. When invoking a procedure, the corresponding arguments are of the form **keyword=value**, and can appear anywhere in the argument sequence.

When you call a procedure, the arguments are evaluated and then bound to the parameters. In the simplest case, there is a one-to-one correspondence between arguments and parameters; the first argument is bound to the first parameter, the second argument to the second parameter, and so on. The presence of default values and keyword parameters can change this correspondence, as described in this section.

Required Positional Parameters

A required positional parameter is called *required* because a corresponding argument must have been passed in the function call that invoked the procedure if the parameter is used during the execution of the procedure. It is called *positional* because the argument's position within **argumentSequence** must correspond to the position of the parameter in **parameterDeclarations**.

The syntax to declare a required positional parameter is:

```
parameterName :: parameterType
```

The **parameterName** must be a valid symbol, and is used to refer to the parameter within the procedure body. The **:: parameterType** is optional. If it is present and the corresponding argument does not match the specified type, an exception is raised.

In this example, the procedure **Adder** is defined with two parameters, **a** and **b**. The procedure returns the sum of its two arguments. For the parameter **a**, **Adder** expects an argument of type **integer**.

```
> Adder := proc( a::integer, b ) a+b end proc:
> Adder(2,3);
```

$$5 \tag{6.3}$$

The next call to **Adder** raises an exception because the second argument is missing.

```
> Adder(3);
Error, invalid input: Adder uses a 2nd argument, b, which is missing
```

This call raises an exception because the supplied first argument does not match the parameter's specified type.

```
> Adder(2.5,4);
Error, invalid input: Adder expects its 1st argument, a, to be of type
 integer, but received 2.5
```

If a procedure has both required and ordered parameters (described below), all of the required parameters must appear *before* the ordered parameters.

Optional Ordered Parameters

An optional ordered parameter is declared in the same way as a required positional parameter, with the addition of a default value:

```
parameterName :: parameterType := defaultValue
```

The presence of **defaultValue** allows the parameter to be optional. If there are no remaining arguments or the next unused argument does not match the specified **parameterType**, the parameter receives the default value. The non-matching argument, if any, remains available for binding to a later parameter.

As was the case with a required positional parameter, **:: parameterType** can be omitted. The parameter will receive its default value only when there are no more available arguments, since any available argument would have been valid for an untyped parameter.

Usually, **defaultValue** will be of the type specified by **parameterType**, but this need not be the case. The default value can be a *literal* value of any other type, or **NULL**. If the default value is not a literal value, but is an expression that evaluates to something other than itself, then the result of that evaluation must conform to **parameterType**.

This class of parameters is called *ordered* because the arguments are bound to parameters in the order they were passed. If the first unused argument is not bound to the current parameter, it remains as the first available argument for the next parameter.

In this example, the procedure **Adder** is defined with two optional ordered parameters, **a** and **b**, both of type **integer**, and returns their sum:

```
> Adder := proc(a::integer := 10, b::integer := 100.1)
      a + b
  end proc:
> Adder(3,4);
```

$$7 \tag{6.4}$$

```
> Adder(3);
```

$$103.1 \tag{6.5}$$

```
> Adder();
```

$$110.1 \tag{6.6}$$

```
> Adder(3,6.6);
```

$$103.1 \tag{6.7}$$

In the first call to **Adder**, the arguments 3 and 4 were bound to the parameters **a** and **b**, and their sum returned. In the second call, only a single argument was passed, so **b** received its default value. Notice that the default value is not an **integer**, but since it is a literal value, is an acceptable default. In the third call, no arguments were passed and both parameters received their default values.

You may have expected the result of the fourth call to **Adder** to be 9.6, but this is not the case. Why? First, parameter **a** was given the value 3. Next, 6.6 was considered a candidate for parameter **b**, but was rejected because it is not of type **integer**. Instead, **b** received its default value.

This illustrates an important aspect of calling procedures in Maple, which is that in general, it is acceptable to call a procedure with more arguments than it expects. You will see later how to access these within a procedure, allowing you to write procedures that accept a variable number of arguments, or how to disallow the passing of extra arguments.

Expected Ordered Parameters

An expected ordered parameter is similar to an optional ordered parameter, except that the corresponding argument can be omitted only if all further arguments are also omitted. If there is an argument available, it must match **parameterType** or an exception is raised.

The declaration of an expected ordered parameter declaration differs from that of an optional ordered parameter by enclosing **parameterType** in expects():

```
parameterName :: expects( parameterType ) := defaultValue
```

The procedure below is identical to the one from the previous section, except that parameter **b** has been declared as an expected parameter. When it is called with a second argument of the wrong type, instead of saving that argument for a later parameter, Maple raises an exception:

```
> Adder := proc(a::integer := 10, b::expects(integer) := 100.1)
      a + b
  end proc:

> Adder(3,6.6);

Error, invalid input: Adder expects its 2nd argument, b, to be of type
 integer, but received 6.6
```

Keyword Parameters

Keyword parameters are not positional and not ordered. A keyword parameter is bound to a value when an argument of the form **keyword=value** appears in a procedure invocation. The left-hand side of such an argument specifies the keyword parameter name, and the right-hand side specifies the value it will receive. If **true** is an acceptable value for the parameter, then an argument of the form **keyword** is equivalent to **keyword=true**.

The declaration of a keyword parameter looks very much like that of an optional ordered parameter, except that all keyword parameter declarations are enclosed in braces, much like a **set** is:

```
{ ... parameterName :: parameterType := defaultValue ... }
```

The **:: parameterType** can be omitted, in which case any value can be passed as the right-hand side of the keyword argument. If **parameterType** is specified, then the passed value must be of that type.

As is the case with an ordered parameter, if **defaultValue** is a *literal* value, it need not match **parameterType**.

A procedure can have multiple keyword parameters, which can be declared within a single set of braces, or grouped into multiple sets of braces as desired to improve source code

readability. When a procedure is compiled into Maple's internal form, the keyword parameters are consolidated into a single set. If you then display that procedure using Maple's print command, the keyword parameters are displayed as a single set, sorted lexicographically.

The simplest and most frequently encountered form of keyword parameter declaration has a single Maple symbol for **parameterName**:

```
> Simple := proc( { simple::integer := 2 } )
      sprintf("simple=%d",simple)
  end proc:
> Simple(simple=3);
```

$$\text{"simple=3"} \tag{6.8}$$

```
> Simple();
```

$$\text{"simple=2"} \tag{6.9}$$

```
> Simple(simple=4.5);
Error, invalid input: Simple expects value for keyword parameter simple
 to be of type integer, but received 4.5
```

It is also possible to declare keyword parameters that can be referred to by indexed names when the procedure is called. If **parameterName** is of the form `symbol[symbol]` or `symbol[integer]`, it matches indexed names.

The indexed parameter names are still *symbols* because of the enclosing left single quotes, and are referenced that way within the procedure, but the argument names can be actual indexed names. For more information on indexed keyword arguments, see *Binding of Arguments to Parameters (page 229)*.

As a convenience to the user of a procedure, multiple spellings of the keyword are allowed by specifying a list of the permitted spellings in the declaration:

```
{ ... [ parameterName1, parameterName2, ... ] :: parameterType :=
defaultValue ... }
```

Within the procedure's **statementSequence**, you can refer to the parameter by any of the declared spellings. If you display the procedure using **print**, however, only the first spelling is used.

```
> Spellings := proc( { [color,colour]::symbol := RED } )
      sprintf("color=%a -- colour=%a", color, colour)
  end proc;
```

$$\begin{aligned} &Spellings := \mathbf{proc}(\{[color, colour]::symbol := RED\}) \\ &\quad sprintf(\text{"color=\%a -- colour=\%a"}, color, color) \\ &\mathbf{end\ proc} \end{aligned} \tag{6.10}$$

```
> Spellings();
```

$$\text{"color=RED -- colour=RED"} \tag{6.11}$$

```
> Spellings(color=BLUE);
```

$$\text{"color=BLUE -- colour=BLUE"} \tag{6.12}$$

```
> Spellings(colour=GREEN);
```

$$\text{"color=GREEN -- colour=GREEN"} \tag{6.13}$$

```
> Spellings(color=ORANGE,colour=PURPLE);
```

$$\text{"color=PURPLE -- colour=PURPLE"} \tag{6.14}$$

```
> Spellings(colour=YELLOW,color=42);
Error, invalid input: Spellings expects value for keyword parameter
[color, colour] to be of type symbol, but received 42
```

If more than one keyword argument matches a keyword parameter, only the last one takes effect.

Alternate spellings and indexed keywords can be combined by including the indexed keyword symbols in the list of alternate spellings.

The End-of-Parameters Marker

Recall from earlier that Maple usually allows extra arguments to be passed to a procedure. This is useful when implementing procedures that can accept a variable number or type of arguments, but for many procedures, the presence of extra arguments indicates a programming error.

A procedure can be declared to disallow extra arguments (that is, arguments that were not bound to any declared parameter) by ending the sequence **parameterDeclarations** with $. If extra arguments remain at the end of argument processing, Maple raises an exception:

```
> TwoSine := proc( x::float := 0.0, $ ) 2 * sin(x) end proc:
```

```
> TwoSine(2.3);
```

$$1.491410424 \tag{6.15}$$

```
> TwoSine();
```

$$0. \tag{6.16}$$

```
> TwoSine(2.3,-4.5);
```

```
Error, invalid input: too many and/or wrong type of arguments passed
 to TwoSine; first unused argument is -4.5
```

```
> TwoSine(42);
```

```
Error, invalid input: too many and/or wrong type of arguments passed
 to TwoSine; first unused argument is 42
```

Default Value Dependencies

You can express the default value **defaultValue** of a parameter in terms of other parameters, as long as the resulting value conforms to the specified type **parameterType**, if any. The parameters on which **defaultValue** depends can appear earlier or later in **parameterDeclarations**. For example, here is a list extraction function that expects a list, a starting index, and an ending index. If the ending index is omitted, the length of the list is used:

```
> SubList := proc( s::list, f::integer := 1, t::integer :=
  numelems(s) )
      s[f..t]
  end proc:
```

```
> SubList([a,b,c,d,e],2,3);
```

$$[b, c] \tag{6.17}$$

```
> SubList([a,b,c,d,e],2);
```

$$[b, c, d, e] \tag{6.18}$$

There can be no cyclic dependencies, such as two parameters' default values depending on each other:

```
> NotGood := proc( s := sin(c), c := cos(s) ) s^2 + c^2 end proc;
```

```
Error, cyclic dependency detected in parameter s := sin(c) in procedure
 NotGood
```

Usually, Maple evaluates the arguments of a function call from left to right. The use of parameter dependencies in default values will alter this order to ensure that the required values are available by the time they are needed. This is only of consequence if the evaluation of one or more arguments has side effects.

Parameter Modifiers

Parameter modifiers change the way that arguments are evaluated and/or bound to parameters. Modifiers appear as part of the parameter's declaration, in the form of a function call enclosing the parameter type **parameterType**.

You have already seen the expects modifier, which changes an optional ordered parameter into an expected ordered parameter.

The seq Modifier

The seq modifier allows the parameter to match multiple arguments. When a parameter with a specified type of the form seq(memberType) is encountered, it is bound to an expression sequence of all arguments (beginning with the next available one) that are of the type specified by **memberType**.

```
parameterName :: seq(memberType)
```

If *no* arguments match **memberType**, the parameter will receive its default value if one was specified, or NULL if there is no default value.

The seq modifier cannot be used together with the expects modifier, because seq is allowed to match zero arguments, whereas expects implies that at least one argument must match. The seq modifier also cannot be used with a keyword parameter.

You must be careful when working with the value of a seq parameter because it might have just one element in it. Such a value is not considered to be a sequence, thus indexing it will not select the element. The safest approach is to enclose the parameter in a list, as in this example:

```
> LargestInteger := proc( x::seq(integer), other::seq(anything) )
      local max, n;
      max := -infinity;
      for n in [x] do
          if n > max then max := n end if
      end do;
      max, [other]
  end proc:
> LargestInteger(4,7,8,2,1);
```

8, [ ] (6.19)

```
> LargestInteger(4,7,"not an integer",8,2,1);
```

7, ["not an integer", 8, 2, 1] (6.20)

The depends Modifier

Usually, a parameter's type is predetermined when the procedure is first written. When arguments are matched to parameters, **parameterType** is not evaluated since this is not expected to yield anything other than what was written. There are cases where this is too restrictive. In that case, use the depends modifier to declare that a parameter's type depends on something that could change. Such a dependency is usually on another parameter.

The syntax for a parameter declaration with the depends modifier is:

```
parameterName :: depends( parameterTypeExpression )
```

where **parameterTypeExpression** is a type expression that can refer to other parameter names.

For example, you might want to write a procedure like this to find one root of a polynomial:

```
> OneRoot := proc( p::depends(polynom(integer,v)), v::symbol )
      local sols;
      sols := [ solve(p=0,v) ];
      if sols = [] then
          error "no solution"
      else
          sols[1]
      end if
  end proc:
> OneRoot(x^2+3*x+5,x);
```

$$-\frac{3}{2}+\frac{1}{2}\,I\sqrt{11} \tag{6.21}$$

```
> OneRoot(x^2+3*x+5,y);
Error, invalid input: OneRoot expects its 1st argument, p, to be of
type polynom(integer,y), but received x^2+3*x+5
```

This procedure expects as an argument for its first parameter, **p**, a polynomial in the variable specified by the second parameter, **v**. If the depends modifier were omitted, the procedure would only accept polynomials in the global variable **v**.

The depends modifier can only be used for required parameters. It cannot be used for optional or expected ordered parameters, nor keyword parameters. If the depends modifier is used together with the seq modifier, it must appear within it. That is, **parameterType** must be written in the form seq(depends(memberType)).

The uneval Modifier

Unlike the other modifiers described so far, the uneval modifier takes no arguments. That is, it does *not* enclose another type or modified type. Instead it is used *as* the **parameterType**.

```
parameterName :: uneval
```

A parameter with the uneval modifier prevents the corresponding argument from being evaluated when the procedure is called. The effect is the same as if the argument had been enclosed in unevaluation quotes ('...').

The uneval modifier can only be used for required positional parameters, and cannot be used in conjunction with any other modifiers. It also cannot be used for any parameter declaration after one that uses the seq modifier.

```
> Square := proc( x::uneval ) x^2 end proc:
> (a, b) := (3, 4.5):
> r := Square(a+b);
```

$$r := (a+b)^2 \tag{6.22}$$

```
> eval(r);
```

$$56.25 \tag{6.23}$$

The evaln Modifier

A parameter declared with the evaln modifier expects an argument that can be evaluated to a name (that is, an assignable object). This modifier can be used in two different forms, evaln or evaln(valueType). In the second form, the resulting name is expected to have a value that matches the type **valueType**.

```
parameterName :: evaln
```

```
parameterName :: evaln(valueType)
```

In effect, declaring a *parameter* with the evaln modifier is equivalent to enclosing the *argument* with evaln at procedure invocation time, and allows you to write procedures where the user of the procedure does not have to remember to do so.

Like uneval, the evaln modifier can only be used for required positional parameters, and cannot be used for a parameter declaration after one having a seq modifier. The only other modifier that can be used together with evaln is the depends modifier, in the form depends(evaln(valueType)).

```
> SquareName := proc( x::evaln(integer) ) x^2 end proc:
> (a, b) := (3, 4.5):
```

In the first call, the argument **a** is evaluated to 'a', which is a name with an integer value.

```
> SquareName(a);
```

$$a^2 \tag{6.24}$$

In the next call, the argument **b** is evaluated to 'b', which is a name, but not with an integer value.

```
> SquareName(b);
```

```
Error, invalid input: SquareName expects its 1st argument, x, to be
of type evaln(integer), but received b := 4.5
```

In the next call, the argument does not evaluate to a name.

```
> SquareName(a+b);
```

```
Error, illegal use of an object as a name
```

In the next example, the procedure **Accumulate** accumulates all the values of its second argument in the variable passed as its first argument. Notice that the first call fails, because **Accumulate** expects a name with a numeric value, but **total** has not been initialized yet.

```
> Accumulate := proc( r::evaln(numeric), n::numeric )
      r := eval(r) + n
  end proc:
```

```
> Accumulate(total,2);
```

```
Error, invalid input: Accumulate expects its 1st argument, r, to be
of type evaln(numeric), but received total := 'total'
```

```
> total := 0;
```

$$total := 0 \tag{6.25}$$

```
> Accumulate(total,2);
```

$$2 \tag{6.26}$$

```
> Accumulate(total,3.5);
```

$$5.5 \tag{6.27}$$

```
> total;
```

$$5.5 \tag{6.28}$$

The coercion Modifiers

```
parameterName :: (valueType)
```

```
parameterName :: coerce(valueType,coercion procedure)
```

As stated previously in Chapter 4, coercion refers to the ability to pass one type of data to a procedure and have it receive a different type.

Coercion can be enabled in two ways:

- **Coercion Using ~Type**: You can use a short form notation to invoke Maple built-in coercion functions. This short form notation is a tilde (~) followed by a data type. For example, the command **~Matrix** will accept, among other things, a listlist and return a Matrix. This type of ~ function can be used in place of the data type in a procedure declaration. This tells Maple to try testing if the passed parameter is of that type, and if not, call the ~ function to coerce it into that type.
- **Coercion Using coerce()**: You can use long form notation to enable data coercion by using the coerce() modifier. The coerce modifier allows you to specify a sequence of types and coercion procedures. A coercion procedure is a procedure that accepts a single typed parameter and converts that parameter into a new expression. When the main procedure is called, the argument is type checked against the parameter types handled by the coercion procedure. The first coercion procedure whose parameter's type matches the type of the argument is called. The return value of the matching coercion procedure is then used as the parameter's value.

```
> p_string :=proc(s::coerce(string, (s::name)->convert(s,string)))
   s;
  end proc;
```

$$\begin{aligned} &p\_string := \mathbf{proc}(s::(coerce(string, s::name \rightarrow convert(s, \\ &\quad string)))) \\ &\quad s \\ &\mathbf{end\ proc} \end{aligned} \tag{6.29}$$

```
> p_string("a string");
```

$$\text{"a string"} \tag{6.30}$$

```
> p_string(`a name`);
```

$$\text{"a name"} \tag{6.31}$$

Procedures without Declared Parameters

You can define a procedure without any declared parameters. Some procedures, such as one that generates random numbers, might not depend on any arguments. Other procedures might operate directly on global values, although this is considered poor programming practice.

However, just because a procedure has no declared parameters does not mean that it cannot be passed arguments. Unless a procedure's **parameterDeclarations** ends with $, it is always permissible to pass more arguments than there are declared parameters. All of the arguments are accessible via the special sequence **_passed**, which has one entry corresponding to each argument that was passed. The number of entries is given by **_npassed**. For example, the following procedure produces the sum of all its arguments:

```
> SumOfArgs := proc( )
      add(_passed[i], i=1.._npassed)
  end proc:
> SumOfArgs(42,3.14,sin(-2.5));
```

$$44.54152786 \tag{6.32}$$

For more information on **_passed** and **_npassed** as well as other special names for working with parameters, see *Special Sequences for Referring to Parameters and Arguments (page 223)*.

6.4 Return Type

The closing parenthesis following a procedure's parameter declarations can be followed by :: and a **returnType** assertion. This is optional. Unlike a **parameterType** specification, **returnType** is only an assertion. If kernelopts(assertlevel) is set to 2, the type of the value returned by the procedure is checked against the type specified by **returnType**, and if it does not match, an exception is raised:

```
> ReturnInteger := proc( x ) :: integer;
      x^2
  end proc:
> kernelopts(assertlevel=2):
> ReturnInteger(3);
```

$$9 \tag{6.33}$$

```
> ReturnInteger(Pi);
Error, (in ReturnInteger) assertion failed: ReturnInteger expects its
 return value to be of type integer, but computed Pi^2
```

6.5 The Procedure Body

The body of the procedure is where most of the computation is carried out (although some computation may already have occurred while resolving the **defaultValue** for optional parameters). The procedure body consists of an optional description, option declarations, local and global variable declarations, and executable statements.

The description, option, local variable, and global variable declaration parts are each introduced by their own keyword, and can appear in any order. There can be only one **description** clause and one **option** clause. There can be any number of variable declaration clauses.

Description

Use the description clause to give a procedure a short description that is displayed when the procedure is displayed. The description has no effect on the execution of the procedure. It is only used for documentation purposes.

```
description string, string, ... ;
```

The description keyword is followed by one or more string literals, separated by commas.

```
> Average := proc( x::integer, y::integer )
      description "Compute the average of two integers.",
                  "Returns a rational.";
      (x + y) / 2;
  end proc;
```

$$\begin{aligned}&\mathit{Average} := \mathbf{proc}(x{::}\mathit{integer}, y{::}\mathit{integer})\\&\quad \mathbf{description}\\&\quad \text{"Compute the average of two integers.",}\\&\quad \text{"Returns a rational.";}\\&\quad 1/2*x+1/2*y\\&\mathbf{end\ proc}\end{aligned} \tag{6.34}$$

Options

A procedure can be tagged with one or more *options* which alter the behavior or display of the procedure. Options are specified by the keyword option or options, followed by one or more option names or equations, separated by commas:

```
option optionNameOrEquation, ... ;
options optionNameOrEquation, ... ;
```

Each **optionNameOrEquation** is a symbol or an equation of the form **optionName=value**. Any symbol is allowed as an option name that you can use to tag procedures for your own purposes, but there are several options that are known to Maple.

The arrow and operator Options

The **arrow** option and the **operator** option have meaning when specified together. These options cause Maple to print the procedure using arrow notation:

```
> SumOfSquares := proc( x, y )
      option operator, arrow;
      x^2 + y^2;
  end proc;
```

$$SumOfSquares := (x, y) \rightarrow x^2 + y^2 \tag{6.35}$$

For information on defining a procedure using arrow notation, see *Functional Operators: Mapping Notation (page 252)*

The builtin Option

Maple has two classes of procedures: kernel built-in procedures implemented in the C programming language, and library procedures written in the Maple programming language. Because the kernel built-in functions are compiled, you cannot view their procedure definitions. The **builtin** option identifies a kernel procedure.

This option is shown when you display a purely built-in procedure. Instead of displaying the procedure statements, only the **builtin** option is displayed.

For example, the add procedure is built into the kernel:

```
> print(add);
```

$$\textbf{proc}(\,)\ \textbf{option}\ builtin = add;\ \ \textbf{end proc} \tag{6.36}$$

A procedure can have both the **builtin** option and a statement sequence. In that case, invoking the procedure will first invoke the kernel built-in version. If that indicated that it did not compute a result, the statement sequence is executed instead. This mechanism allows the kernel to process common cases very quickly, and defer to library code to handle other cases.

You can use the type function to test if an expression is a built-in procedure. An expression is of type **builtin** if it is a procedure with option **builtin**:

```
> type(add, 'builtin');
```

$$true \tag{6.37}$$

```
> type(int, 'builtin');
```

$$false \tag{6.38}$$

You cannot create built-in procedures, although there is a mechanism for creating procedures based on externally compiled code. Such procedures have the **call_external** option.

The call_external Option

The **call_external** option appears in procedures generated by the **define_external** procedure. This option indicates that the implementation of the procedure resides in a pre-compiled external library. For more information, see *???*.

The hfloat Option

The **hfloat** option forces all floating-point operations within a procedure to be performed using hardware floating-point values. Depending on the operations performed, this can significantly speed up execution of the procedure at the cost of floating-point accuracy. Procedures that perform many floating-point operations or manipulate the contents of Arrays, Matrices, or Vectors of hardware floating-point values will benefit the most from this option.

The **hfloat** option causes the following differences in the procedure's definition and execution:

Any floating-point constants appearing in the procedure body are converted into hardware floating-point values when the procedure is first created.

Numeric arguments passed to the procedure are converted into hardware floating point values when the procedure is invoked.

Extracting values from hardware floating-point Arrays, Matrices, and Vectors does not incur a conversion to arbitrary precision floating-point form. Instead, the hardware floating-point values are used directly.

Calls to **evalhf** made from within the procedure return a hardware floating point value, and thus do not incur a conversion to arbitrary precision floating-point form.

These differences, together with the rules for contagion of hardware floating-point values in expressions, will usually cause arithmetic operations in the procedure to be performed using hardware floating-point arithmetic.

The use of the **hfloat** option differs from using evalhf in a few ways:

When a procedure is executed within the **evalhf** environment, everything is computed using hardware floats, and the operations available are restricted to those that can be done using hardware floats. No other basic data types, such as integers or strings, are available.

The only data structures available within the **evalhf** environment are Arrays.

Performance of a procedure having option **hfloat** is generally better than one operating with arbitrary precision floats, but usually not as good as a procedure operating within **evalhf**.

But, a procedure with option **hfloat** has the full power of Maple available to it. All Maple operations, data types (except arbitrary precision software floating point), and data structures can be used in such a procedure.

The **hfloat** option cannot be used in conjunction with the **builtin**, **call_external**, or **inline** options.

Hardware floating-point numbers and computations are discussed in detail in *Numerical Programming in Maple (page 267)*. For more information on hardware floating-point contagion, see *Floating-Point Contagion (page 279)*.

The inline Option

Use the **inline** option to create a procedure that can be expanded inline wherever it is called from. An inline procedure avoids the overhead of a procedure invocation by executing the procedure's statements directly as if it were written in-line instead of in a separate procedure. This can result in improved execution speed and reduced memory usage.

Not all Maple procedures can take advantage of the **inline** option. Only procedures whose body consists of a single *expression* or an expression sequence can be expanded in-line. The body cannot consist of a *statement* or statement sequence. For details on further restrictions that may apply, refer to the inline help page.

The overload Option

The presence of option **overload** in a procedure indicates that the procedure will operate only on arguments matching the declared parameters (as is normally the case), and that if the arguments do not match the parameters, the next in a sequence of such procedures is tried.

A sequence of procedures with option **overload** can be combined into a single procedure using the overload command. This will produce a new procedure that will, when called, try each **overload** procedure in turn until one is encountered that will accept the arguments, or no procedures remain. In the latter case, an exception will be raised.

The following example uses the overload command and procedures with the **overload** option to append an entry to either a list (by creating a new list) or a 1-dimensional Array (in-place):

```
> Append := overload(
      [
          proc( L::list, x::anything ) option overload;
              [ op(L), x ]
          end proc,
          proc( A::Array(..), x::anything ) option overload;
              A(ArrayNumElems(A)+1) := x
          end proc
```

```
      ]
  ):
> Append([1,2],3);
```

$$[1, 2, 3] \tag{6.39}$$

Option **overload** can also be used to specify that a procedure exported by a **package** is only applied to arguments of specific type. If non-matching arguments are passed, the default behavior occurs instead.

For example, you can define a new implementation of `+` that works only on set arguments. The system default `+` operator is used for all other cases.

```
> SetOperations := module() option package;
      export `+` := proc( a::set, b::set ) option overload;
          a union b
      end proc;
  end module:
> with(SetOperations);
```

$$[\,`+`\,] \tag{6.40}$$

```
> {1,2,3} + {4,5};
```

$$\{1, 2, 3, 4, 5\} \tag{6.41}$$

```
> 123 + 45;
```

$$168 \tag{6.42}$$

For more information on packages, see *Writing Packages (page 383)*.

The procname Option

As you will read later, the special name **procname** used *within* a procedure refers to the name by which the procedure was called. Among other things, this name is used to describe the location that an exception occurred when displaying an error message. It can also be used to return *unevaluated* calls to the procedure, and to make *recursive* calls.

If a procedure has the **procname** option, then the value of the **procname** special name within the procedure is inherited from the procedure that called it. If an error then occurs within the called procedure, the error is reported as having occurred in the calling procedure. This allows you, for example, to break up your procedure into sub-procedures, yet still have any errors reported as if they occurred in your main procedure.

For more information on the uses of **procname**, see *Returning Unevaluated (page 238)* and *Recursion (page 256)*.

The remember, cache, and system Options

The **remember** option activates a procedure's *remember table*. For a procedure with an active remember table, at the end of each invocation of the procedure, an entry that records the computed result for the specified arguments is made in the procedure's remember table. Subsequent calls to the procedure with the same arguments simply retrieve the result from the remember table instead of invoking the procedure.

The **remember** option allows writing an inherently recursive algorithm in a natural manner without loss of efficiency. For example, the Fibonacci numbers can be computed by the procedure:

```
> Fibonacci := proc( n::nonnegint )
      option remember;
      if n < 2 then
          n
      else
          Fibonacci(n-1) + Fibonacci(n-2)
      end if
  end proc:
```

Without the **remember** option, the time required to compute Fibonacci(n) is exponential in **n**. With option remember, the behavior becomes linear. For a comparison of the efficiency of this procedure with and without option remember, see *???*.

Entries can be explicitly inserted into a procedure's remember table by writing a function call on the left-hand side of an assignment. For example, the Fibonacci procedure can be written:

```
> Fibonacci := proc( n::nonnegint )
      option remember;
      Fibonacci(n-1) + Fibonacci(n-2)
  end proc:

> Fibonacci(0) := 0:

> Fibonacci(1) := 1:
```

A procedure's remember table can grow without bound, and for some procedures, may eventually contain many entries that will never be needed again. Adding the **system** option to a procedure allows Maple's garbage collector to clear out the remember table whenever garbage collection occurs. If a discarded result is needed again later, it will be recomputed.

As an alternative to remember tables, Maple also provides the **cache** option. Unlike a remember table, which can grow without bound, a cache has a maximum number of entries. When the cache becomes full, old entries are removed as new ones are inserted.

The **cache** option can be specified as just the symbol **cache**, or with an optional argument, in the form cache(N) where **N** is an integer specifying the size of the cache. If (N) is not specified, the cache is sized to hold 512 entries.

You can explicitly insert permanent entries into a procedure's cache using the Cache:-AddPermanent function.

When the interface variable **verboseproc** is 3, displaying a procedure also displays the contents of its remember table or cache as comments following the procedure definition:

```
> Fibonacci(7);
```

$$13 \tag{6.43}$$

```
> interface(verboseproc=3):
> print(Fibonacci);
```

$$\begin{aligned} &\textbf{proc}(n::nonnegint) \\ &\quad \textbf{option } remember; \\ &\quad Fibonacci(n-1)+Fibonacci(n-2) \\ &\textbf{end proc} \end{aligned} \tag{6.44}$$

The **remember** and **cache** options are mutually exclusive, and the **system** option can only be used in conjunction with the **remember** option.

The trace Option

If a procedure is given the **trace** option, Maple will log each entry to and exit from the procedure, and the result of any assignment made during the execution of the procedure:

```
> Fibonacci := proc( n::nonnegint )
      option remember, trace;
      Fibonacci(n-1) + Fibonacci(n-2)
  end proc:
> Fibonacci(0) := 0:
> Fibonacci(1) := 1:
```

```
> Fibonacci(3);
{--> enter Fibonacci, args = 3
{--> enter Fibonacci, args = 2
value remembered (in Fibonacci): Fibonacci(1) -> 1
value remembered (in Fibonacci): Fibonacci(0) -> 0

                        1

<-- exit Fibonacci (now in Fibonacci) = 1}
value remembered (in Fibonacci): Fibonacci(1) -> 1

                        2

<-- exit Fibonacci (now in mpldoc/process_example) = 2}
```

$$2 \tag{6.45}$$

Variables in Procedures

A *variable* is a name representing an item of data, such as a numerical value, character string, or list of polynomials. The value of the variable, that is, *which* data item it represents, can change during the execution of a procedure (or sequence of Maple commands at the top-level, outside of any procedure). There are three different classes of variables that can be used within a procedure: global, local, and lexically scoped.

Global Variables

A *global* variable has meaning within an entire Maple session. Many procedures may access a global variable, and all those procedures will refer to the same instance of that variable. A value assigned to a global variable during one function call will still be there the next time the procedure is called (if it was not changed by another procedure in the meantime).

Global variables are introduced by the global keyword, followed by one or more declarations:

```
global variableName := value, ... ;
```

The optional **:= value** part is an assignment that is executed at the beginning of procedure execution. Semantically, it is equivalent to writing a separate assignment statement immediately after all the variable declaration clauses.

A global variable continues to exist and retain its value after the procedure exits, and conceptually, existed (and possibly had a value) before the procedure was executed. Its *lifetime* is thus the duration of the entire Maple session.

Local Variables

A *local* variable has meaning only within a particular procedure. If the same variable name is referenced outside of the procedure or within a different procedure, it refers to a different instance of that name, and is therefore a different variable.

The *lifetime* of a local variable is the time that the procedure is executing. The variable is created when the procedure is first invoked, and is usually discarded when the procedure has finished executing. If the same procedure is later executed again, a new instance of the variable is created. The variable does not retain its value from its previous lifetime.

Local variables are declared using the following syntax:

```
local variableName :: typeAssertion := initialValue, ... ;
```

The only required part of the declaration is **variableName**.

The optional **:: typeAssertion** assertion specifies that the variable is expected to always refer to values of the specified type. Since this is an assertion, if kernelopts(assertlevel) is set to 2, the type is checked every time a new value is assigned to the variable. If the value is not of the specified type, an exception is raised.

The optional **:= initialValue** ensures that the variable is assigned the specified value before its first use. The **initialValue** can be any Maple expression. If the value is a literal expression sequence, it must be enclosed in parentheses, since otherwise the comma separating the elements of the sequence is interpreted as the comma separating individual variable declarations.

Lexically Scoped Variables

When one procedure is defined within another procedure (or within a module), variables in the outer procedure (or module) are visible to the nested procedure. This is called *lexical scoping*. Consider the following procedure, which given a list, produces a new list in which

every element has been divided by the element with the largest magnitude, and then raised to a specified integer power:

```
> PowerList := proc( L::list, power::integer )
      local largest := max(abs~(L));
      map( proc(x) (x / largest) ^ power end proc, L )
  end proc:
> PowerList([1,1/2,-3.14],2);
```

$$[0.1014239929, 0.02535599822, 1.000000000] \tag{6.46}$$

This example uses an *anonymous* nested procedure, declared directly within the expression that uses it. Notice that this inner procedure refers to both of the symbols **power** and **largest**. Because there are no variable or parameter declarations in the inner procedure that declare these symbols, lexical scoping ensures that they are automatically bound to the corresponding symbol in the outer procedure. In other words, **power** in the inner procedure refers to the parameter **power** of the outer procedure, and **largest** in the inner procedure refers to the local variable **largest** of the outer procedure.

Scoping Rules

If you want a variable to be local to a procedure or global, you should declare that variable using a local or global declaration. Declaring the scope of variables makes it easier to debug your code, and also makes it easier for someone else to understand your procedure.

On the other hand, if a variable is intended to refer to a parameter or local variable declared in an enclosing procedure, you must *not* declare it in the enclosed procedure. Doing so would defeat lexical scoping by making the variable local to the enclosed procedure, and thus a different variable with no connection to the one in the enclosing procedure.

If an undeclared variable does not correspond to a parameter or declared variable in a surrounding procedure, Maple determines its scope, and either automatically declare the variable as local or assume that it is global. When the variable is automatically declared local, such an *implicit declaration* generates a warning:

```
> ImplicitLocal := proc( x, y )
      z := x + y;
      if z < 0 then z^2 else z^3 end if
  end proc:
Warning, `z` is implicitly declared local to procedure `ImplicitLocal`
```

Whether a variable is implicitly declared local or assumed to be global depends on how it is used:

If the variable appears on the *left-hand* side of an assignment statement or as the controlling variable of a for loop, Maple adds the variable to the procedure's local declarations. This

means that if an enclosed procedure also refers to the variable (without declaration), lexical scoping binds it to the implicitly declared variable of the enclosing procedure. If a procedure in which such an implicit local declaration is displayed using the print function, the variable appears within the procedure's local declaration clause.

Otherwise, Maple assumes the variable is global. However, the variable is *not* added to the procedure's global declaration clause, which means that it is *not* subject to lexical scoping if the same name is used within an enclosed procedure.

Here is a summary of how the scope of a variable is determined:

If the variable is declared as a parameter, local, or global variable in the procedure in which the variable is encountered, the scope is specified by the declaration.

If the variable is not declared and there is a surrounding procedure (or module), the parameter, local (including implicit local), and global declarations of the surrounding procedure are examined. If the variable is found there, that binding is used. If it is not found, the search continues outward through the layers of surrounding procedures.

If the top level (outside of any procedure or module) is reached, the usage of the variable in the original procedure is examined. If it appears on the left-hand side of an assignment or as the controlling variable of a for loop, it is added to the procedure's local declarations. Otherwise it is assumed to be a global variable.

Non-Variable Name Bindings

In addition to the binding of names to parameters, local variables, and global variables, you can also explicitly bind other names to objects outside of the procedure with the uses clause:

```
uses bindingSpecification, ...
```

The uses keyword is followed by one or more bindings, in a form identical to those of the **use** statement, introduced in *The use Statement (page 189)*. These bindings are in effect over the entire body of the procedure, in the same way they would be if the procedure body had been enclosed in a use statement.

The uses clause must appear at the top of the procedure body, together with the option, description, and initial local and global declarations. If you wish to bind names in a subset of the procedure body, use a use statement instead.

The Statement Sequence

The **statementSequence** section of the procedure can contain any number of Maple statements, nested arbitrarily deeply. Other than *one level evaluation* and references to parameters, the semantics of statements within a procedure are the same as if those statements were executed outside of any procedure.

Referring to Parameters within the Procedure Body

When referring to parameters in the body of a procedure, there are some things to keep in mind.

Parameters Are Not Variables

Although a parameter declaration has a similar form to a local variable declaration, and parameters are referred to by name the same way that variables are, parameters are *not variables*. In Maple, a parameter *always* represents the argument that was bound to it.

Consider this example, which tries to use a parameter on the left-hand side of an assignment statement:

```
> Add2 := proc( x, y )
      x := x + y
  end proc:
> Add2(3,4);
Error, (in Add2) illegal use of a formal parameter
```

This call to **Add2** results in an error because the statement x := x + y is interpreted as 3 := 3 + 4. This is in contrast to languages such as C or C++, where a parameter is effectively a local variable that has been initialized to the argument value.

A parameter can be used on the left-hand side of an assignment if the *value* of the parameter is a name. The evaln parameter modifier can ensure that this is the case. Here is an example you saw earlier:

```
> Accumulate := proc( r::evaln(numeric), n::numeric )
      r := eval(r) + n
  end proc:
> total := 0;
```

$$total := 0 \tag{6.47}$$

```
> Accumulate(total,2);
```

$$2 \tag{6.48}$$

```
> Accumulate(total,3.5);
```

$$5.5 \tag{6.49}$$

```
> total;
```

$$5.5 \tag{6.50}$$

Here, the parameter **r** evaluates to the name **`total`**, an assignable object. Although it appears that an assignment to the parameter **r** is being made within the procedure, it is really the *value* of **r**, which in this case is the global variable **total**, that is being assigned to.

Required Parameters

Recall that a *required parameter* is one for which a corresponding argument must have been passed *if the parameter is used during the execution of the procedure*. Failure to pass an argument for a required parameter only raises an exception if an attempt is made to use that parameter during the particular invocation of the procedure.

For example, a procedure may determine, based on the value of its first required parameter, that it does not have to refer to the second required parameter.

```
> Require := proc( x::integer, y::integer )
      if x < 0 then x^2 else x * y end if
  end proc:
> Require(-3);
```

$$9 \tag{6.51}$$

```
> Require(3,4);
```

$$12 \tag{6.52}$$

```
> Require(3);
Error, invalid input: Require uses a 2nd argument, y (of type integer),
 which is missing
```

Parameters with the seq Modifier

If a required (or optional) parameter was declared with the seq modifier, then the parameter will *always* have a value. That value will be a sequence of the specified type, a single item of that type, or NULL (or the default value for an optional parameter).

To do anything with a seq parameter other than pass it on to another procedure, you should convert the parameter value to a list and then work with the list:

```
> AddAndMax := proc( x::seq(numeric) )
      local a := 0, i;
      for i in [x] do
          a := a + i
      end do;
      a, max(x)
  end proc:
```

Without the [] brackets around **x**, this procedure would produce unexpected results if called with a single floating-point number. A for var in expr loop iterates over the operands of **expr**. If **expr** is a sequence of two or more numbers, it works as expected, but if **expr** were a single float, the loop would iterate over the *float*s operands (the significand and exponent). By enclosing **x** in a list, the loop will always iterate over the arguments bound to **x**.

Parameters with the uneval or evaln Modifiers

Parameters declared with the uneval or evaln modifiers are used like any other. Because Maple uses one level evaluation rules inside procedures, these parameters do not evaluate any further than they did when the arguments were initially evaluated. The eval function can be used to evaluate such parameters further.

Optional and Expected Ordered Parameters

Both *optional* and *expected* ordered parameters are always declared with default values, so using such a parameter within a procedure always yields a value. If an argument was bound to the parameter during procedure invocation, the parameter's value is that argument. Otherwise, the value of the parameter is the declared default value.

Keyword Parameters

Keyword parameters also have declared default values, so using the parameter always yields a value. Unlike ordered parameters, keyword parameters receive their values from arguments of the form **keyword=value**. The value of a keyword parameter is the **value** portion of such an argument, *not* the entire argument.

Special Sequences for Referring to Parameters and Arguments

Maple provides a number of special named expression sequences to make it easy to work with parameters and arguments. These are useful in cases when it would be awkward if they could only be referred to by name.

The special names **_params** and **_nparams** can be used within a procedure to refer to the current values of the positional and ordered parameters. The **_params** symbol represents an expression sequence with **_nparams** members, one corresponding to each declared parameter (excluding keyword parameters). For a given procedure, **_nparams** is constant.

The **_params** symbol can only be used when immediately followed by an index enclosed in square brackets, _params[indexExpr]. It cannot be used in any other context. **indexExpr** can evaluate to one of the following:

An integer, **N**, in the range 1 to **_nparams**, or -_nparams to -1. This is just the selection operation on the sequence **_params**. It yields the value of the **N**th parameter when $\mathbf{N} > 0$, or the (_nparams+1+N)th parameter when $\mathbf{N} < 0$ (negative integers index **_params** from

the end instead of the beginning). If no argument was passed for the requested parameter and no default was declared, the result is NULL.

A range of such integers. This yields an expression sequence of values, *with any* NULL *values omitted*. A sequence of all the non-NULL positional and ordered parameter values can be obtained using _params[..]. Note that due to elision of NULLs, this could produce fewer than **_nparams** values.

An unevaluated parameter name. The notation _params['parameterName'] is equivalent to just writing **parameterName**, except when referring to a required positional parameter that was not bound to an argument. In that case _params['parameterName'] yields NULL whereas referring directly to **parameterName** would raise an exception.

The following example multiplies or divides the last three positional parameters by the first, depending on the value of the keyword parameter **multiply**:

```
> MulDiv := proc( a, b, c, d, { multiply := true } )
      if multiply then
          _params[-3..] * a
      else
          _params[-3..] / a
      end if
  end proc:
  MulDiv(100,1,2,3);
  MulDiv(100,1,2,3,multiply=false);
```

$$100, 200, 300$$

$$\frac{1}{100}, \frac{1}{50}, \frac{3}{100} \tag{6.53}$$

Just as **_params** and **_nparams** can be used to work with positional and ordered parameters in a flexible manner, **_options** and **_noptions** provide similar facilities for working with keyword parameters (often called *keyword options*).

The **_options** symbol represents an expression sequence containing **_noptions** members, one for each declared keyword parameter. Each member of **_options** is an equation of the form **keyword=value**.

If a keyword parameter was declared with multiple spellings, the corresponding member of **_options** uses the first spelling.

Unlike **_params**, **_options** can be used directly, not only through the selection of members of the sequence. Because **_options** returns a sequence of equations, even a member corresponding to an argument with a NULL value is non-NULL. It is an equation of the form **keyword=NULL**.

When **_options** is used with an index, the index must evaluate to an integer (or a range of integers) in the range 1 to **_noptions** or -(_noptions) to -1, or the unevaluated name of a keyword parameter.

The order of the equations in **_options** does not necessarily correspond to the order in which the keyword parameters were declared. Instead, the equations are in lexicographic order by keyword (the first spelling for keyword parameters with multiple spellings). This is the same order in which the keyword parameters are printed when the procedure is displayed by the print command. As a consequence of this, if a new keyword parameter is added to the procedure definition, the numeric index of the **_options** entry corresponding to a particular keyword parameter could change. Thus, when indexing **_options**, it is safest to use the _options['parameterName'] form.

The following example uses **_options** to pass all the keyword arguments on to another procedure:

```
> MyRanMat := proc( a::integer, {density::float := 1.0, generator
  := 0..0.5} )
      LinearAlgebra:-RandomMatrix(a, _options)
  end proc:
```

```
> MyRanMat(2, density=0.75, generator=1..9);
```

$$\begin{bmatrix} 8 & 0 \\ 4 & 0 \end{bmatrix} \tag{6.54}$$

```
> MyRanMat(3, density=0.88);
```

$$\begin{bmatrix} [0.479746213196451499, 0.400140234444400056, 0. \\ ], \\ [0., 0., 0.478753417717148799], \\ [0.210880641313137496, 0.485296390880307849, \\ 0.139249109433524199]] \end{bmatrix} \tag{6.55}$$

The next example selects specific keyword arguments to pass to another procedure:

```
> MulRanMat := proc( a::integer, {density::float := 1.0, generator
   := 0..0.5, mult := 1.0} )
      mult * LinearAlgebra:-RandomMatrix(a, _options['density'],
  _options['generator'])
  end proc:
```

```
> MulRanMat(4, density=0.75, generator=1..9, mult=x/2);
```

$$\begin{bmatrix} 4x & 0 & x & \frac{9}{2}x \\ \frac{3}{2}x & 0 & \frac{3}{2}x & 0 \\ 0 & \frac{1}{2}x & \frac{9}{2}x & \frac{7}{2}x \\ \frac{7}{2}x & 4x & 2x & \frac{3}{2}x \end{bmatrix} \tag{6.56}$$

When there are more arguments in a function call than needed to match the called procedure's parameters, you can access the remaining arguments inside the procedure by using the special sequence **_rest**. The number of members in this sequence is given by **_nrest**.

Because these extra arguments do not correspond to any declared parameters, it is not possible for such an argument to have a NULL value. Recall that the only way for a parameter to be NULL is for no argument to have matched a parameter with no declared default value (or a default value of NULL). Since there is no declared parameter corresponding to any value in **_rest**, these conditions cannot hold.

This example uses **_rest** and **_nrest** to return the number of entries in a sequence of numbers, together with the maximum, and optionally the mean:

```
> MaxMean := proc( {mean := false})
      if mean then
          _nrest, max(_rest), Statistics:-Mean([_rest])
      else
          _nrest, max(_rest)
      end if
  end proc:
> c := MaxMean(6,200,400, mean=true);
```

$$c := 3, 400, 202. \tag{6.57}$$

All of the *arguments* that were *passed* to a procedure can be accessed using the special sequence **_passed**, having **_npassed** elements.

Prior to Maple version 10, **_passed** and **_npassed** were known as **args** and **nargs**. These older names are still accepted as synonyms for the newer names for backwards compatibility. Of historical interest, the earliest versions of Maple did not support declared parameters at all; **args** and **nargs** were the only mechanism for processing arguments.

The **_passed** sequence can be used to do explicit argument processing within the body of the procedure, although this is discouraged for two reasons:

Most argument processing requirements can be handled using the mechanisms described so far in this chapter. Doing so is usually significantly faster (in terms of both execution time and development time) than performing the same operations using your own custom argument processing algorithms within the procedure.

When special argument processing requirements do arise, it is often easier to work with **_params**, **_options**, and **_rest**. In many cases, the provided mechanisms can handle most of the processing, and it is only necessary to look at **_rest** to handle additional arguments.

The clearest and most efficient way to write a procedure to find the maximum of an arbitrary sequence of numbers is to use a single parameter with a seq modifier, and pass that parameter directly to Maple's built-in **max** function. However, the following example uses **_passed**, **_npassed**, and a for loop instead for demonstration purposes:

```
> Maximum := proc( )
      local max := _passed[1], i;
      for i from 2 to _npassed do
          if _passed[i] > max then
              max := _passed[i]
          end if
      end do;
      max
  end proc:
```

Care must be taken when the **_options**, **_rest**, or **_passed** sequences contain only a single entry and that sequence is assigned to a variable (for example, myOpts := _options). The variable will receive the value of that single element rather than an expression sequence. The safest way to use these expression sequences is to transform them into lists (for example, myOpts := [_options]).

6.6 How Procedures Are Executed

When a procedure definition is entered in Maple or read from a file, Maple does not *execute* the procedure. It does however translate the procedure into an internal representation, process all the parameter and variable declarations, perform lexical scoping and implicit local declaration, and *simplify* the procedure's **statementSequence**.

Automatic simplification of **statementSequence** is similar to simplification of expressions when Maple is used interactively, with a few exceptions. Consider the following procedure:

```
> f := proc(x)
      local t := x + 3 + 0/2;
      if true then
```

```
        sqrt(x * 2.0 / 3.0)
    else
        t^2
    end if
end proc;
```

$$f := \textbf{proc}(x)\ \textbf{local}\ t;\ t := x + 3;\ \mathrm{sqrt}(x * 2.0 / 3.0)\ \textbf{end proc} \tag{6.58}$$

During automatic simplification, the division 0/2 has been removed (because it does not contribute to the sum). More significantly, the entire if...then...else...end if statement has been replaced by just the body of the first branch, since the if-condition is true.

Notice that the expression sqrt(x * 2.0 / 3.0) has *not* been simplified to .8164965809*x^(1/2) as it would have been if entered at the top level, outside of a procedure. If this simplification had been performed, then the result produced by the procedure would depend on the setting of **Digits** (and other aspects of the floating-point environment) both when the procedure was simplified, and a possibly different setting of **Digits** when the procedure is later executed. By not performing any floating-point arithmetic during procedure simplification, the procedure will depend only on the state of the floating-point environment at execution time.

A procedure is *executed* after it has been *invoked* by a *function call*. Generally, the process is:

A function call, of the form functionName(functionArguments) is encountered during evaluation of an expression, either at the interactive level or while executing another procedure.

The **functionName** is examined to see if has been assigned a procedure.

The **functionArguments** are evaluated, usually from left to right.

The evaluated arguments are bound to the parameters of the procedure.

All of the procedure's local variables are *instantiated*. That is, for each local variable, a unique instance of the variable's name is created, with no prior value.

Interpretation of the procedure's **statementSequence** begins.

Interpretation continues until the last statement has been executed, an *exception* is raised (either as the result of an operation, or by an explicit error statement), or a return statement is encountered.

Binding of Arguments to Parameters

Argument processing occurs when a function call results in the invocation of a procedure. First, all the arguments are evaluated (except those corresponding to parameters with the uneval or evaln modifiers), and then they are matched to the parameters of the procedure.

Binding of Keyword Arguments

Keyword arguments are always matched first unless the procedure has parameters declared with the uneval or evaln modifiers. Maple makes a pass through the entire sequence of arguments looking for **keyword=value** equations where the **keyword** matches a declared keyword parameter of the procedure.

Whenever a matching keyword parameter is encountered, the right-hand side of the equation becomes the value for that parameter, and the equation is removed from further consideration as an argument. If more than one keyword argument matches a keyword parameter, only the last one takes effect.

Keyword parameter names (the **keyword** part) are Maple symbols like any other. If that symbol is in use as a variable, then using it in a keyword argument may not work as expected since the variable may evaluate to its value. To ensure that this does not happen, it is best to always use unevaluation quotes around the keyword part of a keyword argument:

```
> f := proc( x::integer, { y::integer := 1 }, $ ) x * y end proc:
> y := sin(z):
> f(3,y=2);
Error, invalid input: too many and/or wrong type of arguments passed
 to f; first unused argument is sin(z) = 2
> f(3,'y'=2);
```

$$6 \tag{6.59}$$

This is a good practice when calling any function, whether it is a procedure you defined or a Maple command. See *Protecting Names and Options (page 47)*.

When calling a procedure that accepts a keyword argument from within another procedure that has a parameter with the same name as the keyword argument, you must use both unevaluation quotes *and* the scope resolution operator, :-, to ensure that (the global instance of) the name itself is used instead of the value of the parameter:

```
> f := proc( x::integer, { y::integer := 1 }, $ ) x * y end proc:
> g := proc( y::rational ) f(numer(y), ':-y'=denom(y)) end proc:
> g(3/2);
```

$$6 \tag{6.60}$$

If a keyword parameter has a declared **parameterType** for which true is a valid value (for example, the types **truefalse** or **boolean**), the keyword name alone is interpreted as a synonym for **keyword=true**.

```
> f := proc( x::integer, { square::truefalse := false } )
      if square then x^2 else x end if
  end proc:
> [ f(2), f(3,square=true), f(4,square) ];
```

$$[2, 9, 16] \tag{6.61}$$

If a keyword parameter's **keyword** is a symbol of the form **`symbol[symbol]`** or **`symbol[integer]`**, the parameter is treated specially at during argument processing. Although such a **keyword** is still a symbol (because of the enclosing left single quotes), it matches indexed name keyword arguments. Specifically, if an equation whose left-hand side is an indexed name of the form **symbol[symbol]** or **symbol[integer]** is encountered, it matches the keyword parameter whose **keyword** symbol *looks* like the indexed name. For example, the keyword *argument*,

axis_label[1] = "time"

matches the keyword *parameter*:

`axis_label[1]` :: string := "x"

Keyword arguments with multiple indices are also recognized by attempting to match them using one index at a time. For example, the keyword *argument*,

axis_label[1,2] = ""

matches *both* of the keyword *parameters*,

`axis_label[1]` :: string := "x", `axis_label[2]` :: string := "y"

and sets them both to the empty string.

The following example illustrates these behaviors:

```
> Indexed := proc( { `name[1]`::string := "hello",
                     `name[2]`::string := "goodbye" } )
    sprintf("name[1]=\"%s\" -- name[2]=\"%s\"",`name[1]`,`name[2]`)
  end proc:
> Indexed(name[1]="hi");
```

$$\text{"name[1]="hi" -- name[2]="goodbye""} \tag{6.62}$$

```
> Indexed(name[1]="bonjour",name[2]="aurevoir");
```

"name[1]="bonjour" -- name[2]="aurevoir"" (6.63)

```
> Indexed(name[1,2]="good day");
```

"name[1]="good day" -- name[2]="good day"" (6.64)

```
> Indexed(name[2]=42);
```

```
Error, invalid input: Indexed expects value for keyword parameter
name[2] to be of type string, but received 42
```

The Special Case of evaln and uneval Modifiers

There is one case in which the first stage of argument processing is not keyword matching. If the procedure was declared with any parameter(s) having an uneval or evaln modifier, arguments are first assigned to positional parameters from left to right until the rightmost uneval or evaln parameter has been bound to an argument or until all the arguments have been exhausted, whichever happens first. For each argument/parameter pair:

If the parameter has no **parameterType**, the argument matches trivially, and becomes the value for that parameter.

If the parameter has a **parameterType** specification, the argument may or may not match. If it matches, the argument becomes the value for that parameter. If it does not match, an exception is raised.

```
> Accumulate := proc( r::evaln(numeric), n::numeric,
                      { operation::symbol := `+` } )
      r := operation(eval(r),n)
  end proc:
> total := 0:
> Accumulate(total, 2.3);
```

2.3 (6.65)

```
> Accumulate(total, operation=`*`, 10);
```

23.0 (6.66)

```
> Accumulate(operation=`*`, total, 100);
```

```
Error, illegal use of an object as a name
```

In the last call, an exception is raised because the first argument does not evaluate to a name.

Binding of Arguments to Positional and Ordered Parameters

After all arguments matching keyword parameters have been processed, matching of required positional and optional or expected ordered parameters is carried out. If any parameter had an uneval or evaln modifier, all parameters up to the rightmost of these will already have received arguments, so further matching begins with the next positional or ordered parameter after that.

Matching is done by traversing the parameter declarations from left to right. As each parameter is examined, an attempt is made to match it to the next unused argument as follows:

If the parameter has no **parameterType**, the argument matches trivially, and becomes the value for that parameter.

If the parameter has **parameterType**, but no **defaultValue**, the argument may or may not match. If it matches, the argument becomes the value for that parameter. If it does not match, an exception is raised.

If the parameter has both **parameterType** and **defaultValue**, the argument may or may not match. If it matches, the argument becomes the value for that parameter. If it does not match, the parameter receives its default value, and the argument remains available for matching a subsequent parameter.

In last two cases above, if the parameter's type uses the seq modifier, Maple continues to match additional arguments against the parameter until one is encountered that is not of the correct type. A seq parameter never results in an exception, because even if no arguments match, a valid sequence has been produced (the empty sequence).

At the end of this process, if there are any arguments left over, they are either put into the **_rest** sequence, or, if the procedure was declared with the end-of-parameters marker, $, an exception is raised.

Conversely, if all the arguments were bound to parameters, but there are parameters remaining to be assigned values, these receive their default values if they have one. Otherwise, they have no value, and attempting to use them (by name) within the procedure raises an exception.

Statement Sequence Interpretation

After all the arguments in a function call have been successfully bound to the procedure's parameters, Maple begins interpreting the procedure's *statement sequence*. Each statement is examined in turn and the necessary actions carried out.

For example, an assignment statement is interpreted by evaluating the right-hand side (the expression to be assigned), and resolving the left-hand side (the target of the assignment). The latter involves evaluating any indices if the left-hand side contains indexed names. Fi-

nally, the *value* of the right hand side is assigned to the resolved *variable* on the left-hand side.

When an if-statement is encountered, Maple evaluates the condition. If it is true, statement sequence interpretation continues with the first statement within the first branch of the if-statement. When the statements within that branch have all been executed, interpretation continues with the first statement after the end if. If if-condition was false, Maple looks for an elif or else branch and continues in a similar manner.

When there are no further statements remaining, Maple behaves as if a return statement had been encountered.

Variable Evaluation Rules within Procedures

Maple fully evaluates global variables whenever they are referenced, even within procedures, but local variables are evaluated in a special way. When a local variable is encountered during procedure execution, it is *evaluated only one level*. Consider the following Maple statements, outside of any procedure:

```
> f := x + y;
```

$$f := x + y \tag{6.67}$$

```
> x := z^2 / y;
```

$$x := \frac{z^2}{y} \tag{6.68}$$

```
> z := y^3 + 3;
```

$$z := y^3 + 3 \tag{6.69}$$

Since these statements undergo normal full recursive evaluation, the following result is returned:

```
> f;
```

$$\frac{(y^3+3)^2}{y} + y \tag{6.70}$$

The same sequence of steps within a procedure would yield a different result:

```
> OneLevelEval := proc( )
      local f, x, y, z;
      f := x + y;
      x := z^2 / y;
      z := y^3 + 3;
```

```
      f
  end proc:
> OneLevelEval();
```

$$x + y \tag{6.71}$$

The concept of *one-level evaluation* is unique to symbolic languages like Maple, where the *value* of a variable can be, or include, the *name* of another variable. One-level evaluation avoids arbitrarily deep computation at every step of a procedure and is thus important for efficiency. It has very little effect on the behavior of procedures, because most procedures have a sequential structure. When full evaluation of a local variable is required within a procedure, use the eval function:

```
> FullEval := proc( )
      local f, x, y, z;
      f := x + y;
      x := z^2 / y;
      z := y^3 + 3;
      eval(f)
  end proc:
> FullEval();
```

$$\frac{\left(y^3+3\right)^2}{y} + y \tag{6.72}$$

In addition to illustrating one level evaluation, this example also introduces the idea of an *escaped local*. The expression returned by **OneLevelEval** is x + y and contains the symbols **x** and **y**. However, these are *not* the global variables of the same names; they are the local **x** and **y** declared in **OneLevelEval**. Because these variables have *escaped*, they continue to exist beyond their normal lifetime even though the procedure has finished executing. Usually, an escaped local indicates a programming error such as forgetting to assign a value to a local variable before using it. There are situations where letting a local escape can be useful, such as generating unique instances of a name that will be guaranteed never to evaluate further.

Returning Values from a Procedure

When a procedure has finished executing, a value is returned. If the procedure was invoked by a function call, possibly within a larger expression, the returned value is used as the value of that function. At the interactive level, the returned value is displayed (unless the input was terminated by a colon instead of a semicolon).

Except when a procedure raises an exception, a value is *always* returned. In the absence of an explicit return statement, the returned value is the value of the last statement executed in the procedure. The *value of a statement* means:

The value computed by the right-hand side of an assignment statement.

The value of the expression when the statement is an expression.

The value of the last statement executed within the branches of an if statement or within the body of a loop.

Note that NULL is a valid expression (and thus a valid statement). A procedure that returns NULL is still returning a value, although at the interactive level, nothing is displayed.

You can use an explicit return statement to end the execution of the procedure and return a value immediately:

```
return expression;
```

Upon encountering a return statement during execution, Maple evaluates the expression, and then immediately terminates the execution of the procedure, with the result of the evaluation as the returned value.

This example uses an explicit return statement to immediately return the position **i** of a value **x** in a list when the value is found. If the value is not found, the procedure returns 0:

```
> Position := proc( x::anything, L::list )
      local i;
      for i to numelems(L) do
          if x = L[i] then
              return i
          end if
      end do;
      0
  end proc:
```

```
> Position(3, [2,3,5,7,1,3,7,9,3,9]);
```

$$2 \tag{6.73}$$

```
> Position(4, [2,3,5,7,1,3,7,9,3,9]);
```

$$0 \tag{6.74}$$

The following procedure computes the greatest common divisor, **g**, of two integers **a** and **b**. It returns the expression sequence **g**, **a**/**g**, **b**/**g**. The case **a** = **b** = **0** is treated separately because in that case, **g** is zero:

```
> GCD := proc( a::integer, b::integer, $ )
      local g;
      if a = 0 and b = 0 then
          return 0, 0, 0
      end if;
      g := igcd(a,b);
      g, iquo(a,g), iquo(b,g)
  end proc:
> GCD(0,0);
```

$$0, 0, 0 \tag{6.75}$$

```
> div, quo1, quo2 := GCD(12,8);
```

$$div, quo1, quo2 := 4, 3, 2 \tag{6.76}$$

This example illustrates that you can return a sequence of values from a procedure, and that those values can then be assigned to a sequence of names by the caller. Whenever a procedure returns a sequence of values, the result can be assigned to a sequence of the same number of names (a multiple assignment). If you assigned the result to a single name, then the value of that name would be the entire sequence.

Sometimes, it is convenient to write a procedure which will return a different number of values depending on the context in which it was called. A procedure can use the special variable **_nresults** to determine how many results are expected by the caller. Here is a version of the previous procedure that returns only a single result when called from within an arithmetic expression (the tests for the case **a** = **b** = **0** has been omitted for brevity):

```
> GCD := proc( a::integer, b::integer, $ )
      local g := igcd(a,b);
      if _nresults = 1 or _nresults = undefined then
           g
      else
          g, iquo(a,g), iquo(b,g)
      end if
  end proc:
> div := GCD(12,8);
```

$$div := 4 \tag{6.77}$$

```
> GCD(12,8) ^ 2;
```

$$16 \tag{6.78}$$

```
> { GCD(12,8) };
```

$$\{4\} \tag{6.79}$$

```
> div, quo1, quo2 := GCD(12,8);
```

$$div, quo1, quo2 := 4, 3, 2 \tag{6.80}$$

The **_nresults** variable has the value undefined if the procedure was called from within an expression or within the arguments of another function call. It has an integer value if the call was from the top level of an expression appearing on the right-hand side of an assignment. The value of **_nresults** is the number of variables on the left-hand side of the assignment statement.

Do not use **_nresults** in a procedure with the **remember** or **cache** options. Only the first computed result is stored in the remember table or cache. Subsequent calls with the same input but a different number of expected results will not return the expected number of results. (The **Cache** package can be used to manually implement a simulated remember table that works correctly in conjunction with **_nresults**.)

Another alternative for returning more than one value from a procedure is to assign values to *variables* whose names were passed in as *values*. The following procedure determines whether a list **L** contains an expression of type **T**. If found, the procedure returns the index of the (first matching) expression. If the procedure is called with a third argument, then it also assigns the expression to that name.

```
> FindType := proc( T::type, L::list, V::evaln, $ )
      local i;
      for i to numelems(L) do
          if L[i] :: T then
              if _npassed = 3 then
                  V := L[i]
              end if;
              return i
          end if
      end do
  end proc:
> FindType(string, [2,3/4,"Hello",x+y]);
```

$$3 \tag{6.81}$$

```
> FindType(string, [2,3/4,"Hello",x+y], s);
```

$$3 \tag{6.82}$$

```
> s;
```

$$\text{"Hello"} \tag{6.83}$$

When **FindType** was called with two arguments, the procedure just returned the index of the found list element.

When called with three arguments, parameter **V** received the *name, not the value* of global variable **s**. The evaln declaration of **V** ensures that **V** will always refer to a name. Just before returning, the procedure assigned the found expression to **s**, as referenced by **V**.

If, during the execution of the procedure, you need to refer to the *value* that has been assigned to a name via an evaln parameter, enclose such references to the parameter within a call to eval:

```
> Accumulate := proc( r::evaln(numeric), n::numeric )
      r := eval(r) + n
  end proc:
```

Returning Unevaluated

If a procedure cannot perform a requested computation, it can return the unevaluated form of the function call that invoked it. For example, the procedure below computes the larger of two values if it can, or returns unevaluated if it cannot:

```
> Larger := proc( x, y )
      if x :: numeric and y :: numeric then
          if x > y then
              x
          else
              y
          end if
      else
          'Larger'(x,y)
      end if
  end proc:
> Larger(3.2, 2);
```

$$3.2 \tag{6.84}$$

```
> r := Larger(a, 2*b);
```

$$r := Larger(a, 2\ b) \tag{6.85}$$

The unevaluation quotes around **Larger** within the procedure specify that the function call expression will be constructed, but no procedure invocation will take place (therefore this is *not* a recursive call).

The returned unevaluated function call can later be re-evaluated. If **a** and **b** have numeric values at that time, **Larger** will return a number, otherwise it will return unevaluated once again.

```
> a, b := 3, 2;
```

$$a, b := 3, 2 \tag{6.86}$$

```
> r;
```

$$4 \tag{6.87}$$

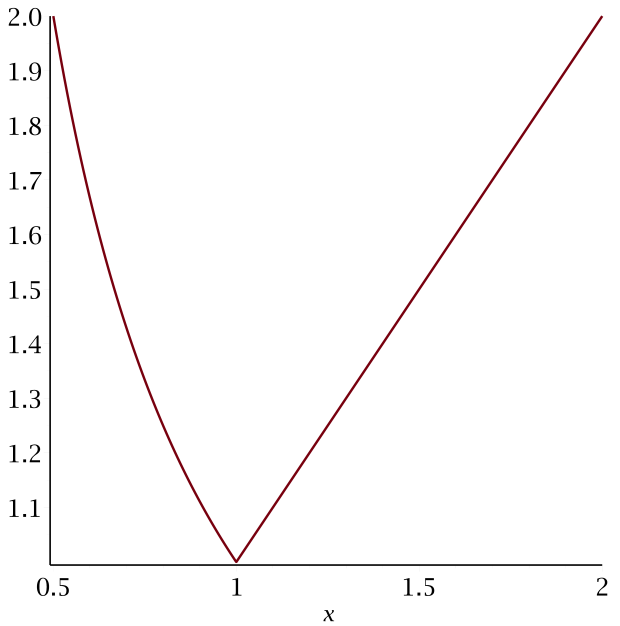
Because of one level evaluation, the last line in the example above would have to be written as r := eval(r) if **r** were a local variable in a procedure.

Rather than using the procedure's name to construct an unevaluated function call to return, you can also use the special name **procname**. The statement, 'Larger'(x,y) could have been written 'procname'(x,y). The advantage to using **procname** is that such unevaluated returns are immediately apparent to anyone reading the source code of your procedure.

Note that if your procedure was called from within another procedure and has the **procname** *option*, then an unevaluated call of the form 'procname'(x,y) refers to the procedure that invoked your procedure.

By writing procedures to return unevaluated when it is not possible to carry out the computation, you make it easier for the user of the procedure to use it in contexts where otherwise it would produce an error:

```
> plot( Larger(x, 1/x), x = 1/2 .. 2 );
```




```
> int( Larger(x, 1/x), x = 0.25 .. 2.0 );
```

$$2.886294361 \tag{6.88}$$

If **Larger** had been implemented without the unevaluated return, both of the above commands would have failed because the first argument to **plot** and **int** could not have been evaluated:

```
> LargerNoUneval := proc( x, y )
      if x > y then
          x
      else
          y
      end if
  end proc:
> plot( LargerNoUneval(x, 1/x), x = 1/4 .. 2 );
Error, (in LargerNoUneval) cannot determine if this expression is true
 or false: 1/x < x
```

```
> int( LargerNoUneval(x, 1/x), x = 0.25 .. 2.0 );
Error, (in LargerNoUneval) cannot determine if this expression is true
 or false: 1/x < x
```

Many Maple functions use the technique of returning unevaluated. For example, the **sin** and **int** functions return a result when they can, or return unevaluated when it is not yet possible to compute a result.

6.7 Using Data Structures with Procedures

The choice of appropriate data structures to solve a particular problem has already been discussed in *Basic Data Structures (page 125)*, but it is worth keeping in mind how your procedure might be used by you or others in the future. If the problem you are solving involves a small amount of data, you may have been tempted to choose a data structure without regard to efficiency or scalability when writing your procedure. If the procedure is used later to solve a larger problem, it may not be able to handle the problem in a reasonable amount of time or memory if you chose a data structure only suitable for small problems.

Passing Data Structures to Procedures

Traditional procedural programming languages such as Pascal or C usually pass arguments to procedures *by value*. This means that the procedure receives a *copy* of the data passed to it. Such languages also allow values to be passed *by reference*. Pascal does this by prefixing the parameter declaration with the var keyword. C requires that the parameter be declared as a *pointer*, using the * prefix, and that the caller explicitly pass the *address* of the argument using the & prefix (except when passing pointers to arrays).

Passing arguments by value ensures that the procedure cannot modify the passed data as a side-effect, but requires making a copy of the data. Passing by reference is more efficient for large data objects, but allows the procedure to (possibly unintentionally) modify the caller's copy of the data.

In Maple, data is always passed by reference, but the immutability of most data types ensures that the procedure cannot modify the caller's copy of the data. The exceptions are Maple's mutable data structures: tables, Arrays, Matrices, Vectors, records, and objects. Modifying these within a procedure will modify the caller's copy. Fortunately, these larger data structures are the ones that you would most often want to pass by reference, since copying such data consumes time and space.

A third argument passing convention seen in some programming languages is passing *by name*. In this case, instead of passing the *value* of a variable, *the variable itself* is passed. The called procedure can then assign a new value to the variable, which will remain in effect when the procedure returns to the caller. Maple allows passing by name via the evaln parameter declaration modifier, or by explicitly quoting the name when calling the procedure.

This does not contradict the earlier statement that Maple always passes by reference, because it is now the *variable name* that is being passed by reference.

Returning Data Structures from Procedures

Just as values are always passed by reference, they are *returned* from procedures by reference, too. Thus, the cost in time and space of returning a large structure such as a list is not any more than that of a small piece of data like an integer.

When returning a table or procedure from a procedure, care must be taken to ensure that it is the data structure itself and not the name referring to it that is returned. This is because tables and procedures use last name evaluation.

```
> IncorrectListToTable := proc( L :: list )
      local T := table(), i;
      for i to numelems(L) do
          T[i] := L[i]
      end do;
      T
  end proc:
> IncorrectListToTable(["String",123,Pi]);
```

$$T \tag{6.89}$$

The example above returns the local variable **T** instead of the actual table. Although the returned value can be used as if it were the actual table, every access to it involves an extra level of addressing behind the scenes, thus consuming more time.

```
> ListToTable := proc( L :: list )
      local T := table(), i;
      for i to numelems(L) do
          T[i] := L[i]
      end do;
      eval(T)
  end proc:
> ListToTable(["String",123,Pi]);
```

$$table([1 = \text{"String"}, 2 = 123, 3 = \pi]) \tag{6.90}$$

Example: Computing an Average

A common problem is to write a procedure that computes the average of **n** data values x_1, x_2, ..., x_n according to the following equation:

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n}$$

Before writing the procedure, think about which data structure and Maple functions to use. You can represent the data for this problem as a list. The numelems function returns the total number of entries in a list **X**, while the **i**th entry of the list is obtained by using X[i]:

```
> X := [1.3, 5.3, 11.2, 2.1, 2.1];
```

$$X := [1.3, 5.3, 11.2, 2.1, 2.1] \tag{6.91}$$

```
> numelems(X);
```

$$5 \tag{6.92}$$

```
> X[2];
```

$$5.3 \tag{6.93}$$

```
> add( i, i=X );
```

$$22.0 \tag{6.94}$$

Using these ideas, write the procedure **Average** which computes the average of the entries in a list. It handles empty lists as a special case:

```
> Average := proc( L::list, $ )
      local n := numelems(L), i, total;
      if n = 0 then
          error "empty list"
      end if;
      total := add(i,i=L);
      total / n
  end proc:
```

Using this procedure you can find the average of list **X** defined above:

```
> Average(X);
```

$$4.400000000 \tag{6.95}$$

The procedure also works if the list contains symbolic entries:

```
> Average([a, b, c]);
```

$$\frac{1}{3}\,a+\frac{1}{3}\,b+\frac{1}{3}\,c \tag{6.96}$$

Calling **Average** with an empty list raises an exception:

```
> Average([]);
```

```
Error, (in Average) empty list
```

A list is a good choice for the data in this example because the data is stored and used in a calculation, but the list itself does not need to be modified.

Example: Binary Search

One of the most basic and well-studied computing problems is that of searching. A typical problem involves searching a list of words (a dictionary, for example) for a specific word **w**. There are many possible methods. One approach is to search the list by comparing each word in the dictionary with **w** until either **w** is found, or the end of the list is reached. Study the code for procedure **LinearSearch** (the first attempt at solving this problem):

```
> LinearSearch := proc( D::list(string), w::string )
      local x;
      for x in D do
          if x = w then
              return true
          end if
      end do;
      false
  end proc:
```

Unfortunately, if the dictionary is large, this approach can take a long time. You can reduce the execution time required by sorting the dictionary before you search it. If you sort the dictionary into ascending order, then you can stop searching as soon as you encounter a word greater than **w**. On average, it is still necessary to search half the dictionary.

Binary searching provides an even better approach. Check the word in the middle of the sorted dictionary. Since the dictionary is already sorted, you can determine whether **w** is in the first or the second half. Repeat the process with the appropriate half of the dictionary until **w** is found, or it is determined not to be in the dictionary.

```
> BinarySearch := proc( D::list(string), w::string )
      local low := 1, high := numelems(D), mid;
      while low <= high do
          mid := trunc((low + high) / 2);
```

```
            if w < D[mid] then
                high := mid - 1
            elif w > D[mid] then
                low := mid + 1
            else
                return true
            end if
        end do;
        false
    end proc:
```

```
> Dictionary := [ "induna", "ion", "logarithm", "meld" ];
```

$$Dictionary := [\text{"induna"}, \text{"ion"}, \text{"logarithm"}, \text{"meld"}] \tag{6.97}$$

```
> BinarySearch( Dictionary, "hedgehogs" );
```

$$false \tag{6.98}$$

```
> BinarySearch( Dictionary, "logarithm" );
```

$$true \tag{6.99}$$

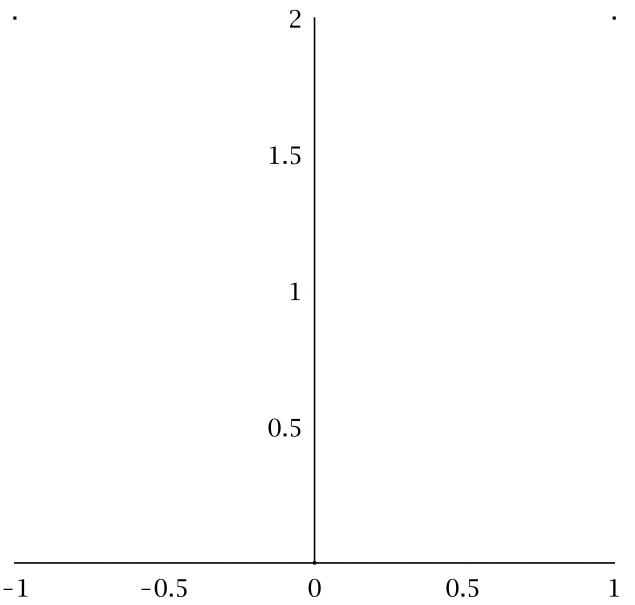
```
> BinarySearch( Dictionary, "melody" );
```

$$false \tag{6.100}$$

Example: Plotting the Roots of a Polynomial

You can construct lists of any type of object, including other lists. A list that contains two numbers can represent a point in the plane, and a list of such list can represent several such points. The Maple plot command uses this structure to generate plots of points and lines.

```
> plot([ [0, 0], [1, 2], [-1, 2] ],
       style=point, symbol=point, color=black);
```



You can make use of this to write a procedure that plots the complex roots of a polynomial. For example, consider the polynomial $x^3 - 1$.

```
> y := x^3-1;
```

$$y := x^3 - 1 \tag{6.101}$$

First, find the roots of this polynomial. You can find the numeric roots of this polynomial by using fsolve. By enclosing the call to fsolve in square brackets, you create a list of the roots.

```
> R := [ fsolve(y=0, x, complex) ];
```

$$R := [-0.500000000000000 - 0.866025403784439\,\mathrm{I},\ -0.500000000000000 + 0.866025403784439\,\mathrm{I},\ 1.] \tag{6.102}$$

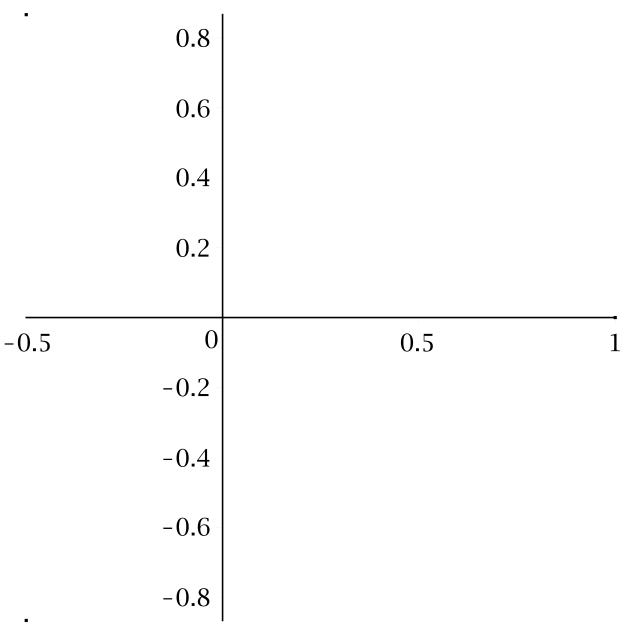
Next, change this list of complex numbers into a list of points in the plane. The Re and Im functions return the real and imaginary parts of a complex number respectively. You can use the map function and an anonymous procedure to convert the entire list at once.

```
> points := map(z -> [Re(z), Im(z)], R);
```

$$points := [[-0.500000000000000, -0.866025403784439], [-0.500000000000000, 0.866025403784439], [1., 0.]] \tag{6.103}$$

Finally, plot the resulting list.

```
> plot(points, style=point, symbol=point, color=black);
```

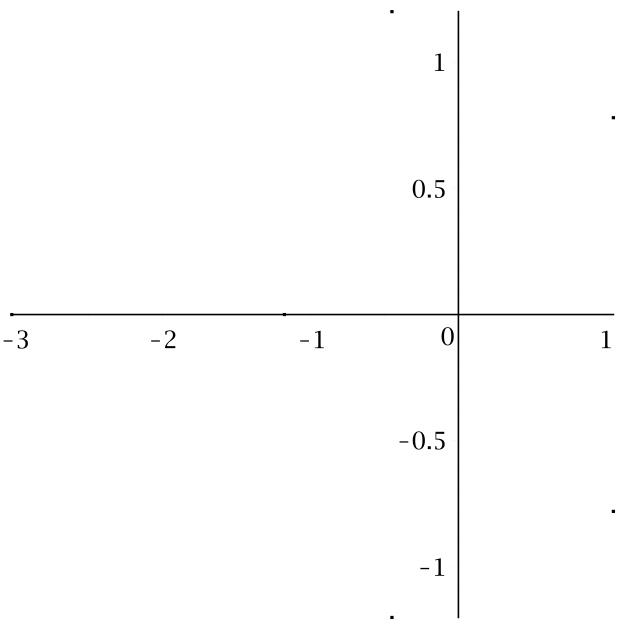


You can automate the process by writing a procedure that follows the same sequence of steps. The input must be a polynomial in x with constant coefficients.

```
> RootPlot := proc( p::polynom(constant,x) )
      description "Plots the roots of a polynomial in x";
      local R := [ fsolve(p, x, complex) ];
      local points := map( z -> [Re(z), Im(z)], R );
      plot(points, style=point, symbol=point, color=black)
  end proc:
```

Test the RootPlot procedure by plotting the roots of the polynomial $x^6 + 3\,x^5 + 5\,x + 10$.

```
> RootPlot( x^6+3*x^5+5*x+10 );
```

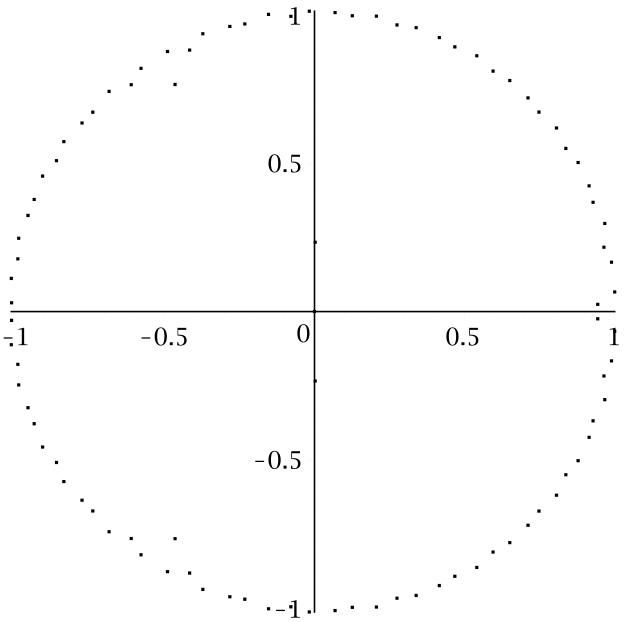


Generate a random polynomial using the randpoly function, and then test the RootPlot procedure again.

```
> y := randpoly(x, degree=100);
```

$$y := -56\,x^{95} - 62\,x^{42} + 97\,x^{8} - 73\,x^{5} - 4\,x^{3} \tag{6.104}$$

```
> RootPlot( y );
```



6.8 Writing Usable and Maintainable Procedures

As with any programming language, it is easy to write a Maple procedure that others cannot easily comprehend (or that you, as the author, have trouble understanding when you look at it, or try to modify it, in the future). Maple's syntax provides you with several facilities to alleviate such problems and produce *maintainable code*.

Formatting Procedures for Readability

Although it is possible to enter an entire procedure on a single very long line, this makes it difficult to understand and edit. For example, the binary search procedure shown earlier *could* have been written this way:

```
> BinarySearch := proc( D::list(string), w::string ) local low :=
  1, high := numelems(D), mid; while low <= high do mid := trunc((low
   + high) / 2); if w < D[mid] then high := mid - 1 elif w > D[mid]
   then low := mid + 1 else return true end if end do; false end
  proc:
```

Procedures are more easily readable if written with one statement per line, and with the statements enclosed within the bodies of loops and if-statements indented:

```
> BinarySearch := proc( D::list(string), w::string )
      local low := 1, high := numelems(D), mid;
      while low <= high do
          mid := trunc((low + high) / 2);
          if w < D[mid] then
              high := mid - 1
          elif w > D[mid] then
              low := mid + 1
          else
              return true
          end if
      end do;
      false
  end proc:
```

Sometimes, a single statement is too long to fit on a single line. Maple's syntax allows you to insert line breaks and white space between any two syntactic *tokens* such as reserved words, variable names, numbers, and punctuation. Indentation can be used within a statement to clarify the grouping of expressions. For example, the polynomial root plotting procedure could have been written like this:

```
> RootPlot := proc( p::polynom(constant,x) )
      description "Plots the roots of a polynomial in x";
      plot(map(z -> [Re(z), Im(z)],
                [fsolve(p, x, complex)]),
           style=point, symbol=point, color=black)
  end proc:
```

In this version of **RootPlot**, the procedure body consists of a description and a single statement. The indentation makes it clear that z -> [Re(z), Im(z)] and [fsolve(p, x, complex)] are arguments of the call to map, and that the result of this call together with the **style**, **symbol**, and **color** options are the arguments of plot.

Commenting Your Code

Comments are one of the most important tools in writing maintainable code. There are two ways of writing comments in Maple procedures:

```
# Comment text until the end of the line.
(* Delimited comment text. *)
```

A # character anywhere within a procedure except inside a "string" or `quoted name` introduces a comment. Everything following # until the end of the line is considered to be a comment and is ignored by Maple. This form is useful for short comments of one or two lines, or to annotate a line.

```
> Average := proc( )
      # Compute total.
      local total := add(_passed[i],i=1.._npassed);
      # Divide total by number of values.
      total / _npassed;
  end proc;
```

$$\begin{array}{l} Average := \mathbf{proc}(\,) \\ \quad \mathbf{local}\ total; \\ \quad total := add(args[\,i\,],\ i = 1\,..nargs);\ total \,/\, nargs \\ \mathbf{end\ proc} \end{array} \tag{6.105}$$

Comments enclosed in (* and *) can begin and end anywhere except within a "string" or `quoted name`. Everything between the delimiters is ignored by Maple. This form can be used within a line or to write a multiline comment.

```
> BetterAverage := proc( )
      (* This procedure computes the average of its
         arguments. It is an error if no arguments were
         passed. *)
      if _npassed = 0 then
          error "too few values"
      else
          add(_passed[i],i=1.._npassed) (*TOTAL*) / _npassed
      end if
  end proc;
```

$$\begin{array}{l} BetterAverage := \mathbf{proc}(\,) \\ \quad \mathbf{if}\ nargs = 0\ \mathbf{then} \\ \quad\quad \mathbf{error}\ \text{"too few values"} \\ \quad \mathbf{else} \\ \quad\quad add(args[\,i\,],\ i = 1\,..nargs) \,/\, nargs \\ \quad \mathbf{end\ if} \\ \mathbf{end\ proc} \end{array} \tag{6.106}$$

Notice that comments are discarded by Maple when the procedure is simplified. Comments are purely for the benefit of the programmer(s) who write, read, and maintain the procedure.

As described earlier, a procedure in Maple can also have a description section. One or more strings can follow the description keyword. Like comments, these have no effect on the execution of the procedure, but they are retained when the procedure is simplified.

```
> AnotherAverage := proc( )
      description "Compute the average of one or more values.",
                  "At least one value must be passed.";
      if _npassed = 0 then
          error "too few values"
      else
          add(_passed[i],i=1.._npassed) / _npassed
      end if
  end proc:
```

You can use Maple's Describe command to print a procedure's declared parameters, return type, and description.

```
> Describe(AnotherAverage);


# Compute the average of one or more values.
# At least one value must be passed.
AnotherAverage( )
```

```
> Describe(RootPlot);


# Plots the roots of a polynomial in x
RootPlot( p::polynom(constant,x) )
```

6.9 Other Methods for Creating Procedures

Enclosing a sequence of statements in proc...end proc is not the only way to create a procedure in Maple. You can also use functional operator notation or the unapply function.

Functional Operators: Mapping Notation

Functional operator notation (or *arrow* notation) is a method by which you can create a special form of procedure which represents a mathematical function or mapping. The syntax is:

```
( parameterSequence ) -> expression
```

The **parameterSequence** can be empty, and the **expression** must be a *single* expression or an if statement.

```
> F := (x,y) -> x^2 + y^2;
```

$$F := (x, y) \rightarrow x^2 + y^2 \tag{6.107}$$

If the procedure requires only a single parameter, you can omit the parentheses around **parameterSequence**:

```
> G := n -> if n < 0 then 0 else 1 end if;
```

$$G := n \rightarrow \textbf{if } n < 0 \textbf{ then } 0 \textbf{ else } 1 \textbf{ end if} \tag{6.108}$$

Internally, a procedure created using operator notation is the same as any other procedure, except that it will have options operator, arrow. You can invoke such a procedure in the usual way:

```
> F(1,2);
```

$$5 \tag{6.109}$$

```
> G(-1);
```

$$0 \tag{6.110}$$

You can use declared parameter types when defining a functional operator:

```
> H := ( n::even ) -> n! * (n/2)!;
```

$$H := n::even \rightarrow n! \left(\frac{1}{2}\, n\right)! \tag{6.111}$$

```
> H(6);
```

$$4320 \tag{6.112}$$

```
> H(5);
```

```
Error, invalid input: H expects its 1st argument, n, to be of type
even, but received 5
```

The arrow notation is designed for simple one-line function definitions. It does not provide a mechanism for specifying local or global variables, options, a description, or more than a single statement. If these are required, use the more general proc...end proc notation.

The unapply Function

Another way to create a procedure is with the unapply function:

```
unapply( expression, parameterSequence )
```

The **expression** must be a *single* expression, and **parameterSequence** a sequence of symbols.

```
> B := x^2 + y^2;
```

$$B := x^2 + y^2 \tag{6.113}$$

```
> F := unapply(B, x, y);
```

$$F := (x, y) \rightarrow x^2 + y^2 \tag{6.114}$$

```
> F(3,4);
```

$$25 \tag{6.115}$$

The functional operator notation (or arrow notation) is a *syntax* for writing an operator. The unapply function is a *function* mapping expressions to procedures. Use the unapply function to create a procedure from an expression that was *computed* instead of one that was *entered*. This works because unapply first evaluates the expression and then encloses the result in a procedure. The arrow notation always produces a procedure containing the expression that was entered.

```
> IntExpr := int(1/(x^3+1), x);
```

$$IntExpr := \frac{1}{3}\ln(x+1) - \frac{1}{6}\ln\left(x^2 - x + 1\right) + \frac{1}{3}\sqrt{3}\arctan\left(\frac{1}{3}(2x-1)\sqrt{3}\right) \tag{6.116}$$

```
> IntFunc := unapply(evalf(IntExpr), x);
```

$$IntFunc := x \rightarrow 0.3333333333\ln(x+1.) - 0.1666666667\ln\left(x^2 - 1.x + 1.\right) + 0.5773502693\arctan(1.154700539x - 0.5773502693) \tag{6.117}$$

```
> IntFunc(3.5);
```

$$0.8664586908 \tag{6.118}$$

If you had tried to use operator notation to create the **IntFunc** procedure, you would not get what you expected:

```
> BadIntFunc := x -> evalf(IntExpr);
```

$$BadIntFunc := x \rightarrow evalf(IntExpr) \tag{6.119}$$

```
> BadIntFunc(3.5);
```

$$0.3333333333\ln(x+1.)-0.1666666667\ln(x^2-1.x+1.)+0.5773502693\arctan(1.154700539x-0.5773502693) \tag{6.120}$$

Notice that the result still contains the symbol **x**. This is because the **x** appearing in **IntExpr** is the *global* variable **x**, not the *parameter* **x** of **BadIntFunc**.

Anonymous Procedures

Recall from the beginning of this chapter that a procedure is a valid Maple expression, independent from any name that it may have been assigned to. You can in fact create, manipulate, and invoke a procedure without ever assigning it to name. Such procedures are called *anonymous*.

Consider the following mapping (a procedure in functional operator notation):

```
> x -> x^2;
```

$$x \to x^2 \tag{6.121}$$

You can invoke this anonymous procedure in the following manner:

```
> (x -> x^2) (t);
```

$$t^2 \tag{6.122}$$

Syntactically, this is a Maple function call. Instead of specifying the procedure to call by giving its name, the procedure is given directly. The same method can be used to directly call a procedure defined using the proc...end proc notation:

```
> proc( x, y ) x^2 + y^2 end proc (u, v);
```

$$u^2+v^2 \tag{6.123}$$

Anonymous procedures are often used with the map function:

```
> map( x -> x^2, [1,2,3,4] );
```

$$[1, 4, 9, 16] \tag{6.124}$$

They are also used to initialize Arrays in *Arrays (page 144)*. You can find numerous other examples of anonymous procedures in this guide.

Procedures, whether anonymous or not, can be combined in expressions, or processed by operators such as D, the differential operator:

```
> D( x -> x^2 );
```

$$x \rightarrow 2\,x \tag{6.125}$$

```
> F := D( exp + 2 * ln );
```

$$F := \exp + 2\left(z \rightarrow \frac{1}{z}\right) \tag{6.126}$$

```
> F(x);
```

$$e^{x} + \frac{2}{x} \tag{6.127}$$

6.10 Recursion

A procedure is termed *recursive* if it contains a call to itself, either directly, or indirectly through another procedure that it calls. In order for a recursive procedure to produce a result, it must test for some condition under which the recursion terminates. Otherwise, it would go on calling itself forever (until Maple runs out of *stack* space).

You have already seen one example of recursion used to compute Fibonacci numbers in *The remember, cache, and system Options (page 215)*. Another well-known example of recursion is the computation of the factorial of an integer. For any integer $0 < n$, the factorial (denoted by $n!$) is defined by $n! = n\,(n-1)!$. For $n = 0$, $n!$ is defined to be equal to 1. This definition naturally lends itself to a recursive implementation:

```
> Fact := proc( n::nonnegint, $ )
      if n > 0 then
          n * Fact(n-1)
      else
          1
      end if
  end proc;
```

$$\begin{aligned} &Fact := \mathbf{proc}(n::nonnegint,\ \$) \\ &\quad \mathbf{if}\ 0 < n\ \mathbf{then}\ n * Fact(n-1)\ \mathbf{else}\ 1\ \mathbf{end\ if} \\ &\mathbf{end\ proc} \end{aligned} \tag{6.128}$$

```
> Fact(0);
```

$$1 \tag{6.129}$$

```
> Fact(4);
```

$$24 \tag{6.130}$$

```
> Fact(-4);
```

```
Error, invalid input: Fact expects its 1st argument, n, to be of type
 nonnegint, but received -4
```

The if-statement ensures that **Fact** only calls itself when $0 < n$.

Rather than using the name to which the procedure has been assigned to make the recursive call, you can also use **procname** or **thisproc**. This ensures that the recursion continues to work even if the procedure body is later assigned to a different name. The special symbol **procname** refers to the name that the procedure was called with. In the **Fact** example, **procname** would be equivalent to **Fact**. The symbol **thisproc** on the other hand refers to the procedure itself. Calling the procedure recursively using **thisproc** is slightly more efficient, and works within anonymous procedures.

This example uses an anonymous version of the **Fact** procedure above to compute the factorials of a list of numbers:

```
> map( n -> if n > 0 then n * thisproc(n-1) else 1 end if,
       [0, 1, 2, 3, 4] );
```

$$[1, 1, 2, 6, 24] \tag{6.131}$$

The **BinarySearch** procedure you saw earlier also lends itself to a recursive implementation.

```
> BinarySearch := proc( D::list(string), w::string,
                        low::integer := 1, high ::integer :=
  numelems(D) )
      local mid;
      if low > high then
          # Nothing left to search. Word is not in list.
          false
      else
          mid := trunc((low + high) / 2);
          if w < D[mid] then
              # Search within the left part of the range.
              thisproc(D,w,low,mid-1)
          elif w > D[mid] then
              # Search within the right part of the range.
              thisproc(D,w,mid+1,high)
          else
              # Word was found in middle of current range.
```

```
            true
        end if
    end if
end proc:
```

```
> Dictionary := [ "induna", "ion", "logarithm", "meld" ];
```

$$Dictionary := [\text{"induna"}, \text{"ion"}, \text{"logarithm"}, \text{"meld"}] \tag{6.132}$$

```
> BinarySearch( Dictionary, "hedgehogs" );
```

$$false \tag{6.133}$$

```
> BinarySearch( Dictionary, "logarithm" );
```

$$true \tag{6.134}$$

```
> BinarySearch( Dictionary, "melody" );
```

$$false \tag{6.135}$$

You use this procedure by passing it a sorted list of strings and a word to search for. The two optional parameters, **low** and **high**, specify which range of list elements to search and have default values specifying the entire list. After determining that the word is lexicographically less than or greater than the middle value, this procedure calls itself recursively, passing the list and word, as well as appropriate values for the **low** and **high** parameters to restrict the search. The recursion (and thus the search) ends when the procedure is asked to search a zero-length section of the list (in which case the word was not found), or when the middle element of the specified range contains the word.

If your procedure has the **procname** *option*, any attempt to make a recursive call via **procname** instead of **thisproc** calls the procedure that invoked your procedure.

6.11 Procedures that Return Procedures

Some of the built-in Maple commands return procedures. For example, rand returns a procedure which in turn produces randomly chosen integers from a specified range. The dsolve function with the type=numeric option returns a procedure which supplies a numeric estimate of the solution to a differential equation.

You can write procedures that return procedures too. This section discusses how values are passed from the outer procedure to the inner procedure.

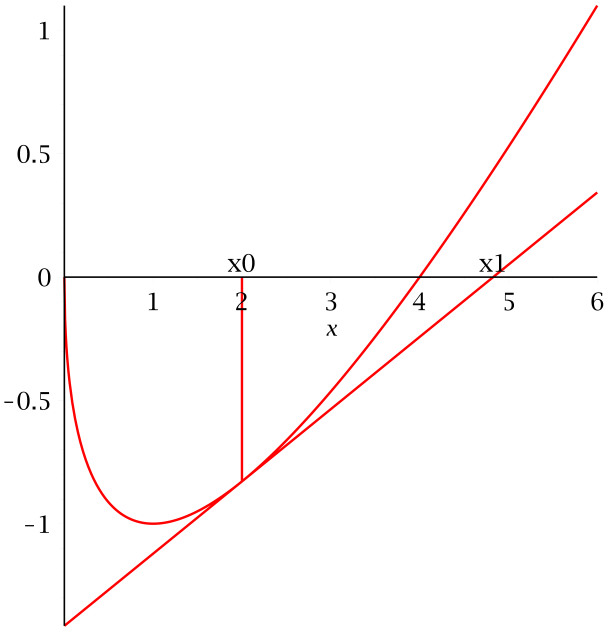
Example: Creating a Newton Iteration

The following example demonstrates how locating the roots of a function by using Newton's method can be implemented in a procedure.

To use Newton's method to find the roots of a function graphically:

Choose a point on the x-axis that you think might be close to a root.

Draw the tangent to the curve at that point and observe where the tangent intersects the x-axis. For most functions, this second point is closer to the real root than the initial guess. Use the new point as a new guess and repeat this process.



The same process can be expressed numerically as an iteration:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

where x_0 is the initial guess, and x_k is the result of the **k**th iteration.

The following procedure takes a function and creates a new procedure which expects an initial guess and, for that particular function, generates the next guess. The new procedure is specific to the function that it was generated for, and does not work for other functions. To find the roots of a new function, use **MakeIteration** to generate a new iterating procedure.

```
> MakeIteration := proc( expr::algebraic, x::name )
      local iteration := x - expr / diff(expr, x);
```

```
    unapply(iteration, x);
end proc:
```

The procedure returned by the **MakeIteration** procedure maps the name **x** to the expression assigned to the iteration.

Test the procedure on the expression $x - 2\sqrt{x}$:

```
> expr := x - 2 * sqrt(x);
```

$$expr := x - 2\sqrt{x} \tag{6.136}$$

```
> iter := MakeIteration(expr,x);
```

$$iter := x \rightarrow x - \frac{x - 2\sqrt{x}}{1 - \frac{1}{\sqrt{x}}} \tag{6.137}$$

The generated procedure, which is assigned to **iter**, returns the solution, $x = 4$ after a few iterations.

```
> x0 := 2.0:
> to 4 do x0 := iter(x0); print(x0) end do:
```

$$4.828427124$$

$$4.032533198$$

$$4.000065353$$

$$4.000000000 \tag{6.138}$$

Observe that the **MakeIteration** procedure above requires its first argument to be an algebraic expression. You can also write a version of **MakeIteration** that works on other procedures (such as functional operators).

```
> MakeIteration := proc( f::procedure )
      (x->x) - eval(f) / D(eval(f));
  end proc:
```

This example uses Maple's ability to treat expressions *containing* procedures *as* procedures. The result of calling this version of **MakeIteration** is an expression with procedures as operands (x->x is just a procedure that maps any value to itself).

Because of *last name evaluation*, **MakeIteration** will accept either a procedure or a *name* whose value is a procedure. The calls to eval within **MakeIteration** ensure that the result refers to the actual procedure that was passed in, instead of to the name of that procedure.

```
> g := x -> x - cos(x);
```

$$g := x \rightarrow x - \cos(x) \tag{6.139}$$

```
> iter := MakeIteration(g);
```

$$iter := (x \rightarrow x) - \frac{x \rightarrow x - \cos(x)}{x \rightarrow 1 + \sin(x)} \tag{6.140}$$

Note that the procedure generated by the call to **MakeIteration** is independent of the name **g** (because of the aforementioned calls to eval). Thus, you can later change **g** without breaking **iter**. You can find a good approximate solution to $x - \cos(x) = 0$ in a few iterations.

```
> x0 := 1.0;
```

$$x0 := 1.0 \tag{6.141}$$

```
> to 4 do x0 := iter(x0); print(x0) end do:
```

$$0.7503638679$$

$$0.7391128909$$

$$0.7390851334$$

$$0.7390851332 \tag{6.142}$$

Example: A Shift Operator

Consider the problem of writing a procedure that takes a function, f, as input and returns a function, g, such that $g(x) = f(x+1)$. You can write such a procedure like this:

```
> ShiftLeft := ( f::procedure ) -> ( x -> f(x+1) ):
```

Try performing a shift on sin(x).

```
> ShiftLeft(sin);
```

$$x \rightarrow \sin(x+1) \tag{6.143}$$

Maple lexical scoping rules declare the **f** within the inner procedure to be the same **f** as the parameter of the outer procedure. Therefore, the **ShiftLeft** procedure works as written.

The previous example of **ShiftLeft** works with univariate functions but it does not work with functions of two or more variables.

```
> h := (x,y) -> x*y;
```

$$h := (x, y) \rightarrow x\,y \tag{6.144}$$

```
> hh := ShiftLeft(h);
```

$$hh := x \rightarrow h(x+1) \tag{6.145}$$

```
> hh(x,y);
```

```
Error, (in hh) invalid input: h uses a 2nd argument, y, which is
missing
```

To modify **ShiftLeft** to work with multivariate functions, rewrite it to generate procedures that accept the additional parameters and pass them on to **f**.

```
> ShiftLeft := ( f::procedure ) -> ( x->f(x+1, _rest) ):
> hh := ShiftLeft(h);
```

$$hh := x \rightarrow h(x+1, \_rest) \tag{6.146}$$

```
> hh(x,y);
```

$$(x+1)\, y \tag{6.147}$$

Because the **ShiftLeft** procedure does not call eval on parameter **f**, the function **hh** depends on **h**. Changing the value assigned to **h** implicitly changes **hh**:

```
> h := (x,y,z) -> y*z^2/x;
```

$$h := (x, y, z) \rightarrow \frac{y\, z^2}{x} \tag{6.148}$$

```
> hh(x,y,z);
```

$$\frac{y\, z^2}{x+1} \tag{6.149}$$

6.12 The Procedure Object

Recall that a Maple procedure is itself an *expression* in Maple which can be (and usually is) assigned to a name. Like any Maple expression, a procedure has a *type*, and has *operands* (not to be confused with its parameters).

The procedure Type

Maple recognizes all procedures (and names to which a procedure has been assigned) as being of type **procedure**. To verify whether a name or an expression is a procedure, use the type function or :: operator:

```
> F := proc( x ) x^2 end proc:
```

```
> type(F, name);
```

$$true \tag{6.150}$$

```
> type(F, procedure);
```

$$true \tag{6.151}$$

```
> type(F, name(procedure));
```

$$true \tag{6.152}$$

```
> type(eval(F), name);
```

$$false \tag{6.153}$$

```
> type(eval(F), procedure);
```

$$true \tag{6.154}$$

The **procedure** type is a *structured type* (see *Structured Types (page 120)*). Using a structured type allows you to verify that a name refers to a procedure, and additionally verify the specified types of the procedure's parameters.

```
> G := proc( n::integer, s::string )
      print(s);
      2 * n * length(s)
  end proc:
> type(G, procedure(integer,string));
```

$$true \tag{6.155}$$

Procedure Operands

Every Maple procedure has eight operands, corresponding to sub-parts of the procedure definition. The following table lists each operand and the corresponding op call that can be used to access it. In the table, the name **P** represents the name of the procedure, and the eval call is necessary so that op will be passed the procedure, not the name (because procedures have last name evaluation).

Table 6.1: Procedure Operands

Operand	op Command
Parameters	op(1,eval(P))
All local variables	op(2,eval(P))
Options	op(3,eval(P))
Remember table	op(4,eval(P))

Operand	op Command
Description	op(5,eval(P))
Declared global variables	op(6,eval(P))
Lexical scoping table	op(7,eval(P))
Return type	op(8,eval(P))

The value of any operand can be a single item, an expression sequence if there are two or more items (such as local variables), or NULL if there were no items (for example, no options).

The lexical scoping table is an internal structure that records the correspondence between undeclared variables within the procedure and locals (or exports), globals, and parameters of surrounding procedures (or modules). It does not correspond to any part of the procedure as written.

The procedure's statement sequence is *not* one of the operands of the procedure, and thus cannot be extracted by op. This is because statements and statement sequences are *not* expressions, and thus cannot be assigned to names or otherwise manipulated.

The following nested procedure illustrates how the parts of the procedure map to the operands. Note that this example refers to the procedure that appears *within*, and is *returned by* procedure **MakeProc** (in order to illustrate lexical scoping). This procedure is not intended to illustrate good programming style, but merely provide an example showing all the possible operands.

```
> MakeProc := proc( offset::integer )
      description "Create and return a procedure";
      proc( n::integer, s::string ) :: integer;
         description "An example to illustrate procedure operands";

          option remember;
          global codes := convert(s,bytes);
          local i;
          total := 0;
          for i to nops(codes) do
              total := total + codes[i]
          end do;
          total * n + offset
      end proc:
  end proc:
```

```
Warning, `total` is implicitly declared local to procedure
```

```
> P := MakeProc(3):
```

```
> P; # The name of the procedure
```

$$P \tag{6.156}$$

```
> eval(P); # The procedure
```

```
proc(n::integer, s::string)::integer;
    option remember;
    local i, total;
    global codes;
    description
    "An example to illustrate procedure operands";
    codes := convert(s, bytes);
    total := 0;
    for i to nops(codes) do
        total := total + codes[i]
    end do;
    total* n + 3
end proc
```

(6.157)

```
> op(1,eval(P)); # Parameters
```

$$n::integer,\ s::string \tag{6.158}$$

```
> op(2,eval(P)); # All local variables
```

$$i,\ total \tag{6.159}$$

```
> op(3,eval(P)); # Options
```

$$remember \tag{6.160}$$

```
> P(3,"nonsense"); # Place an entry in the remember table
```

$$2622 \tag{6.161}$$

```
> op(4,eval(P)); # Show the remember table
```

$$table([(3, \text{"nonsense"}) = 2622]) \tag{6.162}$$

```
> op(5,eval(P)); # Description
```

$$\text{"An example to illustrate procedure operands"} \tag{6.163}$$

```
> op(6,eval(P)); # Declared global variables
```

$$codes \tag{6.164}$$

```
> op(7,eval(P)); # Lexical table
```

$$offset, 3 \tag{6.165}$$

```
> op(8,eval(P)); # Return type
```

$$integer \tag{6.166}$$

6.13 Exercises

1. Implement the function $f(x) = \left(\sqrt{-x^2+1}\right)^3 - 1$, first as a procedure, and then by using the mapping notation. Compute **f(1/2)** and **f(0.5)**, and comment on the different results.

2. You can use $\frac{a\,b}{g}$ to compute the least common multiple of two integers, **a** and **b**, where **g** is the greatest common divisor of **a** and **b**. For example, the least common multiple of 4 and 6 is 12. Write a Maple procedure, **LCM**, which takes as input n>0 integers a_1, a_2, ... , a_n and and computes their least common multiple. By convention, the least common multiple of zero and any other number is zero.

3. Write a Maple procedure called **Sigma** which, given n>1 data values, x_1, x_2, ..., x_n, computes their standard deviation. The following equation gives the standard deviation of n>1 numbers, where mu is the average of the data values.

$$\mathrm{sigma} = \sqrt{\frac{\sum_{i=1}^{n}\left(x_i - \mathrm{mu}\right)^2}{n}}$$

4. Write a Maple procedure which, given a list of lists of numerical data, computes the mean of each column of the data.

5. Write a Maple procedure called **Position** which returns the position **i** of an element **x** in a list **L**. That is, **Position(x,L)** should return an integer i>0 such that **L[i]=x**. If **x** is not in list **L**, **0** is returned.

7 Numerical Programming in Maple

An important part of efficient scientific and mathematical programming is numerical computation. Maple provides many tools for computing with floating-point numbers, some for improving efficiency and some for improving accuracy.

7.1 In This Chapter

- An Overview of Numeric Types in Maple
- An Explanation of Floating-Point Numbers in Maple
- Maple Commands for Numerical Computing
- Efficient Numerical Programs

7.2 Numeric Types in Maple

Before discussing numerical computing in Maple, we will first introduce the various numeric data types used in Maple and briefly describe how they are represented. All of the real numbers discussed in this section will pass checks of type,numeric or type,extended_numeric.

Integers

The most basic numeric type in Maple is the integer. Small integers are represented directly as hardware integers (similar to the **int** type in C or **integer** type in Fortran), which allows for maximum efficiency of both CPU time used for arithmetic and memory used for storage. That is, the number can be stored in one machine word and arithmetic operations can be performed with one CPU operation. On 32-bit architectures, integers in the range $-2^{30}-1$ to $2^{30}-1$ are stored in this way, while on 64-bit architectures, integers in the range $-2^{62}-1$ to $2^{62}-1$. Integers stored in this way are referred to as *immediate integers*.

Larger integers are stored in DAGs of type INTPOS or INTNEG, which contain pointers to arrays of digits that can store integers up to magnitude $10^{2^{28}-218}$ on 32-bit architectures and $10^{2^{35}+2^{32}-18}$ on 64-bit architectures.

```
> dismantle(2^80-1);
```

```
INTPOS(6): 1208925819614629174706175
```

```
> dismantle(-2^101+6);
```

```
INTNEG(6): -2535301200456458802993406410746
```

The arithmetic for these large integers is computed using the GNU Multiple Precision Arithmetic (GMP) library. This library is quite efficient, but still several orders of magnitude slower than arithmetic on immediate integers since each arithmetic operation will require more than one CPU operation and the larger the integer, the more operations and memory will be needed for arithmetic.

```
> CodeTools:-Usage(add(i,i=-2^15..2^16));
```

```
memory used=160.56KiB, alloc change=0 bytes, cpu time=8.00ms, real
time=11.00ms
```

$$1610629120 \tag{7.1}$$

```
> CodeTools:-Usage(add(i,i=2^88-2^15..2^88+2^16));
```

```
memory used=8.92MiB, alloc change=0 bytes, cpu time=28.00ms, real
time=27.00ms
```

$$304239238904873269809912123392 00 \tag{7.2}$$

```
> CodeTools:-Usage(add(i,i=2^4097-2^15..2^4097+2^16));
```

```
memory used=109.36MiB, alloc change=0 bytes, cpu time=320.00ms, real
 time=322.00ms
```

$$\begin{aligned}&205337297974639914340665500453995519859046 7\backslash\\&\quad 7100520612506134414514845817784628791159 1\\&\quad 0478394945398699[...1039\ \text{digits}...]\\&\quad 6395393555619749389086067511784172161221 0\\&\quad 9439600358445025489751475386970283444370 6\\&\quad 806664146972590080\end{aligned} \tag{7.3}$$

Any transitions between GMP integers and immediate integers will be completely transparent and it is not possible to tell them apart in general without use low-level tools such as address-of. However, you can check if an integer is small enough to fit into a single machine word with types integer[4] and integer[8] for 4-byte and 8-byte words respectively.

Integers of all types pass a type,integer type check.

The Integer constructor is guaranteed to return an integer, an extended numeric symbol such as infinity or undefined, a complex number with integer parts, or return unevaluated.

```
> Integer(-2^160);
```

$$-1461501637330902918203684832716283019655932542976 \tag{7.4}$$

```
> Integer(infinity);
```

$$\infty \tag{7.5}$$

```
> Integer(1/2);
```

$$Integer\left(\frac{1}{2}\right) \tag{7.6}$$

The system dependent value for the largest immediate integer can be found with kernelopts(maximmediate), the maximum number of decimal digits in an integer can be found with kernelopts(maxdigits), and the version of the GMP library being used can be found with kernelopts(gmpversion).

Rationals

Exact rational numbers are stored in DAGs of type RATIONAL, which consist of a pair of integers. The first integer is the numerator and can be a POSINT or NEGINT. The second integer is the denominator and is a POSINT. Most low-level programming languages such as C or Fortran do not have an equivalent rational number type.

```
> dismantle(1/2);

RATIONAL(3): 1/2
   INTPOS(2): 1
   INTPOS(2): 2
> dismantle(-10/3);

RATIONAL(3): -10/3
   INTNEG(2): -10
   INTPOS(2): 3
```

Rational numbers can be constructed by using the division operator or the Fraction constructor. In either case, automatic simplification will occur to ensure that the denominator is positive and that the fraction is in lowest terms (the numerator and denominator do not have factors in common). This means that the Fraction constructor may return integers in some cases.

```
> dismantle(Fraction(21,7));

INTPOS(2): 3
> dismantle(Fraction(40,-14));

RATIONAL(3): -20/7
   INTNEG(2): -20
   INTPOS(2): 7
```

Rational number arithmetic is performed in the natural way using integer arithmetic and the igcd and ilcm operations to reduce to lowest terms.

```
> Fraction(2^20+2^12,2^27-2^13) + Fraction(2^12-1,2^13);
```

$$\frac{68141057}{134209536} \tag{7.7}$$

```
> Fraction(2^20+2^12,2^27-2^13) * Fraction(23,187);
```

$$\frac{5911}{6127242} \tag{7.8}$$

Rational numbers of all types will pass a type,rational type check. Only rational numbers that are not also integers will pass a type,fraction type check. Additionally, type,extended_rational includes all rationals as well as the extended numeric symbols infinity, -infinity, and undefined.

```
> type(1, fraction);
```

$$false \tag{7.9}$$

Like the Integer constructor, the Fraction constructor will return unevaluated if it cannot return a value of type extended_rational.

```
> Fraction(x,1);
```

$$Fraction(x, 1) \tag{7.10}$$

```
> Fraction(infinity);
```

$$\infty \tag{7.11}$$

Floating-Point Numbers

Floating-point numbers are stored in DAGs of type FLOAT.

In Maple, as in nearly every programming language, floating-point numbers can be constructed using and visually distinguished from integers with a decimal point symbol, '.'. The floating-point number $1.$ is often treated as equal to the exact integer 1.

```
> evalb(1. = 1);
```

$$true \tag{7.12}$$

Maple floating-point numbers can also be constructed with the SFloat constructor (or the equivalent Float constructor) and can be checked with the nearly equivalent type,sfloat and type,float types. We will generally refer to these numbers as *sfloats* to when we need to distinguish them from hardware floating-point numbers (*hfloats*), introduced below.

```
> Float(1);
```

$$1. \tag{7.13}$$

```
> dismantle(SFloat(0.3333));
```

```
FLOAT(3): .3333
   INTPOS(2): 3333
   INTNEG(2): -4
```

```
> type(.1, float);
```

$$true \tag{7.14}$$

```
> type(.1, sfloat);
```

$$true \tag{7.15}$$

```
> type(1, float);
```

$$false \tag{7.16}$$

A floating-point number represents a rational number with a fixed precision. That rational number can be recovered with convert/rational.

```
> convert(.3333333333, rational, exact);
```

$$\frac{3333333333}{10000000000} \tag{7.17}$$

However, not every rational number can be represented exactly by a floating-point number. For example, the closest floating-point number to $\frac{1}{3}$ is 0.3333333333.

```
> convert(1/3, float);
```

$$0.3333333333 \tag{7.18}$$

Also, unlike numeric types integer and rational, integer and float do not have compatible arithmetic. Floating-point arithmetic has a fixed finite precision, and does round off if the result of arithmetic does not fit into that precision.

```
> 9123456789 + 8123456789 <> convert( 9123456789. + 8123456789.,
  rational, exact);
```

$$17246913578 \neq 17246913580 \tag{7.19}$$

```
> 123456 * 1234567 <> convert( 123456.*1234567., rational, exact);
```

$$152414703552 \neq 152414703600 \tag{7.20}$$

Unlike many other programming languages the precision of sfloat arithmetic can be changed. For this reason, sfloats are known as *arbitrary precision* floating-point numbers.

More information on sfloats and how they differ from the floating-point types in languages such as C and Fortran will be discussed in greater detail in *More about Floating-Point Numbers in Maple (page 275)*.

Hardware Floating-Point Numbers

Floating-point numbers of the type used in languages such as C and Fortran can also be constructed in Maple; they are known as hardware floating-point numbers or *hfloats*. These types are stored as 8-byte double precision IEEE floating-point numbers contained in DAGs of type HFLOAT. Since the . notation is used to construct Maple sfloats, hfloats must be constructed with the HFloat constructor. Maple will display sfloats and hfloats the same way, using decimal notation.

```
> HFloat(1);
```

$$1. \tag{7.21}$$

```
> dismantle(HFloat(0.3333));
```

```
HFLOAT(2): .3333
```

The advantage of hfloats over sfloats is that their arithmetic is computed directly using a single CPU operation for each arithmetic operation. Maple sfloats, however, offer much more flexibility and precision. In many ways the difference is analogous to the difference between immediate integers and GMP integers.

Hardware floats can be distinguished from sfloats with the type,hfloat type.

```
> type(HFloat(1), float);
```

$$true \tag{7.22}$$

```
> type(HFloat(1), sfloat);
```

$$false \tag{7.23}$$

```
> type(HFloat(1), hfloat);
```

$$true \tag{7.24}$$

```
> type(SFloat(1), hfloat);
```

$$false \tag{7.25}$$

For more information on hardware floats and how they differ from sfloats, see *More about Floating-Point Numbers in Maple (page 275)*.

Extended Numeric Types

The special built-in symbols **infinity** (∞), and **undefined** can be used in numeric arithmetic in Maple. In general, operations involving ∞ simplify automatically to a signed infinity or a complex infinity. For details, refer to the type,infinity help page.

```
> -1*infinity;
```

$$-\infty \tag{7.26}$$

```
> 1/2*infinity;
```

$$\infty \tag{7.27}$$

```
> 1/infinity;
```

$$0 \tag{7.28}$$

The **undefined** symbol is usually produced as the result of attempting to carry out an operation that cannot result in a number for the given operands. Almost every arithmetic operation involving **undefined** returns **undefined**. For details, refer to the type,undefined help page.

```
> infinity-infinity;
```

$$undefined \tag{7.29}$$

```
> undefined-undefined;
```

$$undefined \tag{7.30}$$

```
> undefined+1;
```

$$undefined \tag{7.31}$$

Integer and rational numbers share exact undefined and infinite symbols while sfloat and hfloat numbers have their own versions of these, which are displayed differently but treated similarly.

```
> Float(infinity);
```

$$Float(\infty) \tag{7.32}$$

```
> HFloat(undefined);
```

$$HFloat(undefined) \tag{7.33}$$

Complex Numbers

A complex number in Maple is a DAG of type COMPLEX, which consists of a pair of any of the two numeric types. They can be constructed in the natural way using the symbol I for the imaginary unit, or using the Complex constructor.

```
> dismantle(1+I);

COMPLEX(3)
   INTPOS(2): 1
   INTPOS(2): 1
> dismantle(Complex(1/2,1/3));

COMPLEX(3)
   RATIONAL(3): 1/2
      INTPOS(2): 1
      INTPOS(2): 2
   RATIONAL(3): 1/3
      INTPOS(2): 1
      INTPOS(2): 3
```

Automatic simplification will ensure that if one of the parts of a complex number is a float (or hfloat), then other will be made into a float (hfloat).

```
> dismantle(Complex(1., 1/1001));

COMPLEX(3)
   FLOAT(3): 1.
      INTPOS(2): 1
      INTPOS(2): 0
   FLOAT(3): .9990009990e-3
      INTPOS(2): 9990009990
      INTNEG(2): -13
```

```
> dismantle(Complex(HFloat(1.), 1/1001));

COMPLEX(3)
   HFLOAT(2): 1.
   HFLOAT(2): .000999000999
> dismantle(Complex(HFloat(1.), 1.));

COMPLEX(3)
   HFLOAT(2): 1.
   HFLOAT(2): 1.
```

Complex numbers are not of type type,numeric but can be checked with type type,complex which can also be structured to check for the numeric subtypes of its two components.

```
> type(1+I,numeric);
```

$$\mathit{false} \tag{7.34}$$

```
> type(1+I,complex(integer));
```

$$\mathit{true} \tag{7.35}$$

Non-numeric Constants

Many Maple expressions represent constants, but are not considered to be of type numeric. This means that arithmetic performed on these constants will be more generic symbolic operations on DAGs of type SUM, PROD, NAME, or FUNCTION. Some examples of non-numeric constants are **Pi** (π), *Catalan*, $\sin(1)$, $\sqrt{5}$, and $\pi + e^2 - \sqrt{1 + 5\ Catalan}$.

```
> type(Pi, numeric);
```

$$\mathit{false} \tag{7.36}$$

```
> type(sqrt(5)-1, constant);
```

$$\mathit{true} \tag{7.37}$$

7.3 More about Floating-Point Numbers in Maple

To take full advantage of floating-point numbers and to avoid many common pitfalls in numerical computing, it is important to understand exactly what floating-point numbers are and how they are represented.

Representation of Floating-Point Numbers in Maple

The dismantle command shows that the two numbers 1 and $1.$ have different internal representations. 1 is simply stored as an integer while $1.$ is stored as a pair of integers.

```
> dismantle(1);

INTPOS(2): 1
```

```
> dismantle(1.);

FLOAT(3): 1.
   INTPOS(2): 1
   INTPOS(2): 0
```

Similarly, the numbers $\frac{1}{2}$ and 0.5 are also different even though they are both stored as pairs of integers.

```
> dismantle(1/2);

RATIONAL(3): 1/2
   INTPOS(2): 1
   INTPOS(2): 2
```

```
> dismantle(0.5);

FLOAT(3): .5
   INTPOS(2): 5
   INTNEG(2): -1
```

In Maple, the FLOAT DAG-type represents a floating-point number in the form S * 10^E where both S and E are integers. For $1.$, the *significand* (or *mantissa*) is $S = 1$ and the *exponent* is $E = 0$. In addition to being specified in decimal notation, floats of this form can be constructed by using scientific notation, or the Float constructor.

```
> Float(2,0);
```

$$2. \tag{7.38}$$

```
> 2*1e0;
```

$$2. \tag{7.39}$$

The advantage of using this significand-exponent representation is that fixed precision approximations of large and small numbers can be stored compactly and their arithmetic can

be done efficiently. Storing the integer 10^50 needs at least 167 bits or 3 words on a 64-bit machine. The floating-point number 1e50 can be stored in less than 8 bits but in in practice uses 2 words (one for each integer).

```
> dismantle(10^50);

INTPOS(8): 10000000000000000000000000000000000000000000000000

> dismantle(1e50);

FLOAT(3): .1e51
   INTPOS(2): 1
   INTPOS(2): 50
```

Using two immediate integers, a float can store a much larger range of numbers than a rational number with two immediate integers. The range a rational can represent is about $1.\,10^{-9}..1.\,10^{9}$ while a float can represent a range of about $1.\,10^{-1073741823}..9.\,10^{1073741823}$. This is a **much** larger range for the same storage cost. Of course, that larger range means that floats of a fixed size can represent fewer numbers in that range. And since floating-point numbers are always of a fixed size, this means that arithmetic will always be of limited precision. That is, each operation will have to round the result to a number that can be represented as another floating-point number.

In Maple, the significand is limited to 10 decimal digits of precision by default but can be changed while the exponent is restricted to being a word-sized integer.

More information on the restrictions on the size of software floats in Maple can be found by using the Maple_floats command.

By contrast, hfloats, are represented in base 2, rather than base 10. So they represent numbers using the form S * 2^E, where the significand, S, is a 52-bit integer and the exponent, E, is a 10-bit integer. Thus, the range of numbers representable as hardware floats is $2.225073859\,10^{-308}..1.797693135\,10^{308}$. Because the largest possible significand of a hardware float has about $\mathrm{floor}\left(\log_{10}\left(2^{52}\right)\right) = 15$ base-10 digits of precision, hardware floats can be converted to software floats without meaningful loss of precision when Digits is 15. Conversely, so long as their exponent is smaller than 307 and their significand had fewer than 15 digits sfloats can be converted to hfloats without loss of precision.

Precision and Accuracy

By default, 10-digit precision is used for floating-point arithmetic, which means that the arithmetic will be rounded to 10 digits. This means any single floating-point operation will be accurate to 10 digits.

For example, storing 10^50+1 requires 50 decimal digits so it will be rounded in floating-point arithmetic. By contrast, 10^50+10^41 can be stored with 10 digits so it will still be computed accurately.

```
> .1e51 + 1.;
```

$$1.\,10^{50} \tag{7.40}$$

```
> .1e51 + .1e42;
```

$$1.000000001\,10^{50} \tag{7.41}$$

The Digits environment variable can be used to change the working precision used by Maple. Larger values of Digits will allow more accurate computation, but at the cost of slower arithmetic.

```
> Digits := 100:
> .1e51 + 1.;
```

$$1.00000000000000000000000000000000000000000000000\backslash 00000001\,10^{50} \tag{7.42}$$

The maximum value for Digits is system dependent and can be found with the Maple_floats command.

```
> Maple_floats(MAX_DIGITS);
```

$$38654705646 \tag{7.43}$$

For the default value of Digits, the significand is an immediate integer and so arithmetic will be fast in general. It also means that some numerical function evaluations (such as **sin** in the following example) will be able to use the CPU's native hardware floating-point arithmetic to achieve the needed precision. However, raising Digits about the default value will lead to slower arithmetic and slower function evaluation.

```
> Digits:=10:
> CodeTools:-CPUTime(add(sin(1e-6*i),i=1..100000));
```

$$2.500, 4995.884639 \tag{7.44}$$

```
> Digits:=22:
> CodeTools:-CPUTime(add(sin(1e-6*i),i=1..100000));
```

$$6.080, 4995.8846386821409 98954 \tag{7.45}$$

Reducing Digits below its default value does not usually lead to large improvements in performance.

```
> Digits:=5:
> CodeTools:-CPUTime(add(sin(1e-6*i),i=1..100000));
```

$$2.460, 4996.0 \tag{7.46}$$

It is also important to note that changing Digits does not necessarily change the accuracy of sequences of multiple floating-point computations; it changes only the precision of the individual operations performed. The following example computes two additions using 10 digits of precision, but *catastrophic cancellation* leads to a mere one digit of accuracy in the final answer.

```
> Digits := 10:
> x := 1234567890.;
```

$$x := 1.234567890\ 10^{9} \tag{7.47}$$

```
> y := -x+1;
```

$$y := -1.234567889\ 10^{9} \tag{7.48}$$

```
> z := 3.141592654;
```

$$z := 3.141592654 \tag{7.49}$$

```
> x+z+y<>z+1;
```

$$4. \neq 4.141592654 \tag{7.50}$$

Ensuring accuracy requires careful study of the problem at hand. In this example, you need 19 digits of precision to get 10 digits of accuracy.

```
> Digits := 19:
> x+z+y=z+1;
```

$$4.141592654 = 4.141592654 \tag{7.51}$$

Floating-Point Contagion

An important property of floating-point numbers in Maple, and nearly every other computing environment, is *contagion*. When numerical expressions are created involving both floating-point numbers and exact numbers, the floating property is *contagious* and causes the answer to become a floating-point number.

```
> 1. * 10;
```

$$10. \tag{7.52}$$

```
> 0. + 10;
```

$$10. \tag{7.53}$$

As you can see, this contagion property can be used as a quick method to convert exact values to floating-point numbers. However, while floating-point contagion extends to all Maple structures of type *numeric* (except, in some cases, hfloats), it does not apply to non-numeric constants.

```
> type(3/4, numeric);
```

$$true \tag{7.54}$$

```
> 4/3 + 0.;
```

$$1.333333333 \tag{7.55}$$

```
> 1.*sqrt(3);
```

$$1.\sqrt{3} \tag{7.56}$$

The hfloat type is also contagious, but the precise behavior of the contagion is determined by the UseHardwareFloats environment variable. By default, hfloats are contagious for small values of Digits:

```
> type(4/3 + HFloat(0.), hfloat);
```

$$true \tag{7.57}$$

```
> type(1. + HFloat(0.), hfloat);
```

$$true \tag{7.58}$$

```
> HFloat(1.1) * sin(4*Pi/7) -1;
```

$$1.10000000000000 \sin\left(\frac{3}{7}\pi\right) - 1 \tag{7.59}$$

For large values of Digits, hfloats in computations will be converted to sfloats so that the results are sfloats.

```
> Digits := 20;
```

$$Digits := 20 \tag{7.60}$$

```
> type(1 + HFloat(0.), hfloat);
```

$$false \tag{7.61}$$

```
> type(1 + HFloat(0.), sfloat);
```

$$true \tag{7.62}$$

If UseHardwareFloats=true then hfloats are completely contagious.

```
> UseHardwareFloats := true;
```

$$UseHardwareFloats := true \tag{7.63}$$

```
> Digits := 20;
```

$$Digits := 20 \tag{7.64}$$

```
> a := 10.^19+1;
```

$$a := 1.0000000000000000001\ 10^{19} \tag{7.65}$$

```
> b := a + HFloat(0.1);
```

$$b := 1.00000000000000\ 10^{19} \tag{7.66}$$

```
> type(b, hfloat);
```

$$true \tag{7.67}$$

If UseHardwareFloats=false then hfloats will always be converted to sfloats in computations, regardless of the setting of Digits. The HFloat constructor will still create hfloats, however.

```
> UseHardwareFloats := false;
```

$$UseHardwareFloats := false \tag{7.68}$$

```
> Digits := 10;
```

$$Digits := 10 \tag{7.69}$$

```
> c := 1 + HFloat(0.1);
```

$$c := 1.100000000 \tag{7.70}$$

```
> type(c, hfloat);
```

$$false \tag{7.71}$$

```
> type(HFloat(0.1), hfloat);
```

$$true \tag{7.72}$$

Table 7.1 summarizes the floating-point contagion rules.

Table 7.1: Floating-Point Contagion Rules

UseHardwareFloats	true	false	deduced	deduced
Digits	any	any	1...15	16...

hfloat + hfloat	hfloat	sfloat	hfloat	sfloat
hfloat + sfloat	hfloat	sfloat	hfloat	sfloat
sfloat + sfloat	sfloat	sfloat	sfloat	sfloat

More on the Floating-Point Model

The software floating-point system is designed as a natural extension of the industry standard for hardware floating-point computation, known as IEEE 754. Thus, there are representations for **infinity** and **undefined** (what IEEE 754 calls a *NaN*, meaning *Not a Number*) as discussed in *Extended Numeric Types (page 273)*.

The IEEE 754 standard defines five rounding algorithms. Two methods called nearest and simple round to nearest values, and the other three are directed roundings that round up or down (as needed) towards $-\infty$, ∞, or 0. Maple implements all of these rounding modes and the desired mode can be selected by setting the Rounding environment variable.

```
> Rounding;
```

$$nearest \tag{7.73}$$

```
> 1.4^10;
```

$$28.92546550 \tag{7.74}$$

```
> Rounding := 0;
```

$$Rounding := 0 \tag{7.75}$$

```
> 1.4^10;
```

$$28.92546549 \tag{7.76}$$

Another important feature of this system is that the floating-point representation of zero, **0.**, retains its arithmetic sign in computations. That is, Maple distinguishes between **+0.** and **-0.** when necessary. In most situations, this difference is irrelevant, but when dealing with functions that have a discontinuity across the negative real axis, such as $\ln(x)$, preserving the sign of the imaginary part of a number on the negative real axis is important.

For more intricate applications, Maple implements extensions of the IEEE 754 notion of a **numeric event**, and provides facilities for monitoring events and their associated status flags. For more information about this system, refer to the numerics help page.

7.4 Maple Commands for Numerical Computing

In this section we will discuss some of the commands available in Maple for floating-point computation.

The evalf Command

The evalf command is the primary tool in Maple for performing floating-point calculations in software floating-point mode. You can use **evalf** to compute approximations of non-numeric constants.

```
> evalf(Pi);
```

$$3.141592654 \tag{7.77}$$

You can alter the number of digits of the approximation by changing the value of the environment variable **Digits**, or by specifying the number as an index to **evalf** (which leaves the value of Digits unchanged).

```
> Digits := 20:
> evalf(Pi);
```

$$3.1415926535897932385 \tag{7.78}$$

```
> evalf[200](Pi);
```

$$\begin{aligned}&3.14159265358979323846264338327950288419716\backslash\\&39937510582097494459230781640628620899862\backslash\\&80348253421170679821480865132823066470938\backslash\\&44609550582231725359408128481117450284102\backslash\\&70193852110555964462294895493038 20\end{aligned} \tag{7.79}$$

```
> evalf(sqrt(2));
```

$$1.4142135623730950488 \tag{7.80}$$

```
> Digits := 10:
```

Remember that the **Digits** command specifies the precision in decimal digits, unlike hardware floating-point numbers which specify precision in binary digits.

All floating-point computations are performed in *finite precision*, with intermediate results generally being rounded to **Digits** precision. As such, it is possible for round-off errors to accumulate in long computations. Maple ensures that the result of any *single* floating-point arithmetic operation (+, -, *, /, or **sqrt**) is fully accurate. Further, many of the basic functions in Maple, such as the trigonometric functions and their inverses, the exponential and logarithmic functions, and some of the other standard special functions for mathematics, are ac-

curate to within **.6** *units in the last place* (ulps), meaning that if the **Digits + 1st** digit of the true result is a 4, Maple may round it up, or if it is a 6, Maple may round it down. Most mathematical functions in Maple, including numerical integration, achieve this accuracy on nearly all inputs.

It is possible to create software floats with different precisions. Changing the value of **Digits** will not change these numbers; it affects only the precision of subsequent operations on those numbers.

```
> Digits := 50;
```

$$Digits := 50 \tag{7.81}$$

```
> a := evalf(Pi);
```

$$a := 3.1415926535897932384626433832795028841971693993751 \tag{7.82}$$

```
> Digits := 10;
```

$$Digits := 10 \tag{7.83}$$

```
> a;
```

$$3.1415926535897932384626433832795028841971693993751 \tag{7.84}$$

```
> a+1;
```

$$4.141592654 \tag{7.85}$$

```
> evalf(a);
```

$$3.141592654 \tag{7.86}$$

From this example, you can see that **evalf** can be used to create a lower precision float from one of higher precision. This can be used to round a result to a desired number of digits. However, **evalf** will not increase the precision of a low precision float.

```
> evalf[100](1.0);
```

$$1.0 \tag{7.87}$$

```
> evalf[10000](a);
```

$$3.1415926535897932384626433832795028841971693993751 \tag{7.88}$$

The evalf command also provides an interface to purely numerical computations of integrals, limits, and sums.

Some definite integrals have no closed-form solution in terms of standard mathematical functions. You can use **evalf** to obtain a numerical answer directly using numerical techniques.

```
> r := Int(exp(x^3), x=0..1);
```

$$r := \int_0^1 e^{x^3}\, dx \tag{7.89}$$

```
> value(r);
```

$$\int_0^1 e^{x^3}\, dx \tag{7.90}$$

```
> evalf(r);
```

$$1.341904418 \tag{7.91}$$

In other cases, Maple can find an exact solution, but the form of the exact solution is almost incomprehensible. The function Beta in the following example is a special function that appears in mathematical literature.

```
> q := Int( x^99 * (1-x)^199 / Beta(100, 200), x=0..1/5 );
```

$$q := \int_0^{\frac{1}{5}} \frac{x^{99}\,(1-x)^{199}}{\mathrm{B}(100, 200)}\, dx \tag{7.92}$$

```
> value(q);
```

$$\begin{aligned}
&27852290545780521179255248650434305998403841\\
&98009096903421704176220527155238977619068\\
&28166964420518416902474524718187972029459\\
&61766386779717574634134906442572750186110\\
&14357501573520181129894929725484 49/\\
&(\\
&21774128091037151646887384971552115934384\\
&96176725167103101324312241148610308262514\\
&47555252405132308313238717840332750249360\\
&60378263034137682537367383346083183346165\\
&22866113357176260162148352832620593365691\\
&18501246614718189600663973041983050027165\\
&65259568426426994847133755683898925781250\\
&0000\,\mathrm{B}(100, 200))
\end{aligned} \tag{7.93}$$

```
> evalf(q);
```

$$3.546007367\ 10^{-8} \tag{7.94}$$

The two previous examples use the **Int** command rather than **int** for the integration. If you use **int**, Maple first tries to integrate the expression symbolically. Thus, when evaluating the following commands, Maple determines a symbolic answer and then converts it to a floating-point approximation, rather than performing direct numerical integration. In general, the symbolic computation is more difficult, and thus slower than the numerical computation.

```
> evalf( int(x^99 * (1-x)^199 / Beta(100, 200), x=0..1/5) );
```

$$3.546007367\ 10^{-8} \tag{7.95}$$

Similarly, **evalf** can be used on the inert forms **Limit** and **Sum** to compute using numerical algorithms for computing numeric limits and sums.

```
> evalf(Limit(sin(erf(1)*x)/(erf(1)^2*x),x=0));
```

$$1.186660803 \tag{7.96}$$

```
> evalf( Sum(exp(x), x=RootOf(_Z^5+_Z+1)) );
```

$$4.791792042 + 0.\mathrm{I} \tag{7.97}$$

When Not to Use evalf

In general the symbolic commands in Maple are able to handle floating-point numbers in their input, but, by their nature floats are not as precise as rationals or symbolic constants. So, even if you want a numerical answer from a command, you should avoid calling evalf on the input.

The following command does not compute the expected answer of 0.1111111111.

```
> limit(n*(evalf(1/3) - 1/(3+1/n)), n=infinity);
```

$$-Float(\infty) \tag{7.98}$$

It would have been computed correctly with non-float values in the input.

```
> evalf( limit(n*(1/3 - 1/(3+1/n)), n=infinity) );
```

$$0.1111111111 \tag{7.99}$$

Numeric Solvers

There are also a number of numerical algorithms available in Maple in commands other than **evalf**. One of the most important is fsolve which is short for *floating-point solve*. This command computes numerical solutions to equations or systems of equations. In general, it is much more efficient than calling **evalf** on the result of solve, especially if you are interested in only a single solution.

```
> fsolve( exp(x) + 2*sin(x), x);
```

$$-0.3573274113 \tag{7.100}$$

The **fsolve** command is a sophisticated heuristic that chooses among many different algorithms depending on the input. There are several more special purpose solving tools available in the RootFinding package.

Several symbolic solvers in Maple also have numeric modes. The **dsolve** and **pdsolve** commands both accept a **numeric** option, which indicates that a numerical answer should be computed using purely numeric methods. For extensive information on these numeric commands, refer to the dsolve/numeric and pdsolve/numeric help pages.

The evalhf Command

Like evalf, evalhf computes an numerical approximation of its input. However, evalhf uses hardware floats in all intermediate calculations before returning an sfloat.

```
> dismantle( evalhf(1/3) );

FLOAT(3): .333333333333333315
   INTPOS(2): 333333333333333315
   INTNEG(2): -18
```

The evalhf command is affected by the value of Digits, but since intermediate calculations are done with hfloats, at most 18 digits will be returned.

```
> Digits := 100;
```

$$Digits := 100 \tag{7.101}$$

```
> evalhf(1/3) ;
```

$$0.333333333333333315 \tag{7.102}$$

Notice that in this example the result is only correct to 16 digits. In general, the results from evalhf are guaranteed to 15 digits of accuracy.

To find the number of guaranteed digits for your version of Maple, use evalhf(Digits):

```
> evalhf(Digits);
```

$$15. \tag{7.103}$$

In fact, evalhf is, despite superficial similarities, very different from evalf. The evalhf command uses a completely separate evaluation environment which uses only simple types rather that the Maple DAG types. This means that it can be very fast, but at the cost of being limited in the types of computations it can perform.

```
> Digits := 15;
```

$$Digits := 15 \tag{7.104}$$

```
> c := 10.^14;
```

$$c := 1.00000000000000\,10^{14} \tag{7.105}$$

```
> CodeTools:-Usage( evalhf( add( (i+c), i=1..10^6) ) );

memory used=0.89KiB, alloc change=0 bytes, cpu time=44.00ms, real
time=43.00ms
```

$$1.00000000499999867\,10^{20} \tag{7.106}$$

```
> CodeTools:-Usage( ( add( (i+c), i=1..10^6) ) );
```

```
memory used=95.06MiB, alloc change=4.00MiB, cpu time=1.02s, real
time=1.01s
```

$$1.00000000500000\,10^{20} \tag{7.107}$$

```
> c := HFloat(c);
```

$$c := 1.00000000000000\,10^{14} \tag{7.108}$$

```
> CodeTools:-Usage( ( add( (i+c), i=1..10^6) ) );
```

```
memory used=35.03MiB, alloc change=8.00MiB, cpu time=768.00ms, real
time=768.00ms
```

$$1.00000000500001\,10^{20} \tag{7.109}$$

In particular evalhf only handles a specific list of functions. For the list of functions that evalhf recognizes, refer to the evalhf/fcnlist help page.

```
> evalhf(sin(exp(gamma+2)+ln(cos(Catalan))));
```

$$0.09801979012383793 54 \tag{7.110}$$

```
> evalhf( b /3 );
```

```
Error, cannot handle unevaluated name `b` in evalhf
```

evalhf works with Arrays of hardware floats. It cannot handle symbols, lists, sets, and most other Maple data structures.

```
> evalhf(map(t->t+1, [1, 2, 3, 4]));
```

```
Error, unable to evaluate expression to hardware floats: [1, 2, 3, 4]
```

To create an Array of hardware floats, you can use the option datatype=float[8], which specifies that the elements in the Array are 8-byte hardware floats.

```
> A := Array([1, 2, 3, 4], datatype=float[8]);
```

$$A := \left[\begin{array}{cccc} 1. & 2. & 3. & 4. \end{array}\right] \tag{7.111}$$

```
> evalhf(map(t->t+1, A));
```

$$\left[\begin{array}{cccc} 2. & 3. & 4. & 5. \end{array}\right] \tag{7.112}$$

You can also create an Array that can be used by evalhf by using the constructor hfarray. Both constructors create an Array of hardware floats. The only difference is that hfarray defaults to C_order instead of Fortran_order.

```
> A := hfarray(1..4, 1..4, (i,j)->ithprime(i)*isqrt(3*(i+j)));
```

$$A := \begin{bmatrix} 4. & 6. & 6. & 8. \\ 9. & 9. & 12. & 12. \\ 15. & 20. & 20. & 25. \\ 28. & 28. & 35. & 35. \end{bmatrix} \tag{7.113}$$

```
> lprint(A);
```

```
Array(1 .. 4,1 .. 4,{(1, 1) = HFloat(4.), (1, 2) = HFloat(6.), (1, 3)
 = HFloat(6.), (1, 4) = HFloat(8.), (2, 1) = HFloat(9.), (2, 2) =
HFloat(9.), (2, 3) = HFloat(12.), (2, 4) = HFloat(12.), (3, 1) =
HFloat(15.), (3, 2) = HFloat(20.), (3, 3) = HFloat(20.), (3, 4) =
HFloat(25.), (4, 1) = HFloat(28.), (4, 2) = HFloat(28.), (4, 3) =
HFloat(35.), (4, 4) = HFloat(35.)},datatype = float[8],storage =
rectangular,order = C_order)
```

User-defined Maple procedures can be evaluated in the evalhf environment as long as they comply with the restrictions outlined in the evalhf/procedure help page.

```
> SlowPower := proc(a,n) local i, s; s:=1; for i to n do s := a*s;
   end do; end proc;
```

$$\begin{aligned} &SlowPower := \mathbf{proc}(a, n) \\ &\quad \mathbf{local}\ i, s; \\ &\quad s := 1; \mathbf{for}\ i\ \mathbf{to}\ n\ \mathbf{do}\ s := a^* s\ \mathbf{end\ do} \\ &\mathbf{end\ proc} \end{aligned} \tag{7.114}$$

```
> SlowPower(2,10);
```

$$1024 \tag{7.115}$$

```
> evalhf( SlowPower(2,10) );
```

$$1024. \tag{7.116}$$

Numerical Linear Algebra

Maple has access to many libraries for fast numeric computation such as BLAS, CLAPACK, and the NAG® C Library. To take full advantage of the speed provided by these commands, you need to provide them with Matrices and Vectors with the appropriate datatype.

For example, floating-point Matrix times Matrix products can been computed very quickly in the BLAS libraries and quickest dispatch to the BLAS commands will happen if the Matrices are created with datatype=float[8].

```
> A := Matrix(5^3,5^3,(i,j)->(i-j+1)/(i+j));
```

$$A := \begin{bmatrix} 125 \text{ x } 125 \text{ Matrix} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.117}$$

```
> CodeTools:-Usage(A^2);
```

```
memory used=0.57GiB, alloc change=84.01MiB, cpu time=4.20s, real
time=4.16s
```

$$\begin{bmatrix} 125 \text{ x } 125 \text{ Matrix} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.118}$$

```
> Ahf := Matrix(5^3,5^3,(i,j)->(i-j+1)/(i+j), datatype=float[8]);
```

$$Ahf := \begin{bmatrix} 125 \text{ x } 125 \text{ Matrix} \\ \text{Data Type: } float_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.119}$$

```
> CodeTools:-Usage(Ahf^2);
```

```
memory used=300.97KiB, alloc change=0 bytes, cpu time=8.00ms, real
time=303.00ms
```

$$\begin{bmatrix} 125 \text{ x } 125 \text{ Matrix} \\ \text{Data Type: } float_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.120}$$

Of course, many of the linear algebra commands will try to determine if you have a Matrix of low precision floats and will convert to the appropriate datatype automatically. In the

next example, Af has datatype=anything, but the result of Af^2 has datatype=float[8] and requires only a small, but noticeable, copy and conversion overhead.

```
> Af := Matrix(5^3,5^3,(i,j)->(i-j+1.)/(i+j));
```

$$Af := \begin{bmatrix} 125 \text{ x } 125 \text{ Matrix} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.121}$$

```
> CodeTools:-Usage(Af^2);
```

```
memory used=0.54MiB, alloc change=0 bytes, cpu time=28.00ms, real
time=30.00ms
```

$$\begin{bmatrix} 125 \text{ x } 125 \text{ Matrix} \\ \text{Data Type: } float_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.122}$$

We recommend that you specify datatype=float[8] in your constructors explicitly if you intend to perform numeric computations. This makes the numeric nature of the Matrix explicit, and it makes it impossible to accidentally add non-float entries to a Matrix and thus make subsequent computations slower. An exception will be raised if non-numeric entries are assigned into the Matrix.

```
> Ahf[1,1] := sqrt(3);
```

```
Error, unable to store '3^(1/2)' when datatype=float[8]
```

Other numeric types will be automatically converted to float[8].

```
> Ahf[1,1] := 45/111;
```

$$Ahf_{1,1} := \frac{15}{37} \tag{7.123}$$

```
> Ahf[1,1];
```

$$0.405405405405405 \tag{7.124}$$

If a Matrix contains only floats, but does not have a datatype=float[8] restriction, then addition of symbolic elements results in the more expensive symbolic commands to be used.

```
> Af[1,1] := sqrt(3);
```

$$Af_{1,\,1} := \sqrt{3} \tag{7.125}$$

```
> CodeTools:-Usage(Af^2);
```

```
memory used=310.67MiB, alloc change=304.00MiB, cpu time=4.46s, real
time=4.39s
```

$$\begin{bmatrix} \textit{125 x 125 Matrix} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{bmatrix} \tag{7.126}$$

Another advantage of float[8] is that these Matrices are stored in the same way as an hfarray which is analogous to an array of floats in the C or Fortran programming languages and different from a Matrix of datatype=anything or datatype=sfloat which are arrays of Maple DAGs each of which will take more memory than a single 8-byte float. Note the difference in **memory used** in the following two examples.

```
> restart;
```

```
> CodeTools:-Usage(Matrix(10^3,3*10^3,(i,j)->10.^4*j+j,
  datatype=sfloat));
```

```
memory used=0.83GiB, alloc change=50.89MiB, cpu time=14.23s, real
time=14.27s
```

$$\begin{bmatrix} \textit{1000 x 3000 Matrix} \\ \textit{Data Type: sfloat} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{bmatrix} \tag{7.127}$$

```
> restart;
```

```
> B1:=CodeTools:-Usage(Matrix(10^3,3*10^3,(i,j)->10^4*j+i,
  datatype=float[8]));
```

```
memory used=91.66MiB, alloc change=58.90MiB, cpu time=3.06s, real
time=3.09s
```

$$B1 := \begin{bmatrix} 1000 \text{ x } 3000 \text{ Matrix} \\ \text{Data Type: } float_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.128}$$

It is also important to note that elements extracted from a float[8] rtable will be of type hfloat and so hfloat contagion will apply subject to the current setting of UseHardwareFloats.

```
> type(B1[1,1], hfloat);
```

$$true \tag{7.129}$$

There are also many optimized commands for Matrices of complex hfloats. These Matrices can be created using the option datatype=complex[8], and work similarly to those of datatype=float[8].

If you are constructing very large Matrices in your programs, use the ArrayTools package to construct and copy Matrices as efficiently as possible.

7.5 Writing Efficient Numerical Programs

Two main points to keep in mind when trying to write efficient numerical programs are:

- Try to use hardware floating-point arithmetic when **Digits** allows
- Try to minimize memory usage where possible

Writing Flexible Numerical Procedures

You can use the **evalhf(Digits)** construct to determine whether hardware floating-point arithmetic provides sufficient precision in a particular application. If **Digits** is less than **evalhf(Digits)**, then you can take advantage of the faster hardware floating-point calculations. Otherwise, you should use software floating-point arithmetic, with sufficient digits, to perform the calculation.

In the following example, the procedure **myevalf** takes an *unevaluated* parameter, **expr**. Without the **uneval** declaration, Maple would evaluate **expr** symbolically before invoking **myevalf**.

```
> myevalf := proc(expr::uneval)
     if Digits < evalhf(Digits) then
        evalf(evalhf(expr));
     else
        evalf(expr);
     end if;
  end proc:
```

The **evalhf** command evaluates many Maple functions, but not all. For example, you cannot evaluate an integral using hardware floating-point arithmetic.

```
> myevalf( Int(exp(x^3), x=0..1) );
```

```
Error, (in myevalf) unable to evaluate function `Int` in evalhf
```

You can improve the procedure **myevalf** so that it traps such errors and tries to evaluate the expression using software floating-point numbers instead.

```
> myevalf := proc(expr::uneval)
     if Digits < evalhf(Digits) then
        try
           return evalf(evalhf(expr));
        catch:
        end try;
     end if;
     return evalf(expr);
  end proc:
> myevalf( Int(exp(x^3), x=0..1) );
```

$$1.341904418 \tag{7.130}$$

This procedure provides a model of how to write procedures that use hardware floating-point arithmetic whenever possible.

The **myevalf** procedure returns sfloats. A version that returns hfloats is easiest to write using the hfloat procedure option. This option will cause the procedure to use hfloat arithmetic as much as possible so long as digits less than 15. In particular it convert all floats in the procedure definition into hfloats, and causes **evalhf** to not convert its output to an sfloat.

```
> myevalf := proc(expr::uneval)
     option hfloat;
     if Digits < evalhf(Digits) then
        try
           return evalhf(expr);
        catch:
        end try;
```

```
    end if;
    return evalf(1. * expr);
end proc:
```

The multiplication by $1.$ was added to the **evalf** return line to induce hfloat contagion causing the output to be an hfloat when possible.

```
> type( myevalf( Int(exp(x^3), x=0..1) ), hfloat);
```

$$true \tag{7.131}$$

For more information on the **hfloat** option, see *The hfloat Option (page 212)* or refer to the option_hfloat help page.

Example: Newton Iteration

This section illustrates how to take advantage of hardware floating-point arithmetic to calculate successive approximations using Newton's method. You can use Newton's method to find numerical solutions to equations. As *Example: Creating a Newton Iteration (page 258)* describes, if x_n is an approximate solution to the equation $f(x) = 0$, then x_{n+1}, given by the following formula, is typically a better approximation.

$$x_{(n+1)} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The procedure **iterate** takes a function, **f**, its derivative, **df**, and an initial approximate solution, **x0**, as input to the equation $f(x) = 0$. The procedure **iterate** calculates at most **N** successive Newton iterations until the difference between the new approximation and the previous one is small. The procedure prints the sequence of approximations to show successive approximations.

```
> iterate := proc( f::procedure, df::procedure,
                   x0::numeric, N::posint, $ )
    local xold, xnew;
    xold := x0;
    xnew := evalf( xold - f(xold)/df(xold) );
    to  N-1 while abs(xnew-xold) > 10^(1-Digits) do
       xold := xnew;
       print(xold);
       xnew := evalf( xold - f(xold)/df(xold) );
    end do;
    return xnew;
end proc:
```

The following procedure calculates the derivative of **f** and passes all the necessary information to **iterate**.

```
> Newton := proc( f::procedure, x0::numeric, N::posint:=15, $ )
    local df;
    df := D(f);
    print(x0);
    return iterate(f, df, x0, N);
  end proc:
```

Use **Newton** to solve the equation $x^2 - 2 = 0$.

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

$$f := x \to x^2 - 2 \tag{7.132}$$

```
> Newton(f, 1.5);
```

$$1.5$$
$$1.416666667$$
$$1.414215686$$
$$1.414213562$$
$$1.414213562 \tag{7.133}$$

This version of **Newton** uses sfloats unless the arguments passed in are hfloats. If you add option hfloat to the procedure **iterate**, then hfloats are used automatically, provided the value of Digits is small enough.

```
> iterate := proc( f::procedure, df::procedure,
                   x0::numeric, N::posint, $ )
    option hfloat;
    local xold, xnew;
    xold := 1. * x0;
    xnew := 1. * evalf( xold - f(xold)/df(xold) );
    to  N-1 while abs(xnew-xold) > 10^(1-Digits) do
       xold := xnew;
       print(xold);
       xnew := evalf( xold - f(xold)/df(xold) );
    end do;
    return xnew;
  end proc:
```

Now the procedure **Newton** will return **hfloats** instead of **sfloats** when Digits is less than 15.

```
> type( Newton(f, 1.5), hfloat);
```

$$1.5$$

$$1.41666666666667$$

$$1.41421568627451$$

$$1.41421356237469$$

$$\mathit{true} \tag{7.134}$$

In this case, the procedure is simple enough that we can go beyond option hfloat and use the evalhf command to achieve best performance. This next version of **Newton** uses evalhf for floating-point arithmetic if possible and reverts to sfloats otherwise. Since **iterate** only tries to find a solution to an accuracy of **10^(1-Digits)**, **Newton** uses **evalf** to round the result of the hardware floating-point computation to an appropriate number of digits.

```
> Newton := proc( f::procedure, x0::numeric, N::posint:=15, $ )
    local df, result;
    df := D(f);
    print(x0);
    if Digits < evalhf(Digits) then
       try
          return evalf( SFloat( evalhf(iterate(f, df, x0, N)) ));
       catch:
       end try;
    end if;
    return evalf( SFloat( iterate(f, df, x0, N) ) );
  end proc:
```

Newton uses hardware floating-point arithmetic for the iterations and rounds the result to software precision. Hardware floating-point numbers have more digits than the software floating-point numbers, given the present setting of **Digits**.

```
> Newton(f, 1.5);
```

$$1.5$$

$$1.41666666666666674$$

$$1.41421568627450989$$

$$1.41421356237468987$$

$$1.41421356237309515$$

$$1.414213562 \tag{7.135}$$

Newton must use software floating-point arithmetic to find a root of the following Bessel function.

```
> F := z -> BesselJ(1, z);
```

$$F := z \rightarrow \mathrm{BesselJ}(1, z) \tag{7.136}$$

```
> Newton(F, 4);
```

$$4$$

$$3.82649352308792$$

$$3.83170246715760$$

$$3.83170597020591$$

$$3.831705970 \tag{7.137}$$

Software arithmetic is used because **evalhf** does not recognize **BesselJ** and the symbolic code for **BesselJ** uses the **type** command and remember tables, which **evalhf** does not allow.

```
> evalhf( BesselJ(1, 4) );
```

```
Error, unsupported type `('complex')('float')` in evalhf
```

Using a **try-catch** block (as in the previous **Newton** procedure) allows the procedure to work when **evalhf** fails.

The previous **Newton** procedure prints many digits when it is trying to find a ten-digit approximation. The reason is that the **print** command is located inside the **iterate** procedure, which is inside a call to **evalhf**, where all numbers are hardware floating-point numbers, and print as such.

Example: Jacobi Iteration

Jacobi iteration is an iterative method for numerically solving systems of linear equations that are *diagonally dominant* (meaning that the diagonal elements of the matrix representing the system are larger than the sum of all other elements in any given row of the matrix). Given a initial guess of $x0$ for the solution to $A \,.\, x = b$, the process is: if x_k is an approximation for the solution, then the next approximation is $x_{k+1} = S^{-1} \,.\, \left(b - R \,.\, x_k\right)$ where S is the diagonal of A and $A = S + R$.

The procedure **Jacobi** is a straight forward implementation of Jacobi iteration as it is usually presented in a numerical analysis course.

```
> Jacobi := proc(A::Matrix(numeric), b::Vector(numeric),
  x0::Vector(numeric):=b, MAXIter::posint:=25,
  tolerance::positive:=evalf(LinearAlgebra:-Norm(b,2)*10^(1-Digits)),
   $)
  local i,j,k, x_old, x_new, s, residual, n;

     x_new := evalf(x0);

     n := LinearAlgebra:-Dimension(b);
     x_old := Vector(n, 0, rtable_options(x_new));

     residual := evalf(LinearAlgebra:-Norm(A . x_new-b,2));

     for k to MAXIter while residual > tolerance do
         ArrayTools:-Copy(x_new, x_old);
         for i from 1 to n do
             s := 0;
             for j from 1 to n do
                 if i<>j then
                     s := s + A[i,j] * x_old[j];
                 end if;
             end do;
             x_new[i] := (b[i] - s) / A[i,i];
         end do;

         residual := evalf(LinearAlgebra:-Norm(A . x_new-b,2));
     end do;

     if k < MAXIter then
         return x_new;
```

```
    else
        WARNING("Residual %1 greater than tolerance %2 after %3
  iterations", residual, tolerance, k-1);

        return x_new;
    end if;
  end proc:
```

Here we construct a random Matrix that is strongly diagonally dominant to test the procedure. Note that, while in practice Jacobi iteration would not be used on dense Matrices, we use dense Matrices in these examples to illustrate some efficiency principles.

```
> N := 25:
```

```
> M := Matrix(N,N,(i,j)->`if`(i<>j,
  RandomTools:-Generate(integer(range=-100..100))/1000.,
  RandomTools:-Generate(integer(range=100..10000))/10.),datatype=float);
```

$$M := \begin{bmatrix} 25 \text{ x } 25 \text{ Matrix} \\ \text{Data Type: } float_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.138}$$

```
> b := LinearAlgebra:-RandomVector(N,datatype=float);
```

$$b := \begin{bmatrix} 1 \text{ .. } 25 \text{ Vector}_{column} \\ \text{Data Type: } float_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.139}$$

```
> CodeTools:-Usage( Jacobi(M, b) );
```

```
memory used=0.61MiB, alloc change=0 bytes, cpu time=32.00ms, real
time=88.00ms
```

$$\begin{bmatrix} 1 \text{ .. } 25 \text{ Vector}_{column} \\ \text{Data Type: } float_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.140}$$

The code is written in such a way that it will automatically work for software floats at higher values of digits.

```
> Digits := 50:
> M := Matrix(N,N,(i,j)->`if`(i<>j,
  RandomTools:-Generate(integer(range=-100..100))/1000.,
  RandomTools:-Generate(integer(range=100..10000))/10.),datatype=float);
```

$$M := \begin{bmatrix} 25 \text{ x } 25 \text{ Matrix} \\ \text{Data Type: sfloat} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.141}$$

```
> b := LinearAlgebra:-RandomVector(N,datatype=float);
```

$$b := \begin{bmatrix} 1 \text{ .. } 25 \text{ Vector}_{\text{column}} \\ \text{Data Type: sfloat} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.142}$$

```
> CodeTools:-Usage( Jacobi(M, b) );
memory used=11.12MiB, alloc change=0 bytes, cpu time=136.00ms, real
time=201.00ms
```

$$\begin{bmatrix} 1 \text{ .. } 25 \text{ Vector}_{\text{column}} \\ \text{Data Type: sfloat} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.143}$$

This implementation works well for small Matrices, but when the dimension becomes large, it becomes very slow.

```
> Digits := 10:
> N := 500:
```

```
> M := Matrix(N,N,(i,j)->`if`(i<>j,
  RandomTools:-Generate(integer(range=-100..100))/1000.,
  RandomTools:-Generate(integer(range=100..10000))/10.),datatype=float);
```

$$M := \begin{bmatrix} 500 \text{ x } 500 \text{ Matrix} \\ \text{Data Type: float}_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.144}$$

```
> b := LinearAlgebra:-RandomVector(N,datatype=float);
```

$$b := \begin{bmatrix} 1 \text{ .. } 500 \text{ Vector}_{\text{column}} \\ \text{Data Type: float}_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.145}$$

```
> CodeTools:-Usage( Jacobi(M, b) );
```

```
memory used=130.01MiB, alloc change=12.01MiB, cpu time=3.17s, real
time=3.17s
```

$$\begin{bmatrix} 1 \text{ .. } 500 \text{ Vector}_{\text{column}} \\ \text{Data Type: float}_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix} \tag{7.146}$$

Adding option hfloat to Jacobi is not likely to increase performance, since hfloat contagion from the float[8] Matrix elements means that hfloat arithmetic is likely being used everywhere possible already. However, it is possible to rewrite the internal loops as a procedure that can be evaluated with evalhf. (It might be possible to rewrite all of **Jacobi** to be evaluatable to evalhf, but it would be difficult and potential gains would be modest.)

```
> JacobiHelper := proc(A, b, x_old, x_new, n)
  local s, i, j, l;
  option hfloat;
  # this procedure acts by side effect on x_new
        for i from 1 to n do
            s := 0;
            for j from 1 to n do
                if i<>j then
```

```
                    s := s + A[i,j] * x_old[j];
                end if;
            end do;
            x_new[i] := (b[i] - s) / A[i,i];
        end do;
    end proc:
```

And the rest of the procedure with option hfloat.

```
> Jacobi := proc(A::Matrix(numeric), b::Vector(numeric),
  x0::Vector(numeric):=b, MAXIter::posint:=25,
  tolerance::positive:=evalf(LinearAlgebra:-Norm(b,2)*10^(1-Digits)),
   $)
  option hfloat;
  local i,j,k, x_old, x_new, s, residual, n;

     x_new := evalf(x0);

     n := LinearAlgebra:-Dimension(b);
     x_old := Vector(n, 0, rtable_options(x_new));

     residual := evalf(LinearAlgebra:-Norm(A . x_new-b,2));

     for k to MAXIter while residual > tolerance do
         ArrayTools:-Copy(x_new, x_old);
         # JacobiHelper acts by side effect on x_new
         if Digits <= evalhf(Digits) then
              evalhf( JacobiHelper(A, b, x_old, x_new, n) );
         else
              ( JacobiHelper(A, b, x_old, x_new, n) );
         end if;

         residual := evalf(LinearAlgebra:-Norm(A . x_new-b,2));
     end do;

     if k < MAXIter then
         return x_new;
     else
         WARNING("Residual %1 greater than tolerance %2 after %3
  iterations", residual, tolerance, k-1);

         return x_new;
```

```
    end if;
  end proc:
> CodeTools:-Usage( Jacobi(M, b) );
memory used=3.98MiB, alloc change=0 bytes, cpu time=472.00ms, real
time=473.00ms
```

$$\begin{bmatrix} 1 .. 500\ Vector_{column} \\ Data\ Type: float_8 \\ Storage: rectangular \\ Order: Fortran\_order \end{bmatrix} \tag{7.147}$$

Using evalhf here achieves an impressive speed-up but you can achieve even better speed by taking advantage of the built-in Matrix and Vector operations. In general you code will be faster if you can replace nested loops with calls to external commands for Vectors or Matrices. Those commands will be highly optimized for your platform taking advantage of multiple cores and cache hierarchy where possible.

```
> Jacobi := proc(A::Matrix(numeric), b::Vector(numeric),
  x0::Vector(numeric):=b, MAXIter::posint:=25,
  tolerance::positive:=evalf(LinearAlgebra:-Norm(b,2)*10^(1-Digits)),
   $)
  local k, x_new, S, S_inv, residual;

     x_new := evalf(x0);
     S := LinearAlgebra:-Diagonal(A,datatype=float);
     S_inv := 1 /~ S;

     residual := evalf(LinearAlgebra:-Norm(A.x_new-b,2));

     for k to MAXIter while residual > tolerance do
         # computing R.x as A.x - S.x is probably a bad idea
  numerically
         # but we do it anyway to avoid making the code overly
  complicated
         x_new := S_inv *~ (b - ( A . x_new - S *~ x_new));
         residual := evalf(LinearAlgebra:-Norm(A . x_new-b,2));
     end do;

     if k < MAXIter then
         return x_new;
     else
```

```
        WARNING("Achieved tolerance of only %1 after %2 iterations",
    residual, i-1);
          return x_new;
      end if;

  end proc:
> CodeTools:-Usage( Jacobi(M, b) );
memory used=4.13MiB, alloc change=0 bytes, cpu time=60.00ms, real
time=68.00ms
```

$$\begin{bmatrix} 1 \,..\, 500\ Vector_{column} \\ Data\ Type: float_8 \\ Storage: rectangular \\ Order: Fortran\_order \end{bmatrix} \tag{7.148}$$

This sort of speed-up is typical. The built-in numerical linear algebra commands are easily an order of magnitude faster than loops run in Maple, and generally also faster than loops in **evalhf**.

8 Programming with Modules

This chapter describes the structure and flexibility of Maple modules.

Modules allow you to associate related procedures and data in one structure. By creating a module, you can write code that can be reused, transported, and easily maintained. You can also use modules to implement objects in Maple.

This chapter provides several example modules, many of which are available as Maple source code in the **samples** directory of your Maple installation. You can load these examples into the Maple library to modify and extend them, and use them in custom programs.

8.1 In This Chapter

- Syntax and semantics
- Using modules as records or structures
- Modules and use statements
- Interfaces and implementations

8.2 Introduction

You may decide to create a module for one of the purposes described below.

Encapsulation

Encapsulation is the act of grouping code together in one structure to separate its interface from its implementation. By doing so, you can create applications that are transportable and reusable and that offer well-defined user interfaces. This makes your code easier to maintain and understand--important properties for large software systems.

Creating a Custom Maple Package

A *package* is a means of bundling a collection of Maple procedures related to a domain. Most of the Maple library functionality is available in packages.

Creating Objects

Objects can be represented using modules. In software engineering or object-oriented programming, an object is defined as an element that has both a state and behavior. Objects are passed the same way as ordinary expressions, but also provide methods which define their properties.

Creating Generic Programs

Generic programs accept objects with specific properties or behaviors. The underlying representation of the object is transparent to generic programs. For example, a generic geometry program can accept any object that exports an **area** method, in addition to other objects. The framework of the program would rely on information in each given object to determine specific behaviors, while the overall program implements a common pattern between the objects.

8.3 A Simple Example

In the following example, a module generates a sequence of numbers.

```
> Counter := module()
      description "number generator";
      export    getNext;
      local    count;

      count := 0;

      getNext := proc()
          count := 1 + count;
      end proc;

  end module:
  Counter:-getNext();
  Counter:-getNext();
  Counter:-getNext();
```

$$1$$

$$2$$

$$3 \tag{8.1}$$

The *module definition* format, which will be described in more detail in the next section, is similar to a procedure definition in that the body is contained within a delimited code block. Also, elements such as local variables, options, and description are declared at the top of the module. Unlike a procedure, the body of the module is evaluated only once when it is declared. The values that are defined during this evaluation process, and the values that are defined in subsequent usage of the module, are stored and can be used again.

In a module definition, you can declare *exported variables*, which are names that will be made available once the module has been run. These exported variables can be accessed by using the member selection operator (:-) or the indexing operation (**[]**) , while *local variables*

remain private (that is, they are accessible only by methods within the module). The example above declares and uses one exported local variable called **getNext** and one local variable called **count**.

8.4 Syntax and Semantics

Module definitions have the following general syntax.

```
module()
     local L;
     export E;
     global G;
     options O;
     description D;
     B
 end module
```

The Module Definition

All module definitions begin with the keyword **module**, followed by an empty pair of parentheses. This is similar to the parentheses that follow the **proc** keyword in a procedure definition. Following that is an optional declaration section and the module body. The keywords **end module** (or simply **end**) terminate a module definition.

The simplest valid module definition is

```
> module() end;
```

$$\mathbf{module}(\,)\quad \mathbf{end\ module} \tag{8.2}$$

which does *not* contain exported variables, local variables, references, global variables, or a body of statements.

The Module Body

The body of a module definition contains the following components:

- Zero or more Maple statements. The body is executed when the module definition is evaluated, producing a module.
- Assignment statements that assign values to the exported names of the module.

Also, the body can optionally contain the following components:

- Assignments to local variables and arbitrary computations.
- A **return** statement, which cannot contain a **break** or **next** statement outside of a loop. Running a **return** statement terminates the execution of the body of the module definition.

Module Parameters

Unlike procedures, module definitions do not have explicit parameters because modules are not called (or invoked) with arguments.

Implicit Parameters

All module definitions have an implicit parameter called **thismodule**. Within the body of a module definition, this special name evaluates to the module in which it occurs. You can, therefore, refer to a module within its own definition before the result of evaluating it has been assigned to a name.

thismodule is similar to **thisproc** in procedures, but is not the same as **procname**. The difference between **thismodule** and **procname** is that **procname** evaluates to a *name*, while **thismodule** evaluates to the module expression itself. There is no concept of a **modulename** implicit variable because the invocation phase of evaluating a module definition is part of its normal evaluation process, and it occurs immediately. Procedures, on the other hand, are not invoked until they are called with arguments. Normally, at least one name for a procedure is known by the time it is called; this is not the case for modules.

Implicit parameters related to passing arguments (for example, **_params**, **_options**, **_passed**, and others) *cannot* be referenced in module definitions. They are only available within the scope of a procedure.

For more information on procedures, see *Procedures (page 195)*.

Named Modules

In a module definition, an optional symbol can be specified after the **module** keyword. Modules created in this way are called *named modules*.

Semantically, named modules are almost identical to normal modules, but the exported variables of named modules are printed differently, allowing the module from which it was exported to be identified visually. In the following example, a normal module is assigned to the name **NormalModule**.

```
> NormalModule := module() export e; end module;
  NormalModule:-e;
```

$$NormalModule := \mathbf{module}(\,)\ \mathbf{export}\ e;\ \ \mathbf{end\ module}$$

$$e \tag{8.3}$$

In the following example, the symbol (the name of the module) after the **module** keyword is **NamedModule**.

```
> module NamedModule() export e; end module;
```

$$\textbf{module}\ \mathit{NamedModule}(\,)\ \textbf{export}\ \mathit{e};\ \ \textbf{end module} \tag{8.4}$$

```
> NamedModule:-e;
```

$$\mathit{NamedModule}\text{:-}\mathit{e} \tag{8.5}$$

When the definition of a named module is evaluated, the name (which appears immediately after the **module** keyword) is assigned the module as its value and the name is protected (that is, it cannot be modified). Therefore, a named module is usually created only once. For example, an error occurs when the same named module definition above is executed.

```
> module NamedModule() export e; end module;
```

```
Error, (in NamedModule) attempting to assign to `NamedModule` which
is protected.  Try declaring `local NamedModule`; see ?protect for
details.
```

Executing the normal module definition again creates a *new* instance of the module and does not result in an error. It simply reassigns the variable **NormalModule** to the new module instance.

```
> NormalModule := module() export e; end module;
```

$$\mathit{NormalModule} := \textbf{module}(\,)\ \textbf{export}\ \mathit{e};\ \ \textbf{end module} \tag{8.6}$$

If you save a normal module to a Maple library archive, which is a file used to store a collection of internal files, the normal module becomes a named module the next time it is loaded from the library archive. The savelib command, which is the command used to save a file to a library archive, takes the name of the variable assigned a module, and saving the file associates this name with the module.

For more information about library archive files, see *Writing Packages (page 383)*.

Important: Do not assign a named module to another variable, for example,

```
> SomeName := eval( NamedModule );
```

$$\begin{aligned}&\mathit{SomeName} := \textbf{module}\ \mathit{NamedModule}(\,)\\ &\quad\textbf{export}\ \mathit{e};\\ &\textbf{end module}\end{aligned} \tag{8.7}$$

```
> SomeName:-e;
```

$$\mathit{NamedModule}\text{:-}\mathit{e} \tag{8.8}$$

Exports of named modules are printed using the *distinguished* name that was given to the module when it was created, regardless of whether it has been assigned to another name.

Whether a module has a name also affects the reporting of errors that occur during its evaluation. When the second attempt to evaluate the named module definition above generated an error, the error message reported the location of the error by name. In contrast, when an error occurs during the evaluation of a normal module definition, the name **unknown** is used instead.

```
> NormalModule := module() export e; error "oops"; end module;
```

```
Error, (in unknown) oops
```

This process differs from procedure error reporting. Maple cannot report the name of a normal module (that is, the name of the variable to which the module is assigned) because the evaluation of the right-hand side of an assignment occurs before the assignment to the name takes place. Therefore, the error occurs before the association between a variable and the module has occurred.

Declarations

The declarations section of the module must appear immediately after the parentheses. All of the statements in the declarations section are optional, but, at most, one of each kind can be specified. Most module declarations are the same as those for procedures.

For more information, see *Parameter Declarations (page 197)*.

Description Strings

You can provide a brief description to summarize the purpose and function of your modules. Providing a description is valuable to other users who read your code. Include text after the **description** keyword as you would in a procedure definition.

```
> Hello := module()
      description "my first module";
      export say;
      say := proc()
          print( "HELLO WORLD" )
      end proc;
  end module:
```

When the module is printed, its description string is displayed.

```
> eval( Hello );
```

```
module( )
    export say;
    description "my first module";

end module
```

(8.9)

The **export** declaration is described later in this chapter.

Global Variables

Global variables referenced in a module definition are declared by using the **global** keyword. Following the **global** keyword is a sequence of one or more symbols, which are associated with their global module instances. In certain cases, you must declare a name as a global variable to prevent implicit scoping rules from making it local.

```
> Hello := module()
      export say;
      global message;
      say := proc()
           message := "HELLO WORLD!"
      end proc;
  end module:
> message;
```

message (8.10)

```
> Hello:-say();
```

"HELLO WORLD!" (8.11)

```
> message;
```

"HELLO WORLD!" (8.12)

Local Variables

You can define variables that are local to the module definition by using the **local** declaration. Its format is the same as for procedures. The following example is a variant of the previous **Hello** module, which uses a local variable.

```
> Hello := module()
      local loc;
      export say;
      loc := "HELLO WORLD!";
```

```
    say := proc()
        print( loc )
    end proc;
end module:
```

Local variables (or *locals*) cannot be used or changed outside of the module definition in which they occur. In other words, they are private to the module.

A local variable in a module is a distinct object from a global variable with the same name. While local variables in procedures are typically used only for the duration of the execution time of the procedure body, module local variables are stored after the module definition is executed. They can be used to maintain a state. For example, in the Counter example described at the beginning of this chapter, a local **count** variable stores the current value of the counter. The **count** local variable increments each time the **getNext** procedure is invoked. Its new value is stored and can be used the next time the procedure is called. At the same time, because **count** is local, no external programs can change its value and end the sequence defined by the module.

Exported Local Variables

Procedures and modules both support local variables. However, only modules support *exported* local variables, which are often called *exports*.

Module exports are declared by using the **export** declaration. It begins with the keyword **export**, followed by a (nonempty) sequence of symbols. A name is never exported implicitly; exports *must* be declared.

The result of evaluating a module definition is a module. You can view a module as a collection of its exports, which are also referred to as *members* of the module. These are simply names that can (but need not) be assigned values. You can establish initial values for the exports by assigning values to them in the body of the module definition.

The word export is a short form for *exported local variable*. In most cases, a module export is a local variable such as those declared with the **local** declaration. The difference is that you can access the exported local variables of a module after it has been created.

To access an export of a module, use the member selection operator (**:-**). Its general syntax is

```
modexpr :- membername
```

modexpr must be an expression that evaluates to a module and **membername** must be the name of an export of the module to which **modexpr** evaluates. Anything else signals an exception. You cannot access the local variables of an instantiated module by using this syntax.

The **Hello** example above has one export named **say**. In the following example, **say** is assigned a procedure. To call it, enter

```
> Hello:-say();
```

"HELLO WORLD!" (8.13)

The following expression raises an exception because the name **noSuchModule** is not assigned a module expression.

```
> noSuchModule:-e;
```

```
Error, `noSuchModule` does not evaluate to a module
```

In the following example, a module expression is assigned to the name **m** and the member selection expression **m:-e** evaluates to the value of the exported variable **e** of **m**.

```
> m := module() export e; e := 2 end module:
  m:-e;
```

2 (8.14)

Since **m** does not export a variable named **noSuchExport**, the following expression raises an exception.

```
> m:-noSuchExport;
```

```
Error, module does not export `noSuchExport`
```

In addition to the :- syntax, square brackets can also be used to reference a module export.

```
> m[e];
```

2 (8.15)

The square bracket notation has different evaluation rules than member selection. When using the member selection operator (:-), the export name must be known in advance. When using [], the name of the export can be computed. In this example, an exported variables value can be selected from an arbitrary module.

```
> m := module() export a := 1, b := 2, c := 3; end module:
  FirstExport := proc( m::`module` ) local ex := exports(m); return
   m[ex[1]]; end proc;
  FirstExport(m);
```

```
FirstExport := proc(m::module)
    local ex;
    ex := exports(m); return m[ex[1]]
end proc
```

1 (8.16)

Important: Exports do not need to have assigned values. The following module exports an unassigned name. This illustrates the importance of distinguishing module exports from global variables.

```
> m := module() export e; end module:
```

References to the exported name **e** in **m** evaluate to the name **e**.

```
> m:-e;
```

e (8.17)

Note, however, that this is a *local* name **e** and not the global instance of the name.

```
> evalb( e = m:-e );
```

false (8.18)

The first **e** in the previous expression refers to the global **e**, while the expression **m:-e** evaluates to the **e** that is local to the module **m**. This distinction between a global and export of the same name is useful. For example, you can create a module with an export sin. Assigning a value to the export sin does not affect the protected global name sin.

Determining the Export Names

You can determine the names of the exports of a module by using the exports procedure.

```
> exports( Hello );
```

say (8.19)

```
> exports( NormalModule );
```

e (8.20)

This procedure returns the *global* instances of the export names.

```
> exports( m );
```

e (8.21)

```
> evalb( (8.21) = e );
```

true (8.22)

You can also obtain the local instances of those names by using the option **instance**.

```
> exports( m, 'instance' );
```

e (8.23)

```
> evalb( (8.23) = e );
```

false (8.24)

```
> evalb( (8.23) = m:-e );
```

true (8.25)

You cannot have the same name declared as both a local variable and an exported local variable.

```
> module() export e; local e; end module;
```

```
Error, export and local `e` have the same name
```

(The declared exports and locals actually form a partition of the names that are local to a module.)

Testing for Membership in a Module

As described in previous chapters, the member command can be used to test whether a value is a member of a set or list.

```
> member( 4, { 1, 2, 3 } );
```

false (8.26)

This command can also be used for membership tests in modules.

```
> member( say, Hello );
```

true (8.27)

```
> member( cry, Hello );
```

false (8.28)

The first argument is a global name whose membership is to be tested, and the second argument is the name of a module. The member command returns the value **true** if the module has an export whose name is the same as the first argument.

The member command also has a three-argument form that can be used with lists to determine the first position at which an item occurs.

```
> member( b, [ a, b, c ], 'pos' );
```

true (8.29)

The name **pos** is now assigned the value **2** because **b** occurs at the second position of the list. **[a, b, c]**.

```
> pos;
```

2 (8.30)

When used with modules, the third argument is assigned the *local instance* of the name whose membership is being tested, provided that the return value is **true**.

```
> member( say, Hello, 'which' );
```

true (8.31)

```
> which;
```

say (8.32)

```
> eval( which );
```

proc( ) *print*(*loc*) **end proc** (8.33)

If the return value from the member command is **false**, the name remains unassigned or maintains its previously assigned value.

```
> unassign( 'which' ):
> member( cry, Hello, 'which' );
```

false (8.34)

```
> eval( which );
```

which (8.35)

Module Options

Similar to procedures, a module definition can contain options. The options available for modules are different from those for procedures. Only the options **trace** and **copyright** are common to both procedures and modules. The following four options have a predefined meaning for modules: load, unload, package, and record. The load and unload options cover functionality defined by the ModuleLoad and ModuleUnload special exports described in the next section.

For more information, refer to the module,option help page.

The package Option

A *package* is a collection of procedures and other data that can be treated as a whole. Packages typically gather several procedures that allow you to perform computations in a well-defined problem domain. Packages can contain data other than procedures and can even contain other packages (subpackages).

The package option is used to designate a module as a Maple package. The exports of a module created with the package option are automatically protected.

For more information, see *Writing Packages (page 383)*.

The record Option

The record option is used to identify records, which are fixed-size collections of items. Records are created by using the Record constructor and are represented using modules.

For more information, see *Records (page 329)*.

Special Exports

Certain specially named exports, when defined in a module, affect how modules behave in Maple. These special exports are described below. In most cases, they can be declared as either exported local variables or local variables.

The ModuleApply Procedure

When a procedure named ModuleApply is declared as an export or local of a module, the module name can be used as if it were a procedure name.

Consider the Counter example described at the beginning of this chapter. Since it only has one method, the calling sequence can be shortened by using the ModuleApply function.

```
> Counter := module()
      export ModuleApply;
      local count;

      count := 0;

      ModuleApply := proc()
          count := 1 + count;
      end proc;
  end module:
  Counter();
```

```
Counter();
Counter();
```

$$1$$

$$2$$

$$3 \tag{8.36}$$

In this example, calls to Counter:-ModuleApply() are not needed and the results are the same as those generated by the original Counter example. The ModuleApply function can specify and accept any number of parameters.

You can also use the ModuleApply function to create module *factories*, a standard object-oriented design pattern described later in this chapter.

The ModuleIterator Procedure

The ModuleIterator procedure defines how a module functions when it is used as the **in** clause of a **for** loop.

```
> for e in myModule do
      # Do something with e
  end do;
```

In the example below, the ModuleIterator procedure returns two procedures: **hasNext** and **getNext**. These procedures can have any names, and in fact, do not require names. When the ModuleIterator procedure is called, an iterator is initialized for the instance, the details of which are kept hidden from the caller. The two returned procedures can then be used to iterate over the instance to perform a specific task. For example, consider a class that implements a form of a set of which **mySet** is an instance. You can iterate over this set as follows:

```
> (hasNext,getNext) := ModuleIterator(mySet);
   while hasNext() do
       e := getNext();
       # Do something with e.
   end do;
```

The example above is an explicit use of the ModuleIterator procedure. However, this mechanism is also used implicitly by the Maple for-loop construct,

The hasNext procedure returns a value of **true** or **false** depending on whether remaining elements need to be processed. Successive calls to hasNext with no intervening calls to getNext return the same result. The getNext procedure returns the next element to process, and increments the iterator. These procedures should be implemented so that it is always safe to call getNext after the most recent call to hasNext returns a value of **true**. The result

of calling getNext after hasNext has returned a value of **false**, or before hasNext has ever been called, is up to the implementer of the class.

The Counter example already contains a **getNext** procedure. A *bounded count* should be added so that the iteration can terminate by using **lower** and **upper** exported values. A statement also needs to be added to determine if any numbers are left in the sequence. The **hasNext** and **getNext** procedures are returned directly by **ModuleIterator**.

```
> Counter := module()
      export getNext, ModuleIterator, lower := 0, upper := 5;
      local hasNext, count := 0;

      hasNext := proc()
          evalb( count < upper );
      end proc;

      getNext := proc()
          count := 1 + count;
          return count;
      end proc;

      ModuleIterator := proc()
          return hasNext, getNext;
      end proc;
  end module;

  for e in Counter do
     e;
  end do;
```

Counter := **module**( )
local *hasNext*, *count*;
export *getNext*, *ModuleIterator*, *lower*, *upper*;

end module

5 (8.37)

When the module iterator is used by the seq, add, or mul commands, Maple first checks if the module is an object that exports the numelems command. If so, it will call the numelems command to determine how large to preallocate the result, and the hasNext and getNext procedures will return exactly that many elements. If the module does not export a numelems

method, Maple will increase the result, which will consume more space (as intermediate results are discarded) and time (garbage collection).

The ModuleLoad Procedure

The ModuleLoad procedure is executed automatically when a module is loaded from the Maple library archive in which it has been saved. In a normal session, initialization code can be included in the module body. When loading a saved module, extra initialization code is sometimes required to set up run-time properties for the module. For example, a module that loads procedures from a dynamic-link library (.dll) file may need to call the **define_external** function during the initialization process. For more information on the **define_external** function, see *???*.

Consider the Counter example at the beginning of the chapter. The **count** index can have any value when it is saved. The next time you use it, you might want to reset the count to zero so that it is ready to start a new sequence. This can be done by using the ModuleLoad procedure.

```
> Counter := module()
      export getNext, ModuleLoad;
      local count;

      ModuleLoad := proc()
          count := 0;
      end proc;
      ModuleLoad();

      getNext := proc()
          count := 1 + count;
      end proc;
  end module:
  Counter:-getNext();
```

$$1 \tag{8.38}$$

Note that the initialization code is contained within the ModuleLoad procedure. After that, the ModuleLoad procedure is also called. By defining the module in this way, you will get the same results when executing the module definition as you would when loading a saved module from a library archive.

The results of ModuleLoad can be duplicated using a procedure with a different name by using the load=pname option in the option sequence of the module.

ModulePrint

If a module has an export or local named ModulePrint, the result of the ModulePrint command is displayed instead of the module when a command in that module is executed.

The ModulePrint procedure does not display output. Instead, it returns a standard Maple expression that will be displayed. The expression returned can be customized to another object that portrays or summarizes the module.

In the following example, the Counter example will be extended from the ModuleIterator example to display a summary of what the module does.

```
> Counter := module()
      export ModuleIterator, getNext, lower := 0, upper := 5;
      local ModulePrint, hasNext, count := 0;

      hasNext := proc()
          evalb( count < upper );
      end proc;

      getNext := proc()
          count := 1 + count;
          return count ;
      end proc;

      ModuleIterator := proc()
          return hasNext, getNext;
      end proc;

      ModulePrint := proc()
          return [[ sprintf("Counter from %d to %d", lower, upper)
   ]];
      end proc;
  end module;
```

$$Counter := [["Counter\ from\ 0\ to\ 5"]] \tag{8.39}$$

ModuleUnload

The ModuleUnload procedure is called immediately before a module is discarded. A module is discarded either when it is no longer accessible and is garbage collected, or when you end your Maple session.

```
> M := module()
     export ModuleUnload;
```

```
   ModuleUnload := proc() print("I am gone"); end proc;
end module:
unassign(M);
1;2;3;4; gc();
```

$$1$$

$$2$$

$$3$$

$$4$$

$$\text{"I am gone"} \tag{8.40}$$

You may not see the "I am gone" message after executing the code above because several factors determine exactly when memory is free to be garbage collected. At a minimum, no references can be left in the module. It must not be assigned or contained in any other live expression. This includes the ditto operators and the list of display reference handles (that is, the undo/redo buffer of the GUI). Also, it must not be identified as being alive by the garbage collector (i.e. a reference to the module is not found by the collector).

A module can become inaccessible, and therefore subject to garbage collection before the unload= procedure is executed, but can then become accessible again when that procedure is executed. In that case, the module is *not* garbage collected. When it eventually is garbage collected, or if you end your Maple session, the unload= procedure is *not* executed again.

The behavior of ModuleUnload can be duplicated using a procedure with a different name by using the unload=pname option in the option sequence of the module.

Implicit Scoping Rules

The bindings of names that appear within a module definition are determined when the module definition is simplified. Module definitions are subject to the same implicit scoping rules that apply to procedure definitions. Under no circumstances is a name ever implicitly determined to be exported by a module; implicitly scoped names can resolve only to non-exported local variables or global names.

Lexical Scoping Rules

Module definitions, along with procedure definitions, follow standard lexical scoping rules.

Modules can be nested, in the sense that a module can have any of its exports assigned to a module whose definition occurs within the body of the outer module.

Here is a simple example of a submodule.

```
> m := module()
      export s;
      s := module()
          export e;
          e := proc()
              print( "HELLO WORLD!" )
          end proc;
      end module
  end module:
```

The global name **m** is assigned a module that exports the name **s**. Within the body of **m**, the export **s** is assigned a module that exports the name **e**. As such, **s** is a *submodule* of **m**. The **Shapes** package, which is described in *Writing Packages (page 383)*, illustrates the use of submodules.

Modules and procedures can both be nested to an arbitrary depth. The rules for the accessibility of local variables (including exported locals of modules) and procedure parameters are the same as the rules for nested procedures.

Module Factory

The Counter example used up to this point would be more useful if you could have many Counter modules running at the same time, and if they could be specialized according to specified bounds. Modules do not take explicit parameters, but you can write a generic module that could be specialized by using the *factory* design pattern.

To do this, write a *constructor* procedure for the module that accepts the lower and upper bound values as arguments. The following module creates a Counter.

```
> MakeCounter := proc( lower::integer, upper::integer )
      return module()
          export ModuleIterator, getNext;
          local ModulePrint, hasNext, count := lower;

          hasNext := proc()
              evalb( count < upper );
          end proc;

          getNext := proc()
              count := 1 + count;
              return count ;
          end proc;

          ModuleIterator := proc()
```

```
            return hasNext, getNext;
        end proc;

        ModulePrint := proc()
            return [[ sprintf("Counter from %d to %d", lower,
upper) ]];
        end proc;
    end module;
end proc;
c1 := MakeCounter(6,10);
c1:-getNext();
c1:-getNext();
c2 := MakeCounter(2,4);
c2:-getNext();
c1:-getNext();
```

```
MakeCounter := proc(lower::integer, upper::integer)
    return module( )
        local ModulePrint, hasNext, count;
        export ModuleIterator, getNext;
        count := lower;
        hasNext := proc( )
            evalb(count < upper)
        end proc;
        getNext := proc( )
            count := 1 + count; return count
        end proc;
        ModuleIterator := proc( )
            return hasNext, getNext
        end proc;
        ModulePrint := proc( )
            return [[sprintf("Counter from %d to %d",
            lower, upper)]]
        end proc
    end module
end proc
```

```
c1 := [["Counter from 6 to 10"]]

7

8

c2 := [["Counter from 2 to 4"]]

3

9
```

(8.41)

In the above example, two specialized Counters operate at the same time with different internal states.

Modules and Types

Two Maple types are associated with modules. First, the name **module** is a type name. Naturally, an expression is of type **module** only if it is a module. When used as a type name, the name **module** must be enclosed in name quotes (`).

```
> type( module() end module, '`module`' );
```

$$true \tag{8.42}$$

```
> type( LinearAlgebra, '`module`' );
```

$$true \tag{8.43}$$

Second, a type called **moduledefinition** identifies expressions that are module definitions. In the previous example, the module definition

```
> module() end module:
```

was evaluated before being passed to type, so the expression that was tested was not the definition, but the module to which it evaluates. You must use unevaluation quotes (') to delay the evaluation of a module definition.

```
> type( 'module() end module', 'moduledefinition' );
```

$$true \tag{8.44}$$

Other important type tests satisfied by modules are the types **atomic** and **last_name_eval**.

```
> type( module() end module, 'atomic' );
```

$$true \tag{8.45}$$

The procedure map has no effect on modules; modules passed as an argument to map remain unchanged.

```
> map( print, module() export a, b, c; end module );
```

$$\textbf{module}(\,)\ \textbf{export}\ a, b, c;\ \ \textbf{end module} \tag{8.46}$$

Modules also follow last name evaluation rules. For more information on last name evaluation rules, refer to the last_name_eval help page.

```
> m := module() end module:
  m;
  type( m, 'last_name_eval' );
```

$$m$$

$$true \tag{8.47}$$

Although the type **module** is a surface type, which checks information at the top level of your code, it acts also as a structured type. Parameters passed as arguments to the unevaluated name **module** are interpreted as export names. For example, the module

```
> m := module() export a, b; end module:
```

has the structured module type **`module`(a, b)**:

```
> type( m, '`module`( a, b )' );
```

$$true \tag{8.48}$$

It also has the type **`module`(a)**

```
> type( m, '`module`( a )' );
```

$$true \tag{8.49}$$

because any module that exports symbols **a** and **b** is a module that exports the symbol **a**.

For more information about structured types, refer to the type,structure help page.

8.5 Records

The Record command, which was introduced in *Records (page 155)*, is an example of a module factory that can help you to write reusable code. Like an Array, a record is a fixed-size collection of items but, like a table, individual items stored within the record can be referenced by a name, rather than a numeric offset. In Maple, records, which are called structures in C++, are implemented as modules.

Creating Records

To create a record, use the Record constructor. In the simplest form, it takes the field names as arguments.

```
> rec := Record( 'a', 'b', 'c' );
```

$$rec := Record(a, b, c) \tag{8.50}$$

The name **rec** is now assigned a record with fields named **a**, **b**, and **c**. You can access and assign values to these fields by using the expressions **rec:-a**, **rec:-b**, and **rec:-c**.

```
> rec:-a := 2;
```

$$a := 2 \tag{8.51}$$

```
> rec:-a;
```

$$2 \tag{8.52}$$

If unassigned, a record field evaluates to the *local* instance of the field name.

```
> rec:-b;
```

$$b \tag{8.53}$$

```
> evalb( (8.53) = b );
```

$$false \tag{8.54}$$

This is useful because the entire record can be passed as an *aggregate* data structure.

The record constructor accepts initializers for record fields. That is, you can specify an initial value for any field in a new or unassigned record by passing an equation with the field name on the left side and the initial value on the right.

```
> r := Record( 'a' = 2, 'b' = sqrt( 3 ) );
```

$$r := Record\left(a = 2, b = \sqrt{3}\right) \tag{8.55}$$

```
> r:-b;
```

$$\sqrt{3} \tag{8.56}$$

In addition, you can associate Maple types with record fields. To associate a type, use the **`::`** operator with the field name specified as the first operand.

Type assertions can be used in combination with initializers. An incompatible initializer value triggers an assertion failure when the **assertlevel** kernel option is set to 2. For more information, refer to the kernelopts help page.

```
> kernelopts( 'assertlevel' = 2 ):
> Record( a::integer = 2.3, b = 2 );
```

$$Record(a::integer = 2.3, b = 2) \tag{8.57}$$

```
> r := Record( 'a'::integer = 2, 'b'::numeric );
```

$$r := Record(a::integer = 2, b::numeric) \tag{8.58}$$

```
> r:-b := "a string";
```

$$b := \text{"a string"} \tag{8.59}$$

If the initializer for a record field is a procedure, you can use the reserved name **self** to refer to the record you are creating. This allows records to be self-referential. The name **self** is applicable only to creating records and not to modules in general. For example, you can write a complex number constructor as follows.

```
> MyComplex := ( r, i ) ->
      Record( 're' = r, 'im' = i, 'abs' = (() -> sqrt(
              self:-re^2 + self:-im^2 )) ):
> c := MyComplex( 2, 3 ):
> c:-re, c:-im, c:-abs();
```

$$2, 3, \sqrt{13} \tag{8.60}$$

Combined with *prototype-based inheritance*, described in *Object Inheritance (page 332)*, this facility makes the Record constructor a powerful tool for object-oriented programming.

Record Types

Expressions created with the Record constructor are of the type **record**.

```
> type( rec, 'record' );
```

$$\mathit{true} \tag{8.61}$$

This is a structured type that works the same way as the `**module**` type, but recognizes records specifically.

```
> r := Record( a = 2, b = "foo" ):
> type( r, 'record( a::integer, b::string )' );
```

$$\mathit{true} \tag{8.62}$$

Note: In a **record** type, the field types are used to test against the values assigned to the fields (if any), and are not related to type assertions on the field names (if any).

```
> r := Record( a::integer = 2, b::{symbol,string} = "foo" ):
> type( r, 'record( a::numeric, b::string )' );
```

$$\mathit{true} \tag{8.63}$$

Using Records to Represent Quaternions

Records can be used to implement simple aggregate data structures for which you want named access to slots. For example, four real numbers can be combined to form a quaternion and you can represent this using a record structure as follows.

```
> MakeQuaternion := proc( a, b, c, d )
      Record( 're' = a, 'i' = b, 'j' = c, 'k' = d )
  end proc:
```

```
> z := MakeQuaternion( 2, 3, 2, sqrt( 5 ) );
```

$$z := Record\left(re = 2, i = 3, j = 2, k = \sqrt{5}\right) \tag{8.64}$$

In this example, **z** represents the quaternion 2 + 3i + 2j + sqrt(5)*k (where i, j, and k are the nonreal quaternion basis units). The quaternion records can now be manipulated as single quantities. The following procedure accepts a quaternion record as its only argument and computes the Euclidean length of the quaternion that the record represents.

```
> qnorm := proc( q )
      use re = q:-re, i = q:-i, j = q:-j, k = q:-k in
          sqrt( re * re + i * i + j * j + k * k )
      end use
  end proc:
> qnorm( z );
```

$$\sqrt{22} \tag{8.65}$$

A Maple type for quaternions can be introduced as a structured record type.

```
> TypeTools:-AddType( 'quaternion', 'record( re, i, j, k )' );
> type( z, 'quaternion' );
```

$$true \tag{8.66}$$

Object Inheritance

The Record constructor supports a simple form of *prototype-based inheritance*. An object system based on prototypes does not involve classes; instead, it uses a simpler and more direct form of object-based inheritance. New objects are created from existing objects (called *prototypes*) by *cloning*, that is, by copying and augmenting the data and behavior of the prototype.

The Record constructor supports prototype-based inheritance by accepting an index argument, which is the prototype for the new object record.

```
> p := Record( a = 2, b = 3 ); # create a prototype
```

$$p := Record(a = 2, b = 3) \tag{8.67}$$

```
> p:-a, p:-b;
```

$$2, 3 \tag{8.68}$$

```
> r := Record[p]( c = 4 );
```

$$r := Record(a = 2, b = 3, c = 4) \tag{8.69}$$

```
> r:-a, r:-b, r:-c;
```

$$2, 3, 4 \tag{8.70}$$

In this example, the record **p** is the prototype, and the second record **r** inherits the fields **a** and **b**, and their values, from the prototype **p**. It also augments the fields obtained from **p** with a new field **c**. The prototype **p** is not changed.

```
> r:-a := 9;
```

$$a := 9 \tag{8.71}$$

```
> p:-a;
```

$$2 \tag{8.72}$$

Behavior, as well as data, can be copied from a prototype. To copy behavior, use a constructor procedure for both the prototype and its clones.

```
> BaseComplex := proc( r, i )
    Record( 're' = r, 'im' = i )
  end proc:
  NewComplex := proc( r, i )
    Record[BaseComplex(r,i)]( 'abs' =
      (() -> sqrt( self:-re^2 + self:-im^2 )) )
  end proc:
> c := NewComplex( 2, 3 ):
> c:-re, c:-im, c:-abs();
```

$$2, 3, \sqrt{13} \tag{8.73}$$

An object created from a prototype can serve as a prototype for another object.

```
> NewerComplex := proc( r, i )
    Record[NewComplex(r,i)]( 'arg' =
      (() -> arctan(self:-im,self:-re)) )
  end proc:
> c2 := NewerComplex( 2, 3 ):
> c2:-re, c2:-im, c2:-abs(), c2:-arg();
```

$$2, 3, \sqrt{13}, \arctan\left(\frac{3}{2}\right) \tag{8.74}$$

Note: Prototypes are supertypes of their clones.

```
> subtype( 'record( re, im, abs )', 'record( re, im )' );
```

$$true \tag{8.75}$$

For example, NewComplex creates objects of a type that is a subtype of the objects created by BaseComplex.

8.6 Modules and use Statements

The use statement is designed to complement models and to make programming with modules easier in some cases.

This section describes how the use statement can be used with modules. For more information about the **use** statement, see *The use Statement (page 189)*.

A module **m** can appear in the binding sequence of a use statement. The module is regarded as an abbreviation for the sequence of equations **a = m:-a**, **b = m:-b**, ..., where **a, b, ...** are the exports of the module **m**.

For example,

```
> m := module() export a, b; a := 2; b := 3; end module:
  use m in a + b end use;
```

$$5 \tag{8.76}$$

This is useful for programming with packages.

```
> m := Matrix( 4, 4, [[ 26, 0,   0,   30 ],
                      [ 0,  -41, -90, 0],
                      [ 0,  -7,  -56, 0 ],
                      [ 0,  0,   0,   0]] );
  use LinearAlgebra in
      Determinant( m );
      Rank( m );
      CharacteristicPolynomial( m, 'lambda' )
  end use;
```

$$m := \begin{bmatrix} 26 & 0 & 0 & 30 \\ 0 & -41 & -90 & 0 \\ 0 & -7 & -56 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\lambda^4 + 71\,\lambda^3 - 856\,\lambda^2 - 43316\,\lambda \tag{8.77}$$

Note that a name that appears in a binding list for a use statement, which is intended to be a module, must evaluate to a module at the time the use statement is simplified. This is necessary because the simplification of the use statement must be able to determine the exports of the module. For example, the following attempt to pass a module as a parameter to a procedure does *not* work, and an error occurs when the procedure is simplified.

```
> proc( m, a, b )
      use m in e( a, b ) end use
  end proc;
```

```
Error, no bindings were specified or implied
```

The correct way to use a module as a parameter is to specify the names to be bound explicitly, for example,

```
> proc( m, a, b )
      use e = m:-e in e( a, b ) end use
  end proc;
```

$$\textbf{proc}(m, a, b)\ \ m\text{:-}e(a, b)\ \ \textbf{end proc} \tag{8.78}$$

This is necessary because, until the procedure is called with a module expression as first argument, the reference to **e** is ambiguous. The variable **e** could refer to a module export or to another value (such as a global name). To expand the use statement, this must be known at the time the procedure is simplified.

Operator Rebinding

The use statement also allows most infix and prefix operators in the Maple language to be rebound. This is not operator overloading, which can be performed in some programming languages (such as C++), because the rebinding occurs during the automatic simplification process in Maple.

If an operator name appears on the left side of a binding equation for a use statement (consequently, if it is an exported name of a module that is bound with use), then the corresponding operator expressions in the body of the use statement are transformed into function calls. For example,

```
> use `+` = F in a + b end use;
  m := module()
      export `*`, `+`;
      `+` := ( a, b ) -> a + b - 1;
      `*` := ( a, b ) -> a / b;
  end module:
  s * ( s + t );
  use m in s * ( s + t ) end use;
```

$$F(a, b)$$

$$s\,(s + t)$$

$$\frac{s}{s + t - 1} \tag{8.79}$$

When a module-based package is loaded by running the with command, all of exported operators are rebound at the top level so you do not need to write **use** statements to get the overloaded implementations. If a module, **M**, exports a procedure named +, and you use the command with(M), subsequent sums will be processed through **M:-+**.

In most cases, the new operator procedure should contain the **overload** function. This provides a softer binding where your operator implementation will only be invoked when the arguments passed in match the specified type.

```
> PairMath := module()
    option package;
    export `+`;

    `+` := proc( a::PAIR(integer,integer), b )
         option overload;
         if type(b,PAIR(integer,integer)) then
             PAIR( op(1,a) + op(1,b), op(2,a) + op(2,b) );
         else
             PAIR( op(1,a) + b, op(2,a) + b );
         end if;
    end proc;
end module;
with(PairMath);
PAIR(2,3) + 4;
PAIR(1,1) + PAIR(3,4);
1+1;
```

```
PairMath := module( )
    option package; export `+`;
end module
```

$$[\,`+`\,]$$

$$PAIR(6, 7)$$

$$PAIR(4, 5)$$

$$2 \tag{8.80}$$

In the example above, **PairMath:-+** will only be invoked when the left side of + is a PAIR structure. No error occurs when computing 1+1, which is not handled by **PairMath:-+** because **option overload** has been specified for the **PairMath:-+** procedure. When **option overload** is specified, a mismatched type simply moves on to the next + implementation.

Bypassing the current overload occurs on a mismatched parameter type check, or on any **invalid input:** exception raised within the procedure. The module above can be rewritten as follows.

```
> PairMath := module()
    option package;
    export `+`;

    `+` := proc( a, b )
         option overload;
         if type(a,PAIR(integer,integer)) then
             if type(b,PAIR(integer,integer)) then
                 PAIR( op(1,a) + op(1,b), op(2,a) + op(2,b) );
             else
                 PAIR( op(1,a) + b, op(2,a) + b );
             end if;
         elif type(b,PAIR(integer,integer)) then
             PAIR( a + op(1,b), a + op(2,b) );
         else
             error("invalid input: a or b should be a PAIR
structure");
         end if;
    end proc;
end module;
with(PairMath);
```

```
1 + PAIR(2,3);
2 + 2;
```

```
PairMath := module( )
    option package; export `+`;
end module
```

$$[\,`+`\,]$$

$$PAIR(3, 4)$$

$$4 \tag{8.81}$$

Another option is to use the **overload** function to achieve polymorphism.

```
> PairMath := module()
    option package;
    export `+`;
    local PP, PA, AP;

   PP :=  proc( a::PAIR(integer,integer), b::PAIR(integer,integer)
  )
        option overload;
        print("in PP");
        PAIR( op(1,a) + op(1,b), op(2,a) + op(2,b) );
    end proc;

    PA :=  proc( a::PAIR(integer,integer), b )
        option overload;
        print("in PA");
        PAIR( op(1,a) + b, op(2,a) + b );
    end proc;

    AP :=  proc( a, b::PAIR(integer,integer) )
        option overload;
        print("in AP");
        PAIR( a + op(1,b), a + op(2,b) );
    end proc;

    `+` := overload( [ PP, PA, AP ] );
end module;
with(PairMath);
1 + PAIR(2,3);
PAIR(2,3) + 4;
```

```
PAIR(1,1) + PAIR(3,4);
5+5;
```

```
PairMath := module( )
    option package;
    local PP, PA, AP;
    export `+`;

end module
```

$$[\ `+`\ ]$$

"in AP"

$$PAIR(3, 4)$$

"in PA"

$$PAIR(6, 7)$$

"in PP"

$$PAIR(4, 5)$$

$$10 \tag{8.82}$$

For more information, see the overload help page.

8.7 Interfaces and Implementations

Generic programming is a programming style and a software engineering methodology for writing reusable code. Many Maple built-in operations are generic, for example, the addition operator + computes sums of integers, rational numbers, complex numbers, polynomials, special functions, and so on. When using the addition operator +, you do not need to define how an expression is represented-- the automatic simplifier recognizes how Maple expressions are represented. As with any dynamically typed language, Maple allows for generic programming. Most built-in Maple operations (including many standard library commands) are naturally *polymorphic* in that they can perform successfully with many data formats.

Generic Programming as a Good Software Engineering Practice

When working on any large project, it is important to write *reusable* code; that is, code that can perform a well-defined function in a variety of situations. Generic programs do not rely on the details of how their inputs are represented. They can perform their function on *any* inputs that satisfy a specified set of constraints. Normally, these constraints are described

in terms of the *behavior* of the inputs rather than on their physical representation or the storage layout of their concrete representation. This behavior is sometimes called a *contract*. Generic programs rely *only* on the object behavior specified by the contract. They do *not* rely on information of how an object is implemented; therefore, generic programs *separate* interfaces from implementations.

Distinction between Local and Exported Variables

The behavior specified by the contract for a module includes any module exports. Whatever is expressed through its local variables is private to the module, and is not to be relied on, or even known, by clients of the module. (Client access is, in fact, the only technical difference between module locals and exports.)

Before the introduction of the module system, design by contract was enforced in Maple only by convention. Maple commands whose names had to be enclosed in name quotes (`) were considered private, and not for client use. However, this was only a convention. Also, it was necessary to use global variables to communicate information and state among the commands that comprised a subsystem (such as solve or assume). Now, using modules, it is possible to design software systems that enforce their contracts by a mechanism embedded in the Maple language.

Interfaces

In Maple, contracts are represented by an *interface*, which is a special kind of structured type. It has the form

```
`module`( symseq );
```

where **symseq** is a sequence of symbols or expressions of the form **symbol::type**. For example, an interface for a ring can be written as

```
> `type/ring` := ``module`( `+`, `*`, `-`, zero, one )':
```

while an (additive) abelian group can take the form

```
> `type/abgroup` := ``module`( `+`, `-`, zero )':
```

These symbols are the ones to which clients have access as module exports.

A module is said to *satisfy*, or to *implement*, an interface if it is of the type defined by the interface.

```
> z5 := module()
      description "the integers modulo 5";
      export `+`, `*`, `-`, zero, one;
      `+` := (a,b) -> a+b mod 5;
      `*` := (a,b) -> a*b mod 5;
      `-` := s -> 5-s mod 5;
```

```
      zero := 0;
      one := 1;
  end module:
> type( z5, 'ring' );
```

$$\mathit{false} \tag{8.83}$$

A module can satisfy more than one interface.

```
> type( z5, 'abgroup' );
```

$$\mathit{false} \tag{8.84}$$

Interfaces are an abstraction that form part of the Maple type system. They provide a form of *constrained* polymorphism. Not every Maple type is an interface; only those that have the form described are interfaces. You can define a Maple type (that, as it happens, is not itself an interface) to describe interfaces.

```
> `type/interface` := 'specfunc( {symbol,symbol::type},
                        `module` )':
```

This is a structured type. It describes expressions that are themselves structured types. They have the form of an unevaluated function call with the operator symbol `**module**` and all arguments of type **symbol**, or of type **symbol::type**. In the two previous examples in this section, the types **type/ring** and **type/abgroup** are the interface expressions, and the names **ring** and **abgroup** are the respective names of those interfaces.

A Package for Manipulating Interfaces

The following example illustrates a package for manipulating interfaces. The package is small enough that it can be included here, in full, but it is also available in the **samples/ProgrammingGuide** directory of your Maple installation.

```
> Interface := module()
      description "a package for manipulating interfaces";
      global `type/interface`;
      export define,     # define an interface
             extend,     # extend an interface
             extends,     # test for an extension
             equivalent,# test equivalence
             savelib,    # save an interface
             satisfies;    # test whether a module satisfies
                         # an interface
      local  gassign,    # assign to a global variable
             totype,     # convert from interface name to type
             toset,    # convert from interface name to a set
```

```
        setup;    # install `type/interface` globally
option package, load = setup;

# Define a global type for interfaces.
# This assignment takes care of installing the type
# in the Maple session in which this module definition
# is evaluated. Calling `setup()' ensures that this also
# happens when the instantiated module is read from a
# Maple library archive.
`type/interface`
         := 'specfunc( {symbol, `::`}, `module` )';

# Ensure that `type/interface` is defined. This thunk is
# called when the instantiated `Interface' module is read
# from a Maple library archive.
setup := proc()
    global `type/interface`;
    `type/interface`
      := 'specfunc( {symbol, `::`}, `module` )';
    NULL # quiet return
end proc;

# Assign to the global instance of a name
gassign := proc( nom::symbol, val )
    option inline;
    eval( subs( _X = nom,
                proc()
                  global _X;
                  _X := val
                end proc ) )()
end proc;

# Convert an interface name to the corresponding type.
totype := ( ifc::symbol ) -> ( `type/` || ifc );

# Convert an interface name to a set of symbols.
toset := ( ifc::symbol ) -> { op( ( `type/` || ifc ) ) };

# Install a new interface into the type system.
define := proc( ifc )
    description "define an interface";
    if map( type, {args}, 'symbol' ) <> { true } then
```

```
        error "arguments must all be symbols"
    end if;
    gassign( `type/` || ifc,
       '`module`'( args[ 2 .. nargs ] ) );
    ifc # return the interface name
end proc;

# Implement subtyping.
extend := proc( new, old )
    description "extend an existing inteface";
    if map( type, {args}, 'symbol' ) <> { true } then
        error "arguments must all be symbols"
    end if;
    if not type( totype( old ), 'interface' ) then
        error "cannot find an interface named %1", old
    end if;
    define( new, op( totype( old ) ), args[3..nargs] )
end proc;

# Test whether ifc2 is an extension of ifc1.
extends := proc( ifc1, ifc2 )
    description "test whether the second interface "
                "extends the first";
    local t1, t2;
    t1, t2 := op( map( totype, [ ifc1, ifc2 ] ) );
    if not type( [t1,t2], '[interface,interface]' ) then
        if not type( t1, 'interface' ) then
            error "arguments must be interface names, "
                  "but got %1", ifc1
        else
            error "arguments must be interface names, "
                  "but got %1", ifc2
        end if
     end if;
     toset( ifc1 ) subset toset( ifc2 )
end proc;

# Save an interface to the Maple library archive.
savelib := proc()
    description "save a named interface to a "
                "Maple library archive";
    local ifc;
```

```
        for ifc in map( totype, [ args ] ) do
            if not type( ifc, 'interface' ) then
                error "arguments must be interfaces, "
                      "but got %1", ifc
            end if;
            :-savelib( totype( ifc ) )
        end do
    end proc;

    # Test whether a module satisfies an interface.
    # This is simply an alternative to a call
    # to `type()'.
    satisfies := proc( m, ifc )
       description "test whether a module satisfies an interface";

        if not type( totype( ifc ), 'interface' ) then
            error "second argument must be an interface name, "
                  "but got %1", ifc
        end if;
        type( m, ifc )
    end proc;

    # Test whether two interfaces are equivalent.
    # Since unevaluated function calls compare
    # differently if their arguments are in a
    # different order, we convert them to sets first,
    # and then test for equality.
    equivalent := proc( ifc1, ifc2 )
        description "test whether two interfaces "
                    "are equivalent";
        local t1, t2;
        t1, t2 := totype( ifc1 ), totype( ifc2 );
        if not type( t1, 'interface' ) then
            error "expecting an interface name, "
                  "but got %1", ifc1
        elif not type( t2, 'interface' ) then
            error "expecting an interface name, "
                  "but got %1", ifc2
        end if;
        evalb( { op( t1 ) } = { op( t2 ) } )
    end proc;
end module:
```

This package implements the interface abstraction. It allows you to manipulate interfaces without having to consider how they fit into the Maple type system.

```
> with( Interface );
```

$$[define, equivalent, extend, extends, satisfies, savelib] \tag{8.85}$$

```
> define( 'abgroup', `+`, `-`, 'zero' );
```

$$abgroup \tag{8.86}$$

```
> type( `type/abgroup`, 'interface' );
```

$$true \tag{8.87}$$

```
> type( z5, 'abgroup' );
```

$$false \tag{8.88}$$

```
> satisfies( z5, 'abgroup' );
```

$$false \tag{8.89}$$

```
> extend( 'ring', 'abgroup', `*`, 'one' );
```

$$ring \tag{8.90}$$

```
> type( `type/ring`, 'interface' );
```

$$true \tag{8.91}$$

```
> extends( abgroup, ring );
```

$$false \tag{8.92}$$

```
> satisfies( z5, 'ring' );
```

$$false \tag{8.93}$$

```
> type( z5, 'ring' );
```

$$false \tag{8.94}$$

The load Option

This package provides an abstraction of the interface concept in Maple and illustrates a module feature that was not previously demonstrated: the **load=procedure_name** option. In the **Interface** package, this option is used in a typical way. The declaration

```
option load = setup;
```

that appears in the module definition indicates that, when the instantiated module is read from a Maple library archive, the procedure **setup** is to be called. The procedure named must be a local variable or an exported local variable of the module. The local procedure **setup** in this module simply ensures that the global variable **type/interface** is assigned an appropriate value. This assignment is also made in the body of the module so that the assignment is also executed in the session in which the module is instantiated. This is done for illustrative purposes. A better approach would be to invoke **setup** in the body of the module definition.

9 Object Oriented Programming

9.1 In This Chapter

- A brief introduction to Object Oriented Programming will be presented.
- A description of how Object Oriented Programming is implemented in Maple.
- How to override operators and engine routines using Objects.

9.2 Introduction To Object Oriented Programming

Objects are a programming tool that allows data and procedures to be encapsulated together. For example, an object could be created to represent a car. A car object might have variables to track its position, velocity and steering position. The car object might also have procedures to accelerate the car and to adjust the steering. A further procedure could be implemented to update the car's position and velocity based on the current acceleration, velocity and steering. Multiple car objects could be used to represent multiple cars, each with their own positions and velocities, but sharing the same procedures for how the cars move.

Objects can also restrict access to certain variables and procedures. For example the car object would allow other code to call a routine to adjust the steering, but may not allow external code to set the value of the steering variable directly. Although this may seem restrictive, it allows the object to control its internal state. In the car example, it could limit the range of steering.

Terminology

The variables in an object that store the data and procedures are referred to as the object's *members*. Procedures associated with an object are called *methods*. Object members have access controls which limit where the members can be accessed from, similar to modules. Members declared *exported* can be accessed from anywhere. Members declared *local* can only be accessed from within the object's methods. Objects are instances of a *class*. A class describes the exports and locals that each instance of the class (the objects) will have.

Benefits of Object Oriented Programming

Benefits of object oriented programming are:

- The implementation of a class can be changed radically without changing the interface of exported methods. Thus code that uses the objects will not need to change when the internal implementation changes.
- As objects are self contained, they can be reused.
- Objects can define methods that allow them to integrate with existing Maple routines. Thus users can create objects that can be used like built-in types.

- A set of classes can implement a common set of exports. Thus a procedure that uses only the common exports will work with any objects from any of the classes without needing to know which classes the objects belong to.

Good object oriented design can be difficult. In particular, identifying which concepts should be represented as objects can be tricky. A good rule of thumb is that objects should be your "nouns" and methods should be "verbs". Thus you would create an object to represent a car and you call a method to accelerate the car.

9.3 Objects in Maple

Creating a New Class of Objects

To create a new class of objects, use the named module declaration syntax with option object.

```
> module NewObject()
    option object;
    ...
  end module;
```

This will create a new object and assign the new object to the module name (NewObject in the example above). An object created this way will be referred to as a *prototype object*. In Maple, any object (prototype or other) can be used as a representative of the class.

When declaring an object the members must be declared as either local to the object, using a **local** declaration or exported, using an **export** declaration. A member that is declared **local** can only be accessed from the object's methods, or other object's methods of the same class. A member that is exported can be accessed anywhere.

By default, the values assigned to the object's members are unique to the object. That is, two objects of the same class can have different values assigned to their members. However some members, member procedures in particular, are shared among all objects of a class. Thus members can also be declared as **static**. A **static** member stores only one value that is common to all objects of a class.

Creating More Objects

Once a prototype object exists, it can be used to create new objects using the Object routine. The **Object** routine creates a new object of the same class as the object passed into **Object**.

```
> newObj := Object( existingObject );
```

By default, the newly created object will have its members assigned the same values as the object passed to **Object**. However by implementing a ModuleCopy routine, the object can perform different actions when new instances are created. A **ModuleCopy** routine can accept additional arguments that are passed into the **Object** routine.

```
> newObj := Object( existingObject, arg1, arg2, ... );
```

Objects and Types

All objects are of type **object**. In addition **type** and **::** can be used to determine if an object is an instance of a particular class by passing an object of the expected class as the type. You can refine this type checking by defining the ModuleType method.

9.4 Methods

Methods are procedures assigned to the members of an object. Methods have a few differences from normal procedures.

Methods Can Access Object Locals

A method belonging to a particular class can access both the local and exported members of any object of that class. This allows methods to access and manipulate the internal states of their objects without requiring the objects to export accessor procedures.

Method Names Should Be Declared static

In Maple, most method names should be declared as **static**. In most cases, all objects of the same class use the same procedures for their methods. If the method name is not declared **static**, each object will have a separate copy of the procedure. This can be quite wasteful.

There are some instances where an object will have a non-**static** method. However unless you intend different objects to have different procedures assigned to the method, your method should be **static**.

Methods Are Passed the Objects They Manipulate

Some object oriented languages associate method calls with a particular object. That object is represented via a **self** variable or by allowing direct access to that object's members. Maple does not give a particular object special significance in that way. Instead, all objects that a method needs to manipulate must be passed as parameters.

Calling Methods

To call an object's method, call the method as a standard function call and pass the object in as an argument.

```
> method( ..., object, ... );
```

When a function call is evaluated and an object is passed in as an argument, the object is searched for an exported procedure with a matching name. If one is found, that member procedure is called with the given arguments.

This search proceeds from left to right, so the first object with a matching method is used as the class whose method is invoked.

Objects in Indexed Function Calls

When making an indexed function call (of the form **func[index](args)**) Maple will also check the indices (**index**) for a matching object as well as the arguments. If a matching object is found in the indices, that object will be used before one found in the arguments.

Searching an index sequence is also performed from left to right.

Special Methods

There are a set of special methods that a class can define that will automatically be used in various situations. Not all of these methods make sense for all objects. See the method specific help pages for complete details.

ModuleCopy: The **ModuleCopy** method is invoked when a object is copied via the Object routine.

ModuleType: The **ModuleType** method is invoked when an object is passed into the type routine. It allows a module to have a more precise type check of objects of a particular class.

ModulePrint: The **ModulePrint** method is invoked when an object is pretty-printed.

ModuleDeconstruct: The **ModuleDeconstruct** method is invoked when an object is converted to a one-dimensional form, usually Maple syntax.

ModuleApply: The **ModuleApply** method is invoked when an object is used as a function in a function call expression.

ModuleLoad: The **ModuleLoad** method is invoked when the object class is read in from a library.

ModuleUnload: The **ModuleUnload** method is invoked when an object is garbage-collected.

ModuleIterator: The **ModuleIterator** method creates an interface that can be used to iterate over the contents of the object.

9.5 Overloading Operators

Objects can define methods which allow them to control what happens when those objects are used with various operators. For example, if an object implements a + method, then that method will be invoked if the object appears in a sum expression.

```
> 1 + Obj1 + n;
```

By overloading operators, objects can be used in Maple expressions. This, combined with overloading built-in routines, allows objects to be used naturally in general Maple expressions.

Supported Operators

The following operators can be overloaded by an object:

+	-	*	/	^	!	.	=	<>	<	<=	>	>=
and	or	not	xor	implies	intersect	union	minus	subset	in			
[]	{}	?[]	@	@@	&*	*&name*						

The following operators, in particular, **cannot** be overridden:

:	?()	:-	,	->	:=

Note: These lists are not the same as the operators that can and cannot be overridden using a use statement.

Implementing Operators

In general implementing operators is similar to implementing normal methods. However particular operators have rules that must be followed if they are to be implemented correctly.

The rules for the various operators are documented on the Object,operators help page.

9.6 Overloading Built-in Routines

Objects can implement methods to override some built-in routines (like convert or abs). These methods will be invoked when objects are passed as arguments to the corresponding built-in routines. By overriding built-in routines, user-defined objects can be used in normal Maple expressions. This, combined with overloading operators, allows objects to be used naturally in general Maple expressions.

Any routine implemented in Maple code can be overloaded. However, not all built-in routines (routines implemented in the Maple kernel) can be overloaded.

Overridable Built-in Routines

The following built-in routines can be overloaded by object methods:

abs	aname	conjugate	convert	diff	eval	evalhf	evalf
expand	has	hastype	Im	Re	implies	indets	length
map, map2, map[n]	member	normal	numboccur	subs	trunc	type	

Some overloadable built-in routines have a specific interface that must be followed. The interfaces for the overloadable built-ins can be found on the object,builtins help page.

9.7 Examples

The following example shows a class for objects representing integers modulo a given base.

```
(* create a new class of objects with a prototype object
       named 'IntMod' *)
module IntMod()
    option object;

    (* These locals maintain the internal state of the ModInt objects.
       base is the modulus, value is the integer.  These members are not
       declared as 'static' so each object has its own values for these
       members. *)

    local base := 1;
    local value := 0;

    (* We implement the 'ModuleApply' and 'ModuleCopy' routines to create
a
       nice object factory.  With these defined the prototype object can
       be applied to generate new objects.  These routines are declared
       as 'static', so they are shared between objects of this class. *)
    export ModuleApply::static := proc()
        Object( IntMod, _passed );
    end;

    (* The ModuleCopy routine initializes 'self' using 'proto' and the
       passed arguments. If a value or base is given as a parameter,
       those are used.  Otherwise, these values are copied from 'proto'. *)

    export ModuleCopy::static := proc( self::IntMod, proto::IntMod,
v::integer, b::integer, $ )
        if ( _npassed < 4 ) then
            self:-base := proto:-base;
        else
            self:-base := b;
        end;

        if ( _npassed < 3 ) then
            self:-value := proto:-value;
        else
            self:-value := v mod self:-base;
        end;
```

```
    end;

    (* Implement a 'ModulePrint' routine to allow the objects to
        display nicely *)
    export ModulePrint::static := proc( self::IntMod )
        nprintf( "%d mod %d", self:-value, self:-base );
    end;

    (* We implement a 'ModuleType' routine to allow better type checking.
       This allows a base to be specified in the type check *)
    export ModuleType::static := proc( self, type, b, $ )
        if ( _npassed = 2 ) then
            true;
        else
            evalb( self:-base = b );
        end;
    end;

    (* A getter function to access the value field *)
    export getValue::static := proc( self::IntMod )
        self:-value;
    end;

    (* Overload the + operator.  This routine accepts any number of
       arguments, it sums any IntMod objects and integers appropriately.
       If there are other terms, a sum expression is returned with one
       IntMod and the remaining terms. *)
    export `+`::static := proc( )
       local ints, imods, total, base, other;

       ( ints, other ) := selectremove( type, [_passed], { 'IntMod',
'integer' } );

       ( imods, ints ) := selectremove( type, ints, 'IntMod' );

       base := imods[1]:-base;
       if ( not andmap( type, imods, 'IntMod'( base ) ) ) then
           error "all IntMods must be of the same base"
       end;

      total := ( `if`( numelems(ints) > 0, ints[1], 0 ) + add( getValue(i),
 i in imods ) );

       IntMod( total, base ), op( other );
    end;
```

```
    (* Overload the * operator.  Similar to the + operator, we multiply
       out all the IntMod's and integers, and maintian other terms to be
       returned as part of a product expression. *)
    export `*`::static := proc( )
        local ints, imods, total, base, other;

        ( ints, other ) := selectremove( type, [_passed], { 'IntMod',
'integer' } );

        ( imods, ints ) := selectremove( type, ints, 'IntMod' );

        base := imods[1]:-base;
        if ( not andmap( type, imods, 'IntMod'( base ) ) ) then
            error "all IntMods must be of the same base"
        end;

       total := ( `if`( numelems(ints) > 0, ints[1], 1 ) * mul( getValue(i),
 i in imods ) );

        IntMod( total, base ), op( other );
    end;

    (* ^ operator.  We need to handle 3 cases, IntMod as base, IntMod as
       exponent, and both base and exponent are IntMods *)
    export `^`::static := proc( b, e, $ )
        if ( b::IntMod ) then
           if ( e::IntMod ) then
               IntMod( b:-value^e:-value, b:-base );
           elif ( e::integer ) then
               IntMod( b:-value^e, b:-base );
           else
               error( "integer expected for exponent" );
           end;
        else
           b^e:-value;
        end;
    end;

    (* For the comparison operators, handle the cases where there is only
       one argument or one of the arguments is not an IntMod by returning
       false. *)
    export `=`::static := proc( l, r, $ )
        if ( _npassed <> 2 or not l::IntMod or not r::IntMod ) then
           return false;
```

```
    end;
    evalb( l:-base = r:-base and l:-value = r:-value )
end;

export `<`::static := proc( l, r, $ )
    if ( _npassed <> 2 or not l::IntMod or not r::IntMod ) then
       return false;
    end;
    evalb( l:-base = r:-base and l:-value < r:-value )
end;

export `<=`::static := proc( l, r, $ )
    if ( _npassed <> 2 or not l::IntMod or not r::IntMod ) then
       return false;
    end;
    evalb( l:-base = r:-base and l:-value <= r:-value )
end;

export `>`::static := proc( l, r, $ )
    if ( _npassed <> 2 or not l::IntMod or not r::IntMod ) then
       return false;
    end;
    evalb( l:-base = r:-base and l:-value > r:-value )
end;

export `>=`::static := proc( l, r, $ )
    if ( _npassed <> 2 or not l::IntMod or not r::IntMod ) then
       return false;
    end;
    evalb( l:-base = r:-base and l:-value >= r:-value )
end;

(* override the convert function to allow conversions from IntMods
   to integers. *)
export convert::static := proc( v, toType, $ )
    if ( v::IntMod ) then
        if ( toType = ':-integer' ) then
            v:-value;
        else
            error "cannot convert from IntMod to %1", toType;
        end;
    else
        error "cannot convert into IntMod from %1", v
    end;
```

```
    end;
end:
```

```
> i0m5 := IntMod( 0, 5 );
```

$$i0m5 := 0 \bmod 5 \tag{9.1}$$

```
> i1m5 := Object( i0m5, 1 );
```

$$i1m5 := 1 \bmod 5 \tag{9.2}$$

```
> type( i1m5, 'IntMod' );
```

$$true \tag{9.3}$$

```
> type( i1m5, 'IntMod'(3) );
```

$$false \tag{9.4}$$

```
> type( i1m5, 'IntMod'(5) );
```

$$true \tag{9.5}$$

```
> i2m5 := i1m5 + 1;
```

$$i2m5 := 2 \bmod 5 \tag{9.6}$$

```
> i3m5 := i2m5 + 1;
```

$$i3m5 := 3 \bmod 5 \tag{9.7}$$

```
> i4m5 := i3m5 + 1;
```

$$i4m5 := 4 \bmod 5 \tag{9.8}$$

```
> i4m5 + 1;
```

$$0 \bmod 5 \tag{9.9}$$

```
> i3m5+i4m5;
```

$$2 \bmod 5 \tag{9.10}$$

```
> i1m5 + 9 + i4m5;
```

$$4 \bmod 5 \tag{9.11}$$

```
> convert( i3m5, 'integer' );
```

$$3 \tag{9.12}$$

```
> convert( 3, IntMod );
Error, (in IntMod:-convert) cannot convert into IntMod from 3
```

```
> i2m5 * i4m5 * y * f(x);
```

$$3 \bmod 5\, y\, f(x) \tag{9.13}$$

```
> i2m5^1;
```

$$2 \bmod 5 \tag{9.14}$$

```
> i2m5^2;
```

$$4 \bmod 5 \tag{9.15}$$

```
> i2m5^3;
```

$$3 \bmod 5 \tag{9.16}$$

```
> i2m5^4;
```

$$1 \bmod 5 \tag{9.17}$$

```
> i2m5^5;
```

$$2 \bmod 5 \tag{9.18}$$

```
> evalb(i2m5 < i4m5);
```

$$\mathit{true} \tag{9.19}$$

```
> evalb(i3m5 > i2m5);
```

$$\mathit{true} \tag{9.20}$$

```
> evalb(i2m5 <= i4m5);
```

$$\mathit{true} \tag{9.21}$$

```
> evalb(i3m5 >= i2m5);
```

$$\mathit{true} \tag{9.22}$$

```
> evalb(i3m5 = i2m5);
```

$$\mathit{false} \tag{9.23}$$

9.8 Avoiding Common Mistakes

Overloaded Operators and Built-in Routines Must Handle All Possibilities

An object's method will be invoked whenever that object appears in a matching function call, regardless of the object's position in the argument sequence. Thus when implementing

operators and overloading built-in routines, it is important to handle all the cases where the object could appear.

In the following example it might be easy to assume that when **member** is called the object will be the first argument (the container). However, it is also possible that the object will appear as the second argument (the element being searched for).

```
> module Container()
    option object;

    local t := table();

    export insert::static := proc( c::Container, a, $ )
        c:-t[a] := 1;
        NULL;
    end;

    export member::static := proc( c, e, $ )
        if ( c::Container ) then
            if ( c:-t[e] = 1 ) then
                return e;
            else
                return 0;
            end;
        else
            return ':-member'( c, e );
        end;
    end;
  end:
> container := Object( Container ):
> insert( container, a ):
> member( container, a );
```

$$true \tag{9.24}$$

```
> member( [container], container );
```

$$true \tag{9.25}$$

For some possibilities, the correct approach is to simply pass the arguments on to the Maple routine. When doing so, care must be taken to access the correct version of the routine.

Make Sure to Access the Correct Routine

When overloading operators and built-in routines, those overloads will be used within the implementation of the object itself. This means that care should be taken to call the global version of a routine when it is required. In the **member** overload shown earlier, the code invokes the global version of **member**, by using quotes and **:-**. Failing to do so can lead to infinite recursions and other unexpected behavior.

Be Aware of NULL

Be careful when assuming that operators and built-in routines will always be passed a certain number of arguments. Many will accept **NULL** as an argument, and this may lead to fewer arguments than expected.

```
> module Wrapper()
    option object;

    local value := 10;

    export `=`::static := proc( l, r, $ )
        ( l::Wrapper and r::Wrapper and l:-value = r:-value );
    end;
  end:
> cp := Object( Wrapper ):
> evalb( cp = Wrapper );
```

true (9.26)

```
> evalb( cp = 11 );
```

false (9.27)

```
> evalb( cp = NULL );
```

"Error, invalid input: Wrapper:-= uses a 2nd argument, r, which is missing" (9.28)

Lexical Scoping Does Not Circumvent local

Members that are declared as local can only be accessed from within the class's methods. This means that methods cannot use lexical scoping to pass values to nested procedures.

```
> module LexicalObj()
    option object;
    local a;
```

```
    export b :: static := proc(mm :: m, f, lst :: list, $)
        print(mm:-a);
        return map(x -> f(mm:-a, x), lst);
    end;
end:
> b(m, `+`, [1,2,3]);
```

$$b(m, `+`, [1, 2, 3]) \tag{9.29}$$

In this example, we can print the value of **a** in **b** because **b** is a method. However the **map** fails because the arrow procedure is not a member and thus does not have access to **a**.

10 Input and Output

10.1 In This Chapter

- Introduction
- Input and output in the worksheet
- Input and output with files
- Reading and writing formatted data
- Useful utilities
- 2-D math

10.2 Introduction

This chapter explores the ways in which you can read input and write output programmatically. Here are a few examples of I/O operations in Maple.

- **Example 1:** An integral can be printed in various ways. The two outputs below show the integral in 2-D and 1-D representations, respectively.

```
> y := Int(x^2, x=1..2);
```

$$y := \int_1^2 x^2 \, dx \tag{10.1}$$

```
> lprint(y);
```

```
Int(x^2, x = 1 .. 2)
```

- **Example 2:** The checkfile procedure defined below uses commands in the FileTools package to examine the properties of a file.

```
> checkfile := proc(fname :: string)
      return FileTools:-Exists(fname) and
  FileTools:-IsReadable(fname);
  end proc:
```

- **Example 3:** A Matrix is written to a file using the ExportMatrix command.

```
> M := LinearAlgebra:-RandomMatrix(5, 4);
```

$$M := \begin{bmatrix} -98 & -76 & -4 & 29 \\ -77 & -72 & 27 & 44 \\ 57 & -2 & 8 & 92 \\ 27 & -32 & 69 & -31 \\ -93 & -74 & 99 & 67 \end{bmatrix} \tag{10.2}$$

```
> ExportMatrix("testfile", M);
```

$$67 \tag{10.3}$$

- **Example 4:** The sscanf command is used here to read three floating-point numbers from the string given as the first argument.

```
> z := sscanf("X=123.4 Y=-27.9 Z=2.3E-5", "X=%f Y=%f Z=%f");
```

$$z := [123.4, -27.9, 0.000023] \tag{10.4}$$

The first example shows the difference between 1-D and 2-D output in Maple. Note that input can also be provided in both forms. In Maple, 1-D math is character-based, is available in all interfaces, and can be controlled by many of the basic I/O commands discussed in this chapter. Typeset or 2-D math is available only with the standard worksheet interface and is generally manipulated interactively using the Maple GUI tools. However, it can be controlled programmatically in a limited way.

Most of this chapter is devoted to the manipulation of files, which is the main way data is shared between Maple and external applications. Many of the file-processing commands also apply to interactive input and output, when 1-D mode is used. For example, the command for formatted writing, printf, produces output in a Maple worksheet or document. However, it is essentially identical to the fprintf command for printing to a file.

This chapter starts with a discussion of input and output in the worksheet, including some notes on using other interfaces. The next section covers manipulation of files. Tools for importing and exporting general files as well as those specially designed for numerical data are discussed. Later in the chapter, low-level commands for formatted reading and writing, along with other useful utilities, are shown. The chapter concludes with an explanation of 2-D math and how it can be customized programmatically.

10.3 Input and Output in the Worksheet

This section introduces common ways of reading from the keyboard and writing to the screen. In contrast, the following section discusses I/O through the use of files. There is some overlap between the two sections, as the keyboard and screen themselves can be considered files. This is explained further in the "Default and Terminal Files" section of the file_types help page.

Interfaces

Maple has several user interfaces, all described on the versions help page. In this chapter, it is assumed you are using either the standard worksheet interface or the command-line interface. Most of the I/O operations described here apply to either interface. The major exception is typeset or 2-D math, which is available only with the standard worksheet interface.

You can use the interface command to communicate with the user interface. It allows you to query or set certain user interface variables. This is one way of controlling the look of the output. Note that the **interface** command does not affect the actual computation. A few examples are shown below. A complete list of the variables is available on the interface help page.

The **version** variable returns the interface version, platform information, build date and build number.

```
> interface(version);
```

Standard Worksheet Interface, Maple 17.00, Linux, March 5 2013 Build ID 816628 (10.5)

The **prettyprint** variable controls how Maple expressions are rendered as output.

```
> interface(prettyprint=1):
  Diff(f(x), x);
```

```
  d
--- f(x)
 dx
```
(10.6)

```
> interface(prettyprint=3):
  Diff(f(x), x);
```

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) \tag{10.7}$$

The **rtablesize** variable specifies the largest-sized **rtable** that will be displayed inline. If an **rtable** has a dimension that is larger than this integer, then it is displayed as a placeholder.

```
> interface(rtablesize);
```

$$10 \tag{10.8}$$

```
> Matrix(5, 5, (i,j)->i+j);
```

$$\begin{bmatrix} 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \\ 6 & 7 & 8 & 9 & 10 \end{bmatrix} \tag{10.9}$$

```
> Matrix(15, 15, (i,j)->i+j);
```

$$\begin{bmatrix} \textit{15 x 15 Matrix} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{bmatrix} \tag{10.10}$$

Interactive Output

By default, the output from a statement entered in the worksheet is automatically printed to the screen, unless the statement is terminated by a colon. In the previous section, you saw how to use the interface command to customize certain aspects of the output. Another way to adjust the output is to set the printlevel environment variable. The default value of **printlevel** is 1. When it is set to a higher value, additional information pertaining to procedure calls is printed. This is one way of tracing a procedure for debugging purposes. For more information about debugging programs, see *???*.

The print command prints Maple expressions using the current setting of the **prettyprint** interface variable. In the worksheet, the default output is 2-D math and in the command-line version, the default is a simulated math notation using text characters. Note that the **print** command returns **NULL** and thus the output cannot be regenerated with the ditto commands.

The **print** command is particularly useful in two situations. First, it can be used to print intermediate results calculated within a procedure. Normally, only the returned value of a procedure is printed when it is called.

```
> p := proc(n)
          local i;
          for i to n do
              i^2;
```

```
            end do;
        end proc:
> p(5);
```

$$25 \tag{10.11}$$

```
> q := proc(n)
            local i;
            for i to n do
                print(i^2);
            end do;
        end proc:
> q(5);
```

$$1$$
$$4$$
$$9$$
$$16$$
$$25 \tag{10.12}$$

The **print** command can also be used to print procedures and other expressions that follow last name evaluation rules. For more information, refer to the last_name_eval help page.

```
> print(q);
```

```
proc(n)
    local i;
    for i to n do
        print(i^2)
    end do
end proc
```
(10.13)

The lprint command prints Maple expressions in a character-based notation that is similar to the format used for 1-D input. Like the **print** command, the value returned by a call to **lprint** is **NULL**, and the output cannot be recalled using the **ditto** operator. When the **prettyprint** interface variable is set to 0, Maple uses **lprint** to print all expressions to the interface.

```
> lprint(expand(x+y)^5);
```

```
(x+y)^5
```

Another commonly used command is printf, which produces formatted output. This command will be discussed in *Reading and Writing Formatted Data (page 376)*.

If you want to redirect all output that normally goes to the screen to a file, use the writeto and appendto commands. This is an easy way to log the input and output of a Maple session, particularly if you are using the command-line interface. In the standard worksheet interface, you can simply save the current worksheet or document. For more information on writing to files, see *Input and Output with Files (page 367)*.

Interactive Input

Normally, input is passed to Maple procedures directly through the procedure's parameters. In the standard worksheet interface, input can also be provided through Maplets and components. For more information about these topics, see *???*.

The readline and readstat commands are also available for interactive input, though these are less commonly used. The **readline** command reads the next line from the terminal or a file and returns it as a string, while the **readstat** command reads the next statement from the terminal and returns the value of that statement.

Customization

You can customize the prettyprinting of a function **fnc** in a limited way by defining a **print/fnc** procedure. In the following example, expressions of the form **g(x)** should be printed so that the argument is repeated three times in a list.

```
> `print/g` := proc(x) [x, x, x] end proc:
```

```
> g(b^2);
```

$$[b^2, b^2, b^2] \tag{10.14}$$

```
> g(5.8);
```

$$[5.8, 5.8, 5.8] \tag{10.15}$$

There is a similar facility for prettyprinting a module. If a module has an export or local named ModulePrint, then the result of the **ModulePrint** command is displayed instead of the module when a command in the module is executed. For more information, see *Programming with Modules (page 307)*.

10.4 Input and Output with Files

Introduction

This section covers input and output using files, which is recommended when you have a large amount of data to process. This also provides a way for Maple to share data with external applications. In this chapter, the term "file" is not limited to a disk file, but can include the default output stream to a terminal or worksheet output region. Below is a brief introduction to a few concepts related to files. For more detailed information, refer to the file and file_types help pages.

- **Text and binary files:** The Maple I/O library works with both text files (streams of characters) and binary files (streams of bytes). The I/O commands allow you to specify the type of file and generally assume a text file if no information is given.
- **Read and write modes:** At any given time, a file may be open either for reading or for writing. If you attempt to write to a file which is open for reading, Maple closes and reopens the file for writing. If you do not have permission to write to the file, then an error occurs.
- **The default and terminal files:** The Maple I/O library treats the user interface as a file. The identifier **default** refers to the current input stream, the one from which Maple reads and processes commands. The identifier **terminal** refers to the top-level input stream, the current input stream when you started Maple. When Maple is run interactively, **default** and **terminal** are equivalent.
- **File names and descriptors:** Maple I/O commands refer to files in one of two ways: by name (given as a string) or by descriptor. A file descriptor identifies a file after you have opened it using its name and offers slight advantages in terms of efficiency and flexibility. The commands described in this section accept either a name or a descriptor as the file identifier.
- **Current directory:** If you create files using the examples in this chapter, the files are saved in the current directory. To query or set the current working directory, use the currentdir command.

Working with General Files

This section covers the manipulation of general files. If you are working with files of numerical data, it is recommended that you use the commands described in *ImportMatrix and ExportMatrix (page 372)*.

There are two sets of commands that you can use. The first subsection below describes the basic top-level commands for file manipulation. Alternatively, you can use the FileTools package, which provides a simpler interface to the other commands and offers additional functionality. For most file operations, the **FileTools** package is recommended, but the two sets of commands are generally compatible and can be used interchangeably on a file.

The Maple I/O Library

Below is a description of commonly used commands in the Maple I/O library.

- **Opening and closing files**

  Before you can read from or write to a file, you must open it. When referring to files by name, this happens automatically with the first file operation. When you use descriptors, however, you must explicitly open the file first to create the descriptor. The fopen command takes as arguments the filename, a mode (**READ**, **WRITE** or **APPEND**) and optionally, the file type (**TEXT** or **BINARY**).

```
> f := fopen("testfile", 'WRITE', 'TEXT');
```

  When you are finished with a file, you can close it with the fclose command, which takes the file identifier as its argument. This operation ensures that all information is written to the disk. When you exit Maple or use the **restart** command, Maple automatically closes any open files, whether they were opened explicitly with **fopen** or implicitly through one of the other I/O commands.

- **Reading and writing lines of text**

  The readline command reads one newline-terminated line from a file and returns a string containing that line. The writeline command writes one or more strings to a file, separated with newline characters, and returns a count of the characters. If the file is not already open with type **TEXT**, then the readline or writeline command will open the file automatically.

```
> writeline("testfile", "The first line", "The second line");
```

- **Reading and writing bytes**

  The readbytes command reads one or more or bytes from a file and returns a list of integers. Optionally, you can specify that the file is to be opened in text rather than binary mode, and in this case, a string is returned. You can also provide a previously created **rtable** to the **readbytes** command and it will return the data in the **rtable**. Similarly, the writebytes command writes bytes from a string or list to a file. More information about both commands can be found on their help pages.

- **Reading and writing formatted input and output**

  The fscanf command parses expressions from a file based on a format string. Similarly, the fprintf command prints expressions to a file based on a format string. Both commands are similar to the C standard library commands of the same names. Both **fscanf** and **fprintf** are described in greater detail in *Reading and Writing Formatted Data (page 376)*.

- **Other file utilities**

There are a number of other useful file utilities, including iostatus (obtain the status of an open file), fremove (remove a file), fflush (flush output), filepos (sets or returns the position), and feof (check if the current position is at the end). For more information about these commands, refer to their help pages.

Below is a simple example that uses a few of the commands introduced here. The file generated will be placed in your current working directory, which you can set with the currentdir command.

First, define a **Vector** of floating-point values.

```
> V := Vector([1.20, 4.85, 6.23, 2.45, 7.99]):
> n := LinearAlgebra:-Dimension(V):
```

Next, create a new file called prices1.txt and write a number of lines, one for each **Vector** entry.

```
> fid := fopen("prices1.txt", 'WRITE', 'TEXT'):
> writeline(fid, "List of Prices"):
  for i to n do
      fprintf(fid, "Item %d costs %.2f\n", i, V[i]):
  end do:
> fclose(fid):
```

Now open the file again and read the values from each line, adding them up as they are read.

```
> fid := fopen("prices1.txt", 'READ', 'TEXT'):
> readline(fid):
> pricesum := 0.:
  while not feof(fid) do
      t := fscanf(fid, "Item %d costs %f\n"):
      pricesum := pricesum + t[2]:
  end do:
> fclose(fid):
```

Finally, reopen the file to append a line showing the sum of the prices.

```
> fid := fopen("prices1.txt", 'APPEND', 'TEXT'):
> fprintf(fid, "\nThe sum of the prices is: %.2f\n", pricesum):
> fclose(fid):
```

If you encounter an error while using any of the I/O commands listed in this section, refer to the IO_errors help page for more information about the source of the error. Common

mistakes include reading from a file that does not exist and writing to a file for which you do not have permission to alter.

The FileTools Package

The FileTools package is a collection of file manipulation utilities. It covers most of the functionality described in the previous section and provides an easy-to-use interface. It also contains a large number of additional commands that are useful when working with files.

The **FileTools** package contains two subpackages: FileTools:-Text and FileTools:-Binary. These subpackages contain commands to work with text files and binary files, respectively.

Some of the commonly used commands are listed below. A full list of commands is available in the FileTools help page.

- **Opening and closing files**

  The FileTools:-Text:-Open command allows a file to be opened, with options to indicate whether Maple can create the file if it does not already exist, overwrite it, or append to it. It returns a file descriptor. As with the situation described in the previous section, it is not always necessary to open a file before using it, as a file is automatically opened when you use a **FileTools** command to access it. The FileTools:-Text:-Close command closes a file and ensures all data is written to disk. When you exit Maple, all open files are automatically closed. The FileTools:-Text:-OpenTemporaryFile command causes a temporary file to be opened. Corresponding commands are available in the **Binary** subpackage: FileTools:-Binary:-Open, FileTools:-BinaryClose, and FileTools:-Binary:-OpenTemporaryFile.

- **Reading from and writing to binary files**

  The FileTools:-Binary:-Read and FileTools:-Binary:-Write commands are available for reading and writing binary data. Unlike the readbytes and writebytes commands, the **FileTools** commands support a number of hardware data types and allows the byte order to be specified. There is also a FileTools:-Binary:-CountBytes command for returning the total number of bytes left in a file.

- **Reading from and writing to text files**

  The **FileTools:-Text** subpackage has a large number of commands for reading and writing text. The FileTools:-Text:-Readline and FileTools:-Text:-Writeline commands read and write a line at a time. The FileTools:-Text:-ReadFile command reads all lines in a file at once.

  The FileTools:-Text:-ReadFloat and FileTools:-Text:-WriteFloat commands offer simple ways to read and write a single float. The FileTools:-Text:-ReadNextFloat command is useful if you want to read the next float while ignoring all characters preceding it. The FileTools:-Text:-CountFloats command counts the number of floating-point numbers

remaining in the file. Similar commands are available for integers, characters, and strings as well.

- **Checking and modifying properties of files**

  The **FileTools** package has commands that allow you to examine the properties of a file, such as FileTools:-Status, FileTools:-Exist, and FileTools:-AtEndOfFile. There are a number of additional commands that check if a file is open, readable, writable, lockable, or executable. The package also includes commands that return a file's size and position.

  It is possible to modify files, by using, for example, the FileTools:-Rename and FileTools:-Remove commands. There are also commands for copying, locking, and unlocking files.

- **Performing directory operations**

  The **FileTools** package also includes commands to work with directories and file paths, such as FileTools:-ListDirectory, FileTools:-MakeDirectory and FileTools:-AbsolutePath.

The following example is similar to the one shown in the previous section using the basic I/O commands, but this time, you will use the **FileTools** package.

First, create a new file prices2.txt containing a title and a line for each of the values in **V**. Here, you can use the commands for writing strings, integers and floats, without worrying about specifying the formatting precisely.

```
> with(FileTools:-Text):
> V := Vector([1.20, 4.85, 6.23, 2.45, 7.99]):
> fid := Open("prices2.txt", 'overwrite'):
> WriteLine(fid, "List of Prices"):
  for i to n do
      WriteString(fid, "Item");
      WriteInteger(fid, i, 'delim'=" ");
      WriteString(fid, "costs");
      WriteFloat(fid, V[i], 'leftdelim'=" "):
      WriteLine(fid, ".");
  end do:
> Close(fid);
```

Now, open the file again and read the floating-point values from each line. The **CountLines** and **ReadNextFloat** commands make this task easier, as you do not have to check for the end of file or explicitly read other characters in each line.

```
> fid := Open("prices2.txt"):
> ReadLine(fid):
```

```
> pricesum := 0.:
  numlines := CountLines(fid):
  for i to numlines do
      ReadNextInteger(fid):
      pricesum := pricesum + ReadNextFloat(fid):
  end do:
> Close(fid):
```

Finally, open the file again to append a line showing the sum of the prices.

```
> fid := Open("prices2.txt", 'append'):
> WriteLine(fid, "", "The sum of the prices is:"):
> WriteFloat(fid, pricesum):
> Close(fid):
```

Importing and Exporting Numerical Data

The basic I/O commands and the FileTools package can be used to read from and write to any text or binary file. However, if the file that you want to read or write consists exclusively of numeric data, then it is much easier to use one of the commands designed for this type of file.

ImportMatrix and ExportMatrix

The ImportMatrix and ExportMatrix commands read and write data that can be stored in a Matrix or Vector.

These commands support different types of files, including some that are generated or recognized by other software applications. The formats supported are: MATLAB®, Matrix Market, comma-separated values (.csv), and generic delimited files. The **source** and **target** options are used to indicate the desired format.

Files created with MATLAB versions 5, 6 or 7 can be imported. By default, the **ExportMatrix** command generates a MATLAB Version 7 binary file, using data compression as described on the StringTools:-Compress help page, when the **target=Matlab** option is provided. However, it is possible to generate a Version 6 file without compression by adding the **mode=v6** option. You can also read and write MATLAB ASCII files using the **mode=ascii** option. Import and export of both dense and sparse Matrices are supported with MATLAB format.

Matrix Market files are imported and exported using the **MatrixMarket** value for the **source** and **target** options. The Matrix Market coordinate and array formats are supported; the pattern format is not supported.

For .csv and general delimited files, the **format** option can be used to indicate whether the storage is dense or sparse. In the latter case, only the nonzero entries are present in the imported or exported file.

Below is a small example showing how **ImportMatrix** and **ExportMatrix** work with MATLAB arrays. The file will be placed in your current working directory.

Generate two random Matrices and export them to a MATLAB Version 6 file. The number of bytes written is returned by the **ExportMatrix** command.

```
> A := LinearAlgebra:-RandomMatrix(3, 4, 'datatype'=float[8]);
```

$$A := \begin{bmatrix} -70. & -94. & -53. & 40. \\ 13. & -7. & 21. & 97. \\ -58. & 12. & -25. & 43. \end{bmatrix} \tag{10.16}$$

```
> B := LinearAlgebra:-RandomMatrix(2, 5, 'datatype'=float[8]);
```

$$B := \begin{bmatrix} 96. & -80. & -29. & 89. & -67. \\ 93. & -92. & 96. & -55. & 77. \end{bmatrix} \tag{10.17}$$

```
> ExportMatrix("testfile.mat", [A, B], 'target'='Matlab',
  'mode'='v6');
```

$$432 \tag{10.18}$$

Now, read the MATLAB arrays back into Maple using the **ImportMatrix** command. It is not necessary to use the **source** and **mode** options in this case. The **ImportMatrix** command can automatically recognize MATLAB binary files. With most text files (MATLAB or otherwise), you will have to specify the source type.

```
> M := ImportMatrix("testfile.mat");
```

$$M := \left[\left["mat1", \begin{bmatrix} -70. & -94. & -53. & 40. \\ 13. & -7. & 21. & 97. \\ -58. & 12. & -25. & 43. \end{bmatrix} \right], \left["mat2", \begin{bmatrix} 96. & -80. & -29. & 89. & -67. \\ 93. & -92. & 96. & -55. & 77. \end{bmatrix} \right] \right] \tag{10.19}$$

A sequence of two lists is returned, with each list containing a string and a Matrix. The string shows the name stored with each matrix in the MATLAB file. The names are automatically assigned by the **ExportMatrix** command, but you can specify your own names with the **arraynames** option.

Now, export the Matrix **A** to a text file with values delimited by spaces.

```
> ExportMatrix("anotherfile.txt", A, 'target'='delimited',
  'delimiter'=" ");
```

$$282 \tag{10.20}$$

Import the contents of the file back into Maple. In this case, it is necessary to specify the source and the character used as delimiter.

```
> ImportMatrix("anotherfile.txt", 'source'='delimited', 'delimiter'="
   ");
```

$$\begin{bmatrix} -70.000000000000000, -94.000000000000000, \\ -53.000000000000000, 40.000000000000000], \\ [13.000000000000000, -7.0000000000000000, \\ 21.000000000000000, 97.000000000000000], \\ [-58.000000000000000, 12.000000000000000, \\ -25.000000000000000, 43.000000000000000]] \end{bmatrix} \tag{10.21}$$

Notice that only a single Matrix is returned. Multiple Matrices can be exported to MATLAB, but with other formats, only a single Matrix can be saved in a file. Also, only MATLAB arrays have names associated with them.

Other Commands

The readdata command reads numeric data from a text file into Maple. The data in the file must consist of integers or floating-point values arranged in columns, separated by white space, and it is returned in a list or list of lists.

The writedata command writes numeric data from a Maple vector, matrix, list or list of lists into a text file. This command accepts an optional argument in the form of a procedure that allows you to control the formatting of the output.

Files Used by Maple

In additional to the general files that can be manipulated by the I/O commands described earlier in this section, several other files are used implicitly by Maple. A few are described briefly below. For more information, refer to the file help page.

- **Maple language files**

  A Maple language file contains statements conforming to the syntax of the Maple language. These are the same as statements that can be entered interactively. Any filename can be used for a Maple language file, but the name cannot end with ".m". The standard file extension for Maple language files is ".mpl".

Maple language files can be created using a text editor or the save statement. Maple procedures and complex scripts of commands are usually written in a text editor, while the **save** statement is used to save results or procedures that were entered into Maple interactively.

Maple language files may be read using the read statement. The statements within the file are read as if they were being entered into Maple interactively, except that they are not echoed to the screen unless the **echo** interface variable has been set to 2 or higher.

Maple includes a preprocessor modeled on the C preprocessor and Maple language files may include preprocessor directives such as **$include** and **$define**.

- **Internal format files**

  Maple internal format files are used to store expressions compactly. Expressions stored in this format can be read by Maple faster than those stored in the Maple language format. These files are identified by filenames ending with the two characters ".m" (or ".M" on platforms where filenames are not case-sensitive).

  Like Maple language files, Maple internal format files are read and written using the read and save statements. The presence of the ".m" ending in the filename specifies that the file is an internal format file, and not a language file.

- **Library archives**

  Maple uses library archive files to store collections of internal format files. These files end with the extension ".mla" (or, for older library archive files, with extension ".lib"). For more information about creating Maple libraries, see *Writing Packages (page 383)*.

- **Help databases**

  A Maple help database is a file that stores a collection of files representing help pages in the Maple help system. It contains the information required to index, navigate, and search the help system, and its filename has the extension ".hdb". For more information, refer to the worksheet/reference/maplehdb help page.

- **Worksheet files**

  If you are using Maple with a graphical user interface, you can save your worksheet. In the standard worksheet interface, files are identified by names ending with ".mw". In the classic worksheet interface, files end in ".mws". Both types of files are portable between the graphical user interfaces on different platforms.

- **Maplet Files**

  Maple worksheets can be saved as ".maplet" files. The MapletViewer runs such files independent of the Maple worksheet environment.

10.5 Reading and Writing Formatted Data

The scanf and printf Commands

The scanf and printf commands allow you to read from and write to the terminal using a specified format. The formatting information is provided by a format string. Below is an example showing how the **printf** command is used to display floating-point values.

Enter the following Vector of values:

```
> V := Vector([.8427007929, .9953222650, .9999779095, .9999999846,
   1.000000000]);
```

$$V := \begin{bmatrix} 0.8427007929 \\ 0.9953222650 \\ 0.9999779095 \\ 0.9999999846 \\ 1.000000000 \end{bmatrix} \tag{10.22}$$

Print each value on a single line, preceded by an integer indicating its position. The format string is the first argument to the **printf** command. This string consists of two conversion specifications, "%d" and "%.2e", along with other characters to be printed, including the newline character "\n". The first conversion specification indicates that the first argument following the format string should be printed as an integer. The second conversion specification indicates that the second argument following the format string should be printed in scientific notation, with two digits after the decimal point.

```
> for i to LinearAlgebra:-Dimension(V) do
      printf("%d%12.2e\n", i, V[i]);
  end do;
```

```
1    8.43e-01
2    9.95e-01
3    1.00e+00
4    1.00e+00
5    1.00e+00
```

The **scanf** and **printf** commands belong to a family of related commands that provide formatted I/O capabilities. The other commands will be discussed later in this chapter. These commands are based on similarly named functions from the C programming language library.

For example, the sscanf command below reads an integer, a space, a character, and a floating-point value from the string given as the first argument. The conversion specifications, "%d", "%c" and "%f", will be explained in the next section.

```
> sscanf("892 123.456E7","%d %c%f");
```

$$[892, "1", 2.3456\,10^{8}] \tag{10.23}$$

Format Strings

As you saw in the previous examples, the format string passed to **scanf** or **printf** specifies exactly how Maple is to parse the input or write the output. It consists of a sequence of conversion specifications that may be separated by other characters.

First, consider the specification for the **scanf** command, which has the format shown below. What follows is a brief explanation of the specification. For more information, refer to the scanf help page.

%[*][*width*][*modifiers*]*code*

- The character "%" begins each conversion specification.
- The optional character "*" indicates that the item scanned is to be discarded and not returned as part of the result.
- The optional *width* indicates the maximum number of characters to be scanned for this object. You can use this to scan one larger object as two smaller objects.
- The optional *modifier* affects the type of value to be returned. The most common of these is "Z", which, when preceding any of the numeric format codes, indicates that a complex value is to be scanned.
- Several format codes are available for use with **scanf**. A few of the more commonly used ones are mentioned here.

  "d" -- integer

  "f" -- floating-point number

  "c" -- character

  "s" -- string

  "a" -- Maple expression

The specification for the **printf** command is similar to that for **scanf**. The differences are summarized here. For more information, refer to the printf help page. The specification has the following format.

%[*flags*][*width*][.*precision*][*modifiers*]*code*

- As with **scanf**, the conversion specification for **printf** begins with "%". The optional *width* and *modifiers* are similar to those described earlier. The *width* value indicates the minimum number of characters to output for the field.
- The optional *flag* can be one of several characters affecting how numeric values are displayed. For example, the flag "+" indicates that signed numeric values are output with a leading "+" or "-" sign.
- The format codes for **printf** are similar to those for **scanf**. One notable difference is that, while "e" and "g" are equivalent to "f" for **scanf**, they produce different output in **printf**. The code "e" causes a numeric value to be printed in scientific notation, while output using the code "g" uses one of integer, fixed-point or scientific notation, depending on the numeric value.

The **scanf** and **printf** commands can also be used to print **rtables**. For more information about the flags used for this purpose, refer to the rtable_printf help page.

Related Commands

Several commands are related to **scanf** and **printf**:

- **fscanf** and **fprintf**

  These commands read from and write to a file instead of the terminal. They take a filename or descriptor as an additional argument, but otherwise use the same calling sequence as **scanf** and **printf**.

- **sscanf** and **sprintf**

  These commands read from and write to a string (which is then returned) instead of the terminal. The **sscanf** command takes a string as an additional argument, but otherwise these commands use the same calling sequence as **scanf** and **printf**.

- **nprintf**

  This command is the same as **sprintf** except that it returns a Maple symbol instead of a string.

All these commands are described fully on the scanf and printf help pages.

10.6 Useful Utilities

This section describes other tools that are useful for input and output.

The StringTools Package

The StringTools package is a collection of utilities for manipulating strings. These commands are frequently used in conjunction with the basic input and output commands, for analyzing

or converting data that is read or written. The **StringTools** package includes numerous commands; for brevity, we will describe only a few commands that may be of interest to users performing input/output operations in Maple. These include commands for

- converting the case of characters (*e.g.*, StringTools:-LowerCase)
- performing character class tests (*e.g.*, StringTools:-HasDigit)
- comparing strings (*e.g.*, StringTools:-IsPrefix)
- doing pattern-matching and text searching (*e.g.*, StringTools:-Substitute)
- handling whitespace (*e.g.*, StringTools:-TrimRight)

Two commands that are relevant to file I/O are StringTools:-Compress and StringTools:-Uncompress. The first command uses an algorithm from the zlib library to compress the input into a lossless and more compact format, while the second reverses the process. These commands are compatible with the commands for reading and writing bytes described in *Input and Output with Files (page 367)*.

For more information about the zlib library, visit **http://www.zlib.net**.

Conversion Commands

Some additional commands may be useful when you are performing input and output operations in Maple.

- The convert/bytes help page shows how to transform strings into bytes using the **convert** command.
- The parse command allows you to parse a string as a Maple statement. For example, the following command parses the given string, evaluates it, and returns the expression 4*x^2.

```
> parse("x^2+3*x^2");
```

$$4\,x^2 \tag{10.24}$$

10.7 2-D Math

Introduction

Typeset or 2-D math is available with the standard worksheet interface. Normally, input and output of 2-D math is done interactively using the Maple GUI tools. However, certain aspects of the input and output can be controlled programmatically in a limited way.

There are two available modes for typesetting: standard and extended. The mode can be changed by using the interface command. The following command shows the current setting in your worksheet or document:

```
> interface(typesetting);
```

standard (10.25)

Standard typesetting uses default rules for displaying expressions. With extended typesetting, the rules can be customized using the Typesetting Rule Assistant (TypesettingRuleAssist) or exports from the Typesetting package. The Typesetting Rules Assistant and the **Typesetting** package exports can also be used to adjust how 2-D input is parsed, regardless of the typesetting mode used for output.

The Typesetting Package

The Typesetting package provides commands for programmatically customizing extended typesetting output in certain situations and for controlling how particular 2-D expressions are parsed. It also includes internal-use commands that are not intended for general use. Additionally, the package exports a number of names that act as Maple typesetting tags similar to MathML tags.

The commands available to users are described on the Typesetting help page. A subset of the commands are listed below:

- Typesetting:-Settings: adjust general extended typesetting settings, such as whether dot notation for derivatives is used and whether functions such as $2(x)$ should be interpreted as implicit multiplication.
- Typesetting:-Suppress: suppress dependencies of functions (so that f can be interpreted as $f(x)$, for example).
- Typesetting:-EnableTypesetRule, Typesetting:-EnableParseRule and Typesetting:-EnableCompletionRule: control specific typesetting, parsing and command-completion rules.
- Typesetting:-UseSymbolForTypeset: control the display of operator symbols.

Extended typesetting output is produced by the **Typeset** command When this command is called, an unevaluated function is returned. This output, which is recognized by the Maple GUI, is not intended to be altered by users. Because the structure is meant for internal use, the tag names and format of the structure may change from one Maple release to another.

```
> lprint(Typesetting:-Typeset(BesselJ(v, x)));
Typesetting:-mrow(Typesetting:-msub(Typesetting:-mi("J", fontstyle =
 "normal", Typesetting:-msemantics = "BesselJ"), Typesetting:-mi("v")),
 Typesetting:-mo("&ApplyFunction;"),
Typesetting:-mfenced(Typesetting:-mi("x")))
```

Additional Tips

- Users are discouraged from manipulating the typesetting structures created for internal use. However, in rare circumstances, it may be useful to call the **Typesetting:-Typeset** command. For example, standard typesetting mode is generally used for typeset text in plots. Extended typesetting output produced by the **Typeset** command may be passed to plays inside the **typeset** structure. For more information, see *???*.
- Occasionally, you may find it necessary to manipulate a typeset expression programmatically without having the expression evaluate. For example, you want to print $\frac{1}{2}+\frac{1}{3}$ without having it evaluate to $\frac{5}{6}$, or you want to use $x+$, which gives an error when evaluated in Maple. In theses situations, it is useful to create an atomic identifier. To do this, you must be working in the standard worksheet interface. Enter the expression in the input line, select it and then use the 2-D math context menu to convert to an atomic identifier. If you **lprint** the result, you will see a name (such as **`#mrow(mi("x"),mo("+"))`**, for $x+$) that can now be used within a Maple program written in 1-D math.

10.8 Exercises

1. Write a loop (with a single statement in its body) that prints strings listing the cubes of the integers **1** to **10**.
2. Create a file in a text editor that contains the following lines.

```
x := 1;           # valid input line
if := 2;}         # invalid assignment
y := 3;           # valid input line
two words := 4;   # invalid assignment
```

   Save the file. In a Maple session, open the file by using the read statement. Observe how Maple reacts to invalid statements.
3. Create a data file in a text editor that contains the following information.

```
1   2   3
4   5   6
```

Save the file. Read this file into a Maple session, convert the data to a list, and reverse its order. Write the reversed data in the same format to a different file.

11 Writing Packages

This section describes how to collect a large software project in Maple into a *package* that is easy to maintain. Packages can be configured to load automatically when you start Maple and distributed to other users as a library rather than as Maple source code.

11.1 In This Chapter

- What is a package
- Writing Maple packages by using modules
- Examples of custom packages

11.2 What Is a Package

A *package* is a collection of procedures and other data that can be treated as a whole. Packages typically gather a number of procedures that enable you to perform computations in a well-defined problem domain. Packages may contain data other than procedures, and may even contain other packages (subpackages).

Packages in the Standard Library

A number of packages are shipped with the standard Maple library. For example, the **group**, **numtheory**, **CodeGeneration**, and **LinearAlgebra** packages are all provided with Maple, along with several dozen others. The **group** package provides procedures for computing with groups that have a finite representation in terms of permutations, or of generators and defining relations. The **LinearAlgebra** package provides numerous procedures for computational linear algebra.

Packages Are Modules

Modules are the implementation vehicle for packages. A module represents a package by its exported names. The exported names can be assigned arbitrary Maple expressions, typically procedures, and these names form the package.

For more information about modules, see *Programming with Modules (page 307)*.

Some older and deprecated Maple packages such as **simplex** and **networks** are not implemented using modules; they are implemented using tables. In table-based packages, the name of a package command is used as the index into a table of procedures. It is not recommended to write new packages using tables since modules allow much more flexibility.

Package Exports

Some of the data in a package is normally made accessible to the user as an export of the package. For packages implemented as modules, the package exports are the same as the exports of the underlying module. For packages implemented as tables, the package exports are the names used to index the underlying table.

Accessing the exports of a package is a fundamental operation that is supported by all packages. If **P** is a Maple package, and **e** is one of its exports, you can access **e** by using the fully qualified reference **P[e]**. If **P** is a module, you can also use the syntax **P:-e**. These methods of accessing the exports of a module are normally used when programming with a package.

Note that the member selection operator (**:-**) is left-associative. If **S** is a submodule of a module **P**, and the name **e** is exported by **S**, then the notation **P:-S:-e** is parsed as **(P:-S):-e**, and so it refers to the instance of **e**, which is local to **S**. This concept is important for referencing members of subpackages. For example,

```
> CodeTools:-Profiling:-Coverage:-Print();
```

calls the procedure **Print** in the subpackage **Coverage** in the subpackage **Profiling**, which is part of the **CodeTools** package. You can use indexed notation for this.

```
> CodeTools[Profiling][Coverage][Print]();
```

Using Packages Interactively

For interactive use, it is inconvenient to enter fully qualified references to all of the exports of a package. To facilitate the process of entering package command names, the Maple procedure with is provided for the interactive management of package namespaces. By using with, you can globally impose the exported names of a package. This allows you to access the package exports, without typing the package prefix, by making the names of the exports accessible at the top level of the Maple session. For example, to use the **numtheory** package, enter the command

```
> with( numtheory );
```

$$[GIgcd, bigomega, cfrac, cfracpol, cyclotomic, divisors, factorEQ, factorset, fermat, imagunit, index, integral\_basis, invcfrac, invphi, iscyclotomic, issqrfree, ithrational, jacobi, kronecker, \lambda, legendre, mcombine, mersenne, migcdex, minkowski, mipolys, mlog, mobius, mroot, msqrt, nearestp, nthconver, nthdenom, nthnumer, nthpow, order, pdexpand, \phi, \pi, pprimroot, primroot, quadres, rootsunity, safeprime, \sigma, sq2factor, sum2sqr, \tau, thue] \quad (11.1)$$

This command makes the names exported by the **numtheory** package (a list of which is returned by the call to with) available temporarily as top-level Maple commands.

```
> cfrac( ( 1 + x )^k, x, 5, 'subdiagonal', 'simregular' );
```

$$\cfrac{1}{1-\cfrac{k\,x}{1+\cfrac{1}{2}\cfrac{(k+1)\,x}{1-\cfrac{1}{6}\cfrac{(k-1)\,x}{1+\cfrac{1}{6}\cfrac{(k+2)\,x}{1+\ldots}}}}} \quad (11.2)$$

11.3 Writing Maple Packages By Using Modules

A Simple Example

The simplest type of package is a collection of related procedures that are bundled together for convenience, for example, PolynomialTools or ArrayTools are packages which are both included with Maple. The following is an example of a custom package.

```
> SomeTools := module()
  description "Some useful tools";
  option package;
  export axpy, sqrm1, identity;

      axpy := proc(a::algebraic, x::algebraic, y::algebraic, $)
      description "compute a times x plus y";
          return a*x + y;
      end proc; # axpy
```

```
    identity := proc()
    description "return the arguments";
        return _passed;
    end proc; # axpy

    sqrm1 := proc(x::algebraic, $)
    description "square minus one";
        return x^2 - 1;
    end proc;

end module; # Some Tools
```

```
SomeTools := module( )
    option package;
    export axpy, sqrm1, identity;
    description "Some useful tools";

end module
```
(11.3)

This example is simply a module that consists of a few exported members that are procedures and an option called **package**. As with all modules, its members can be accessed by using the **:-** and **[]** operators.

```
> SomeTools:-axpy(1,1,1);
```

$$2 \tag{11.4}$$

```
> SomeTools[sqrm1](2);
```

$$3 \tag{11.5}$$

The **package** option also allows you to call the package by using the with command to access the package exports.

```
> with(SomeTools);
```

$$[axpy, identity, sqrm1] \tag{11.6}$$

```
> identity("nothing");
```

$$\text{"nothing"} \tag{11.7}$$

If the with command is used to call a module that does not include the **package** option, an exception will occur:

```
> SomeOtherTools := module() export Nothing; Nothing := x->x; end
  module;
```

$$\begin{aligned} &\mathit{SomeOtherTools} := \mathbf{module}(\,) \\ &\quad \mathbf{export}\ \mathit{Nothing}; \\ &\mathbf{end\ module} \end{aligned} \tag{11.8}$$

```
> with(SomeOtherTools);
```

```
Warning, SomeOtherTools is not a correctly formed package - option
`package' is missing
```

$$[\mathit{Nothing}] \tag{11.9}$$

Packages generally include many lines of Maple code, so you will probably want to create and modify them in a specialized editor designed for programming such as *vim* or *emacs*. In the example above, the definition of the module SomeTools have been put in a file called **SomeTools.mpl** in the **samples/ProgrammingGuide/** directory of your Maple installation). If you copy this file into the current directory, it can then be loaded in Maple by using the read command.

```
> read("SomeTools.mpl");
```

This allows you to access the **SomeTools** package commands in a Maple worksheet or document. You can include a **read** statement as the first executed command or in the startup code of a Maple worksheet or document.

Custom Libraries

If you prefer not to call the read command to load your custom package, you can save your package as a Maple library archive (.mla) file. This allows your package to be available whenever Maple is started; however, it is loaded into memory only if it is required. In contrast, the read method automatically loads the package into memory.

Before a package can be saved to a library archive, it must be loaded into Maple, either directly in a worksheet or by using the read command. Then, the simplest method for saving it to a library archive is to call the savelib command.

Before calling the **savelib** command, you may want to set up a directory to store your custom library. For example, you can create a directory called **maple** in your home directory and use the LibraryTools commands to create an empty .mla file in which to store your library.

```
> mylibdir := cat(kernelopts(homedir), kernelopts(dirsep), "maple",
   kernelopts(dirsep), "toolbox", kernelopts(dirsep), "personal",
  kernelopts(dirsep), "lib");
```

"C:\Documents and Settings\juser\maple\toolbox\personal\lib" (11.10)

```
> FileTools:-MakeDirectory(mylibdir, 'recurse');
> LibraryTools:-Create(cat(mylibdir, kernelopts(dirsep),
  "packages.mla"));
> libname := mylibdir, libname;
```

libname := "C:\Documents and Settings\juser\maple\toolbox\personal\lib", "C:\Program Files\Maple 15\lib" (11.11)

To save a new value for **libname** (a predefined variable, which specifies the location of the main Maple library, and package and user libraries), and to make sure that this directory is the default location for saving in the future after you use the **restart** command, add the line above to your Maple initialization file, which specifies initialization settings for Maple.

For more information on initialization files, see the worksheet/reference/initialization help page.

Note: Maple automatically adds **lib** subdirectories of directories in **HOMEDIR/maple/toolbox** to the predefined variable **libname**. Therefore, modifying the **.mapleinit** or **maple.ini** file is only necessary if you want to designate this directory as the default location in which the **savelib** command will save your library files.

You can modify the Maple initialization file manually in a text editor or by using the FileTools commands in Maple:

```
> mapleinitfile := cat(kernelopts(homedir), kernelopts(dirsep),
     `if`(kernelopts(platform)="unix", ".mapleinit", "maple.ini"));
```

"C:\Documents and Settings\juser\maple.ini" (11.12)

```
> FileTools:-Copy(mapleinitfile, cat(mapleinitfile, ".mpl.bak"));
> FileTools:-Text:-Open( mapleinitfile, 'append');
> FileTools:-Text:-WriteLine(mapleinitfile, cat("libname :=
  \"",mylibdir,"\", libname:"));
```

```
> FileTools:-Text:-Close( mapleinitfile );
```

Restart, and verify that your libname is set now correctly.

```
> restart;
```

```
> libname;
```

Finally, save the package to the library archive by calling the savelib command.

```
> savelib( 'SomeTools' );
```

Enter the restart command, followed by the ShowContents command in LibraryTools to verify that the package has been added to the library archive. If everything has worked correctly, you can now use the with command to access the package commands.

```
> restart;
```

```
> LibraryTools:-ShowContents(libname[1]);
```

```
> with(SomeTools);
```

Running the savelib command saves the module to the first library archive found in the path specified by the global variable **libname** or the library archive in the path specified by the global variable **savelibname**, if it is defined. (At least one of these values must be defined.) You can save your package to a different library archive by using the LibraryTools:-Save command or by providing a file name as a second argument to the savelib command.

You will want to remove this example package from your library when you are done. This can be done using the Delete command in the LibraryTools package.

```
> LibraryTools:-Delete('SomeTools', libname[1]);
```

You can confirm that it has been deleted.

```
> LibraryTools:-ShowContents(libname[1]);
```

Important: Always make sure that the standard Maple library directory is write-protected to avoid saving expressions in it. If you accidentally save a file to the standard Maple library, you may need to reinstall Maple to restore the main library archive.

11.4 A Larger Example

Several additional techniques are useful for writing larger packages. Some of these techniques will be described in the context of an example package called **RandomnessTests** whose full source can be found in the **samples/ProgrammingGuide/RandomnessTests** directory, which is located in the directory where Maple is installed. It is a package containing procedures to analyze the randomness of a sequence of bits.

ModuleLoad

Often, a package needs to initialize the internal or global state when it is loaded. For example, many packages define new types for the type system. Generally, these types are not needed unless the package is loaded, and so they are created by the **ModuleLoad** local member of the package. In this example, we will define a type **BinarySequence** that is a linear data type containing only zeros and ones.

```
> ModuleLoad := proc()
       TypeTools:-AddType
        (
            ':-BinarySequence',
            proc(L)
                type(L, list({identical(0),identical(1)}))
                or type(L, 'Vector'({identical(0),identical(1)}))
                or ( type(L, 'Array'({identical(0),identical(1)}))
                    and nops([rtable_dims(L)])=1
                    and op([1,1], [rtable_dims(L)])=1 );
            end proc
        );
    end proc;
```

The local named **ModuleLoad** will automatically run when a module is loaded from a library. If you are using the **read** command to load the definition of the module into Maple or if you enter it in a worksheet, this procedure will not run automatically.

You can also define the local **ModuleUnload**, which will run if the module is removed from memory because it is not being used. In this case, the custom type definition **BinarySequence** is removed.

```
> ModuleUnload := proc()
         TypeTools:-RemoveType(':-BinarySequence');
     end proc;
```

It is also possible to use the **load=proc** and **unload=proc** options in the module definition to specify different procedures to be invoked when a package is loaded and unloaded respectively. However, the use of the **ModuleLoad** and **ModuleUnload** procedures is recommended.

The Preprocessor and Structured Source Files

If a package has many exports, it often useful to put each export into its own source file. The Maple preprocessor is similar to the preprocessor of a C compiler. It allows you to specify the names of source files and macro definitions to include in a master .mpl file, which defines the contents of your package.

To include the file name of an export in an .mpl file, you can use the **$include** directive.

To define macros, you can use the **$define** directive.

In our example, we have a file called RandomnessTests.mpl, which has several exports stored in the same directory in the files: WaldWolfowitz.mm, BitFrequency.mm, Compressibility.mm, BinaryRank.mm. The following example shows the contents of the RandomnessTests.mpl file.

Table 11.1: RandomnessTests

```
##MODULE RandomnessTests
##
##DESCRIPTION
##- A package containing commands for testing the randomness of
## binary sequences.

$define RANDINFO ':-RandomnessTests'
$define MAINLEVEL 2

RandomnessTests := module()
option package;

export
    WaldWolfowitz,
    BitFrequency,
    Compressibility,
    BinaryRank;

local
    Runs,
    ModuleLoad,
    ModuleUnload;

    ModuleLoad := proc()
       TypeTools:-AddType
        (
            ':-BinarySequence',
            proc(L)
                type(L, list({identical(0),identical(1)}))
                or type(L, 'Vector'({identical(0),identical(1)}))
                or ( type(L, 'Array'({identical(0),identical(1)}))
                    and nops([rtable_dims(L)])=1
                    and op([1,1], [rtable_dims(L)])=1 );
            end proc
        );
    end proc;
```

```
    ModuleUnload := proc()
        TypeTools:-RemoveType(':-BinarySequence');
    end proc;

$include "WaldWolfowitz.mm"
$include "BitFrequency.mm"
$include "Compressibility.mm"
$include "BinaryRank.mm"

end module; # RandomnessTests
```

When this file is loaded into Maple from the command-line interface or by using the **read** command in a Maple worksheet or document, Maple automatically replaces each **$include** directive with the contents of the file specified.

The file also includes two **$define** directives which are macros that are not used in this file, but will be used in the included files. Including them in the top-level source file allows us to make package-wide changes by editing the macros in one place. In our example, these macros will be used to control the userinfo definitions throughout the package. The following line appears in the WaldWolfowitz.mm file:

```
userinfo(MAINLEVEL, RANDINFO, nprintf("sequence has %d runs", runs));
```

When loaded, the preprocessor will transform this line into the following:

```
> userinfo(2, ':-RandomnessTests', nprintf("sequence has %d runs",
    runs));
```

Macros can also have parameters such as the following macro in the BinaryRank.mm file:

```
$define GF2RankProbability(m,n,r)
2^(r*(n+m-r)-m*n)*mul(((1-2^(1-n))*(1-2^(1-m)))/(1-2^(1-r)),1=0..(r-1))
```

which allows **GF2RankProbability** to be used as if it were a procedure. However, it will be replaced inline by the preprocessor. This is similar to how procedures with option inline function, but with more restrictions.

Subpackages

When creating large packages, it is useful to organize commands into smaller subpackages. Our package does this with a **Visualization** subpackage and a **Data** submodule to store sample random inputs.

Achieving this is as simple as including the definition for these other modules and packages within the top-level package.

11.5 Example: A Shapes Package

In this section, a sample package is presented to illustrate concepts that may be helpful when working with modules and submodules, and putting them together into a package.

Modules allow you to create packages with a hierarchical structure; this cannot be done with table-based implementations of packages. This section covers the following topics:

- Organizing the source code for a large package that has a nontrivial substructure.
- A description of the **Shapes** package, including details of its design and implementation
- Hints related to source code organization.

The package presented in this section provides the means to compute areas and circumferences of various planar figures, which are called shapes.

> **Note:** Only portions of the source code for this package are shown. The fully commented source code can be found in the **samples/ProgrammingGuide/shapes** directory of your Maple installation.

Source Code Organization

The **Shapes** package is organized into several source files:

- shapes.mpl
- point.mm
- segment.mm
- circle.mm
- square.mm
- triangle.mm

To avoid platform-specific differences, all of the source files are located in the same directory or folder.

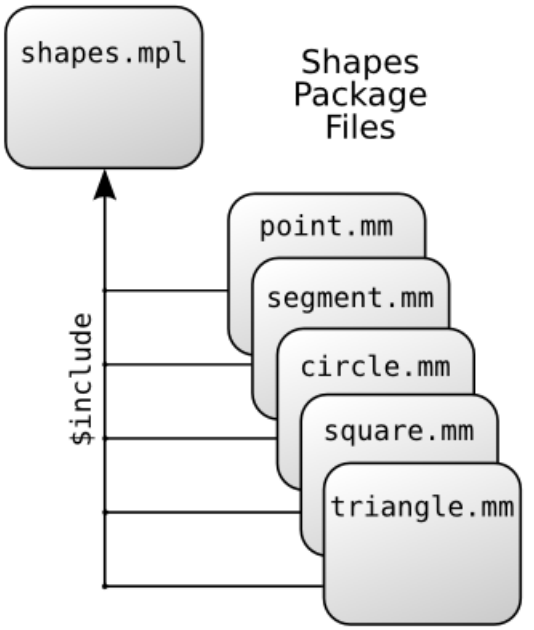


Figure 11.1: Organization of Package Source Files

To define the module that implements this package, use the Maple preprocessor to include the remaining source files at the appropriate point in the master source file shapes.mpl. A number of **$include** directives are included in shapes.mpl, such as

```
$include        "point.mm"
$include        "segment.mm"
```

Splitting a large project into several source files makes it easier to manage and allows several developers to work on a project at the same time. The source file is divided into shape-specific functionality. Most of the functionality for points, for instance, is implemented by the source code stored in the point.mm file.

Package Architecture

The **Shapes** package is structured as a module with several exported procedures. Individual submodules provide specific functionality for each shape type supported by the package. Each of these shape-specific submodules is stored in its own source file; these files are included into the main package source file, shapes.mpl.

The package module **Shapes** has a submodule, which is also called **Shapes**. The submodule **Shapes:-Shapes** contains one submodule for each shape supported. The submodule hierarchy is illustrated in **Figure 11.2**.

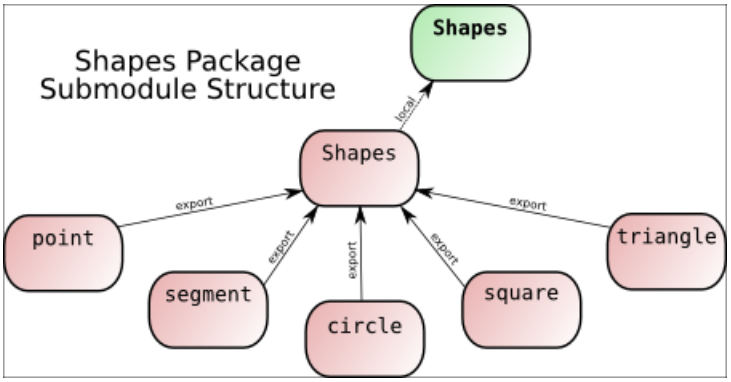


Figure 11.2: Design of Package

The result of preprocessing the main file shapes.mpl produces a module whose source has the following general form.

```
Shapes := module()
     option package;
     export make, area, circumference;
     local Shapes, circum_table;
     Shapes := module()
         export point, segment, circle, square, triangle;
         point := module() ... end module;
         segment := module() ... end module;
         .....
     end module;
     make := proc() ... end proc;
     area := proc() ... end proc;
     circum_table := table(); ...
     circumference := proc() ... end proc;
 end module:
```

The Package API

The **Shapes** package exports the following procedures:

- **make**
- **area**
- **circumference**

The make Procedure

The exported procedure **make** creates shapes. It is used to create a shape expression from the input data. For example, points are derived from their x and y coordinates.

```
> org := make( 'point', 0, 0 );
```

$$org := make(point, 0, 0) \tag{11.13}$$

A circle is created from its center and radius.

```
> circ := make( 'circle', org, 2 );
```

$$circ := make(circle, make(point, 0, 0), 2) \tag{11.14}$$

In each case, the name of the shape is passed as the first argument to specify which kind of shape to return.

The area Procedure

To compute the area of a shape, call the exported procedure **area** with the shape as its argument.

```
> area( circ );
```

$$area(make(circle, make(point, 0, 0), 2)) \tag{11.15}$$

The circumference Procedure

The exported procedure **circumference** computes the circumference of a given shape.

```
> circumference( circ );
```

$$circumference(make(circle, make(point, 0, 0), 2)) \tag{11.16}$$

Shape Representation

Shapes are represented as unevaluated function calls. The arguments to the call are the instance-specific data for the shape. For example, a point with coordinates (2,3) is represented by the unevaluated function call **POINT(2, 3)**. Some instance data are shapes themselves. For example, a segment is represented, using its endpoints, as an unevaluated function call of the form **SEGMENT(start_point, end_point)**. The start and end points of the segment can be obtained by calling the point constructor.

Procedure Dispatching

The **Shapes** package illustrates three types of procedure dispatching.

- Dispatching on submodule exports
- Conditional dispatching
- Table-based dispatching

Dispatching on Submodule Exports

The **make** procedure, which is exported from the **Shapes** package, uses the **Shapes:-Shapes** submodule for procedure dispatching.

To test whether a method for a given shape is available, the **make** procedure tests whether there is a submodule by that name in the **Shapes:-Shapes** submodule. If no such submodule is found, an exception is raised. Otherwise, the **make** export from the submodule is passed the arguments that were given to the top-level **Shapes:-make** procedure. The **make** source code is as follows.

```
> make := proc( what::symbol )
           description "constructor for shapes";
           local   ctor,     # the shape constructor,
                             # if found
                   theShape; # the submodule for the
                             # kind of shape requested

           if not member( what, Shapes, 'theShape' ) then
                   error "shape `%1' not available", what
           end if;
           if member( ':-make', theShape, 'ctor' ) then
                   ctor( args[ 2 .. nargs ] )
           else
                   error "no constructor provided for "
                         "shape %1", what
           end if
   end proc:
```

The first argument to **make** is a symbol that specifies the shape to create (**point**, **circle**, **triangle**). This symbol is used as an index in the **Shapes:-Shapes** submodule. The first statement uses the member command to test whether the symbol passed in the **what** parameter is exported by the **Shapes:-Shapes** submodule. If it is not found, an appropriate diagnostic is issued and an exception raised. If member returns the value **true**, then its third argument, the local variable **theShape**, is assigned the export found in the submodule.

For example, if **what** is the symbol **circle**, then the local variable **theShape** is assigned the submodule **Shapes:-Shapes:-circle** that implements operations on circles. The same idea is used to select the shape-specific constructor; it is the value assigned to the local variable **ctor** when the value **true** is returned from the second call to the member command. Any remaining arguments are used as data to construct the shape. These are passed to the **make**

export in a shape-specific submodule, if found, and are not checked further at this level. This design localizes the shapes to the corresponding submodule.

Conditional Dispatching

The procedure **area** uses a simple conditional dispatching mechanism. The tag of the input shape is extracted and is used in direct comparisons with hard-coded values to determine which shape-specific **area** subcommand to call to perform the area computation.

```
> area := proc( shape )
       description "compute the area of a shape";
       local   tag;

       if not type( shape, 'function' ) then
           error "expecting a shape expression, "
                     "but got %1", shape
       end if;

       # Extract the "tag" information from the shape
       tag := op( 0, shape );

      # Dispatch on the "tag" value
       if tag = ':-POINT' then
           Shapes:-point:-area( shape )
       elif tag = ':-SEGMENT' then
           Shapes:-segment:-area( shape )
       elif tag = ':-CIRCLE' then
           Shapes:-circle:-area( shape )
       elif tag = ':-SQUARE' then
           Shapes:-square:-area( shape )
       elif tag = ':-TRIANGLE' then
           Shapes:-triangle:-area( shape )
       else
           error "not a recognized shape: %1", tag
       end if
   end proc:
```

Table-based Dispatching

The third dispatch method illustrated in the **Shapes** package is table-based. This technique is used by the exported procedure **circumference**, which references the table **circum_table** to look up the appropriate procedure to call. This table is built by assigning its entries in the body of the **Shapes** package.

```
> circum_table := table();
> circum_table[ 'POINT' ] := Shapes:-point:-circumference;
> circum_table[ 'SEGMENT' ] := Shapes:-segment:-circumference;
> circum_table[ 'CIRCLE' ] := Shapes:-circle:-circumference;
> circum_table[ 'SQUARE' ] := Shapes:-square:-circumference;
> circum_table[ 'TRIANGLE' ] := Shapes:-triangle:-circumference;
```

The source code for the procedure **circumference** is as follows.

```
> circumference := proc( shape )
      description "compute the circumference of a "
                  "shape expression";
      if not type( shape, 'function' ) then
          error "expecting a shape, but got %1", shape
      end if;
      if assigned( circum_table[ op( 0, shape ) ] ) then
          circum_table[ op( 0, shape ) ]( shape )
      else
          error "no circumference method available "
                    "for shape %1. Supported shapes "
                    "are: %2", tag,
                        sprintf( "%q", op( ALL_SHAPES ) )
      end if
   end proc:
```

Minimal checking is done to ensure that the input has the right structure. If an entry is found in the table **circum_table** for the shape tag (as with the **area** procedure), the corresponding procedure is called with the given shape as an argument. (The shape must be passed as an argument, so that the shape-specific submodule can extract the instance data from it.) Otherwise, a diagnostic is issued and an exception is raised.

Shape-specific Submodules

As already noted, each shape is implemented in a shape-specific submodule. The set of exports of each module varies, but each supports the required exports **make**, **area**, and **circumference** in the top-level **Shapes** module. Certain shapes support other operations. Only two submodules are described here. You can see the source for the other submodules in the sample source code.

The point Submodule

The submodule that implements points is fairly simple. In fact, it makes no reference to any lexically scoped variables in its parent modules (**Shapes** and **Shapes:-Shapes**).

```
> point := module()
      description "support commands for points";
      export make, area, circumference, xcoord, ycoord;
      option package;

      make := ( x, y ) -> 'POINT'( x, y );
      area := () -> 0;
      circumference := () -> 0;
      xcoord := p -> op( 1, p );
      ycoord := p -> op( 2, p );
   end module:
```

Since the area and circumference of a point are both 0, these procedures are easy to implement. In addition to the required exports, the **point** submodule also exports two utility procedures, **xcoord** and **ycoord**, for retrieving the x and y coordinates of a point. Providing these values makes it possible for clients of this submodule to use it without requiring information about the concrete representation of points. This makes it easier to change the representation later, if required.

Within this submodule, the names **make**, **area**, and **circumference** are the same as the names with the same external representation at the top-level **Shapes** module.

The circle Submodule

This submodule provides the circle-specific commands for the **Shapes** package.

```
> circle := module()
      export  make, center, radius, diameter,
              area, circumference;
      option package;

      make := proc( cntrPt, radius )
              'CIRCLE'( cntrPt, radius )
      end proc;

      center := circ -> op( 1, circ );
      radius := circ -> op( 2, circ );
      diameter := circ -> 2 * radius( circ );
      circumference := circ ->  Pi * diameter( circ );
      area := circ -> Pi * radius( circ )^2;
   end module:
```

Again, some extra commands are provided in addition to those required at the top level of the **Shapes** package. The exported procedure **radius** is used to define other commands. It can be made local to this submodule.

Appendix A Internal Representation

The table below lists the structures that are currently implemented in Maple.

Each structure, along with the constraints on its length and contents, is described in the sections that follow.

Table A.1: Maple Structures

AND	ASSIGN	BINARY	BREAK	CATENATE
COMPLEX	CONTROL	DCOLON	DEBUG	EQUATION
ERROR	EXPSEQ	FLOAT	FOR	FOREIGN
FUNCTION	GARBAGE	HASH	HASHTAB	HFLOAT
IF	IMPLIES	INEQUAT	INTNEG	INTPOS
LESSEQ	LESSTHAN	LEXICAL	LIST	LOCAL
MEMBER	MODDEF	MODULE	NAME	NEXT
NOT	OR	PARAM	POLY	POWER
PROC	PROD	RANGE	RATIONAL	READ
RETURN	RTABLE	SAVE	SERIES	SET
SDPOLY	STATSEQ	STOP	STRING	SUM
TABLE	TABLEREF	TRY	UNEVAL	USE
XOR	ZPPOLY			

A.1 Internal Functions

The internal functions in Maple are divided into five groups:

Evaluators

The evaluators are the main functions responsible for evaluation. There are six types of evaluations: statements, algebraic expressions, Boolean expressions, name forming, arbitrary precision floating-point arithmetic, and hardware floating-point arithmetic. The user interface calls only the statement evaluator, but thereafter there are many interactions between evaluators. For example, the statement

```
if a > 0 then b||i := 3.14/a end if;
```

is first analyzed by the statement evaluator, which calls the Boolean evaluator to resolve the **if** condition. Once completed (for example, a **true** result is returned), the statement evaluator is invoked again to perform the assignment, for which the name-forming evaluator is invoked with the left-hand side of the assignment, and the expression evaluator

with the right-hand side. Since the right-hand side involves floating-point values, the expression evaluator calls the arbitrary precision floating-point evaluator.

Normally, you do not specifically call any of the evaluators. However, in some circumstances, when a nondefault type of evaluation is needed, you can directly call **evalb** (the Boolean evaluator), **evaln** (the name-forming evaluator), **evalf** (the arbitrary precision floating-point evaluator), or **evalhf** (the hardware floating-point evaluator).

Algebraic Functions

Algebraic functions are commonly called *basic functions*. Some examples are taking derivatives (**diff**), dividing polynomials (**divide**), finding coefficients of polynomials (**coeff**), computing series (**series**), mapping a function (**map**), expanding expressions (**expand**), and finding indeterminates (**indets**).

Algebraic Service Functions

These functions are algebraic in nature, but serve as subordinates of the functions in the previous group. In most cases, these functions cannot be explicitly called. Examples of such functions are the internal arithmetic packages, the basic simplifier, and retrieval of library functions.

Data Structure Manipulation Functions

These are similar to the algebraic functions, but instead of working on mathematical objects, such as polynomials or sets, they work on data structures, such as expression sequences, sums, products, or lists. Examples of such functions are operand selection (**op**), operand substitution (**subsop**), searching (**has**), and length determination (**length**).

General Service Functions

Functions in this group are at the lowest hierarchical level. That is, they can be called by any other function in the system. They are general purpose functions, and not necessarily specific to symbolic or numeric computation. Some examples are storage allocation and garbage collection, table manipulation, internal I/O, and exception handling.

A.2 Flow of Control

The flow of control does not need to remain internal to the Maple kernel. In many cases, where appropriate, a decision is made to call functions that are written in Maple and are a part of the Maple library. For example, many uses of the **expand** function are handled in the kernel. However, if an expansion of a sum to a large power is required, the internal **expand** function calls the external Maple library function **'expand/bigpow'** to resolve it. Functions such as **diff**, **evalf**, **series**, and **type** make extensive use of this feature.

Therefore, for example, the basic function **diff** cannot differentiate any function. All of that functionality is included in the Maple library in procedures named **'diff/functionName'**. This is a fundamental feature of Maple since it permits:

- Flexibility (the ability to change the Maple library)
- Customization (by defining your refined handling functions)
- Readability (much of the Maple functionality is visible at the user level)

Maple allows the kernel to remain small by offloading nonessential functions to the library.

A.3 Internal Representations of Data Types

The parser and some internal functions build all of the data structures used internally by Maple. All of the internal data structures have the same general format:

Header	$Data_1$	...	$Data_n$

The header field, stored in one or more machine words, encodes the length of the structure and its type. Additional bits are used to record simplification status, garbage collection information, persistent store status, and various information about specific data structures (for example, whether a **for** loop contains a **break** or **next** statement).

The length is encoded in 26 bits on 32-bit architectures, resulting in a maximum single object size of 67,108,863 words (268,435,452 bytes, or 256 megabytes). On 64-bit architectures, the length is stored in 32 bits, for a maximum object size of 4,294,967,295 words (34,359,738,360 bytes or 32 gigabytes).

Every structure is created with its own length, and that length does not change during the existence of the structure. Furthermore, the contents of most (but not all) data structures are never changed during execution because it is unpredictable how many other data structures are referring to them and relying on them not to change. The normal process for modifying a structure is to copy it and then to modify the copy. Structures that are no longer used are eventually reclaimed by the garbage collector.

The following sections describe each of the structures currently implemented in Maple, along with the constraints on their lengths and contents. The 6-bit numeric value identifying the type of structure is of little interest, so symbolic names will be used.

The notation ^something in the data structure depictions indicates that the value stored in that field of the structure is a pointer to the value (something), rather than being the something itself.

AND: Logical AND

AND	^expr1	^expr2

Maple syntax: **expr1 and expr2**

Length: 3

ASSIGN: Assignment Statement

ASSIGN	^name-seq	^expr-seq

Maple syntax: **name1, name2, ... := expr1, expr2, ...**

Length: 3

The left-hand side *name* entries must evaluate to assignable objects: **NAME**, **FUNCTION**, **MEMBER** or **TABLEREF** structures, or a sequence thereof. If the left-hand side is a sequence, the right-hand side must be a sequence of the same length.

BINARY: Binary Object

BINARY	data	...

Maple syntax: none

Length: arbitrary

The **BINARY** structure can hold any arbitrary data. It is not used directly as a Maple object, but is used as storage for large blocks of data within other Maple objects (currently only **RTABLE** structures). It is also sometimes used as temporary storage space during various kernel operations.

BREAK: Break Statement

BREAK

Maple syntax: **break**

Length: 1

CATENATE: Name Concatenation

CATENATE	^name	^expr

Maple syntax: name || expr

Length: 3

- If the *name* entry is one of **NAME**, **CATENATE**, **LOCAL**, or **PARAM**, and if the *expr* entry evaluates to an integer, **NAME**, or **STRING**, the result is a **NAME**.
- If the *name* entry is a **STRING** or **CATENATE** that resolves to a **STRING**, and if the *expr* entry evaluates to an integer, **NAME**, or **STRING**, the result is a **STRING**.

- If *expr* is a **RANGE**, the result is to generate an **EXPSEQ** of the **NAME** or **STRING** structures.

COMPLEX: Complex Value

COMPLEX	^re	^im
COMPLEX	^im	

Maple syntax: Complex(re,im), Complex(im), re + im * I or im * I

Length: 2 or 3

The *re* and *im* fields must point to **INTPOS**, **INTNEG**, **RATIONAL**, or **FLOAT** structures, one of the **NAME**s **infinity** or **undefined**, or a **SUM** structure representing **-infinity**. In the length 3 case, if either **re** or **im** is a **FLOAT**, the other must be a **FLOAT** as well.

CONTROL: Communications Control Structure

CONTROL	^integer

Maple syntax: none

Length: 2

This is an internal structure used for communication between the kernel and user interface. Such a structure never reaches the user level, or even the mathematical parts of the kernel.

DCOLON: Type Specification or Test

DCOLON	^expr	^type-expr

Maple syntax: expr :: typeExpr

Length: 3

This structure has three interpretations depending on the context in which it is used. When it appears in the header of a procedure definition, it is a parameter declaration that has a type. When it appears in the **local** section of a procedure or on the left-hand side of an assignment, it is a type assertion. When it appears elsewhere (specifically, in a conditional expression), it is a type test.

DEBUG: Debug

DEBUG	^expr1	^expr2	...

Maple syntax: none

Length: 2 or more

This is another structure that is only used internally. It is used by the kernel when printing error traceback information to transmit that information up the call stack.

EQUATION: Equation or Test for Equality

EQUATION	^expr1	^expr2

Maple syntax: expr1 = expr2

Length: 3

This structure has two interpretations depending on the context in which it is used. It can be either a test for equality, or a statement of equality (not to be confused with an assignment).

ERROR: Error Statement

ERROR	^expr

Maple syntax: **error "msg"**, **arg**, ... **arg**

Length: 2

This structure represents the Maple **error** statement. The *expr* is either a single expression (if only a message is specified in the **error** statement), or an expression sequence (if arguments are also specified). The actual internal tag used for the **ERROR** structure is **MERROR** to prevent a conflict with a macro defined by some C compilers.

EXPSEQ: Expression Sequence

EXPSEQ	^expr1	^expr2	...

Maple syntax: **expr1**, **expr2**, ...

Length: 1 or more

An expression sequence is an ordered sequence of expressions. It is most commonly used to construct lists, sets, and function calls. Extracting an expression sequence from a list or set L can be done by using the command op(L). This operation is very efficient as it does not involve creation of a new structure. Similarly, if E is an expression sequence, then constructing a list using [E] involves almost no work and is also very efficient. Constructing a set using {E} requires **E** to be sorted. A function call data structure is made up of the function name plus the expression sequence of arguments. During evaluation of a function call, the argument sequence gets flattened into one expression sequence. That is, f(E1,E2) is turned into f(e11,e12,...e1n,e21,e22,...e2m) where **e1i** constitutes the members of the expression sequence **E1**, and **e2i** constitutes the members of the expression sequence **E2**. Thus it is not possible to pass raw expression sequences as arguments to functions. Typically sequences are wrapped in lists, as f([E1],[E2]) in order to keep the

element groupings intact. The special value **NULL** is represented by an empty expression sequence. Thus, [NULL] is equivalent to [], and f(NULL) is equivalent to f().

FLOAT: Software Floating-Point Number

FLOAT	^integer1	^integer2	^attrib-expr

Maple syntax: 1.2, 1.2e3, Float(12,34), Float(infinity)

Length: 2 (or 3 with attributes)

A floating-point number is interpreted as integer1 * 10^integer2. A floating-point number can optionally have attributes, in which case, the length of the structure is 3 and the third word points to a Maple expression. This means that several floating-point numbers with the same value but different attributes can exist simultaneously.

The *integer2* field can optionally be one of the names, **undefined** or **infinity**, in which case the **FLOAT** structure represents an undefined floating-point value (not-a-number, or NaN, in IEEE terminology), or a floating-point infinity. When *integer2* is **undefined**, *integer1* can accept different small integer values, allowing different NaN values to exist. When *integer2* is **infinity**, *integer1* must be 1 or -1.

FOR: For/While Loop Statement

FOR	^name	^from-expr	^by-expr	^to-expr	^cond-expr	^stat-seq

FOR	^name	^in-expr	^cond-expr	^stat-seq

Maple syntax:

```
for name from fromExpr by byExpr to toExpr
      while condExpr do
            statSeq
  end do
```

Maple syntax:

```
for name in inExpr
      while condExpr do
            statSeq
  end do
```

Length: 7 or 5

The *name* follows the same rules as the *name* field of the **ASSIGN** structure, except that it can also be the empty expression sequence (**NULL**), indicating that there is no controlling variable for the loop.

The *from-expr*, *by-expr*, *to-expr*, and *cond-expr* entries are general expressions. All are optional in the syntax of **for** loops and can therefore be replaced with default values (1, 1, **NULL**, and **true** respectively) by the parser.

The *stat-seq* entry can be a single Maple statement or expression, a **STATSEQ** structure, or **NULL** indicating an empty loop body. An additional bit in the header of the **FOR** structure is used to indicate whether the *stat-seq* entry contains any **break** or **next** statements.

FOREIGN: Foreign Data

FOREIGN	...

Maple syntax: none

Length: 1 or more

This structure is similar to the **BINARY** structure, except that it is for use by Maple components outside the kernel, such as the user interface. A **FOREIGN** structure is exempt from garbage collection, and the external component is responsible for freeing this structure when it is finished using it.

FOREIGN data structures can be created and managed in external code by using the **MaplePointer** API functions. For more information, refer to the OpenMaple,C,MaplePointer help page.

FUNCTION: Function Call

FUNCTION	^name	^expr-seq	^attrib-expr

Maple syntax: name(exprSeq)

Length: 2 (or 3 with attributes)

This structure represents a function invocation (as distinct from a procedure definition that is represented by the **PROC** structure). The *name* entry follows the same rules as in **ASSIGN**, or it can be a **PROC** structure. The *expr-seq* entry gives the list of actual parameters; this entry is always an expression sequence (possibly of length 1, which indicates that no parameters are present).

GARBAGE: Garbage

GARBAGE	...

Maple syntax: none

Length: 1 or more

This structure is used internally by the Maple garbage collector as a temporary object type for free space.

HFLOAT: Hardware Float

HFLOAT	floatword

HFLOAT	floatword	floatword

Maple syntax: none

Length: 2 on 64-bit architectures; 3 on 32-bit architectures

This structure is used to store a hardware floating-point value. The one or two words (always 8 bytes) after the header store the actual double-precision floating-point value. **HFLOAT** objects can appear as the result of floating-point computations, I/O operations, or by extracting elements from hardware floating-point **RTABLE** structures. They look like and are treated as indistinguishable from software **FLOAT** objects.

IF: If Statement

IF	^cond-expr1	^stat-seq1	^cond-expr2	^stat-seq2	...	...	^stat-seqN

Maple syntax:

```
if condExpr1 then
   statSeq1
elif condExpr2 then
   statSeq2
...
else statSeqN
end if
```

Length: 3 or more

This structure represents the **if** ... **then** ... **elif** ... **else** ... **end if** statements in Maple. If the length is even, the last entry is the body of an **else** clause. The remaining entries are interpreted in pairs, where each pair is a condition of the **if** or **elif** clause, followed by the associated body.

IMPLIES: Logical IMPLIES

IMPLIES	^expr1	^expr2

Maple syntax: **expr1 implies expr2**

Length: 3

INEQUAT: Not Equal or Test for Inequality

INEQUAT	^expr1	^expr2

Maple syntax: expr1 < > expr2

Length: 3

This structure has two interpretations, depending on the context in which it is used. It can be either a test for inequality or an inequality statement.

INTNEG: Negative Integer

INTNEG	GMP-integer

Maple syntax: -123

Length: 2 or more

This data structure represents a negative integer of arbitrary precision. For a complete description of the integer representation, including positive integers, see the following section.

INTPOS: Positive Integer

INTPOS	GMP-integer

Maple syntax: 123

Length: 2 or more

This data structure represents a positive integer of arbitrary precision. Integers are represented internally in a base equal to the full word size of the host machine. On 32-bit architectures, this base is 4294967296. On 64-bit architectures, the base is 2^64. Integers in this range use the GNU Multiple Precision Arithmetic (GMP) library for integer arithmetic.

Small integers are not represented by data structures. Instead of a pointer to an **INTPOS** or **INTNEG** structure, a small integer is represented by the bits of what would normally be a pointer. The least significant bit is 1, which makes the value an invalid pointer (since pointers must be word-aligned). Such an integer is called an *immediate integer*.

The range of integers that can be represented in this way is -1,073,741,823 to 1,073,741,823 (that is, about +-10^9) on 32-bit architectures, and -4,611,686,018,427,387,903 to 4,611,686,018,427,387,903 (that is, about +-410^18) on 64-bit architectures. (Note that the maximum (non-immediate) integer magnitude in Maple is about 2^2,147,483,488 on 32-bit architectures and 2^274,877,906,688 on 64-bit architectures.)

LESSEQ: Less Than or Equal

LESSEQ	^expr1	^expr2

Maple syntax: expr1 <= expr2, expr2 >= expr1

Length: 3

This structure has two interpretations, depending on the context. It can be interpreted as a relation (that is, an inequation) or as a comparison (for example, in the condition of an **if** statement, or the argument to a call to **evalb**). Maple does not have a greater-than-or-equal structure. Any input of that form is stored as a **LESSEQ** structure.

LESSTHAN: Less Than

LESSTHAN	^expr1	^expr2

Maple syntax: expr1 < expr2, expr2 > expr1

Length: 3

Similar to the **LESSEQ** structure above, this structure has two interpretations, depending on the context. It can be interpreted as a relation (that is, an inequation), or as a comparison (for example, in the condition of an **if** statement, or the argument to a call to **evalb**).

Maple does not have a greater-than structure. Any input of that form is stored as a **LESS** structure.

LEXICAL: Lexically Scoped Variable within an Expression

LEXICAL	integer

Maple syntax: **name**

Length: 2

This represents an identifier within an expression in a procedure or module that is not local to that procedure, but is instead declared in a surrounding procedure or module scope. The *integer* field identifies which lexically scoped variable of the current procedure is being referred to. The *integer*, multiplied by 2, is an index into the *lexical-seq* structure referred to by the **PROC** DAG of the procedure. Specifically, |integer| * 2 - 1 is the index to the **NAME** of the identifier, and |integer| * 2 is the index to a description (**LOCAL**, **PARAM**, or **LEXICAL**) relative to the surrounding scope. The value of *integer* can be positive or negative. If *integer* is a positive value, the original identifier is a local variable of a surrounding procedure; if *integer* is a negative value, it is a parameter of a surrounding procedure.

LIST: List

LIST	^expr-seq	^attrib-expr

Maple syntax: **[expr, expr, ...]**

Length: 2 (or 3 with attributes)

The elements of the *expr-seq* are the elements of the list. The list can optionally have attributes.

LOCAL: Local Variable within an Expression

LOCAL	integer

Maple syntax: **name**

Length: 2

This structure indicates a local variable when it appears within an expression in a procedure or module. The *integer* is an index into the procedure *local-seq*. At procedure execution time, it is also an index into the internal data structure storing the active locals on the procedure activation stack, and stores private copies of the **NAME**s of the local variables (private copies in the sense that these **NAME**s are not the same as the global **NAME**s of the same name).

MEMBER: Module Member

MEMBER	^module	^name

Maple syntax: module:-name

Length: 3

This structure represents a module member access in an expression. **MEMBER** objects typically do not persist when a statement is simplified. Instead, they are replaced by the actual member that they refer to (an instance of a **NAME**).

MODDEF: Module Definition

MODDEF	param-seq	local-seq	option-seq	export-seq	stat-seq	desc-seq

global-seq	lexical-seq	mod-seq	static local-seq	static export-seq	static name-seq

Maple syntax:

```
module modName ( )
     description d1, d2, ...;
     local l1, l2, ...;
     local sl1::static, sl2::static, ...;
     export e1, e2, ...;
```

```
    export se1::static, se2::static, ...;
    global g1, g2, ...;
    option o1, o2, ...;
    statSeq
end module
```

Length: 13

The parameter sequence (*param-seq*), which occurs between the parentheses after **modName**, points to an expression sequence describing the formal parameters of the module. Currently, Maple does not support parameterized modules, so this field always points to the sequence containing only an instance of the name **thismodule**.

The local sequence (*local-seq*) points to an expression sequence listing the explicitly and implicitly declared local variables. Each entry is a **NAME**. The explicitly declared variables appear first. Within the module, locals are referred to by **LOCAL** structures, the local variable number being the index into the local sequence. The instances of these names appear in the **MODULE** structure.

The export sequence (*export-seq*) points to an expression sequence listing the exported module members. Each entry is a **NAME**. Within the module, exports are referred to by **LOCAL** structures, the local variable number being the number of elements in the local sequence, plus the index into the export sequence. The instances of these names appear in the **MODULE** structure.

The option sequence (*option-seq*) points to an expression sequence of options to the module (for modules, options are the same as attributes). Each entry is a **NAME** or **EQUATION** specifying an option. Typical options are **package**, **load=**... and **unload=**...

The statement sequence (*stat-seq*) field points to a single statement or a statement sequence (**STATSEQ**). If the module has an empty body, this is a pointer to **NULL** instead.

The description sequence (*desc-seq*) field points to an expression sequence of **NAME**s or **STRING**s. These sequences are meant to provide a brief description of what the module does and are displayed even when the value of **interface(verboseproc)** is less than 2.

The global sequence (*global-seq*) field points to a list of the explicitly declared global variables in the module (those that appeared in the **global** statement). This information is never used at run time, but is used when simplifying nested modules and procedures to determine the binding of lexically scoped identifiers (for example, an identifier on the left-hand side of an assignment in a nested procedure can be global if it appears in the **global** statement of a surrounding context). This information is also used at printing time, so that the **global** statement contains exactly the global identifiers that were declared originally.

The lexical sequence (*lexical-seq*) field points to an expression sequence of links to identifiers in the surrounding scope, if any. The sequence consists of pairs of pointers. The first pointer of each pair is to the globally unique **NAME** of the identifier; this is needed at simplification and printing time. The second pointer is a pointer to a **LOCAL**, **PARAM**, or **LEXICAL** structure which is understood to be relative to the surrounding scope. When a module definition is evaluated, the lexical sequence is updated by replacing each of the second pointers with a pointer to the actual object represented. The name pointers are not modified, so that the actual identifier names are still available. The *lexical-seq* for a module contains entries for any surrounding-scope identifiers used by that module or by any procedures or modules contained within it.

The module name (*mod-name*) field points to the optional name of the module. If a module name is specified when the module is declared, the name appears there. If no module name is specified, this field will contain a value of **NULL**.

The *static local-seq* points to an expression sequence listing the local variables that were explicitly declared as :static. Each entry is a **NAME**. Within the module, static locals are referred to by **LOCAL** structures, the local variable number being the index into the *static local-seq* minus the number of nonstatic locals and exports. A static local shares its value among all instances of a class.

The *static export-seq* points to an expression sequence listing the exported module members declared as static. Each entry is a **NAME**. Within the module, exports are referred to by **LOCAL** structures, the local variable number being the number of elements in the *local-seq*, *static local-seq*, and *export-seq*, plus the index into the *static export-seq*.

The *static name-seq* stores the instances of the static locals and exports. It appears in the **MODDEF** structure as these static variables are shared among all modules with the same definition.

MODULE: Module Instance

MODULE	^export-seq	^mod-def	^local-seq

Maple syntax: none

Length: 4

Executing a module definition (**MODDEF**) results in a module instance. Each local or exported member of the module is instantiated and belongs to that instance of the module. The *export-seq* field points to an expression sequence of names of the instantiated exports (as opposed to the global names, as stored in the module definition). The *mod-def* field points back to the original module definition. The *local-seq* field points to an expression sequence of names of the instantiated local variables of the module.

NAME: Identifier

NAME	^assigned-expr	^attrib-expr	characters	characters	...

Maple syntax: name

Length: 4 or more

The *assigned-expr* field points to the assigned value of the name. If the name has no assigned value, this field is a null pointer (not a pointer to **NULL**). The next field points to an expression sequence of attributes of the name. If there are no attributes, this field points to the empty expression sequence (**NULL**). The remaining fields contain the characters that form the name, stored 4 or 8 for each machine word (for 32-bit and 64-bit architectures respectively). The last character is followed by a zero-byte. Any unused bytes in the last machine word are also zero. The maximum length of a name is 268,435,447 characters on 32-bit architectures and 34,359,738,351 characters on 64-bit architectures.

NEXT: Next Statement

NEXT

Maple syntax: **next**

Length: 1

NOT: Logical NOT

NOT	^expr

Maple syntax: **not expr**

Length: 2

OR: Logical OR

OR	^expr1	^expr2

Maple syntax: **expr1 or expr2**

Length: 3

PARAM: Procedure Parameter in an Expression

PARAM	integer

Maple syntax: **name**

Length: 2

This structure indicates a parameter when it appears in a procedure. The *integer* is an index into the procedure *param-seq*. Several special **PARAM** structures exist:

PARAM	0

This structure represents the Maple symbol **_npassed** (formerly **nargs**), the number of arguments passed when the procedure was called.

PARAM	-1

This structure represents the Maple symbol **_passed** (formerly **args**), the entire sequence of arguments passed when the procedure was called.

PARAM	-2

This structure represents the Maple symbol **procname**, referring to the currently active procedure.

PARAM	-3

This structure represents the Maple symbol **_nresults**, the number of results expected to be returned from the procedure.

PARAM	-4

This structure represents the Maple symbol **_params**, the sequence of declared positional arguments passed when the procedure was called.

PARAM	-5

This structure represents the Maple symbol **_nparams**, the number of declared positional arguments passed when the procedure was called.

PARAM	-6

This structure represents the Maple symbol **_rest**, the sequence of undeclared arguments passed when the procedure was called.

PARAM	-7

This structure represents the Maple symbol **_nrest**, the number of undeclared arguments passed when the procedure was called.

PARAM	-8

This structure represents the Maple symbol **_options**, the sequence of options in the procedure.

PARAM	-9

This structure represents the Maple symbol **_noptions**, the number of options in the procedure.

PARAM	-10

This structure represents the Maple symbol **thisproc**, referring to the instance of the currently active procedure.

At procedure execution time, the *integer* (if positive) is used as an index into the internal data structure **Actvparams**, which is part of the Maple procedure activation stack, and stores pointers to the values (which are also Maple structures) of the actual parameters passed to the procedure.

POLY: Multivariate Polynomials with Integer Coefficients

POLY	^indet_seq	m[i] degrees	m[i] coeff	m[i+1] degrees	m[i+1] coeff	...

Maple syntax:

```
x^3 + 2*x*y + 1;
```

Length: 2*(number of monomials) + 2

This is an internal representation for multivariate polynomials of limited degree and integer coefficients. **SUM** dags are automatically simplified to POLY dags if possible, provided the polynomial has at least two terms and its total degree is greater than 1.

Each degree word stores the total degree of the monomial and the individual degrees. For example, 5*x^2*y^3 + 1 is a two-variable polynomial whose first term has total degree 5: degree 2 in x, and degree 3 in y. The numbers 5, 2, and 3 are packed into a single *degree* word. The packing depends on the number of variables in the polynomial and the machine word length. Because the packing must fit in one word of memory, not all polynomials can be represented in this way. But many polynomials are stored in this data structure, which can be operated on efficiently.

Each coefficient word must be an integer data structure.

The *indet_seq* is the sequence of indeterminates that occur in the polynomial. The indeterminates must be Maple **NAME**s or **TABLEREF**s. They are always sorted into descending order under the ordering used for sets.

The terms of the polynomial are always stored in graded lexicographical order. That is, monomials are compared first by their total degree, with ties broken by degree in the first variable, then degree the second variable, and so on.

If the sort command is used to sort a polynomial, and it would reorder either the terms or the variables, then the **POLY** dag is automatically converted to a **SUM** dag in place. Should this occur, it is not possible to convert the **SUM** back into a **POLY**.

The precise representation of monomials is as follows. For univariate polynomials, the entire degree word is used and the maximum degree of a **POLY** is the largest immediate

integer kernelopts(maximmediate). For polynomials in n variables, we require n < WORDSIZE/2, so the maximum number of variables is 31 on a 64-bit machine and 15 on a 32-bit machine. The total degree and all of the exponents are given floor(WORDSIZE/(n+1)) bits each, flush against the bottom of the word. For example, on a 64-bit machine a polynomial in x,y will use the lowest 21 bits for y, the next 21 bits for x, and the next 21 bits for the total degree. Any unused bits at the top of a word must remain unset.

POWER: Power

POWER	^expr1	^expr2

Maple syntax: expr1 ^expr2

Length: 3

This structure is used to represent a power when the exponent is not an integer, rational, or floating-point value. When the exponent is numeric, the **POWER** structure is converted to a length 3 **PROD** structure.

PROC: Procedure Definition

PROC	^param-seq	^local-seq	^option-seq	^rem-table	^stat-seq	^desc-seq
^global-seq	^lexical-seq	^eop	^return-type			

Maple syntax:

```
proc ( paramSeq ) :: returnType;
        description descSeq;
    local localSeq;
    export exportSeq;
    global globalSeq;
    option optionSeq;
    statSeq
end proc
```

Length: 10 or 11 (the return type is optional)

The *param-seq* points to an expression sequence describing the formal parameters of the procedure. Each entry is either a **NAME** or a **DCOLON** (which, in turn, contains a **NAME** and an expression specifying a type). Within the procedure, parameters are referred to by **PARAM** structures, the parameter number being the index into the *param-seq*.

The *local-seq* points to an expression sequence listing the explicitly and implicitly declared local variables. Each entry is a **NAME**. The explicitly declared variables appear first. Within the procedure, locals are referred to by **LOCAL** structures, the local variable number being the index into the *local-seq*.

The *option-seq* field points to an expression sequence of options to the procedure (for procedures, options are the same as attributes). Each entry is a **NAME** or **EQUATION** specifying an option. Commonly used options are **cache**, **operator**, and `**Copyright ...**`.

The *rem-table* field points to a hash table containing remembered values of the procedure. Entries in the table are indexed by the procedure arguments, and contain the resulting value. If there is no remember table, this field contains a pointer to **NULL**, which is the empty expression sequence.

The *stat-seq* field points to a single statement or a statement sequence (**STATSEQ**). If the procedure has an empty body, this is a pointer to **NULL** instead. For each procedure that is built into the kernel, there is a wrapper **PROC** that has the option **builtin** in its *option-seq*, and a single Maple integer pointed to by its *stat-seq*. The integer gives the built-in function number.

The *desc-seq* field points to an expression sequence of **NAME**s or **STRING**s. These are meant to provide a brief description of what the procedure does, and are displayed even when the **interface(verboseproc)** command is less than 2.

The *global-seq* field points to a list of the explicitly declared global variables in the procedure (those that appeared in the global statement). This information is never used at run time, but it is used when simplifying nested procedures to determine the binding of lexically scoped identifiers. For example, an identifier on the left-hand side of an assignment in a nested procedure can be global if it appears in the global statement of a surrounding procedure. This information is also used at procedure printing time, so that the **global** statement will contain exactly the same global identifiers that were declared in the first place.

The *lexical-seq* field points to an expression sequence of links to identifiers in the surrounding scope, if any. The sequence consists of pairs of pointers. The first pointer of each pair is to the globally unique **NAME** of the identifier; this is needed at simplification and printing time. The second pointer is a pointer to a **LOCAL**, **PARAM**, or **LEXICAL** structure which is understood to be relative to the surrounding scope. When a procedure is evaluated (not necessarily called), the *lexical-seq* is updated by replacing each of the second pointers with a pointer to the actual object represented. The name pointers are not modified, so that the actual identifier names are still available. The *lexical-seq* for a procedure contains entries for any surrounding-scope identifiers used by that procedure or by any procedures contained within it.

The *eop* field is **BINARY**. The first entry specifies the number of positional parameters of the procedure. The remaining entries, if any, specify the evaluation order permutation for the procedure (that is, an evaluation order for the arguments that is consistent with any dependencies among the parameter specifications).

The *return-type* field is present only if a return type has been specified for the procedure. A return type is an assertion about the type of the value returned by the procedure; if **kernelopts(assertlevel)** is set to 2, then this type is checked as the procedure returns.

PROD: Product, Quotient, Power

PROD	^expr1	^expon1	^expr2	^expon2	...	...

Maple syntax: expr1 ^ expon1 * expr2 ^ expon2 ...

Length: 2n + 1

This structure is interpreted as pairs of factors and their numeric exponents. Rational or integer expressions to an integer power are expanded. If a rational constant is in the product, this constant is moved to the first entry by the simplifier. A simple power, such as a^2, is represented as a **PROD** structure. More complex powers involving non-numeric exponents are represented as **POWER** structures.

RANGE: Range

RANGE	^expr1	^expr2

Maple syntax: **expr1 .. expr2**

Length: 3

RATIONAL: Rational

RATIONAL	^integer	^pos-integer

Maple syntax: **1/2**

Length: 3

This structure is one of the basic numeric objects in Maple. Note that this is not a division operation, but only a representation for rational numbers. Both fields must be integers (**INTPOS**, **INTNEG**, or an immediate integer) and the second must be positive.

READ: Read Statement

READ	^expr

Maple syntax: **read expr**

Length: 2

The Maple **read** statement. The expression must evaluate to either a string or symbol (**STRING** or **NAME** structure), and specifies the name of the file to read.

RETURN: Return Statement

RETURN	^expr-seq

Maple syntax: **return expr1, expr2, ...**

Length: 2

The Maple **return** statement. The expression sequence is evaluated, giving the value(s) to return.

RTABLE: Rectangular Table

RTABLE	^data	^maple-type	^index-func	^attrib	flags	num-elems

L_1	U_1	...	...	L_N	U_N	P_1	P_2	

Maple syntax: **rtable(...)**

Length: 2n + p where **n** is the number of dimensions (0 to 63), and **p** is 0, 1, or 2, depending on the number of P_i parameters.

The *data* field points to either a block of memory (for dense and NAG-sparse **RTABLE**s), or to a **HASHTAB** structure (for Maple-sparse **RTABLE**s). The data block is either an object of type **BINARY**, or memory allocated directly from the storage manager of the operating system when the block is too large to be allocated as a Maple data structure. If the data block is a **BINARY** object, the *data* pointer points to the first data word, not to the object header.

The *maple-type* field points to a Maple structure specifying the data type of the elements of an **RTABLE** of Maple objects. If the **RTABLE** contains hardware objects, the *maple-type* field points to the Maple **NAME anything**.

The *index-func* field points to either an empty expression sequence (**NULL**), or an expression sequence containing at least one indexing function and a pointer to a copy of the **RTABLE** structure. The copy of the **RTABLE** is identical to the original structure, except that its *index-func* field refers to one less indexing function (either **NULL**, or another expression sequence containing at least one indexing function and a pointer to another copy of the **RTABLE** with one less indexing function again).

The *attrib* field points to an expression sequence of zero or more arbitrary attributes, which can be set by the **setattribute** command and queried by using the **attributes** command.

The *flags* field is a bit field containing the following subfields.

- data type - 5 bits - indicates that one of several hardware data types or a Maple data type (as specified by *maple-type*) is being used.

- subtype - 2 bits - indicates if the **RTABLE** is an Array, Matrix, or Vector.
- storage - 4 bits - describes the storage layout (for example, sparse, upper triangular, and so on)
- order - 1 bit - indicates C or Fortran ordering of **RTABLE** elements.
- read only - 1 bit - indicates that the **RTABLE** is to be read-only once created.
- foreign - 1 bit - indicates that the space pointed to by the *data* field does not belong to Maple, so Maple should not garbage collect it.
- eval - 1 bit - indicates if full evaluation should occur on lookup. For more information, refer to the rtable_eval help page.
- literal - 1 bit - optimization for internal type checking of data contained in an **RTABLE**.
- number of dimensions - 6 bits - the number of dimensions of the **RTABLE**, from 0 to 63.

The *num-elems* field indicates the total number of elements of storage allocated for the data. For a Maple-sparse **RTABLE**, *num-elems* is not used. For a NAG-sparse **RTABLE**, and for other formats that grown in size since initial allocation, *num-elems* specifies the number of elements currently allocated, some of which might not be in use.

The $L_i..U_i$ fields specify the upper and lower bounds of each dimension; they are stored directly as signed machine integers. The limits on bounds are -2,147,483,648 to 2,147,483,647 for 32-bit architectures and -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 for 64-bit architectures. The total number of elements cannot exceed the upper limit numbers either. Space is always reserved for at least 4 dimensions in case the rtable is redimensioned.

The remaining P_i fields refer to storage specific properties such as the number of bands above and below the diagonal and the number of elements that are sorted in NAG-sparse storage.

SAVE: Save Statement

SAVE	^expr-seq

Maple syntax: **save expr, expr, ...**

Length: 2

The Maple **save** statement. The expression sequence gives a list of names of objects to save, and either a file name or Maple library archive name (.mla) in which to save them. The file or library archive name can be specified as a **NAME** or **STRING**.

SDPOLY: Sparse Distributed Multivariate Polynomial

SDPOLY	^expr	^term_ordering	^coeff_domain	^coeff_1	exp_1	...	exp_n
...	...	^coeff_m	exp_1	...	exp_n		

Maple syntax: none

Length: For a polynomial of m terms with n variables, the length is $4 + m\,(n + 1)$

The *expr* entry stores the indeterminates of the polynomial (symbol for univariate cases or expression sequence of symbols for multivariate cases).

The *term_ordering* is either null or a pointer to a Maple procedure that is used to compare the *exponent_vector* to sort the polynomial terms. When *term_ordering* is null, lexicographic order is used to sort the polynomial terms.

The *coeff_domain* is either null or a pointer to a Maple module that is used to perform coefficient arithmetic (addition and multiplication). When *coeff_domain* is null, ordinary arithmetic is used. Each of the following **m** terms consists of a coefficient coeff_i (i=1..m) followed by an exponent_vector [exp_j] (j=1..n). Coefficient coeff_i is a non-zero Maple expression. Exponent_vector [exp_j] is an array of **n** hardware integers. Each integer stores the exponent of the corresponding indeterminate. By default, the polynomial terms are sorted by lexicographic order (that is, sorted by descending powers of indeterminate).

SERIES: Series

SERIES	^expr1	^expr2	integer	^expr3	integer	...	...

Maple syntax: none

Length: 2n + 2

This is the internal representation of a series in Maple. There is no input syntax for a series; one can only be generated from a computation. The first expression has the general form x-a, where **x** denotes the variable of the series used to perform that expansion, and **a** denotes the point of expansion. The remaining entries are interpreted as pairs of coefficients and exponents. The exponents are integers, *not* pointers to integers or immediate integers. The exponents appear in increasing order. A coefficient **O(1)** (a function call to the function **O**, with parameter 1) is interpreted specially by Maple as an order term.

SET: Set

SET	^expr-seq	^attrib-expr

Maple syntax: { expr, expr, ... }

Length: 2 (or 3 with attributes)

The entries in the expression sequence of the set are sorted in a deterministic order. For details, see the set help page.

STATSEQ: Statement Sequence

STATSEQ	^stat1	^stat2	...

Maple syntax: **stat1; stat2; ...**

Length: 3 or more

This structure represents a sequence of two or more statements, and can be used wherever a single statement (for example, **ASSIGN**, **IF**, **FOR**) can appear. A statement sequence, containing only a single statement, is replaced by that statement. A statement sequence containing no statements is replaced by the empty expression sequence (**NULL**). Nested **STATSEQ** structures are flattened. All of the above transformations are made by the simplifier.

STOP: Quit Statement

STOP

Maple syntax: **quit**, **done**, or **stop**

Length: 1

STRING: Character String

STRING	reserved	^attrib-expr	characters	characters	...

Maple syntax: "This is a string"

Length: 4 or more

A Maple string is structurally similar to a **NAME**, except that it has no *assigned-value* field. The *attrib-expr* field points to an expression sequence of attributes of the string. If there are no attributes, this field points to the empty expression sequence (**NULL**). The remaining fields contain the characters that form the string, stored 4 or 8 per machine word (for 32-bit and 64-bit architectures respectively). The last character is followed by a zero-byte. Any unused bytes in the last machine word are also zero.

The maximum length of a string is 268,435,447 characters on 32-bit architectures and 34,359,738,351 characters on 64-bit architectures.

SUM: Sum, Difference

SUM	^expr1	^factor1	^expr2	^factor2	...	...

Maple syntax: expr1 * factor1 + expr2 * factor2 ...

Length: 2n + 1

This structure is interpreted as pairs of expressions and their numeric factors. Rational or integer expressions with an integer factor are expanded and the factor replaced with 1. If there is a rational constant in the sum, this constant is moved to the first entry by the simplifier. Simple products, such as a*2, are represented as **SUM** structures. More complex products involving non-numeric factors are represented as **PROD** structures.

TABLE: Table

TABLE	^index-func	^array-bounds	^hash-tab

Maple syntax: N/A

Length: 4

This is a general table type, as created by the **table** and **array** commands in Maple. The *index-func* points to either a **NAME** or a **PROC**. For general tables, the *array-bounds* field points to the empty expression sequence (**NULL**). For **array**s (not to be confused with **Array**s, which are implemented as **RTABLE**s), the *array-bounds* field refers to an expression sequence of **RANGE**s of integers. The *hash-tab* field points to a **HASHTAB** structure containing the elements.

TABLEREF: Table Reference

TABLEREF	^name	^expr-seq	^attrib-expr

Maple syntax: **name [expr]**

Length: 3 (or 4 with attributes)

This data structure represents a table reference, or indexed name. The *name* entry follows the same rules as for **ASSIGN**, or it may be a **TABLE** or **MODULE** structure. (The parser will not generate a **TABLEREF** with a **TABLE** structure for the *name* entry, but this can occur internally.) The expression sequence contains the indices.

TRY: Try Statement

TRY	^try-stat-seq	^catch-str	^catch-stat-seq	...	...	^final-stat-seq

Maple syntax:

```
try tryStat
     catch "catchStr": catchStat
     ...
     finally finalStat;
 end try
```

Length: 3 or more

This structure represents a **try** statement, and can have an arbitrary length, depending on how many **catch** blocks are contained within it, and whether it has a **finally** block. The *catch-strs* point to the catch string of the corresponding **catch** block. If no catch string is specified, the *catch-str* points to **NULL**. Empty *catch-stat-seqs* are also represented by pointers to **NULL**, as is an empty (but present) **finally** block.

The actual internal tag used for the **TRY** structure is **MTRY** to prevent collision with a macro defined by some C exception handling libraries.

UNEVAL: Unevaluated Expression

UNEVAL	^expr

Maple syntax: **'expr'**

Length: 2

USE: Use Statement

USE	^bindings	^statseq

Maple Syntax:

```
use bindings in
 statseq
end use
```

Length: 3

The *bindings* component points to an expression sequence of equations whose left-hand sides are symbols, and the *statseq* component points to a sequence of statements that form the body of the **use** statement. The right-hand sides of the binding equations can be arbitrary expressions.

The **use** statement introduces a new binding contour and binds the names that appear on the left-hand side of the equations in *bindings*. For convenience, on input, a module **'m'** can appear among the *bindings*, and is treated as if it were the sequence **e1 = m:-e1, e2 = m:-e2, ...**, where the **ei** are the exports of **'m'**. Within the sequence *statseq* of statements, the symbols appearing on the left-hand side of the equations in *bindings* are bound to the corresponding right-hand sides. The previous bindings of those symbols are restored upon exit from the **use** statement. Bindings are resolved during automatic simplification.

XOR: Logical Exclusive-Or

XOR	^expr1	^expr2

Maple syntax: **expr1 xor expr2**

Length: 3

ZPPOLY: Polynomials with Integer Coefficients modulo n

ZPPOLY	^indet	mod	coef0	coef1	...
ZPPOLY	^indet_seq	mod	^zppoly0	^zppoly1	...

Maple syntax: modp1(ConvertIn(expr, indet), n);

Maple syntax: modp2(ConvertIn(expr, indet1, indet2), n);

Length: degree(zppoly) + 2 (for the zero polynomial)

Length: degree(zppoly) + 3 (otherwise)

This is the internal representation of univariate and bivariate polynomials modulo some integer. The modp1() and modp2() front ends provide a suite of functions to work on this data structure operating in the domain of polynomials in one or two variables with integer coefficients modulo n, written Z_n_x or Z__n[x,y], respectively. *indet_seq* is an expression sequence of the indeterminates of the polynomial: (x), or (x,y). *mod* is the integer modulus of the integer domain. In a univariate polynomial, the coefficients are stored in the following order.

(coef0*indet^0 + coef1*indet^1 + ... + coefi*indet^i) mod n

A bivariate polynomial contains pointers to univariate ZPPOLY structures representing the coefficients of the first indeterminate.

(coef0(indet2)*indet1^0 + coef1(indet2)*indet1^1 + ...) mod n

where each **coefi** is a univariate polynomial in **indet1 mod** n.

All coefficients are stored, including zero coefficients. The leading coefficient is always non-zero.

A.4 Hashing in Maple

An important factor in achieving the overall efficient performance of Maple is the use of hash table-based algorithms for critical functions. Tables are used in both simplification and evaluation, as well as for less critical functions. For simplification, Maple keeps a single copy of each expression, or subexpression, during a session. This is done by keeping all objects in a table. In procedures, the **cache** and **remember** options specify that the result of each computation of the procedure is to be stored in a *remember table* associated with the procedure. Finally, tables are available to the user as one of the Maple data types.

All table searching is done by hashing. Three types of hash tables are available: basic, dynamic, and cache. Basic hash tables are used for most Maple hashing. They are auto-

matically promoted to dynamic hash tables when they are filled with a large number of elements. Dynamic hash tables are designed to work with a large number of elements. Cache tables are a type of hash table that store only *recently* inserted items.

Basic Hash Tables

The algorithm used for the basic hash tables is direct chaining, except that the chains are dynamic vectors instead of the typical linked lists. The two data structures used to implement hash tables are **HASHTAB** and **HASH**.

Hash Table

HASHTAB	^hash-chain1	^hash-chain2	...

Maple syntax: none

Length: $2^n + 1$

This is an internal data structure with no Maple syntax equivalent. It is used in the representation of tables within Maple. Each entry points to a hash chain (a **HASH** structure), or is a null pointer if no entry has been created in that hash chain yet (that is, with that entry location as its hash value). The size of a **HASHTAB** structure depends on the type of table and the platform, but is always a power of 2 plus one.

Hash Chain

HASH	key	^expr1	key	^expr2	...	...

Maple syntax: none

Length: 2n + 1

Each table element is stored as a pair of consecutive entries in a hash bucket vector. The first entry of this pair is the hash key, and the second is a pointer to a stored value. In some cases (for example, procedure remember tables and user-defined tables), the key is also a pointer. In other cases, the key is a hashed value (for example, the simplification table, the symbol table). The key cannot have the value zero (or the null pointer) since this is used to indicate the bottom of the bucket.

Dynamic Hash Tables

The Maple dynamic hash table is a complicated data structure. a brief overview is presented here.

Instead of using a flat, fixed-length directory, Maple dynamic hash tables use a tree structure with contiguous bits from the hash key to select a child. A child of a directory can be a subdirectory or a hash chain. For example, a top-level directory may use the first

10 bits to index 1024 children. One of its children may be a directory that uses, for example, the next 8 bits of the key to index 256 children.

A hash chain in a dynamic table stores elements using key value pairs (in the same way that a hash chain does in a basic hash table). The first n bits of the keys in a hash chain are identical, where n is the number of bits required to locate the hash chain. The remaining bits are arbitrary. Using the example in the previous paragraph, the elements of a hash chain that is a child of the directory with 256 children have hash keys that are identical in the first 18 bits.

When a hash chain with unused bits overflows, it is split into two. This may require creating a subdirectory with two children or doubling the size of the hash chain's parent directory. In either case, another bit from the hash key is introduced for indexing. This bit is used to divide the elements of the old chain into the two new chains. If the hash chain has no unused bits for indexing, the chain grows as needed. This growth occurs only if many elements are inserted with identical hash keys.

Cache Hash Tables

Cache tables have two classes of entries: permanent and temporary. Each bucket in the table has 4 entries reserved as temporary, followed by a pointer to a variable-sized chain.

Permanent entries, as designated by the way they are inserted, are stored exclusively in the variable-sized chain, which can grow as needed.

Temporary entries are inserted in the normal way you would include a value in a basic hash table or remember table. These are hashed to identify the bucket in which they are to be stored. The existing entries in that bucket are pushed right by one, and the new entry is put in the leading, "most-recent" spot. Reinserting an expression will cause it to be promoted to the "most-recent" spot. Inserting a fifth element that hashes to the same bucket will cause the least recently inserted temporary element to be removed from the table.

The maximum size of the cache table can be specified at creation time. Because cache tables have a maximum size, and because as new elements are added old ones may be removed, the cache table does not grow continuously as values are added. When used as a remember table, they are useful for temporarily storing elements that were recently computed, and likely to be needed again. Over time, as more elements are inserted, the old elements will be discarded.

Cache tables can be created by using the **Cache** command, or as a remember table in a procedure with the **cache** option specified. The advantage of using a cache table over standard remember tables is that a cache table has a maximum size. This means that a cache table does not act as a memory trap, storing a large number of values that cannot

be reclaimed by the garbage collector. As cache tables allow permanent elements to be added, they can be used in procedures that cannot use option system remember tables.

The Simplification Table

The most important table maintained by the Maple kernel is the *simplification table*. All simplified expressions and subexpressions are stored in the simplification table. The main purpose of this table is to ensure that simplified expressions have a unique instance in memory. Every expression which is entered into Maple or generated internally is checked against the simplification table. If it is found in the simplification table, the new expression is discarded and the old one (the one in the simplification table) is used. This task is done by the simplifier, which recursively simplifies (applies all the basic simplification rules) and checks against the table. The garbage collector deletes the entries in the simplification table that cannot be reached from a global name or from a live local variable.

The task of checking for equivalent expressions within thousands of subexpressions would not be feasible if it were not done with the aid of hashing. Every expression is entered in the simplification table using its signature as a key. The signature of an expression is a hashing function itself, with one important attribute: signatures of trivially equivalent expressions are equal. For example, the signatures of the expressions a+b+c and c+a+b are identical; the signatures of a*b and b*a are also identical. If the signatures of two expressions disagree, the expressions cannot be equal at the basic level of simplification.

In Maple 13, the use of the basic and dynamic hash tables as the data structure behind the simplification table was phased out in favor of a new structure that worked better in a multithreaded environment. In particular, the new table guarantees atomic inserts. This removed the need for locking, and, because the simplification table is used so often, greatly improved performance when running many threads.

Searching for an expression in the simplification table is done by:

- Simplifying recursively all of its components
- Applying the basic simplification rules
- Computing its signature and searching for this signature in the table

If the signature is found, then a full comparison is performed (taking into account that additions and multiplications are commutative) to verify that it is the same expression. If the expression is found, the one in the table is used and the searched one is discarded. A full comparison of expressions has to be performed only when there is a collision of signatures.

Since simplified expressions are guaranteed to have a unique occurrence, it is possible to test for equality of simplified expressions using a single pointer comparison. Unique representation of identical expressions is significant for the efficiency of tables, and

therefore the **remember** option. Also, since the relative order of objects is preserved during garbage collection, sequences of objects can be ordered by machine address. For example, sets containing mutable objects are represented this way. The set operations, such as union or intersection, can be done in linear time by merging sorted sequences. Sorting by machine address is also available by using the **sort** command.

The Name Table

The simplest use of hashing in the Maple kernel is the *name table*. This is a symbol table for all of the global names. Each key is computed from the character string of the name and the entry is a pointer to the data structure for the name. The name table is used to locate global names formed by the lexical scanner or by name concatenation. It is also used by functions that perform operations on all global names. These operations include:

- Marking for garbage collection
- Saving a Maple session environment in a file
- The Maple commands **anames** and **unames**, which return all assigned and unassigned global names, respectively

Remember Tables

A remember table is a hash table in which the argument(s) to a procedure call are stored as the table index, and the result of the procedure call is stored as the table value. Because a simplified expression in Maple has a unique instance in memory, the address of the arguments can be used as the hash function. Therefore, searching a remember table is very fast.

Several kernel functions use remember tables including **evalf**, **series**, **divide**, **normal**, **expand**, **diff**, **readlib**, and **frontend**. The functions **evalf**, **series**, and **divide** are handled internally in a special way for the following reasons:

- **evalf** and **series** need to store some additional environment information (**'Digits'** for evalf and **'Order'** for series). Consequently, the entries for these are extended with the precision information. If a result is requested with the same or less precision than what is stored in the table, the table value is retrieved and rounded. If a result is produced with more precision than what is stored, it is stored in the table, replacing the lower precision value.
- **evalf** remembers only function calls (this includes named constants); it does not remember the results of arithmetic operations.
- If a division operation succeeds and the divisor is a nontrivial polynomial, the **divide** function stores the quotient in its remember table. Otherwise, no value is stored in the remember table.

If **option remember** is specified together with **option system**, at garbage collection time, the remember table entries which refer to expressions no longer in use elsewhere in the

system are removed. This provides a relatively efficient use of remembering that does not waste storage for expressions that have disappeared from the expression space. As garbage collection time can be unpredictable, cache remember tables provide an alternate approach similar to option system, by remembering only the most recently computed results.

Maple Language Arrays and Tables

Tables and arrays are provided as data types in the Maple language through the **table** and **array** commands.

Note: Unlike the array command, the Array command creates a rectangular table, which is described in the following subsection. An array is a table for which the component indices must be integers within specified bounds. Tables and arrays are implemented using the Maple internal hash tables. Because of this, sparse arrays are equally as efficient as dense arrays. A table object consists of the following.

- Index bounds (for arrays only)
- A hash table of components
- An indexing function

The components of a table **T** are accessed using a subscript syntax (for example, T[a,b*cos(x)]). Since a simplified expression is guaranteed to have a unique instance in memory, the address of the simplified index is used as the hash key for a component. If no component exists for a given index, then the indexed expression is returned.

The semantics of indexing into a table are described by its indexing function. Aside from the default, general indexing, some indexing functions are provided by the Maple kernel. Other indexing functions are loaded from the library or are supplied by the user.

Maple Language Rectangular Tables

Rectangular tables (as implemented by the **RTABLE** structure) can use a variety of storage formats. One format, Maple-sparse, is identical to that used in tables and arrays, namely a hash table. For Matrices, there is another sparse format, NAG-sparse, which uses one vector for each dimension to record indices, and one more vector to record the values of the entries. Most **RTABLE** storage formats are dense, the simplest being the rectangular format. Other dense formats include upper-triangular and band, where storage is allocated only for the upper triangle or a band of elements respectively. To the user, rectangular tables appear as objects of type **Array**, **Matrix**, **Vector[row]**, and **Vector[column]**. Note that an **Array** is not the same as an **array**. For more information, refer to the Array and array(deprecated) help pages.

Portability

The Maple kernel and the command-line interface are not associated with any one operating system or hardware architecture. The Maple kernel is designed to be portable to any system which supports a C compiler, a flat address space, and a 32-bit or 64-bit word size. Refer to the **Install.html** file on your product installation disc for a list of currently supported operating system versions.

Most of the source code comprising the kernel is the same across all platforms. Extensive use of macros and conditional compilation take care of platform dependencies, such as word size, byte ordering, storage alignment requirements, differences in hardware floating point support, and sometimes, C compiler bugs.

The Maple library is interpreted by the Maple kernel. Therefore, other than issues such as maximum object size, it is completely independent of the underlying architecture.

The Standard worksheet graphical user interface is implemented in Java, which is platform-independent. This includes custom GUI features such as embedded components and Maplets.

Index

Symbols

!, 31
#, 31
& operator, 107
.. operator, 107
1-D output, 361
2-D math, 379
2-D output, 361
:, 330
:-, 67, 314
:: operator, 111
:= operator, 3
?, 31
?[], 62
@ operator, 105
@@ operator, 105
[], 30
{}, 30
~, 17

A

anyfunc type, 122
argument
    definition, 195
Arrays
    applying a function to contents, 152
    automatic resizing, 148
    copying, 150
    creating, 144
    getting bounds, 149
    getting number of elements, 149
    numeric, 152
    testing for equality, 151
arrow notation, 252
assignment operator, 3, 43
automatic resizing
    Arrays, 148
automatic simplification, 50

B

backslash, 24, 27, 31
braces
    forming sets, 30
built-in commands, 8

C

case-sensitivity in Maple, 20
colon, 2, 28, 169
comma
    forming expression sequence, 30
comments, 7
Complex constructor, 59
complex numbers, 274
    evalc command, 61
    Re and Im commands, 60
concatenation
    names, 23
    strings, 23
constants
    special, 44
    symbolic, 44, 275
copying Arrays, 150
copying tables, 142
creating
    Arrays, 144
    lists, 126
    queues, 160
    records, 155
    sets, 132
    tables, 137

D

DAG, 31
data structures
    Arrays, 144
    converting, 154
    filtering elements, 153
    immutable, 135
    lists, 126
    mutable, 151
    queues, 159

records, 155
sets, 132
tables, 137
data types, 35
internal representation, 405
definition
argument, 195
function call, 195
delaying evaluation, 29, 46
dismantle command, 36
dot character, 84
double colon operator, 111
double quotes
displaying a text string, 2, 25

E

equality
records, 156
error statement handling, 185
escape characters, 31
eval command, 10, 118
difference between eval and subs, 118
evalc command, 61
evalf command, 283
evalhf command, 287
evalindets command, 120
evaluating expressions, 116
evaluation
delaying, 29, 46
evaluation rules, 9, 32, 155, 164, 315
tables, 141
exception handling, 185
expand command, 104
expression sequence, 30
expression statements, 170
expressions
converting to strings, 26
evaluating and simplifying, 116
grouping terms, 29
rational, 79
set-theoretic, 101
tree form, 36
union, 103
extended numeric, 273
extracting data from tables, 141

F

file input and output
files used by Maple, 374
general files, 367
FileTools package, 370
Maple I/O library, 368
importing and exporting numerical data, 372
introductory concepts, 367
floating-point numbers
catastrophic cancellation, 279
Digits, 278
hardware, 272
IEEE 754, 282
precision, 277
representation, 276
software, 270
floats
exponent, 55
hardware, 55
Maple_floats command, 55
significand, 55
software, 55
flow of control, 404
for loop, 132, 227
ModuleIterator procedure, 320
scoping rules, 219
format strings, 377
fprintf command, 378
fractions
denom command, 54
Fraction constructor, 54
numer command, 54
fscanf command, 378
full evaluation, 9
function call, 68, 196
definition, 195
function type, 122

G

garbage collection, 324
global variables
  modules, 313
  procedures, 196

H

hash tables, 430
hashing in Maple, 429
  Arrays and Tables, 434
  basic hash tables, 430
  cache hash tables, 431
  dynamic hash tables, 430
  name table, 433
  portability, 435
  rectangular Tables, 434
  remember tables, 433
  simplification table, 432
help databases, 375
HFloat
  constructor, 57
hfloat
  option, 295

I

Im command, 60
imaginary unit
  changing default, 61
in operator, 101
indets command, 123
indexed expression
  extracting individual elements, 62
indexed expressions constructor, 62
indexing
  mathematical, 145
  negative, 146
  programmer, 145
indices function, 141
  nolist option, 141
infinity, 273
input and output
  interactive input, 366
  interactive output, 364
  with files
    files used by Maple, 374
    files used by Maple,help databases, 375
    files used by Maple,internal format files, 375
    files used by Maple,library archives, 375
    files used by Maple,Maple language files, 374
    files used by Maple,Maplet files, 375
    files used by Maple,worksheet files, 375
    general files, 367
    general files,FileTools package, 370
    general files,Maple I/O library, 368
    importing and exporting numerical data, 372
    introductory concepts, 367
  worksheet
    interfaces, 363
integers
  determining length, 53
  GMP, 268
  immediate, 267
  signed, 53
interactive input
  worksheet, 366
interactive output
  worksheet, 364
interface command, 363
  variables
    echo, 194
    imaginaryunit, 61
    prettyprint, 363
    rtablesize, 363
    typesetting, 379
    verboseproc, 10
    version, 363
interfaces
  worksheet input and output, 363
internal format files, 375

internal representation
data types, 405
assignment statement, 406
binary object, 406
break statement, 406
character string, 426
communications control structure, 407
complex value, 407
debug, 407
difference, 426
equation, 408
error statement, 408
expression sequence, 408
for loop statement, 409
foreign data, 410
function call, 410
garbage, 410
hardware float, 411
identifier, 417
if statement, 411
less than, 413
less than or equal, 413
lexically scoped variables, 413
list, 414
local variables, 414
logical AND, 405
logical IMPLIES, 411
logical NOT, 417
logical OR, 417
logical XOR, 428
module definition, 414
module instance, 416
module member, 414
multivariate polynomials with integer coefficients, 419
name concatenation, 406
negative integer, 412
Next statement, 417
not equal, 412
polynomials with integer coefficients modulo n, 429
positive integer, 412
Power, 420
procedure definition, 420
procedure parameters, 417
product, 422
quit statement, 426
quotient, 422
range, 422
read statement, 422
rectangular table, 423
return statement, 423
save statement, 424
series, 425
set, 425
software float, 409
sparse distributed multivariate polynomial, 425
statement sequence, 426
sum, 426
table, 427
table reference, 427
test for equality, 408
test for inequality, 412
try statement, 427
type specification, 407
unevaluated expressions, 428
use statement, 428
while loop statement, 409
interrupt a Maple computation, 10
command-line, 11
hard interrupt, 11
worksheet
interrupt icon, 11
stop icon, 11
intersect operator, 104

K

kernel, 1
kernelopts
maxdigits, 53

L

last name evaluation, 9, 99, 141, 242, 260
left single quotes

forming names, 20
library archives, 375
line continuation character, 27
lists, 126
accessing data stored in, 127
creating, 126
nested, 126
local variables, 6
modules, 313
loops
commands, 182

M

macro definitions, 390
map command, 131
Maple character set, 15
Maple internal functions
algebraic functions, 404
algebraic service functions, 404
data structure manipulation functions, 404
evaluators, 403
general service functions, 404
Maple keywords, 16
Maple language files, 374
Maple library, 1
Maple library archive, 387
Maple library commands
printing, 10
Maple preprocessor, 390
Maple types, 35
Maple User Interface, 1
Maplet files, 375
Matrix
creating, 30
maximum number of digits, 53
member function, 129
member selection, 67, 314
members
exported, 347
local, 347
memory
clearing, 11
methods, 349
module definitions
body, 309
declaring statements, 312
implicit scoping rules, 324
lexical scoping rules, 324
named modules, 310
parameters, 310
syntax, 309
ModuleIterator procedure, 320
modules
exports, 314
members, 314
options, 318

N

name tables, 433
names, 3, 20
equality of, 42
multiple assignment, 171
unassigning, 49
with blank spaces, 20
with international characters, 20
nested lists, 126
nolist option
indices function, 141
nprintf command, 378
numelems command, 130
numeric types, 267

O

object oriented programming
class
creating new, 348
introduction, 347
members
definition, 347
methods, 349
calling, 349
static vs non-static, 349
modules
definition, 347
objects, 348

definition, 347
indexed function calls, 350
special methods, 350
op command, 34, 127, 133
operators
&, 107
.., 107
:-, 67
::, 111
@, 105
@@, 105
addition, 70
arithmetic, 3
binary, 17
division, 74
element-wise, 17
if, 177
in, 101
intersect, 104
member selection, 67
multiplication, 74
set-theoretic, 101
subtraction, 70
unary, 17
union, 102

P

packages, 319, 383
exports, 384
organizing, 392
saving, 387
packed records, 156
parameter
definition, 195
prettyprinting, 366
print
defining custom printing, 366
printf command, 376
procedure call, 8
procedure definition, 4
procedures
adding comments, 7
declaring parameters
parameter modifiers, 204
defining, 4
invocation, 8
maintainable code, 249
adding comments, 250
formatting procedures for readability, 249
options
cache, 215
syntax, 6
protected names, 45

Q

queues
creating, 160
dequeue, 160
enqueue, 160
quit statement, 192
quotes
double, 2, 25
left single, 20
right single, 29

R

rational numbers, 269
Re command, 60
read statement, 193
records, 333
creating, 155
equality, 156
remember tables, 433
restart command, 11
return statement, 185
right single quotes
delaying evaluation, 29
rtables, 144

S

save statement, 193
scanf command, 376
selection operation, 127, 138, 315
semicolon, 2, 28, 169

separating statements, 28
sequence, 30
setattribute command, 115
sets, 30, 132
  accessing data stored in, 134
  arithmetic, 133
SFloat
  constructor, 56
simplifying expressions, 116
sort command, 131
specfunc type, 122
special characters, 15
special methods, 350
  ModuleApply, 350
  ModuleCopy, 350
  ModuleDeconstruct, 350
  ModuleIterator, 350
  ModuleLoad, 350
  ModulePrint, 350
  ModuleType, 350
  ModuleUnload, 350
sprintf command, 378
square brackets, 30
  and braces, 30
sscanf command, 378
stacks
  creating, 157
  popping, 158
  pushing, 158
strings, 2, 25
  concatenation, 23
  length, 22
  mutability, 24
  parsing, 25
  searching, 23
StringTools package, 378
StringTools:-StringBuffer command, 24
structured types, 32, 120
sub-Array access, 146
subexpressions
  substituting, 117
subs command, 118
  difference between eval and subs, 118
subsindets command, 120
subsop command, 117
substituting subexpressions, 117
substrings
  extracting, 22

T

table indexing, 138
tables
  applying a function to contents, 143
  checking index, 140
  copying, 142
  creating, 137
  evaluation rules, 141
  extracting data, 141
  getting number of elements, 140
  removing an element from, 139
try statement, 186
type checking, 32
type command, 45
typeset math, 379
typesetting package, 380

U

undefined, 273
unevaluation quotes, 29, 46
union operator, 102
use statement, 337

V

Vector
  creating, 30
viewing help pages, 2

W

white space characters, 27
worksheet files, 375
worksheet input and output
  interactive input, 366
  interactive output, 364
  interfaces, 363