EdScheme

for the Macintosh

User's Guide and Reference Manual

Schemers Inc.

E^dScheme

for the Macintosh

User's Guide
and
Reference Manual

Schemers Inc.

This User's Guide and Reference Manual was written by Edward C. Martin and Iain Ferguson.

ISBN 0–9628745–6–6

First published 1993
Typeset in 11pt Computer Modern Roman using TEX
Published by Schemers Inc.
4250 Galt Ocean Drive, Suite 7U
Fort Lauderdale, FL
USA 33308

Contents

Part I The Programming Environment **1**

1 Starting Out 3
- 1.1 Scheme—A Modern Lisp 3
- 1.2 The *EdScheme* Advantage 4
- 1.3 **About This Book** 5
- 1.4 When You First Use *EdScheme* 8

2 An *EdScheme* Session 11
- 2.1 Starting Up 11
- 2.2 Organizing the Desktop 12
- 2.3 The Current Expression 14
- 2.4 Indentation and Formatting 15
- 2.5 Indexing Document Windows 16
- 2.6 **Using Multiple Documents** 17
- 2.7 Saving your Documents 17

3 Programming Environment Windows 19
- 3.1 **Transcript Windows** 20
- 3.2 The Trace Window 23
- 3.3 Document Windows 23

3.4 The Expression Window 25
3.5 The Clipboard Window 26

Part II The Programming Language 27

4 Lists and Vectors 29
4.1 Lists 29
4.2 Vectors 31

5 Numbers and Numeric Procedures 35
5.1 Types of Numbers 35
5.2 Numeric Constants 36
5.3 Prefixes 36
5.4 Digits 37
5.5 Integers 37
5.6 Rational Numbers 39
5.7 Real Numbers 41
5.8 Complex Numbers 43
5.9 Exactness 44
5.10 Mathematical Information 45

6 Files and Ports 49
6.1 Files 49
6.2 Ports and File-handling 50

7 Graphics Windows and Bitmaps 53
7.1 Graphics Windows 53
7.2 A Graphics Programming Example 54
7.3 Bitmaps 58

8 Text Windows and User Menus 61
8.1 Text Windows 61
8.2 Menus 65

Part III Language Reference 67

9 The *EdScheme* Menus 69
9.1 The File Menu 69
9.2 The Edit Menu 83
9.3 The Search Menu 86
9.4 The Evaluate Menu 89
9.5 The Windows Menu 93

10 Data Expressions 95
10.1 Identifiers: Keywords and Variables 95
10.2 Booleans 97
10.3 Pairs and Lists 97
10.4 Numbers 99
10.5 Characters 99
10.6 Strings 99
10.7 Vectors 100
10.8 Procedures 100
10.9 Continuations 101
10.10 Atoms 102
10.11 Streams and Delayed Objects 103
10.12 Environments 103
10.13 Ports 104
10.14 Miscellaneous Data Expressions 105
10.15 Comments 106

11 Syntax and Semantics 107
11.1 Abbreviations 107
11.2 The Syntax of *EdScheme* 109

12 Language Elements 245
12.1 Constants, Booleans, and Equivalence Predicates 245
12.2 Characters 246
12.3 Strings 246
12.4 Vectors 247
12.5 Numbers 247

12.6 Symbols and Lists 248
12.7 Graphics and Text Windows 248
12.8 Keyboard and Ports 250
12.9 Events, Menus, and the Mouse 251
12.10 Debugging 251
12.11 Keywords and Special Forms 251
12.12 Miscellaneous 252

Index 253

– o O o –

Part I

The Programming Environment

Starting Out

.1 Scheme—A Modern Lisp

Programming computers is not just about encoding algorithms with a view to processing large amounts of data. Certain programming languages—Scheme is one—lend themselves to more interesting pursuits, including those that fall under the catch-all 'artificial intelligence'. Scheme programmers have a special responsibility, namely, to ensure not only that their programs *work* but that they are imbued with a sense of *elegance*, even beauty. A good program is greater than the sum of its parts, just as a masterful painting is more than just streaks of oil on canvas. The Scheme programmer does not seek elegance simply for its own sake, however, for the well-written program is more efficient, more reliable, and easier to debug. Above all else, a good Scheme program readily lends itself to use in unexpected applications, and it is this particular quality that sets Scheme apart from many other programming languages. So often, the Scheme programmer who sets out to solve a specific problem instead finds solutions to an entire *class* of problems, many of which seemed previously unrelated, and of which the initial problem is just an instance.

Scheme is a dialect of the Lisp programming language; it was invented by Guy Lewis Steele Jr. and Gerald Jay Sussman of the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. It demonstrates that a clear and simple language with few syntactic rules is sufficient to form the basis of a practical, efficient, and powerful programming language, flexible enough to support most of the major programming paradigms, including object-oriented programming. Scheme's simple structure is due in part to the fact that the language's designers rated simplicity of form above gratu-

itous compatibility with the older Lisps, and in part to recent advances in the mathematics of programming language design. The result is a language powerful enough for professional programmers, yet with a structure so simple that even complete computer novices can pick up the language and write significant programs.

Scheme's 'tail-recursive' design means that the traditional control structures may all be described in terms of a single paradigm: recursion. Armed with this tool, students have immediate access to algorithms that would previously have been beyond their immature skills. One of the features of Scheme is that all data objects are treated alike. Consequently, the addition of a new data object may add significantly to the Scheme's expressive power while in no way increasing the complexity of the language. Moreover, the skills required to manipulate the simpler data objects are transferable to the manipulation of more complex objects. No wonder, then, that so many educational institutions have turned to Scheme as the basis for both their introductory and advanced computer science courses.

Scheme is a language whose future is assured. Increasingly, computer scientists are exploring the possibility of applying mathematical proof techniques to verifying or predicting the behavior of computer programs, and it is no accident that both the language itself and the behavior of Scheme programs are susceptible of rigorous mathematical interpretation.[1] Furthermore, Scheme is poised to exploit the new generation of parallel processing architectures. And in terms of software engineering, Scheme supports—indeed, *encourages*—the most up-to-date programming paradigms.

.2 The *EdScheme* Advantage

EdScheme for the Macintosh™ is an incremental optimizing compiler for the Scheme language designed specifically with the learner in mind, but providing a complete implementation of the Scheme specification[2] for the intermediate and advanced user. The programming environment takes advantage of the special capabilities of the Macintosh computer, and includes:

[1] In fact, Scheme is an outworking of the highly theoretical investigations of the American mathematician and logician Alonzo Church, who first described his pioneering work in an article published in 1941, *The Calculi of Lambda Conversion.*

[2] ***EdScheme*** implements the *Revised[4] Report on the Algorithmic Language Scheme.*

- A full-featured integrated editor, with special capabilities such as parenthesis-matching, program formatting, file indexing, and template editing.
- Customized transcript and debugging windows featuring colored and styled text in addition to all the facilities provided by the integrated editor.
- A powerful and comprehensive turtle graphics interface, providing users with access to the Macintosh's Color QuickDraw™ graphics toolbox and many additional graphics capabilities.

At the same time, ***EdScheme*** is a powerful implementation, with:

- Unlimited precision, 'bignum' integral and rational arithmetic, double-precision floating point arithmetic to approximately 16 significant digits, and complex number arithmetic.
- Comprehensive file-handling facilities, as well as access through the Macintosh's serial ports to external devices such as the HyperBot™ robotic controller.
- Language extension using macros and transformers, support for advanced programming techniques such as delayed evaluation and streams, and first-class continuations and environments.

The result is a simple-to-learn but powerful language in a simple-to-use yet comprehensive programming environment.

.3 About This Book

This User's Guide is designed to complement the ***EdScheme*** interpreter. For your convenience it is divided into three parts. Part I deals with the programming interface. If you are already familiar with using the Macintosh computer then you will feel right at home, since ***EdScheme*** provides all the usual features that you will have come to expect from a Macintosh application. However, to help you take full advantage of ***EdScheme***'s advanced programming interface it would be wise to familiarize yourself with the various aspects of ***EdScheme***'s programming environment by reading Chapter 2, which takes you through a brief but informative ***EdScheme*** session. It will

also help you decide how to set up the ***EdScheme*** programming environment to best suit your needs and abilities. Chapter 3 discusses the programming environment from a more technical standpoint.

Part II of this Guide describes how ***EdScheme*** implements the Scheme programming language, and includes several example programs that may be entered directly into ***EdScheme***. Chapter 4 provides a gentle introduction to programming with lists (the backbone of many ***EdScheme*** programs) and vectors. Chapter 5 describes ***EdScheme***'s exceptional 'number crunching' abilities, including its ability to process 'bignums' (integers of practically unlimited size), rational numbers (or 'fractions') and complex numbers. Although including technical, mathematical descriptions of certain aspects of ***EdScheme***'s numeric processing, it begins with an intuitive introduction to the topic suitable for beginners. Chapter 6 describes ***EdScheme***'s file-handling capabilities and hardware support, including a description of the serial interface which is used to link ***EdScheme*** to robotic controllers. Chapters 7 and 8 describe the special ***EdScheme*** extensions that allow you to take full advantage of the Macintosh's powerful graphics and text processing abilities. Chapter 7 covers the ***EdScheme*** turtle graphics interface and discusses graphics windows and bitmaps. Chapter 8 describes ***EdScheme***'s program-generated text windows and menus, and introduces you to some of the techniques used to write Macintosh applications within the ***EdScheme*** programming environment.

Users who are not familiar with the Scheme programming language will find it informative to read through Chapter 2. It should be understood, however, that this Guide is not designed to be a Scheme primer. For that purpose there are several excellent introductory texts, among them those in the list that follows. Novice users are encouraged to take advantage of one or more of these books.

[1] *The Schemer's Guide* by Iain Ferguson with Edward Martin and Burt Kaufman, Schemers Inc., Fort Lauderdale, FL, 1992.

This book teaches the Scheme programming language from the very beginning. An understanding of elementary school arithmetic is all that is required as the book takes you from simple concepts all the way to more advanced programming techniques such as streams and object-oriented programming. *The Schemer's Guide* is currently in increasingly wide use as a textbook in high schools and as a freshman text in colleges. (***EdScheme*** provides a special 'Schemer's Guide Mode' that ensures compatibility with *The Schemer's Guide*. In this mode,

EdScheme uses strict error checking to catch the programming errors typically made by beginners. For details of how to set ***EdScheme*** so that it operates in Schemer's Guide Mode, see page 9.)

[2] *Scheme and the Art of Programming* by George Springer and Daniel P. Friedman, MIT Press, Cambridge, MA, 1989.

This college freshman-level textbook covers all the major aspects of Scheme programming, and includes a substantial section on programming with continuations (this is one of the few textbooks that deals with this difficult and advanced topic well). The book includes many example programs, and discusses many computer science issues from a Scheme viewpoint, including sorting procedures and object-oriented programming techniques.

[3] *Structure and Interpretation of Computer Programs* by Harold Abelson, Gerald Jay Sussman with Julie Sussman, MIT Press, Cambridge, MA, 1985.

A landmark book in the history of computer science education, *SICP* is an undergraduate textbook covering many aspects of computer science from an engineering standpoint. It includes sections on representations of digital circuits, constraints, streams, logic programming, and compiler construction. *SICP* is a ***must-read*** for students interested in constructing large Scheme programs. (Although this book employs non-standard Scheme programming elements, ***EdScheme*** interprets its programming examples without modification.)

[4] *Essentials of Programming Languages* by Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes, MIT Press, Cambridge, MA, 1992.

This book is suitable for upper-division and graduate-level courses in computer science, and demonstrates Scheme's amazing ability to 'prototype' other computer languages. The book provides a thorough introduction to interpreter and compiler design, with a great many program examples, all of which may be run in ***EdScheme***.

Part III of this Guide constitutes an exhaustive reference, a place you can turn to whenever you get into difficulty, whether you are seeking a solution to a problem concerning the programming environment—Chapter 9 provides a complete breakdown of ***EdScheme***'s menus—or the Scheme programming

language—Chapters 10, 11, and 12 combine to provide a comprehensive description of the Scheme programming language, including many examples.

.4 When You First Use *EdScheme*

If you are not familiar with the jargon of computers, compilers, and interpreters—perhaps this is your first foray into computer science—then don't worry! Using ***EdScheme*** is as simple as double-clicking on the ***EdScheme*** icon (the 'ink-pot λ' icon). However, the first time you start up the application you may want to take steps to 'configure' ***EdScheme*** so that it behaves according to your needs. You can also change the way ***EdScheme*** appears, to suit your personal taste.

If you are running under a multiprocessing environment such as Multifinder™ or System 7™, you may wish to begin by increasing ***EdScheme***'s 'partition size'. The partition size determines how much memory is allocated to ***EdScheme*** by the Macintosh computer when ***EdScheme*** starts up. The more memory ***EdScheme*** has, the faster it is able to run your programs, and the larger the programs it can process. The partition size that is right for you depends on the total amount of memory in your system, and on whether you require ***EdScheme*** to run concurrently with other applications. ***EdScheme*** provides a default partition size of 1500K, but on systems with a large amount of memory (4MB or more) you should consider increasing this figure. To change the partition size, first exit from ***EdScheme*** (if it is already running), select the ***EdScheme*** icon by clicking on it, then choose **Get Info** from the Finder's **File** menu and alter the partition size.

Most other changes to ***EdScheme***'s behavior or appearance are made by starting up the application and then choosing **Preferences** from the **File** menu. A cascading sub-menu then provides a further choice of five items:

Windows, **Language**, **Debugging**, **Memory**, and **Specify ...**.

Detailed information about each of these is provided in Chapter 9: *The **EdScheme** Menus* starting on page 71. Here are two changes you may want to make right away:

1. If you intend to use ***EdScheme***'s extensive graphics capabilities, you will need to set aside some memory for this purpose. To do so, start ***EdScheme*** by double-clicking on its icon and then choose the **Memory**

item from the Preferences sub-menu in the File menu. *EdScheme* displays a dialog showing the various uses to which *EdScheme* puts the memory provided by the Macintosh. One of the entries in this dialog shows the amount of memory allocated to the **Graphics** buffer; by default, this amount is zero. To use *EdScheme*'s powerful graphics commands you must allocate sufficient memory to the **Graphics** buffer. All you need to know in order to perform this task is explained on page 80 in Chapter 9.

2. If you are learning to program in Scheme using *The Schemer's Guide* (see [1] in the bibliography in the previous section), you should set *EdScheme* to Schemer's Guide Mode. This forces *EdScheme* to conduct more thorough error checking of the programs you write, trapping the kinds of errors that are often made by beginners. (These errors are ignored when *EdScheme* is in Standard Scheme Mode, and this sometimes leads to unexpected and confusing results.) You can set *EdScheme* to Schemer's Guide Mode by choosing Language from the Preferences sub-menu of the File menu and clicking on the radio button labeled Schemer's Guide Mode. As you do this, certain other selections will automatically be made for you, so as to guarantee compatibility with *The Schemer's Guide*. For further information about this automatic behavior and how to reverse it (if you so desire), see page 75 in Chapter 9: *The EdScheme Menus*.

-o O o-

2 An *EdScheme* Session

In this chapter we describe a short ***EdScheme*** session that shows how the ***EdScheme*** programming environment may be used to best advantage.

2.1 Starting Up

Before starting your first ***EdScheme*** session, you should make sure that all the files the application needs have been installed on your Macintosh hard drive (if you have one) and that you have a backup copy of the installation diskette. (See also Section 1.4: *When You First Use* ***EdScheme*** in Chapter 1.)

To launch ***EdScheme***, double click on the ***EdScheme*** icon (an ink-pot over the Greek letter lambda, λ). There will be a short pause, during which ***EdScheme*** displays a 'splash screen' copyright notice, and then a window entitled 'Transcript' will appear. This window is called 'the **Transcript Window**'. If the file 'EdScheme Init.s' is in your 'EdScheme' folder, the message 'Initializing EdScheme.....Done.' will gradually be printed in this window, and beneath this a right arrow symbol, '$\Longrightarrow$', called 'the **prompt**', will appear. To the right of this prompt there will be a flashing black line, called 'the **caret**'. If you are impatient to see ***EdScheme*** in action, type **(+ 3/2 1/2 5)** and then press the RETURN key. The contents of the Transcript Window will resemble the diagram at the top of the next page (except that if you are working on a color system, your Transcript Window will now contain multicolored text!).

You may have noticed that, when you typed the right parenthesis and before you pressed the RETURN key, ***EdScheme*** flashed the left parenthesis.

```
EdScheme (with integrated editor and turtle graphics interface).
Release 4.0, © 1991, 1993 Schemers Inc. SN# 007.
Standard Scheme Mode. Tuesday, January 5, 1993, 9:42 PM.

Initializing EdScheme.....Done.
⟹ (+ 3/2 1/2 5)
7
⟹ |
```

This is an important service provided by all of *EdScheme*'s edit windows. (In making these comments, we are assuming that you have not yet altered *EdScheme*'s default settings. In particular, this 'parenthesis-matching' behavior only occurs if the Format & Paren-Matching checkboxes are checked in the Windows dialog accessed through the Preferences sub-menu in the File menu.)

The expression **(+ 3/2 1/2 5)** is called a **Scheme expression.** When you pressed RETURN after the right parenthesis, *EdScheme* evaluated the Scheme expression to an 'answer', technically called a **data expression.** *EdScheme* makes it easy for you to distinguish a Scheme expression from a data expression by printing them in different colors (and in different typefaces).

If you are already familiar with Scheme then you may want to try evaluating a few more Scheme expressions in order to get a feel for how the Transcript Window behaves. However, before you tackle any real projects you should read the next section, which explains how to optimize *EdScheme*'s programming environment.

.2 Organizing the Desktop

At any time during an *EdScheme* session you may type Scheme expressions into the Transcript Window and evaluate them to data expressions, but experience has shown that in general this is seldom the most productive way to work. An important feature of the Transcript Window is that it provides a true transcript (or history) of an *EdScheme* session. In particular, you cannot go back and 'change history' by editing a previously evaluated Scheme expression, even if that Scheme expression generated an error. You may,

of course, use the standard Macintosh editing facilities to Copy a Scheme expression, Paste it into the Transcript Window at the current prompt, and then edit it before pressing RETURN, but this is usually not very convenient.

An alternative is to use a **document window**, as follows: First, use the 'size box' in the bottom right-hand corner of the window to resize the Transcript Window so that it takes up about one-third of the height of the desktop. Then move the Transcript Window by clicking in its title bar and dragging with the mouse until it lies in the bottom one-third of the desktop. Next, choose New from the File menu. An empty window, called a 'document window', will appear with the title 'Untitled'. If necessary, resize and move this window so that it occupies the top two-thirds of the screen, thereby allowing the Transcript Window to remain in view in the bottom one-third. A caret will be flashing in the top left corner of the document window.

Now type the Scheme expression **(+ 2 -1/2 5 3)**. This time, what you type should appear in the document window in a 'simple' black font. Note that pressing the RETURN key now has no effect other than to move the caret down by one line each time it is pressed—*EdScheme* does not attempt to evaluate the expression. However, when the caret is placed immediately after the right parenthesis,[1] the matching left parenthesis flashes, just as it did in the Transcript Window. Make sure that the caret is placed immediately after the right parenthesis, and then press the ENTER key on the Macintosh's 'keypad'—the small group of keys on the right of the keyboard. The Scheme expression **(+ 2 -1/2 5 3)** appears in the Transcript Window next to the last prompt, and beneath it appears the answer: **19/2**.

Of course, you could just as easily have typed the Scheme expression directly into the Transcript Window. But let's suppose you had really meant to type the number 5/2 instead of 3. In the Transcript Window, you would have no alternative but to copy-and-correct or retype the entire expression before re-evaluating it. In a document window, however, you can correct errors by editing your previous work. In this case, you would use the mouse or the left arrow key to move the caret just to the right of the '3', press

[1] Each press of the right or left arrow key moves the caret one character in the corresponding direction. Similarly, each press of the up or down arrow key moves the caret one line in the corresponding direction. This behavior holds true both in the Transcript Window and in any document window. In addition, if you hold down the OPTION key while pressing the right, left, up, or down arrow key, the caret will move to the end of the current line, the start of the current line, the top of the file, or the bottom of the file, respectively. Finally, you can move the caret to any desired location in the current window by moving the mouse pointer to that location and clicking once.

the DELETE key to erase it, and then type '5/2'. Finally, move the caret so that it once again lies immediately after the right parenthesis (causing the left parenthesis to flash), and press Keypad-ENTER. The amended Scheme expression will be evaluated in the Transcript Window as before (except, of course, that the answer will now be 9).

.3 The Current Expression

The ease with which Scheme expressions can be edited and re-evaluated is just one reason why you will find it convenient to do most of your work in a document window rather than directly in the Transcript Window. But there are additional features of document windows that make them truly indispensable! Leaving the amended Scheme expression as it is, press the RETURN key two or three times, then type the second Scheme expression shown in the following diagram:

```
(+ 2 -1/2 5 5/2)

(first (quote (monday tuesday wednesday)))
```

As you type, note that each time you type a right parenthesis, *EdScheme* flashes the matching left parenthesis. You can tell when you have finished typing this Scheme expression by watching which left parentheses flash as you type right parentheses; when the parenthesis to the left of the symbol **first** flashes, the expression is complete. Making sure that the caret is immediately after the final right parenthesis, press the Keypad-ENTER key once more. The Transcript Window again responds by evaluating the Scheme expression to produce the data expression **monday**. Now move the caret back up to the top line, again immediately after the right parenthesis, and hit Keypad-ENTER. This time, the Scheme expression on the top line is re-evaluated.

This example illustrates the concept of the **current expression**. In general, the current expression is the Scheme expression whose left parenthesis is flashing, or whose left parenthesis was the last one that *EdScheme* flashed. Pressing the Keypad-ENTER key tells *EdScheme* to 'paste' the current expression into the Transcript Window, to evaluate it, and then print the resulting data expression.

You may care to try experimenting with changing the current expression. *EdScheme* provides a window, called the **Expression Window**, whose

sole purpose is to display the current expression. To view this window, choose Show Expression from the Edit menu. Then, using the mouse or the arrow keys, move the caret to various places in the document window (or even in the Transcript Window) and notice how the current expression changes. You may evaluate the expression shown in the Expression Window at any time by pressing Keypad-ENTER or, equivalently, by choosing Expression from the Evaluate menu.

.4 Indentation and Formatting

Next, let's try something a little more ambitious! Leave some space under the last expression in the document window (or, if you prefer, erase the contents of the document window by choosing Select All and then Cut from the Edit menu), and then type the following:

```
(define cube
  (lambda (n)
    (* n n n)))
```

Notice that each time you press the RETURN key, *EdScheme* indents the next line.[2] *EdScheme* uses a sophisticated indentation routine to calculate where each line should begin, so that your Scheme programs automatically appear in standard layout. Make sure the definition for **cube** is the current expression (the easiest way to ensure this is to move the caret so that the first left parenthesis—preceding the keyword **define**—is flashing) then press Keypad-ENTER to evaluate the definition. To test the definition, try evaluating something like **(cube 3)**, either from the document window using Keypad-ENTER, or directly in the Transcript Window. (If you are feeling a little more adventurous, try evaluating **(cube 12345678987654321)**—*EdScheme* will tell you that the answer is 1881676411868862234942354805142998028003108518161.)

Of course, very few people are able to create flawless procedure definitions every time, so you will usually find yourself having to edit the definitions you write. This action will often transform what began as a neatly indented definition into a ragged collection of lines of code, arranged apparently at random. For our purposes, we can simulate this effect by just adding or

[2] As mentioned earlier, we are assuming that *EdScheme*'s default settings have not yet been altered.

deleting leading spaces from the definition of **cube**. To reinstate the original, neatly indented format of the definition, all you have to do is place the caret on one of the three lines of the definition, and then choose [Format] from the [Edit] menu. *EdScheme* will rapidly change the indentation of each line of the definition so that it once again appears in standard form.

For your convenience, *EdScheme* also provides two alternative 'hot key' methods of choosing the [Format] menu item: You can simply press the TAB key while holding down the OPTION key, or you can press the M key while holding down the COMMAND key.

EdScheme's formatting capabilities extend beyond the reformatting of individual procedure definitions, however. If you wish, you can format a single line so that its leading character is correctly indented relative to the previous lines. To do this, place the caret on the line in question and press the TAB key—this time without the OPTION key. At the other extreme, you can format several definitions—or even a whole document—all at once by using the mouse to mark the definitions (or by selecting the whole document by choosing [Select All] from the [Edit] menu), and then choosing [Format] (or using one of the OPTION-TAB or COMMAND-M hot key combinations).

.5 Indexing Document Windows

Here's a more complex definition that you should type below the definition for **cube**:

```
(define factorial
  (lambda (n)
    (if (zero? n)
        1
        (* n (factorial (- n 1))))))
```

When you have finished typing this definition, press the OPTION key and, while still holding it down, click the mouse on the title bar of the document window. A 'popup' menu will appear containing the two items [cube] and [factorial], listed in alphabetical order. If you choose either of these items, the corresponding procedure definition will be selected in the document window (that is, it will marked as though ready to be cut or copied, say). This automatic indexing of the contents of document windows is invaluable when

working with larger documents; keeping track of the location of many definitions is a task best left to the computer!

.6 Using Multiple Documents

The ***EdScheme*** implementation disk includes several extensive Scheme documents, some of which contain a large number of definitions. To open a document window for an existing document, choose **Open ...** from the **File** menu. You will be shown a standard Macintosh File Selector dialog; double-click on the document you wish to open.

Very often, the development of a large or medium-sized program involves definitions spread over several Scheme files, so ***EdScheme*** provides a facility for fast-switching between open document windows. Document windows are numbered in the order in which they are opened, and as each document window is opened, its title is entered into the **Windows** menu on the main ***EdScheme*** menu bar. Associated with each such menu item is a hot-key combination comprising the COMMAND key and the document window's number. For example, if you have two open document windows (each one either having been opened using **Open ...** from the **File** menu or created using **New** from the same menu), alternately pressing COMMAND-1 and COMMAND-2 (or choosing the corresponding items from the **Windows** menu) will switch you between the two documents.

.7 Saving your Documents

Before you end your ***EdScheme*** session you may wish to save your documents (and possibly your Transcript Window) to disk. This couldn't be easier! Just make sure that the window whose contents you want to save is the currently selected window (the easiest way to ensure this is to click on the window's title bar), then choose **Save** from the **File** menu (or use the **Save** item's alternative hot-key combination, COMMAND-S). You will first be prompted to provide a name for the file, unless the contents of the document window have previously been read from or written to a file. If you are an experienced computer user, you will know that it is very unsafe to leave the important task of saving documents until the end of the session. Whether experienced or not, you should try to get into the habit of saving your documents at

regular intervals, say, every few minutes, by pressing COMMAND-S while in each document window in turn. That way, you can avoid the frustrating experience of losing hours of work because of an unexpected, momentary fluctuation in electrical power.

– o O o –

3 Programming Environment Windows

EdScheme provides six types of windows within its programming environment:

- The **Transcript Window** is a channel of communication between you and the *EdScheme* interpreter, and it maintains an historical record of the current *EdScheme* session.
- **Debug Transcripts** provide a more specialized form of communication with the interpreter within the context of 'error environments'.
- The **Trace Window** allows you watch programs as they run.
- **Document Windows** enable you to store Scheme programs in disk files, and they serve as a convenient canvas on which to create new programs.
- The **Expression Window** displays the current expression.
- The **Clipboard Window** displays the contents of the clipboard.

All these windows are described in detail below. In addition, *EdScheme* programs can generate two other types of windows, namely, graphics windows and text windows. These are discussed in Chapters 7 and 8.

.1 Transcript Windows

This section describes the operation of the primary Transcript Window and Debug Transcript windows. Each Transcript Window is a colored, styled-text editor whose behavior and appearance may be customized to suit your needs.

Text in Transcript Windows may appear in as many as three different colors and fonts. System messages, such as error messages and the *EdScheme* banner, are printed in one color, Scheme expressions (your programs) print in a second color, and data expressions (the output produced by your programs) print in a third color. The following table lists the default colors and fonts for the three types of expressions.

	System	*Scheme*	*Data*
Font	Geneva	Chicago	Monaco
Color	Blue	Black	Red

By default, each font is printed in a plain style, but it is possible to customize the styles, as well as the colors and fonts, by means of the **Windows** dialog that is accessed through the **Preferences** sub-menu of the **File** menu.

The same dialog gives you the option of switching on and off *EdScheme*'s automatic expression-formatting and parenthesis-matching feature in the context of Transcript Windows. By default, this feature is activated.

You can save the contents of the Transcript Window or any Debug Transcript Window using the **Save** or **Save As ...** items in the **File** menu. Having saved in one of these ways, however, the only way to reinstate the saved contents *as a Transcript Window* is to use the **Open As Transcript ...** item in the **File** menu; such saved contents cannot be read into a document window. To save the contents of a transcript or debug transcript window so that they *can* be read into a document window, use the **Save As Text ...** item in the **File** menu. Be warned, though, that this method of saving makes it impossible for the saved contents to be reinstated as a Transcript Window.

If you would like to open a fresh Transcript Window, you may do so using the **New Transcript ...** item in the **File** menu. Before the new window opens, you will be given the choice of saving or discarding the current Transcript Window.

The (primary) Transcript Window appears when *EdScheme* is started. Scheme expressions typed at the prompt ($\Longrightarrow$) are evaluated and their values are printed into this window. All the usual Macintosh editor functions (cut and paste, search and replace, for example) are available in the Transcript

Window, with one important proviso: *EdScheme* will not allow you to change anything that appears *before* the last prompt. This behavior ensures that the contents of the window provide a true transcript (or history) of the current *EdScheme* session.

On start-up, ***EdScheme*** allocates a block of memory for use by the Transcript Window. The amount of memory set aside is determined by the setting for the **Transcript** buffer in the Memory dialog accessed through the Preferences sub-menu in the File menu. As a session continues, the Transcript Window gradually fills this block of memory until eventually there is no room for any more input into the allocated memory. At this point, ***EdScheme*** automatically deletes text from the top of the transcript, releasing memory so that the session may continue. If you need to ensure that all of a session's transcript is kept, and you anticipate that an extensive record will be generated, then you should increase the memory allocated to the Transcript Window. (As with all changes to ***EdScheme***'s memory configuration, any alteration you make will not come into effect until after ***EdScheme*** is restarted.)

The Transcript Window may be closed (temporarily) in any of the standard Macintosh ways or from within an ***EdScheme*** program by evaluating the expression **(window-close *transcript*)**. Closing the Transcript Window does not cause it to lose its contents (in effect it is 'hidden' rather than 'closed'). It may be re-opened from the desktop by choosing Transcript from the Windows menu, or from within an ***EdScheme*** program by evaluating the expression **(window-select *transcript*)**. Alternatively, you can remove the window from view by evaluating the expression **(window-hide *transcript*)**, and then make it visible again by evaluating the expression

(window-show *transcript*).

If your ***EdScheme*** preferences are set appropriately—see the section of Chapter 9 (starting on page 77) that deals with the Debugging dialog accessed through the Preferences sub-menu in the File menu)—an error arising during the evaluation of a Scheme expression will cause a Debug Transcript to open. The environment that is current in this Transcript Window includes bindings for all the local variables that are in play at the time the error occurred. You may evaluate Scheme expressions in this window, and even generate new errors, which in turn will cause additional (dependent) Debug Transcripts to open.

If Debug Transcripts are enabled in the Debugging dialog just referred

to, then they may be temporarily inhibited by choosing **Debug Transcripts** from the **Evaluate** menu. If Debug Transcripts are enabled, a black diamond appears next to the menu item when the **Evaluate** menu drops down. If Debug Transcripts are enabled, but have been temporarily inhibited by choosing **Debug Transcripts** from the **Evaluate** menu, then this black diamond is not present. If Debug Transcripts are not enabled, then this menu item is 'grayed out', that is, it cannot be chosen.

Debug Transcripts are numbered in hierarchical fashion. Those that arise from errors generated in the Transcript Window are numbered sequentially in the form $1.n$ in order of generation. Those that arise from errors generated in a Debug Transcript Window $m.x$ are numbered sequentially in the form $(m+1).n$ in order of generation. The bindings that are current in a given Debug Transcript Window are cumulative; they include all the bindings in force in all of the Debug Transcripts in the 'ancestral line' that traces the given window's generation history all the way back to the Transcript Window itself.

You can define procedures and macros in a Debug Transcript window using **define**, **define-macro**, **define-alias**, and **define-transformer**, or you can load them from a file using **load** or by choosing **Load ...** from the **Evaluate** menu. However, such procedures and macros are inaccessible to all 'older generation' Debug Transcripts whose environment (or 'context') differs from that of the Debug Transcript into which those procedures and macros were introduced. If your preferences are set so that Debug Transcripts are only generated in a new context, then the procedures and macros are only accessible in the Debug Transcript into which they were introduced and its 'descendants'.

If you evaluate an expression that involves a mutator procedure such as **set!**, the relevant binding will be changed in the uppermost (that is, 'most ancient') Debug Transcript whose environment involves such a binding, and in all Debug Transcripts that trace their existence back to that Transcript.

The fonts, colors, and type styles used in Debug Transcripts are determined by the selections that have been made for the Transcript Window. The memory allocated to Debug Transcripts may be set by modifying the **Debugging** entry in the **Memory** dialog (see the similar note above concerning the memory allocated for the Transcript Window). The setting in this dialog specifies the amount of memory available to each Debug Transcript individually. Normally, Debug Transcript windows are short-lived, so they do not require as large an allocation of memory as the Transcript Window.

To close a Debug Transcript, click in its Close Box (the box at the left end of the window's title bar). A Debug Transcript differs from the Transcript

Window, however, in that by closing it you will lose its contents and—if its environment is different from that of its 'parent'—the environment in which it was created.

Pressing Keypad-ENTER causes the current expression to be pasted into the most recently active Transcript Window. The expression is then evaluated and the result printed in that window.

If an error occurs in a Scheme program, and the Transcript Window from which the program was run is currently hidden, ***EdScheme*** automatically shows that Transcript Window.

.2 The Trace Window

A third, more rudimentary type of Transcript Window is the Trace Window. This is the window to which trace information is written if you have checked the Output to Trace Window checkbox in the dialog accessed through the [Trace ...] item in the [Evaluate] menu. (See the detailed description on page 90 in Chapter 9: *The **EdScheme** Menus*.) The contents of this window are generated by ***EdScheme***. You may copy text from the Trace Window to the clipboard, but you will not be able to do any editing in the window itself. To show the Trace Window, choose [Trace Window] from the [Windows] menu. To erase everything from the Trace Window, choose [Clear Trace] from the [Evaluate] menu. The amount of memory allocated to the Trace Window is the same as that allocated to each Debug Transcript, and its behavior once that allocation is exceeded is the same as for the Transcript Window.

3.3 Document Windows

Documents are text files that may be created or opened in the usual way by choosing [New] or [Open ...], respectively, from the [File] menu. They may be edited using the standard Macintosh editing facilities. Finally, they may be saved and/or closed in the usual way by means of the [Save], [Save As ...], and [Close] items in the [File] menu. You may also close a document by clicking on its window's Close Box, whereupon you will be given the option of saving the document if you have made any changes since opening it.

Document windows are not directly manageable from within an ***EdScheme*** program. So anything you want to do with a document window must be

achieved using the keyboard, the mouse, and/or the main menu bar. In particular, choosing [Close all] from the [Windows] menu closes all open document windows. (It has no effect on any other kind of windows that may be open.)

If one or more document windows are open, you can use the bottommost items in the [Windows] menu or the corresponding hot key combinations to switch rapidly between all such open document windows. These menu items do not apply to any other kind of window. (See also Section 2.6: *Using Multiple Documents* on page 17 in Chapter 2.)

If you choose [Whole file] from the [Evaluate] menu, then all the Scheme expressions in the currently selected document will be evaluated, and the value of the final expression will be printed to the Transcript Window. (The same effect is produced if you choose [Select All] from the [Edit] menu and then choose [Selection] from the [Evaluate] menu.) Further related information is provided in the entry for the [Whole file] menu item on page 89 in Chapter 9: *The **EdScheme** Menus*.

With the help of the mouse, you can also select a block of text of any size in the currently selected document using one of the following methods:

1. Click once on the mouse button and then drag to some other location. All the text between the initial and final positions of the caret will be selected.

2. Quickly click twice on the mouse button and then drag to some other location. All the text from the word initially clicked on through the word on which the mouse button is released will be selected. (If either the initial or final position of the mouse happens to lie between two words, then ***EdScheme*** acts as if the mouse were on the word *following* its actual position.)

3. Quickly click three times on the mouse button and then drag to some other location. All the text from the line initially clicked on through the line on which the mouse button is released will be selected.

If you select a block of text in the current document in one of these ways, and then choose [Selection] from the [Evaluate] menu, all the expressions in the marked block will be evaluated, and the value of the last one will be printed to the Transcript Window. Further related information is provided in the entry for this menu item on page 89 in Chapter 9: *The **EdScheme** Menus*.

The [Windows] dialog box, accessed through the [Preferences] sub-menu in the [File] menu, gives you control over whether or not ***EdScheme***'s expression-

formatting and parenthesis-matching feature is activated in document windows. Let us suppose that you have checked the appropriate **Format & Paren-Matching** checkbox. Then, as explained in Section 2.3: *The Current Expression* in Chapter 2, if you place the caret immediately after a right parenthesis and either hit the **Keypad-ENTER** key or choose **Expression** from the **Evaluate** menu, the current expression will be pasted into the Transcript Window and its value will be printed to the Transcript Window.

In fact, if you have checked both **Format & Paren-Matching** checkboxes in the **Windows** dialog accessed through the **Preferences** sub-menu in the **File** menu, then, whenever you place the caret immediately after a right parenthesis of a Scheme expression in the Transcript Window, any Debug Transcript Window, or any document window, the matching left parenthesis will flash, even if the partners in this matching pair are not both visible in the window at the same time. (In other words, you may have to scroll the window up in order to locate the flashing matching left parenthesis.)

The memory allocated to document windows may be changed by modifying the **Text** buffer entry in the **Memory** dialog, which is accessed through the **Preferences** sub-menu in the **File** menu.

.4 The Expression Window

Whenever the caret is immediately after a right parenthesis either in a document window or in a Transcript Window, then the expression comprising that right parenthesis, its matching left parenthesis, and everything in between is the Current Expression. If the caret is anywhere else, either in a document window, in the Transcript Window, or in a text window, then the Current Expression is whatever it was on the last occasion that the caret immediately followed a right parenthesis. The Current Expression may be directly specified under program control using the procedure **expression-set-text** (see the relevant entry on page 158 in Chapter 11: *Syntax and Semantics*).

The sole function of the Expression Window is to display the Current Expression. If the Current Expression happens to be so extensive that it cannot all appear in the Expression Window at once, then as much of that expression as will fit in the window, starting from the beginning of the expression, will be displayed. In such a case, the remainder of the Current Expression may be inspected by scrolling or resizing the Expression Window.

The Expression Window is activated by choosing **Show Expression** from

the Edit menu or by using the COMMAND-J hot key combination. It is particularly useful when typing the final parentheses of lengthy definitions—type right parentheses one by one as you watch the Expression Window; when sufficient right parentheses have been typed, the Current Expression will start with **(define ....**

The contents of the Expression Window may be accessed under program control using the procedure **expression-text** (see the relevant entry on page 158 in Chapter 11: *Syntax and Semantics*). The Expression Window may be dismissed either by clicking on the Close Box in the upper left corner of the window or by choosing Hide Expression from the Edit menu. The contents of the Expression Window (that is, the Current Expression) are not affected by closing the window.

.5 The Clipboard Window

The sole purpose of the Clipboard Window is to show what was marked at the time of the last Cut or Copy (using the Edit menu) or whatever was designated at the time of the last evaluation involving **text-cut**, **text-copy**, or **clipboard-set-text** (see the entries for these procedures in Chapter 11: *Syntax and Semantics*), whichever was most recent.

Choosing Paste from the Edit menu or evaluating an expression involving **text-paste** (see the relevant entry on page 225 in Chapter 11: *Syntax and Semantics*) will cause the contents of the Clipboard Window to be pasted into the currently selected window, either Transcript, document, or a Scheme-generated text window (see Chapter 8).

The contents of the Clipboard Window may be accessed under program control using the procedure **clipboard-text** (see the relevant entry on page 135 in Chapter 11: *Syntax and Semantics*).

Part II

The Programming Language

4 Lists and Vectors

Programming in Scheme involves dealing extensively with **lists**. Indeed, *Lisp*, the language from which Scheme is derived, is an acronym for '**LIS**t **P**rocessing'. Lists are special kinds of structures in which data may conveniently be stored. Of course, they are not the only such structures; alternatives include those known as **vectors**. This chapter briefly introduces lists and vectors, the objects from which they are built, and some of the primitive procedures with which they are manipulated.

4.1 Lists

A **list** is a flexible data structure that contains zero or more items of data—called **data expressions**—which are accessed sequentially. In this Guide we write lists using the `typewriter` typeface, like this:

```
(this is a list)
```

The expressions that make up Scheme program—these are known as **Scheme expressions**—*look* just like data expressions. This is often very confusing to beginning Scheme programmers, so in this Guide we distinguish Scheme expressions from data expressions by printing Scheme expressions using the **sans serif** typeface, like this:

```
(first '(second third))
```

The list containing no data expressions is called the **empty list**, and prints as `()`. The predicate **null?** tests its input for 'emptiness', returning the boolean

`#t` if the input is the empty list and the boolean `#f` otherwise. Non-empty lists may be constructed using the procedure **cons**. For example, **(cons 'harry '())** evaluates to the list that prints as `(harry)`. Here are some more examples, where '↦' should be read 'evaluates to':

```
(cons 1 (cons 2 '()))            ↦  (1 2)
(cons 'fred '(harry))            ↦  (fred harry)
(cons '(fred harry) '(jane mary)) ↦  ((fred harry) jane mary)
```

If the second input to **cons** is not a list, then the result is an **improper list**, which prints using 'dot notation'. For example,

```
(cons 1 2)                  ↦  (1 . 2)
(cons 1 (cons 2 (cons 3 4))) ↦  (1 2 3 . 4)
```

WARNING: Improper lists are *not* permitted in Schemer's Guide Mode. For more information about ***EdScheme***'s two language modes, see the section—beginning on page 74 in Chapter 9: *The **EdScheme** Menus*—that deals with the Language dialog accessed through the Preferences sub-menu in the File menu, and for information about improper lists refer to Section 10.3: *Pairs and Lists* in Chapter 10: *Data Expressions*. The predicate **list?** returns the boolean `#t` if and only if its input is a (proper) list; the predicate **pair?** returns `#t` if and only if its input is either a list or improper list. (In fact, improper lists and non-empty proper lists are collectively known as **pairs**.)

The principal way to access the data expressions in a pair is to use the procedures **first** and **rest**. The procedure **first** returns the first data expression in the input pair, that is, the data expression that appears furthest to the left in the list's printed representation. If its input is a proper list, the procedure **rest** returns the list consisting of all but the first data expression. Otherwise, it behaves as illustrated by the last two examples below:

```
(first '(a b c d e))        ↦  a
(rest '(a b c d e))         ↦  (b c d e)
(first (rest '(a b c d e))) ↦  b
(first '(1 . 2))            ↦  1        [in Standard Scheme Mode]
(rest '(1 . 2))             ↦  2        [in Standard Scheme Mode]
(rest '(a b . c))           ↦  (b . c)  [in Standard Scheme Mode]
```

The procedures **car** and **cdr**—whose names have historical connections to Scheme's parent language, Lisp—are exactly equivalent to **first** and **rest**, respectively, and may be used in their place, if you so prefer.

Among the other procedures for accessing data expressions in a (proper) list are **last**, **list-ref**, and **nth**. The procedure **last** returns the last data expression in its input, which must be non-empty. (The last data expression is the one that appears furthest to the right in the list's printed representation.) To gain access to data expressions that are in neither first nor last place, without having explicitly to 'chop' your way in from the left using **rest** and picking out the desired expression with a final **first**, you may use either of the procedures **list-ref** and **nth**. Of these, **list-ref** assumes that you take the standard Scheme view that lists are zero-referenced, in that the first data expression in the list has index 0, the next has index 1, and so on. On the other hand, it is sometimes useful for educational purposes to use the more intuitive one-referencing system, whereby the first data expression has index 1. The procedure **nth** assumes that lists are one-referenced.

```
(list-ref '(a b c d e) 2)  ↦  c
(nth 2 '(a b c d e))       ↦  b
```

(Notice that **list-ref** and **nth** do not take their inputs in the same order as each other.)

The procedure **length** returns the length of the input list, that is, the number of data expressions it contains. As is the case for many of the primitive procedures that manipulate lists, it may be written as a derived procedure:

```
(define length
  (lambda (r)
    (if (null? r)
        0
        (+ 1 (length (rest r))))))
```

See Section 12.6: *Symbols and Lists* in Chapter 12: *Language Elements* for a catalogue of ***EdScheme***'s other list-manipulating procedures. The catalogue refers you to the appropriate entries in Chapter 11: *Syntax and Semantics*.

4.2 Vectors

While lists provide the most flexible data structure for writing Scheme programs, they suffer from a major disadvantage. As was hinted in the previous section, in order to access data expressions near the end of the list, you have to 'recur' down the list, repeatedly taking the **rest** of the list until the required

data expression is in first place in what remains of the original list. For example, to obtain the 500th data expression in a list containing 1000 data expressions, you must take the **rest** of the list 499 times, and finish off with a single **first**. This may appear to contradict the description in the previous section of the procedures **list-ref** and **nth**. However, these procedures operate internally by 'recurring down the list' in exactly the way just described; they are provided simply as a convenience to hide the 'messy details' from your eyes! Naturally, this means that the time it takes to access a data expression in a list is longer the further along the list the expression is; this can have a significant impact on the efficiency of certain kinds of programs.

For situations where you frequently want to access the nth element of a list, discarding or ignoring its first through $(n-1)$st elements, Scheme provides a data structure called a **vector**. Vectors are like lists in the sense that they can be used to 'store' many data expressions (including other vectors), but the elements in a vector cannot be accessed using the procedures **first** and **rest**. Instead, each element is accessed directly by reference to its **index**. Every vector is zero-referenced, that is, its first data expression has index 0, the next has index 1, and so on. (Unlike the situation for lists, ***EdScheme*** does ***not*** provide a one-referenced method of accessing the data expressions in a vector.)

For example, we can use the procedure **vector** to make a new vector containing the symbols `fred`, `jane`, and `harry` and assign it the name ***v1***, as follows:

```
(define v1 (vector 'fred 'jane 'harry))
```

To access the elements of this vector we use the procedure **vector-ref** as follows:

```
(vector-ref v1 0)  ↦  fred
(vector-ref v1 1)  ↦  jane
(vector-ref v1 2)  ↦  harry
```

The printed representation of a vector resembles that of a list, the only difference being that the leading left parenthesis is preceded by the hash symbol, **#**:

```
v1 ↦ #(fred jane harry)
```

Vectors may be used explicitly in Scheme expressions provided they are 'quoted', as in the following example:

(vector-ref '#(a b c d e) 4) ↦ **e**

Each element of a vector is accessed equally quickly by ***EdScheme***, no matter where it lies within the vector. In some circumstances, this represents a distinct advantage over lists. Unlike lists, however, vectors cannot be lengthened or shortened; they forever remain the length they are at the time they are created. In fact, an alternative way to bring a vector into existence is to use the procedure **make-vector**, which takes the desired length of the vector being created as one of its inputs. (To discover the length of a given vector, you may use the procedure **vector-length**.) The only way to modify the contents of a vector is to use the procedure **vector-set!**, which requires explicit reference to the index of the item you wish to modify.[1] This 'rigidity' of vectors highlights the fact that, in a sense, vectors trade speed for flexibility.

The procedures mentioned in the previous paragraph are all catalogued in Section 12.4: *Vectors* in Chapter 12: *Language Elements*, together with a few other vector-manipulation procedures. The catalogue refers you to the relevant entries in Chapter 11: *Syntax and Semantics*.

– o O o –

[1] In the Scheme community, '!' is pronounced 'bang', so 'vector-set!' is read as 'vector set bang'.

5 Numbers and Numeric Procedures

This chapter includes information concerning the various kinds of numbers that *EdScheme* supports, as well as some of the large number of procedures it provides for manipulating them.

5.1 Types of Numbers

EdScheme distinguishes numbers from boolean objects, pairs, symbols, characters, strings, vectors, procedures, and the empty list. They cause the predicate **number?** to return `#t`; all other data expressions return `#f` when input into the predicate **number?**.

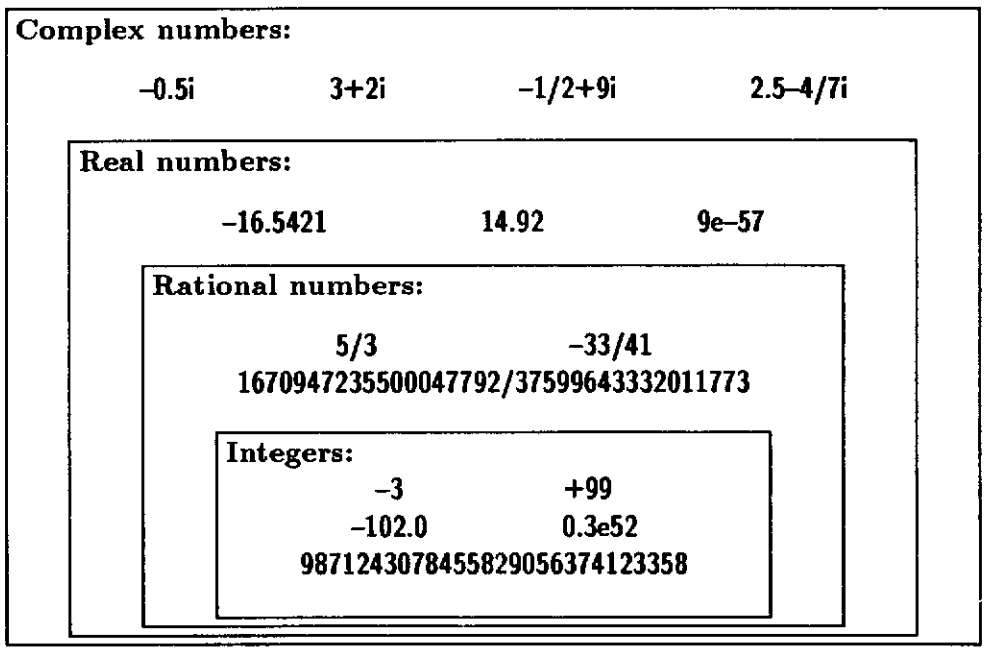
All numbers are complex. Among the complex numbers, some are real, some are rational, and some are integers. These categories are nested in the sense that

- all integers are rational;
- all rational numbers are real;
- all real numbers are complex.

The categories into which a given number falls may be determined by using the predicates **integer?**, **rational?**, **real?**, and **complex?**.

5.2 Numeric Constants

In *EdScheme*, numbers may be named using standard mathematical notation, as follows:



In addition, *EdScheme* provides a powerful facility whereby you may name numbers in a variety of radixes (that is, bases) and for specifying numbers as being exact or inexact. Note however that, unless you specify otherwise using the **number->string** procedure (see the relevant entry on page 188 in Chapter 11: *Syntax and Semantics*), *EdScheme* represents numbers externally—for example, when it prints the result of performing some numerical calculation—in radix 10 notation.

5.3 Prefixes

Numeric constants may be written in binary, octal, decimal, or hexadecimal form, each of which has a corresponding radix prefix, as follows:

Prefix	*Meaning*
#b	binary
#o	octal
#d	decimal
#x	hexadecimal

If no prefix is given, the numeral is assumed to be a decimal, that is, in radix 10.

In addition, numeric constants may optionally be preceded by an 'exactness prefix', either **#i** for inexact, or **#e** for exact. (See Section 5.9: *Exactness* starting on page 44.)

5.4 Digits

The digits available depend on which radix you are using. They are as follows:

Radix	*Digits*
2	**0,1**
8	**0,1,2,3,4,5,6,7**
10	**0,1,2,3,4,5,6,7,8,9**
16	**0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f**

5.5 Integers

Integers in any of the supported radixes are denoted by 'words' made up of an optional prefix (which may be a radix prefix, or an exactness prefix, or a combination in either order of a radix prefix and an exactness prefix), followed (in the case of negative integers) by a minus sign or (in the case of positive integers) by an optional plus sign, followed by one or more digits appropriate to the radix in question. (We say that the portion of such a 'word' following the initial prefix(es) is 'in exact integer form'.) Examples:

#o707 -52 #i#x-2cf #b10101 #d-999 1024 +007

(When printing such integers, ***EdScheme*** strips off any leading zeros and/or plus signs and, if necessary, translates into radix 10 form.) Neither commas, periods, nor spaces may be used to delimit thousands. Integers in this form are exact unless they include a **#i** prefix, in which case they are inexact.

Alternatively, the 'word' naming an integer may have all the digits in its 'tail'—starting anywhere past the first digit and going all the way to the end of the 'word'—replaced by #-signs. Examples:

#o70# #b1#### #e#x-2c##

(We say that the portion of such a 'word' following any initial prefix(es) is 'in unknown digit form'.) ***EdScheme*** prints such integers by replacing each #-sign by 0, translating into radix 10 form if necessary, and—unless a **#e** exactness prefix is present—appending '.0' on the end. For example,

#o70# ↦ 448.0
#e15## ↦ 1500

Integers in this form are inexact unless they include a #e prefix, in which case they are exact.

In addition, integers may be denoted in rational number form (see the next section) using an optional prefix, followed by an integer in exact integer or unknown digit form 'over' one of its exact divisors, also in exact integer or unknown digit form. (If unknown digit form is involved, the question as to whether the denominator is an exact divisor of the numerator is settled *after* all the #-signs have been replaced by zeros.) Examples:

-15/3 **#x2a/e** **#b#e1111##/1010#**

Integers in this form are exact if they involve no #-signs or if they include a **#e** prefix. Otherwise, they are inexact.

In the *decimal* case only, there are three additional ways to denote integers: An optional exactness prefix, followed by

- an optional decimal integer in exact integer form, followed by a decimal point, followed by a (possibly empty) string of zeros, followed by a (possibly empty) string of #-signs. (If there is no integer in front of the decimal point, then there must be at least one zero immediately following the decimal point.) Examples:

  123.0 **#e5040.000000** **5.** **5.09##** **#e.0#**

  If an integer entered in this form has a **#e** exactness prefix, then it is exact and ***EdScheme*** prints it by deleting the decimal point and all that follows. In every other case, it is inexact and ***EdScheme*** replaces whatever follows the decimal point by a single zero.

- **a decimal integer in unknown digit form, followed by a decimal point, followed by a (possibly empty) tail of #-signs. Examples:**

  12#.# **#e5###.######**

If an integer entered in this form has a **#e** exactness prefix, then it is exact and ***EdScheme*** prints it by replacing all the #-signs by zeros and then deleting the decimal point and all that follows. In every other case, it is inexact and ***EdScheme*** changes all the #-signs to zeros and then replaces whatever follows the decimal point by a single zero.

- an 'exponential form' real number in which what follows the exponent marker represents a number greater than or equal to the number of digits from the decimal point to the last non-zero, non-#-sign digit after the decimal point. (See Section 5.7: *Real Numbers* starting on page 41 to find out both what this means and how ***EdScheme*** prints integers named in this way.) Examples:

  #i-30.4020##e3 **0.007d12**

  Integers entered in this form are inexact unless they include a #e prefix, in which case they are exact.

EdScheme supports exact 'bignum' integers of practically unlimited size. You do not have to issue any special commands to activate a 'bignum mode'; ***EdScheme*** automatically uses bignums whenever it is appropriate to do so. Bear in mind, however, that bignums are exact. This means that, if you want a numerical calculation to produce a bignum result, then you must stay away from procedures that only return inexact numbers. The 'square root' procedure **sqrt** falls into this category (see the relevant entry on page 211 in Chapter 11: *Syntax and Semantics*). To produce exact square roots of bignum integers, you will need to define an 'exact-root' procedure, such as the one—based on the iterative Newton's Method for finding square roots—shown at the top of the next page. Then, for example, **(exact-root (expt 11 250))** returns the 131-digit number that is the actual square root of this number. (Check it for yourself against the result of evaluating **(expt 11 125)**.) In comparison,

```
1.493088824218035e+130
```

(which is the result generated by **(sqrt (expt 11 250))**) is pretty feeble!

.6 Rational Numbers

Rational numbers that happen also to be integers are denoted as described in the previous section. Every other rational number is denoted in one of the ways itemized beneath the figure on the next page.

```
(define newton
  (lambda (n a)
    (/ (+ a (/ n a)) 2)))

(define exact-root
  (lambda (n)
    (letrec
      ([loop
        (lambda (newa olda diff)
          (cond
            [(= n (* newa newa)) newa]
            [(<= diff (abs (- newa olda)))
             (sqrt n)]
            [else (loop (floor (newton n newa)))
                        newa
                        (abs (- newa olda)))])])
      (loop (floor (inexact->exact (sqrt n))) n n))))
```

Figure 5.1: An Exact Square Root Procedure

Rational numbers may be denoted:

- as a 'word' formed by an optional **#e** exactness prefix and/or an optional radix prefix, followed possibly by a minus sign or a plus sign, followed by one or more appropriate digits, followed by a 'forward slash' (/), followed by one or more appropriate digits. Examples:

  1/2 -12/35 #o22/7 #b-1010101/11 #e+45/24

- as a number in decimal, unknown digit, or exponential form (see the next section on real numbers), preceded by a **#e** exactness prefix. Examples:

  #e6.52## #e12.5 #e24e-5 #e-43##d-10

Note that *EdScheme* accepts any rational number that is named according to the foregoing description. Whenever it *prints* a rational number, however, that number is always in lowest terms, with any leading zeros in the numerator

and/or denominator removed, and—unless you specify otherwise by using the **number->string** procedure (see the relevant entry on page 188 in Chapter 11: *Syntax and Semantics*)—it will be in radix 10 notation. For example,

If you enter	*EdScheme prints*
-15/6	**-5/2**
#b1111/0101	**3**
#x-2ac/00ba	**-114/31**
#e0.6	**3/5**

The only inexact rational numbers are inexact integers. All other rational numbers are exact.

Note that ***EdScheme*** does *not* support mixed numbers such as three-and-five-sixths; such numbers should instead be represented as 'improper fractions', thus: **23/6**.

.7 Real Numbers

Real numbers that happen to be rational numbers or integers are denoted as described in the preceding two sections. Every other real number is inexact and may only be represented in one of the following four ways:

- in (radix 10) 'decimal' form as an optional plus sign or minus sign and/or an optional integer in exact integer form, followed by a decimal point that in turn is followed by one or more decimal digits. Examples:

  -.007 **3.142** **-0.125** **+1000000.0000001**

- in (radix 10) 'unknown digit' form, which looks just like decimal form except that, from some point beyond the first digit, every digit is replaced by a **#**-sign. Examples:

  3.14# **17##.###** **-.50##**

- in (radix 10) 'exponential' form as a real number in either exact integer, unknown digit, or decimal form, followed by an **exponent marker**—one of **e**, **s**, **f**, **d**, and **l**, in either upper- or lower-case—followed by an integer in exact integer form. Examples:

5e-20 **-17.32L6** **5.3##S-08** **4.9056f+2178**

In this notation, if **m** is one of the exponent markers, **AmB** means **A** times 10 to the power **B**. Thus, the first of the above examples denotes the number that may also be written as zero point nineteen zeros 5, and the last example may also be written as 49056 followed by 2174 zeros.

Note that, although *EdScheme* accepts real numbers in exponential form according to the foregoing description, after replacing any #-signs in **A** by zeros it *prints* such numbers **AmB** as follows:

- if the absolute value of **A** times 10 to the power **B** is between **0.0001** (included) and 10 to the power 16 (not included), the number printed is in decimal form, rounded if necessary to the nearest inexact integer. For example, if you enter

  52.3456789012345678901234567e14

  EdScheme prints **5234567890123457.0**.
- if the absolute value of **A** times 10 to the power **B** falls outside this range, the number printed is in exponential form **CeD** where the absolute value of **C** lies between 1 (included) and 10 (not included) and **C** is rounded if necessary to the nearest 10 to the power −15, and where **D** is provided with a leading zero if it is one of **-5**, **-6**, **-7**, **-8**, or **-9**. For example, if you enter **0.9s-5** , *EdScheme* prints **9e-06**.

As far as *EdScheme* is concerned, different exponent markers do *not* indicate different degrees of precision; all numbers that *EdScheme* prints in exponential form are at least as precise as is required by the IEEE 64-bit floating point standard.

Note that **d**, **e**, and **f** are both hexadecimal digits and decimal notation exponent markers. No confusion can arise, however, since it is impossible for both uses to occur in one and the same numeric constant. (See the entry for the procedure **string->number** on page 216 in Chapter 11: *Syntax and Semantics* where this apparent ambiguity is specifically addressed.)

- in 'rational' form—involving a forward slash (**/**), as described in the previous section—preceded by a **#i** exactness prefix, or where either or both of the denominator and the numerator have 'tails' of #-signs. Examples:

54#/29 3##/2# #i3/5 #i#x2af/bc5

Note that *EdScheme* always *prints* real numbers written in this inexact rational form in radix 10. So, for example, if you type the second of the above numbers and hit the **RETURN** key, *EdScheme* prints `0.2280119482243611`.

5.8 Complex Numbers

Complex numbers that are also real numbers are denoted in one of the ways described in the preceding three sections. Every other complex number is represented

- in one of the following ways, in which **r** and **s** denote (possibly equal) real numbers:

  +i -i +ri -ri r+i r-i r+si r-si r@s

  preceded possibly by an exactness prefix (**#i** or **#e**); or,

- when *both* **r** and **s** denote exact numbers, in one of the above eight forms, preceded (in addition to, or instead of the optional exactness prefix) by a radix prefix.

The leading sign in the case of **+i**, **-i**, **+ri**, and **-ri** is essential; it must not be omitted. The letter **i** may be either upper- or lower-case. Neither spaces nor parentheses may be included in the name of a complex number. Examples:

3-4i #o+15/7i -2.3-5e-10i 23/17-0.0045i 12.5@1.596

The **r@s** form above is known as 'polar form'; the complex number named in this way is **r** * [(cos **s**) + **i** * (sin **s**)], where **s** is interpreted as being an angle measured in radians, irrespective of which Angle Mode you have chosen in the Language dialog accessed through the Preferences sub-menu in the File menu.

EdScheme prints complex numbers by

- supplying any 'pure imaginary' numbers with an exact zero real part (thus, for example, **#x+2b/ci** prints as `0+43/12i`);

- printing the real and imaginary parts of complex numbers entered in 'rectangular' (that is, non-polar) form according to the printing rules outlined in the preceding sections;
- and printing complex numbers entered in polar form in rectangular form (thus, for example, **12.5@1.596** prints as

  ```
  -0.3150125619420707+12.496030053013581).
  ```

  Note that, although you can use the **number->string** procedure—see the relevant entry on page 188 in Chapter 11: *Syntax and Semantics*—to print representations of *exact* complex numbers in radixes other than 10, *EdScheme* always prints inexact complex numbers in radix 10 notation. (See the next section to find out how to tell whether or not a complex number is exact.)

.9 Exactness

In the eyes of *EdScheme* each number is either exact or inexact, a state that can be identified using either one of the predicates **exact?** and **inexact?**. Non-integer rational numbers are exact; integers denoted without the use of a decimal point or a **#**-sign—that is, in exact integer form—are exact (hence the name!) unless preceded by a **#i** exactness prefix; all other integers are inexact. Real numbers named either in decimal, unknown digit, or exponential form are inexact unless preceded by a **#e** exactness prefix. Complex numbers whose real and imaginary parts are *both* exact are themselves exact unless preceded by a **#i** exactness prefix; all other complex numbers—even those in polar form with exact integers either side of the **@**-sign—are inexact unless preceded by a **#e** exactness prefix or unless the number following the **@**-sign is an exact zero.

As a general rule, simple arithmetic performed upon exact numbers produces an exact result, but if at least one of the numbers involved is inexact then the result will be inexact. Some procedures, however, always produce inexact results—for example, **sqrt** and **acos** fall into this category—a state of affairs that is noted in their entries in Chapter 11: *Syntax and Semantics*. All numeric comparison predicates (such as **<=**) and some numeric predicates (such as **zero?**) produce unreliable results if any of their arguments are inexact. The worst offenders are **=** and **zero?**, since the small inaccuracies inherent in the state of 'inexactness' can easily affect the result.

To switch between equivalent exact and inexact numbers you can use the procedures **inexact->exact** and **exact->inexact**, or you can add an appropriate exactness prefix. Thus, **(inexact->exact 0.2)** produces the same result as **#e0.2**. The result of such an 'exactness change' may not, however, always be what you might expect. For example,

```
(inexact->exact 0.2)
        ↦  19999999999999999/100000000000000000
```

rather than the expected `1/5`. This happens because ***EdScheme*** is working with the internal (binary) representation of the number in question, rather than its external (decimal) representation. To achieve the 'expected' result for numbers expressed in decimal form, use the procedure **exact-rationalize** with a 'small' second argument, such as **1e-16**. For example,

```
(exact-rationalize 0.2 1e-16)      ↦  1/5
(exact-rationalize 0.0625 1e-16)   ↦  1/16
```

Simple derived procedures can be written to take care of inexact real numbers whose names are in other forms, and of other types of inexact complex numbers.

10 Mathematical Information

EdScheme provides procedures that give access to a large selection of mathematical functions, most of which accept as arguments any of the numbers described in this chapter. Their behavior when presented with certain kinds of *real* number arguments, however, agrees with what you probably know of these functions from your high school mathematical studies. The following table specifies which arguments produce such 'usual' behavior:

Procedure	*Mathematical Function*	*Argument(s)*
exp	natural exponential function	any real number
expt	general exponential function	any two real numbers such that, if the first is zero, then the second is non-negative

Procedure	*Mathematical Function*	*Argument(s)*
log	natural (or general) logarithm function	one (or two) positive real numbers (the second, if provided, being different from 1)
power	general exponential function	any two real numbers such that, if the first is zero, then the second is non-negative
sqrt	square root function	any non-negative real number
cos	cosine function	any real number
sin	sine function	any real number
tan	tangent function	any real number not an odd multiple of $\frac{\pi}{2}$ (in Radian Mode) or not an odd multiple of 90 (in Degree Mode)
acos	inverse cosine function	any number in the real interval from −1 to 1 (both included)
asin	inverse sine function	any number in the real interval from −1 to 1 (both included)
atan	inverse tangent function	one or two real numbers (not both zero, if there are two)

For all other arguments, these *EdScheme* procedures operate as complex-argument/complex-result procedures and, irrespective of which Angle Mode you have chosen in the **Language** dialog accessed through the **Preferences** sub-menu in the **File** menu, they treat all angles as being measured in radians. For such arguments, $z = a + b\mathrm{i}$ (where a and b are real), they calculate their values as follows:

$$
\begin{aligned}
(\mathbf{exp}\ z) &= (\mathbf{exp}\ a) * [(\mathbf{cos}\ b) + \mathrm{i} * (\mathbf{sin}\ b)] \\
(\mathbf{expt}\ z\ w) &= (\mathbf{exp}\ [w * (\mathbf{log}\ z)]) \qquad (z \neq 0) \\
(\mathbf{power}\ z\ w) &= (\mathbf{exp}\ [w * (\mathbf{log}\ z)]) \qquad (z \neq 0) \\
(\mathbf{cos}\ z) &= [(\mathbf{cos}\ a) * \cosh(b)] - [\mathrm{i} * (\mathbf{sin}\ a) * \sinh(b)] \\
(\mathbf{sin}\ z) &= [(\mathbf{sin}\ a) * \cosh(b)] + [\mathrm{i} * (\mathbf{cos}\ a) * \sinh(b)] \\
(\mathbf{tan}\ z) &= \frac{(\mathbf{tan}\ a) + \mathrm{i} * \tanh(b)}{1 - \mathrm{i} * (\mathbf{tan}\ a) * \tanh(b)}
\end{aligned}
$$

Note that sinh, cosh, and tanh are the hyperbolic sine, cosine, and tangent functions, respectively. They are *not* primitive procedures in ***EdScheme***, but they may easily be defined as derived procedures, as follows:

```
(define sinh
  (lambda (x)
    (/ (- (exp x) (exp (- x))) 2)))

(define cosh
  (lambda (x)
    (/ (+ (exp x) (exp (- x))) 2)))

(define tanh
  (lambda (x)
    (/ (sinh x) (cosh x))))
```

Returning to the evaluation of complex-argument procedures,

$$\begin{aligned} (\textbf{log}\ z) &= (\textbf{log}\ (\textbf{abs}\ z)) + \mathrm{i} * (\textbf{angle}\ z) \\ (\textbf{log}\ z\ w) &= (\textbf{log}\ z) / (\textbf{log}\ w) \end{aligned}$$

(Here, neither z nor w may be 0, and w may not be 1.) Note that all occurrences of the logarithm function on the right side of these equalities are *natural* logarithms to base e (usually denoted by 'ln' in mathematical circles), not logarithms to base 10. For the operation of the procedures **abs** and **angle**, see the relevant entries in Chapter 11: *Syntax and Semantics*.

$$\begin{aligned} (\textbf{sqrt}\ z) &= (\textbf{expt}\ z\ 1/2) \\ (\textbf{asin}\ z) &= -\mathrm{i} * (\textbf{log}\ [\mathrm{i} * z + (\textbf{sqrt}\ (1 - z * z))]) \\ (\textbf{acos}\ z) &= \textbf{pi}/2 - (\textbf{asin}\ z) \\ (\textbf{atan}\ z) &= -0.5\mathrm{i} * [(\textbf{log}\ (1 + \mathrm{i} * z)) - (\textbf{log}\ (1 - \mathrm{i} * z))] \end{aligned}$$

See the entry for the variable **pi** on page 197 in Chapter 11: *Syntax and Semantics*. The argument of **atan** must not be either +i or −i.

– o O o –

6 Files and Ports

EdScheme provides a comprehensive collection of procedures for accessing disk files and for communicating with peripheral devices via the Macintosh's serial ports. This chapter includes the background information you will need in order to interpret the detailed descriptions of *EdScheme*'s file-handling and port-communication procedures given in Chapter 11: *Syntax and Semantics.*

6.1 Files

Macintosh files are identified by means of the volume on which they are located and the file name. Volumes are referenced by exact integers, and they correspond (roughly speaking) to the Finder™ windows that appear on the desktop and the windows that open when the names/icons of folders are clicked on. For example, if you have installed all the *EdScheme* files from the implementation disk into a folder entitled 'EdScheme' on your hard drive, then the file 'game startup.s' is on a volume that corresponds to the window showing the contents of the disk drive from which you usually start the *EdScheme* program. It is also on the volume corresponding to the 'EdScheme' folder, as well as the volume corresponding to the folder entitled 'Game' contained within the 'EdScheme' folder.

In general terms, the current volume will by default be the volume containing the current application. Thus, if you launch *EdScheme* by double-clicking on the *EdScheme* 'ink-pot λ' icon, then the current volume will correspond to the window in which that name/icon appears.

The following *EdScheme* procedures—which are described in detail in

Chapter 11: *Syntax and Semantics*—enable you to determine a volume's reference number and to change the current volume:

EdScheme–volume **last–volume** **set–volume** **volume**

EdScheme allows you to refer to a particular file using either of two methods. You may specify a file using a **full file specification**, that is, a list of three data expressions comprising, in order, a volume reference number, a file version number (usually 0), and a string or symbol naming the file in question. You may also identify a file using a string—known as a **path name**—that specifies the sequence of folders that need to be opened in order to locate the file. The folder names are separated (without any intervening spaces) by colons. Thus, if you have installed all the ***EdScheme*** files from the implementation disk into a folder entitled 'EdScheme' on your hard drive, which you have called 'Macintosh HD', then the string

"Macintosh HD:EdScheme:Game:game startup.s"

identifies the file 'game startup.s' from which you may launch the demonstration game provided with ***EdScheme***. (Note that the disk drive name is *not* preceded by a colon.) On the other hand, if the current volume is the one on which the 'EdScheme' folder appears, then the string

":Game:game startup.s"

identifies the same file. (Path names that begin with a colon are assumed to refer to files located in folders on the current volume.)

Files not in folders, but on the current volume, may alternatively be referred to even more simply by the string or symbol that names them. Thus, if the current volume is the one corresponding to the 'Game' folder, then the startup file is identified by the string **"game startup.s"**.

.2 Ports and File-handling

EdScheme communicates with the outside world via data objects known as **ports**. Like all of Scheme's data expressions, ports can be input to procedures, output from procedures, and stored in data structures. In addition, however, ports may be written to and/or read from (depending on the type of port).

There are three types of ports supported by ***EdScheme***: input ports, output ports, and serial ports (which are specialized kinds of input/output ports). The predicates **input-port?** and **output-port?** enable you to determine what kind of port you are dealing with. In addition, the Transcript Window—see Section 3.1: *Transcript Windows* in Chapter 3: *Programming Environment Windows*—is both an input port and an output port.

The procedures **current-input-port** and **current-output-port** enable you to discover which are the current input and output ports. WARNING: The fact that serial ports and the Transcript Window are both input and output ports can confuse certain test batteries that check for compatibility with the Scheme standard. Since the Transcript Window is frequently both the current input port *and* the current output port, the Scheme expression

```
(input-port? (current-output-port))
```

for example, will usually return the boolean `#t` rather than the expected `#f`.

Files may be opened using the procedures **open-input-file**, **open-output-file**, and **open-extend-file**. However, these procedures require you to refer to the file in question by name. (The Macintosh system of naming files is described in the previous section.) To avoid this requirement, you may use the procedures **choose-input-file** and **choose-output-file** to generate the argument to these three procedures with the help of the Macintosh File Selector dialog. Use **choose-input-file** in conjunction with **open-input-file**, **choose-output-file** with **open-output-file**, and either one with **open-extend-file**. You should refer, however, to the entry for **open-extend-file** on page 189 in Chapter 11 for further information concerning the relative ease of using the two **choose-...-file** procedures, and advice on how to do so. By bringing up a Macintosh File Selector dialog box these two procedures allow you to select the file you wish to open by pointing and clicking. A file that has been opened using the procedure **open-extend-file** is classified as an output port. Files may be closed using the procedures **close-input-port**, **close-output-port**, and **close-port**.

Once a file is open, information concerning its various characteristics may be ascertained and modified using these procedures:

eof?	**eof-object?**	**file-length**	**file-set-length**
file-margin	**file-set-margin**	**file-position**	**file-set-position**

To help you read data from an open input port, ***EdScheme*** provides the procedures **char-ready?**, **peek-char**, **read**, **read-char**, **read-line**, and **string-read**,

and to write data to an open output port you may use the procedures **display**, **freshline**, **newline**, **string–write**, **write**, and **write–char**.

Serial ports may be opened using the procedure **open–serial–port**, and, once open, may be configured using the procedure **configure–serial–port**.

Finally, input to or output from *EdScheme* procedures may be gathered from or directed to disk files using the procedures

call–with–input–file	**call–with–output–file**
with–input–from–file	**with–output–to–file**

All the procedures mentioned in this section are described in detail in Section 11.2: *The Syntax of EdScheme* in Chapter 11: *Syntax and Semantics*, where all of *EdScheme*'s language elements are considered in alphabetical order. The entries in that section that deal with the read/write procedures mentioned above provide specific information concerning how they should be used for communications through serial ports when such usage differs from what is appropriate for other types of ports.

– o O o –

7 Graphics Windows and Bitmaps

EdScheme provides a versatile turtle graphics interface that enables you to engage in all the usual turtle graphics activities, as described in such excellent books as:

[1] Harold Abelson and Andrea A. DiSessa, *Turtle Geometry*, MIT Press, Cambridge, MA, 1983.

[2] Brian Harvey, *Computer Science Logo Style*, Vol. 1, MIT Press, Cambridge, MA, 1985.

[3] Peter Goodyear, *Logo, A Guide To Learning Through Programming*, Ellis Horwood Ltd., Chichester, West Sussex, UK, 1984.

In addition, ***EdScheme*** includes a number of graphics procedures that give you the power to complement and extend the standard turtle graphics capabilities. Among these extended graphics facilities are procedures that allow you to create, save, and manipulate bitmaps. This chapter provides you with some relevant background information.

7.1 Graphics Windows

A graphics window is a window onto a turtle plane, which is centered on an origin and coordinatized in turtle steps by means of a standard right-handed Cartesian coordinate system.

The default dimensions of a turtle plane are 576 turtle steps wide and 720 turtle steps high. Turtle planes of other sizes may be created by supplying a suitable optional input to the procedure **make-graphics-window** (see the relevant entry on page 178 in Chapter 11: *Syntax and Semantics*). No turtle plane may have either dimension less than 20 turtle steps. If you attempt to create a smaller plane, ***EdScheme*** will replace the offending dimension(s) by 20. The smallest graphics window that ***EdScheme*** will open measures 50 by 50, even if the turtle plane beneath it is smaller.

Note that turtle headings, as input to the procedure **turtle-set-heading** and returned by the procedure **turtle-heading**, are measured clockwise from due North in radians or degrees, according to the Angle Mode you have selected in the Language dialog accessed through the Preferences sub-menu in the File menu. (See the relevant section of Chapter 9: *The EdScheme Menus*, starting on page 75.) This clockwise measurement of angles from due North contrasts with the universal mathematical convention—used in every other angle-handling part of the Scheme language—according to which angles are measured counterclockwise from due East. ***EdScheme*** makes this exception so as to be compatible with other standard turtle graphics interfaces.

Graphics windows require quite a lot of your computer's memory, especially on color systems. When supplied to you, ***EdScheme*** by default allocates no memory to graphics. So, if you plan to use graphics windows, you will need to reserve sufficient memory by changing the **Graphics** buffer setting in the Memory dialog accessed through the Preferences sub-menu in the File menu. For further details, see the relevant section of Chapter 9: *The EdScheme Menus* on page 80.

.2 A Graphics Programming Example

A catalogue of the turtle graphics procedures is included in the listing beginning on page 248 in Chapter 12: *Language Elements*. In this section we illustrate the use of some of these procedures by setting up the rudiments of an object-oriented multiple turtle environment. All the Scheme procedures used in this example are described in detail in Section 11.2: *The Syntax of EdScheme* in Chapter 11: *Syntax and Semantics*, where all of ***EdScheme***'s language elements are considered in alphabetical order. The basic 'constructor' procedure for creating turtle-objects and its accompanying syntax-switching procedures are provided in Figure 7.1, on the next page.

```
(define make-turtle
  (lambda (pos heading)
    (let ([pos-at-creation pos]
          [hdg-at-creation heading])
      (lambda (msg)
        (let ([oldpos (turtle-position)]
              [oldhdg (turtle-heading)])
          (let ([restore (lambda ()
                           (begin
                             (pen-up) (turtle-set-position oldpos)
                             (turtle-set-heading oldhdg) (pen-down)
                             'done))])
            (cond
              [(eq? msg 'reset)
               (begin
                 (set! pos pos-at-creation)
                 (set! heading hdg-at-creation)
                 (restore))]
              [(eq? msg 'turn)
               (lambda (angle)
                 (begin
                   (set! heading (+ heading angle))
                   (restore)))]
              [(eq? msg 'ahead)
               (lambda (steps)
                 (begin
                   (pen-up) (turtle-set-position pos)
                   (turtle-set-heading heading) (pen-down)
                   (forward steps) (set! pos (turtle-position))
                   (restore)))])))))))

(define turtle-reset
  (lambda (turtle)
    (turtle 'reset)))

(define turtle-right
  (lambda (turtle angle)
    ((turtle 'turn) angle)))

(define turtle-forward
  (lambda (turtle steps)
    ((turtle 'ahead) steps)))
```

Figure 7.1: Turtle-object Procedures

Having reserved enough buffer space for a graphics window, and restarted *EdScheme* if necessary, let us open a graphics window onto a 200 by 200 turtle plane:

```
(define G (make-graphics-window '(200 200)))
```

Next, we ensure that we are in Degree Mode by choosing the Language item from the Preferences sub-menu in the File menu, clicking on the Degrees radio button (if it is not already activated), and exiting from the dialog by clicking on the OK button. (Any Angle Mode change goes into immediate effect.) Then we use the make-turtle procedure to create six turtles, initially all located at the origin of the turtle plane, but facing in directions spaced at 60-degree intervals:

```
(define t1 (make-turtle '(0 0) 0))     (define t2 (make-turtle '(0 0) 60))
(define t3 (make-turtle '(0 0) 120))   (define t4 (make-turtle '(0 0) 180))
(define t5 (make-turtle '(0 0) 240))   (define t6 (make-turtle '(0 0) 300))
```

To manage the activity of multiple turtles, in Figure 7.2 on the next page we define a procedure—using a special kind of lambda-expression (described in the relevant entry on page 169 in Chapter 11: *Syntax and Semantics*)—that accepts a variable number of arguments, the first of which is a procedure of two inputs (a number and a turtle), the second of which is the number of times the two-input procedure is to be called for each turtle, and the remainder are the turtles on which the two-input procedure is called. Finally, we define two complementary drawing procedures:

```
(define rspiral
  (lambda (n turtle)
    (begin
      (turtle-forward
        turtle
        (* 10 (+ 1 n)))
      (turtle-right turtle 60)
      'done)))
```

```
(define lspiral
  (lambda (n turtle)
    (begin
      (turtle-forward
        turtle
        (* 10 (+ 1 n)))
      (turtle-right turtle -60)
      'done)))
```

We are now ready to give our six turtles some work to do. The three Scheme expressions listed beneath the figure on the next page cause a hexagonal 'stained glass window' design to be drawn in graphics window *G*. If you watch as the drawing takes place, you may be able to discern the separate activities of the six turtles.

```
(define manager
  (lambda args
    (let ([f (first args)] [n (list-ref args 1)]
          [t-list (rest (rest args))] [p-state (pen-state)]
          [show-state (turtle-shown?)])
      (letrec ([time-slice
                (lambda (ctr)
                  (if (= ctr n)
                      'done
                      (begin
                        (map (lambda (turtle) (f ctr turtle))
                             t-list)
                        (time-slice (+ 1 ctr)))))])
        (begin
          (turtle-hide) (time-slice 0)
          (if show-state (turtle-show))
          (pen-set-state p-state)
          'done)))))
```

Figure 7.2: Multiple Turtle Manager

To draw a stained glass window design, evaluate these Scheme expressions:

```
(manager rspiral 6 t1 t2 t3 t4 t5 t6)

(manager (lambda (n turtle) (turtle-reset turtle))
         1 t1 t2 t3 t4 t5 t6)

(manager lspiral 6 t1 t2 t3 t4 t5 t6)
```

(The second expression returns the six turtles to their initial state, ready for the second phase of the drawing.) And before leaving this example, we tidy up after ourselves by reactivating the Language dialog from the Preferences sub-menu in the File menu and resetting Radian Mode—if that was your Angle Mode prior to trying out this example.

Notice that the **manager** procedure hides the turtle before starting to do any drawing. Turtle graphics procedures all operate more efficiently when the turtle is not shown for the simple reason that, under such circumstances, *EdScheme* does not have to take time to redraw the turtle constantly as it moves.

We leave it to you to add to the capabilities of these turtle-objects. As defined here, they operate exclusively in the 'pen-down' mode. It is a relatively straightforward task to give them the ability to understand additional messages so that they can change their pen state or their pen color. Or, if you are more ambitious, you could improve the **manager** procedure by broadening the range of procedures it can cause to be applied to the team of turtles it is managing.

.3 Bitmaps

EdScheme allows you to create off-screen bitmaps for use in conjunction with graphics windows. Bitmaps can be established using the procedure **make-bitmap**, which requires you to specify the dimensions of the bitmap and gives you the opportunity to say how colorful it will be. Once created, an image may be assigned to a bitmap by 'fetching' a rectangular image from an open graphics window using the procedure **bitmap-fetch**. In this process, either the image can be taken 'as is' from a rectangle whose lower left corner is specified and whose dimensions match those of the bitmap, or it can be taken from a specified rectangle and scaled to fit the dimensions of the bitmap. Alternatively, you can create a new bitmap and at the same time assign to it a previously-saved bitmap image using the procedure **bitmap-set-spec**.

Bitmap images may be 'stamped' in a graphics window using the procedure **bitmap-stamp**. By default, the entire bitmap image overprints the rectangular area of the graphics window onto which it is stamped; this is the so-called 'copy mode'. But the stamping mode may be altered using the procedure **bitmap-mode**. Furthermore, **bitmap-stamp** gives you the ability to 'mask' one bitmap with another so that only those portions of the stamping bitmap that 'show through' the masking bitmap are stamped in the graphics window. By masking a bitmap with itself you can make sure that the only parts of a bitmap image that overprint the rectangular area onto which it is stamped are those where the bitmap image is non-white. In certain circumstances—which are explained in the entry for the procedure **bitmap-stamp** on page 124 in Chapter 11: *Syntax and Semantics*—the masking bitmap is scaled to match the dimensions of the stamping bitmap if they happen to be of different sizes.

The details of a bitmap image can be captured in a form suitable for saving to disk using the procedure **bitmap-spec**. As indicated earlier in this section,

bitmap images saved in this way can be reinstated using the procedure **bitmap-set-spec**. Finally, bitmaps can be destroyed using the procedure **bitmap-close**.

All the procedures mentioned in the foregoing survey are described in detail in Chapter 11: *Syntax and Semantics*, where the relevant entries include some simple examples of ***EdScheme***'s versatile bitmap-handling capabilities.

– o O o –

8 Text Windows and User Menus

8.1 Text Windows

Text windows allow you to provide the user of an application written in *EdScheme* with information and to accept input provided by the user while the application is running. They have an automatic word-wrap feature, and their contents may be manipulated under program control as well as by standard mouse-and-menu means. They are created using the procedure **make-text-window**, and—depending on whether or not you supply an optional input to this procedure—the line width at which words are wrapped to the next line can vary with the width of the text window as you resize it and move it about the desktop, or that line width can be rigidly set to a fixed amount. The smallest text window that *EdScheme* will open measures 50 by 50. If you attempt to specify a smaller size using the procedure **window-set-position**, then *EdScheme* will replace the offending dimension(s) by 50.

Text written to text windows can appear flush left, centered, or flush right, the default being flush left. To find out what type of alignment is currently in force, you can use the procedure **text-alignment**, and to alter the alignment setting you can use the procedure **text-set-alignment**.

The preferred method for writing data to text windows and reading data from them is in the form of strings. For writing purposes, *EdScheme* provides the procedures **display**, **string-write**, **text-display**, **text-set-contents**, and **write**, each of which operates slightly differently. (The specific details are provided

in the relevant entries in Chapter 11: *Syntax and Semantics*.) For reading data from text windows, ***EdScheme*** provides the procedures **string–read**, **text–contents**, and **text–readline**, which are also described in detail in Chapter 11.

It is important to realize that strings take up a lot of main memory. So procedures that include many explicit strings will be very greedy of the workspace you have available. This becomes particularly important if you are programming an application with a user-interface that involves your program in displaying a lot of information in text windows. In such circumstances, it is preferable to organize the text to be displayed in one or more 'text resource files', the contents of which may be read using **string–read**, broken up into manageable pieces in a variety of ways, and then written into text windows as required using **string–write** or **text–set–contents**. The demonstration game program in the 'Game' folder on the ***EdScheme*** implementation disk illustrates this method of handling the transmission of information in a text-window-based user-interface. See in particular the function **make–resource–text–object** in the file 'game startup.s'.

Text windows share a memory pool with document windows. There is no limit on the number of text windows that may be open at once, only on the total amount of memory they and any open document windows use. By default, 80 kilobytes of memory are set aside for this purpose. You may change this by altering the **Text** buffer setting in the Memory dialog, which is accessed through the Preferences sub-menu in the File menu. Before any such change comes into effect, however, you will have to restart ***EdScheme***.

Text may be selected, cut from, or pasted into a text window under program control using the primitives **text–set–selection**, **text–cut**, **text–clear**, **text–copy**, and **text–paste**, or directly by using standard mouse-and-Edit-menu methods. This ability of text windows to exchange data with the clipboard makes possible other uses for text windows besides their involvement (mentioned earlier in this section) in application user-interfaces. For example, the procedure **leave–strings**, whose definition is given in Figure 8.1 on the next page, may be used to extract just the strings out of a data file. Such a procedure might be useful, for example, for creating the kind of 'text resource file' referred to earlier in this section.

For demonstration purposes, let us construct a suitable data file. Open a new document window by choosing the New item from the File menu, and type into it the following text (which continues at the top of page 64).

```
This is a "demonstration" data "file" created for the
"purpose" of illustrating "how the" "leave-strings"
```

```
(define leave-strings
  (lambda (xwin)
    (letrec
      ([work
         (lambda (start current cut-flag)
           (let ([n (text-length xwin)])
             (if (= n current)
                 (if cut-flag
                     (begin
                       (text-set-selection (list (- start 1) current) xwin)
                       (text-cut xwin) (newline xwin)
                       'done)
                     (begin
                       (text-set-selection (list start current) xwin)
                       (text-cut xwin)
                       'done))
                 (let ([next
                        (begin
                          (text-set-selection
                            (list current (+ 1 current)) xwin)
                          (text-copy xwin) (clipboard-text))])
                   (if (string=? next "\"")
                       (if cut-flag
                           (begin
                             (text-set-selection
                               (list start current) xwin)
                             (text-cut xwin)
                             (work start (+ 1 start) (not cut-flag)))
                           (let ([new-start (+ 1 current)])
                             (begin
                               (text-set-selection
                                 (list new-start new-start) xwin)
                               (newline xwin)
                               (work (+ 1 new-start) (+ 1 new-start)
                                     (not cut-flag)))))
                       (work start (+ 1 current) cut-flag)))))))])
      (work 0 0 #t))))
```

Figure 8.1: A Text Window Procedure

```
procedure works.  Once the "procedure" finishes its
"job", there should be a "list" of nine strings
remaining in the "window".
```

Then choose Save As ... from the File menu, enter the name `test.dat`, and click on the OK button. Dismiss the document window by clicking on its Close Box (in its upper left corner), and open up a new one using the New menu item. Type the definition of the procedure **leave-strings** into this document window, and evaluate it by placing the caret immediately to the right of the final parenthesis and pressing the Keypad-ENTER key. (By typing this definition into a document window, you avoid having to retype the whole thing if you make a typing error.) Move into the Transcript Window simply by clicking on it once, and open a text window by evaluating the following Scheme expression:

```
(define T (make-text-window))
```

Use the mouse to resize and drag the text window so that it does not overlap the Transcript Window, and then evaluate the following Scheme expression:

```
(begin
  (text-set-contents
    (string-read (open-input-file "test.dat")))
  (close-port)
  (leave-strings T))
```

The contents of the file 'test.dat' will appear in the text window, and before your very eyes all the non-strings will be 'eaten away', leaving only nine strings, each on a separate line.

Roughly speaking, the **leave-strings** procedure works as follows: It copies the characters, one at a time, onto the clipboard, checking each one to see if it is a 'double quote' mark. It recognizes the first such double quote it finds as the start of the first string, so it cuts everything from the start of the window contents up to (but not including) that double quote. It then switches the ***cut-flag*** to `#f` and looks for the next double quote, which it recognizes as the end of the first string. Next, it inserts a newline character into the text window, switches the ***cut-flag*** back to `#t` and begins a new search, starting with the character following the second double quote mark. It continues like this, finding double quotes, and alternately cutting and leaving, until it reaches the end of the text window's contents. (The portion of the procedure comprising the first Scheme expression following the line

(if (= *n current*)

deals with the tail end of the window's contents.) If you would like to save the list of strings that results from this application of the **leave-strings** procedure, you may do so by evaluating the following Scheme expression:

```
(begin
  (string-write
    (text-contents T) (open-output-file "string.dat"))
  (close-port)
  (window-close T)
  'done)
```

which also closes the text window ***T***. After this, the file 'string.dat' will contain the list of strings just obtained.

We leave it to you to modify and improve the **leave-strings** procedure so that it reads a Scheme program from a disk file and creates from it two disk files, one containing all the strings in the program in order of appearance, and the other being the original Scheme program in which each string has been replaced by a Scheme expression of the form

(resource-string *n*)

where ***n*** is a number indicating that the string being replaced is the ***n*th** in order from the beginning of the program.

All the procedures used in the foregoing example are described in detail in Section 11.2: *The Syntax of EdScheme* in Chapter 11: *Syntax and Semantics*, where all of *EdScheme*'s language elements are considered in alphabetical order.

.2 Menus

EdScheme allows you to add your own menus to the main menu bar. The items in these menus may be specified, modified, selected, deselected, checked, and unchecked under program control. ***EdScheme*** also enables programs to accept user input in the form of a menu selection from a user-defined menu.

Menus are established using the procedure **make-menu**, and their items may be inspected and modified using the procedures **menu-item** and **menu-set-item**, respectively. When working with user-defined menus, the procedure

menu-number-of-items often proves useful since it returns the number of items such a menu contains. (Note however that a separator line also counts as a menu item.) To close a user-defined menu—that is, to remove it from the menu bar—use the function **menu-close**.

Menu selections from menus (whether user-defined or not) may be read using the procedure **event**, which—together with all the procedures mentioned so far in this section—is described in detail in Chapter 11: *Syntax and Semantics*. In particular, the Scheme expression **(first (event))** will evaluate to the symbol **menu** if the event in question is indeed a menu selection, in which case **(last (event))** will evaluate to a string containing the text of the selected menu item. Of course, the two Scheme expressions just mentioned should not be evaluated one after the other, or else the ***EdScheme*** interpreter will expect *two* separate events. Instead, the event should be captured, and the resulting list inspected, using a Scheme expression such as the following:

```
(let ([e (event)])
  (if (equal? (first e) 'menu)
      (last e)
      "not a menu selection"))
```

There are several examples of the manipulation of user-defined menus in the program files of the demonstration game included on the ***EdScheme*** implementation disk. For example, the file 'game utilities.s' in the 'Game' folder includes the definition of a procedure **menu-items** that returns a list of the items in a user-defined menu.

Part III

Language Reference

9

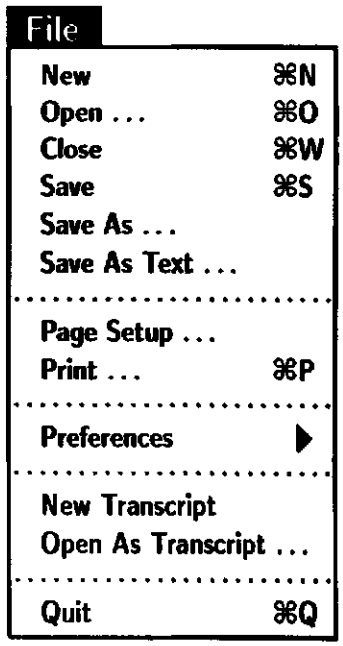
9

9 The *EdScheme* Menus

In this chapter we describe the purpose of each item in the five menus on the ***EdScheme*** Menu Bar. The sections and sub-sections correspond to the menus and the items and any dependent sub-menus they contain.

9.1 The File Menu

The File menu contains twelve menu items, as follows:



New [Hot Key: COMMAND-N]

Opens a new document window, with the title 'Untitled'. To replace this title with one that is more appropriate to the contents you wish to enter into the document, choose the **Save As ...** item of this menu. For more information about document windows, see Section 3.3: *Document Windows* in Chapter 3: *Programming Environment Windows.*

Open ... [Hot Key: COMMAND-O]

Brings up a Macintosh File Selector dialog box that enables you to select a file to be opened.

Close [Hot Key: COMMAND-W]

Closes the currently selected window. If the window in question is a document window whose contents have been modified since the window was opened, a dialog box will appear giving you the opportunity to save the document before closing the window.

Save [Hot Key: COMMAND-S]

Saves the contents of the currently selected document, text, or transcript window, using the window's title as the file name. If you wish to use a different name instead, then choose the Save As ... item in this menu.

If the window is a text window, the 'Save File As:' Macintosh File Selector will appear, just as if you had chosen the Save As ... item. You may then either accept the window's title as the file name, or you may enter an alternative. Note that, if you save the contents of a text window using this item or the Save As ... item, those contents cannot subsequently be read back into a text window using any menu item. They can, however, be read into a document window using the Open ... item in this menu. (To reinstate text window contents that you have saved in a file called 'text'—into a text window, you can open a text window *T* using the procedure **make-text-window**, evaluate the Scheme expression

```
(text-set-contents (string-read (open-input-file "text")) T)
```

and finally close the input file—and every other open port—by evaluating the Scheme expression **(close-port)**.)

If the window is a transcript window, it will be saved in a form that can only be reinstated using the Open As Transcript ... item in this menu. To save the contents of a transcript window in a form that can subsequently be read into a document window, use the Save As Text ... item in this menu.

Save As ...

Brings up a Macintosh File Selector dialog box that enables you to choose a file name under which to save the contents of the currently selected document, text, or transcript window. The window's title appears in the box entitled 'Save File As:'. You may accept this as the file's name by hitting the ENTER key or by clicking on the **Save** button. Alternatively, you may enter some other name simply by beginning to type, and then hitting the ENTER key or clicking on the **Save** button once you have finished typing the new name.

The comments made concerning the saving of text and transcript windows using the Save item in this menu and their subsequent reinstatement or readability apply in the case of this item also.

Save As Text ...

Brings up a 'Save File As:' Macintosh File Selector dialog box allowing you to choose a name under which to save the contents of the current transcript window as a text file that can subsequently be read into a document window.

By saving the contents in this way you will not be able later on to reinstate those contents as a transcript window (using the Open As Transcript ... item in this menu). The reason for this is that saving the contents of a transcript window as text removes all the color and font information that are an essential part of a transcript window's presentation. Furthermore, in the file that is created as a result of saving the contents of a transcript window as text, the *EdScheme* prompt '⟹' is replaced by the symbol '>'.

In contrast, saving the contents of a transcript window by using either the Save or the Save As ... items in this menu causes the additional color and font information to be retained. In these cases, the window can subsequently *only* be reinstated using the Open As Transcript ... item in this menu. The contents of transcript windows saved using these menu items cannot be read into a document window.

Page Setup ...

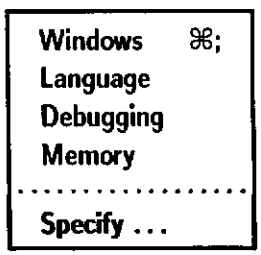
Brings up a dialog box that enables you to modify the settings of the printer driver you have activated using the Chooser item in the Apple menu.

Print ... [Hot Key: COMMAND-P]

Brings up a Macintosh dialog box that enables you to specify how you want some or all of the contents of the currently selected window to be printed. (Any of the various kinds of windows provided by *EdScheme*—including graphics windows—may be printed by choosing this menu item, provided you have previously activated a suitable printer driver using the Chooser item in the Apple menu.)

Preferences

Activates a five-item sub-menu (shown on the right) giving you control over many aspects of *EdScheme*'s operation. The sub-menu items are as follows:



Windows [Hot Key: COMMAND-;]

Brings up a Macintosh dialog box that gives you control over the styles, colors, fonts, and sizes of the characters that appear in document and transcript windows. (The same features of the characters that appear in graphics and text windows are controlled using the procedure **font-style**—see the relevant entry on page 162 in Chapter 11: *Syntax and Semantics*.)

The dialog box also allows you to control whether or not *EdScheme*'s automatic parenthesis-matching and 'pretty-printing' formatting of Scheme expressions is activated in document and/or transcript windows.

The upper portion of the dialog box concerns all of *EdScheme*'s Transcript Windows (including Debug Transcripts and the Trace Window). Automatic parenthesis-matching and expression formatting is activated or deactivated using the **Format & Paren-Matching** checkbox.

The styles, colors, and fonts used for system messages, Scheme expressions, and data expressions may be selected using the pop-up menus in the 3-by-3 array with row headings 'System', 'Scheme', and 'Data', and column headings 'Style', 'Color', and 'Font'. The font selections you make will influence how well *EdScheme* performs its expression formatting (if you have activated that feature). For the best results, you should choose a non-proportional-spacing font for Scheme expressions. (Monaco, which *EdScheme* uses in document windows, is such a font.)

The number entered in the Text Size box determines the point size of the characters used in Transcript Windows.

The default size for text in Transcript Windows is 12 point.

The number entered in the Right Margin box determines how many characters may be displayed in a single line of a transcript window before *EdScheme* begins to look for a 'natural' place to break the line. Such a break will occur after a space, unless no breaking point is found before the transcript line length (determined by the number entered in the Line Length box) is reached. In such a case a forced line break will be inserted, even if this means splitting a word or an atom.

The default transcript right margin is 65 characters, and the default transcript line length is 75 characters.

You will probably want to increase the transcript right margin and transcript line length settings if you are using smaller than the default 12 point

characters and/or you have a large screen monitor. Conversely, you will want to decrease these dimensions if you are using larger than 12 point characters.

A judicious setting for the transcript line length can greatly simplify the task of checking the contents of a data file that would appear as a truncated single line if read into a document window. You can select a transcript line length that is appropriate for the type of data in the data file, read the contents into the current transcript window using code such as the following:

```
(let* ([F (open-input-file "your-data-file-name")]
       [S (string-read F)])
  (close-port F)
  S)
```

The contents of your data file will appear in the Transcript Window, broken into easily-manageable lines. You then have only to copy the result into a document window (using the Edit menu, described below) ready for closer inspection.

Note that the line-breaking described above occurs *only* when data expressions are evaluated or displayed in a transcript window. It does *not* happen when you are typing expressions into the current Transcript Window. As long as you do not type a carriage return, the line you are typing will continue without breaking.

The lower portion of the dialog box concerns *EdScheme*'s document windows. As in the case of transcript windows, automatic parenthesis-matching and expression formatting is activated or deactivated using the Format & Paren-Matching checkbox.

Note that, when creating or working on a document that does *not* consist primarily of *EdScheme* procedure definitions, it is advisable to switch *off* parenthesis-matching and expression formatting in document windows. This allows the *EdScheme* editor to go about its business more speedily, without having to take time to search for matching parentheses and to calculate indentation tab stops. It also prevents *EdScheme* from annoyingly and unnecessarily indenting the second and subsequent lines of parenthetical remarks in your text file.

You should be aware, however, that switching off this feature also disables *EdScheme*'s ability to recognize the Current Expression in a document window—for further information, see Sections 2.3 and 3.4—and it makes it impossible to evaluate any Scheme expressions included in your text file simply by placing the caret after the final right hand parenthesis and hitting the

Keypad-ENTER key (see page 14 in Chapter 2: *An EdScheme Session* for more information about this evaluation technique). It is still possible, though, to evaluate such expressions by selecting them with the mouse, and choosing the Selection item in the Evaluate menu (see later in this chapter for more details).

The number entered in the Text Size box determines the point size of the characters used in document windows.

The default size for text in document windows is 9 point.

To display the default settings for all the parameters that feature in the Windows-Preferences dialog box, click on the Defaults button. Once you have chosen which windows settings you would like, click on the OK button, and your desired settings will go into immediate effect. Alternatively, you can dismiss the dialog box without changing the settings by clicking on the Cancel button.

The *EdScheme* procedures **windows-set-preferences** and **windows-preferences** give you control from within a program over the settings that feature in this dialog box. See the relevant entries on pages 241 and 242 in Chapter 11: *Syntax and Semantics*.

Language

Brings up a Macintosh dialog box that gives you control over the Language Mode in which *EdScheme* operates, the Angle Mode (either radians or degrees), and the presence or absence of case-sensitivity.

The dialog box also lets you instruct *EdScheme* to carry out a garbage collection whenever an error occurs, and it gives you control over some aspects of *EdScheme*'s communication with you.

You may choose either the Standard Scheme Mode or the Schemer's Guide Mode. In Standard Scheme Mode, *EdScheme* procedures operate according to the Scheme Standard and are consistent with the requirements of the *Revised*^{4} *Report on the Algorithmic Language Scheme*. In Schemer's Guide Mode, the procedures behave in a way that is consistent with *The Schemer's Guide* [Schemers Inc., Fort Lauderdale, FL, 1992]. The behavior of all procedures that operate differently in the two modes is explained in detail in their respective entries in Chapter 11: *Syntax and Semantics*. Note that, **if you choose Schemer's Guide Mode, then Degree Mode and Case-sensitive Mode are automatically chosen for you, and Extended Procedure Representations are automatically activated. (This automatic behavior occurs in order to ensure full compatibility with *The Schemer's Guide*. You may, however, reject**

any of these automatic choices by using the radio buttons and checkboxes to make other selections.)

By default, ***EdScheme*** is set to Standard Scheme Mode. Changes in Language Mode setting go into immediate effect, with one exception: If your change of Language Mode involves changing from case-insensitivity to case-sensitivity or *vice versa*, then that aspect of the change will not go into effect until you restart ***EdScheme***.

In addition, you may select either Radian Mode or Degree Mode. This setting affects how ***EdScheme*** interprets inputs to trigonometric functions and certain turtle graphics procedures, and it determines how you should interpret the output from the inverse trigonometric functions and certain other turtle graphics procedures. Note, however, that some procedures (for example, **make-polar**—see the relevant entry on page 180 in Chapter 11: *Syntax and Semantics*) always interpret arguments as being in radians, irrespective of the Angle Mode you have selected. All such procedures are identified specifically in their respective entries in Chapter 11.

By default, ***EdScheme*** is set to Radian Mode. Changes in Angle Mode setting go into immediate effect.

You may choose whether you want ***EdScheme*** to pay attention to the distinction between upper- and lower-case letters. If you reject case-sensitivity by clicking on one of the Lower Case or Upper Case radio buttons, then by so doing you are selecting the default case for all aspects of ***EdScheme***'s operation. Either one of these two single-case settings behaves in accordance with the requirements of the *Revised*^{4} *Report on the Algorithmic Language Scheme*.

By default, ***EdScheme*** is set to lower-case. Changes in case-sensitivity do *not* go into immediate effect; you must first restart ***EdScheme***.

The four checkboxes in this dialog box affect ***EdScheme***'s behavior as follows:

- If you check GC On Error, ***EdScheme*** will perform a partial garbage collection whenever an error occurs. (See the entry for the procedure **gc** on page 164 in Chapter 11: *Syntax and Semantics* for more details.) If you leave this checkbox unchecked, no such garbage collection is forced.

- If you check Extended Procedure Representations, then each derived procedure is represented externally by ***EdScheme*** in such a way as to in-

clude the procedure's parameter list, the 'body' of its definition, and a representation of the environment within which its values are to be calculated. For example, if you define the derived procedure **second** as follows:

```
(define second
  (lambda (x)
    (first (rest x))))
```

and you check Extended Procedure Representations, then the Scheme expression **second** returns

```
<derived procedure (x) (first (rest x)) ()>
```

in Standard Scheme Mode, or

```
(derived (x) (first (rest x)) ())
```

in Schemer's Guide Mode. Or, if the derived procedure **tag-object** is defined as follows:

```
(define tag-object
  (let ([lbl1 'atom] [lbl2 'list])
    (lambda (s)
      (if (atom? s)
          (list lbl1 s)
          (list lbl2 s)))))
```

and the Extended Procedure Representations checkbox is checked, then the Scheme expression **tag-object** returns

```
<derived procedure (s) (if (atom? s)
                           (list lbl1 s)
                           (list lbl2 s))
                       (((lbl1 atom) (lbl2 list)))>
```

in Standard Scheme Mode, or

```
(derived (s) (if (atom? s)
                 (list lbl1 s)
                 (list lbl2 s))
             (((lbl1 atom) (lbl2 list))))
```

in Schemer's Guide Mode. (For further information concerning the external representation of procedures, see Section 10.8: *Procedures* in Chapter 10.)

On the other hand, if you leave the **Extended Procedure Representations** checkbox unchecked, then all derived procedures are represented externally as `<derived procedure>` in Standard Scheme Mode, or as `(derived function)` in Schemer's Guide Mode.

- If you check **Print Values when Loading**, then, whenever you load an ***EdScheme*** file using the [Load ...] item in the [Evaluate] menu (see later in this chapter) or the procedure **load** (see the relevant entry on page 176 in Chapter 11: *Syntax and Semantics*), ***EdScheme*** will display the value of each of the file's Scheme expressions in the Transcript Window in order of appearance and then returns the value of the final expression. Activating this feature makes it easy to identify which is the first expression in a file to generate an error, thus speeding up the debugging process. If this checkbox is left unchecked, the value of the final Scheme expression in the file is returned, but no other values will appear in the Transcript Window.

- If you check **Print Values when Evaluating Selection or File**, ***EdScheme*** will behave in the manner just described whenever you evaluate some or all of the Scheme expressions in a document window using the [Selection] or the [Whole file] items in the [Evaluate] menu. If this checkbox is left unchecked, then the value of the final Scheme expression in the window or the selected portion thereof will be returned to the Transcript Window.

By default, ***EdScheme*** leaves all four of these checkboxes unchecked. Changes in the status of the checkboxes bring about immediate changes in ***EdScheme***'s behavior.

The ***EdScheme*** procedures **language-set-preferences** and **language-preferences** give you control from within a program over many of the settings that feature in this dialog box. See the relevant entries on pages 170 and 170 in Chapter 11: *Syntax and Semantics*.

[Debugging]

Brings up a Macintosh dialog box that gives you control over the manner in which ***EdScheme*** provides you with information concerning errors.

The radio buttons under the heading 'New Debug Transcripts' allow you to choose where you want error messages directed.

- If you click on **Following Each Error**, then each time an error occurs a new Debug Transcript Window is opened and error messages are displayed in accordance with the settings you make in the lower portion of this dialog box.

- If you click **In New Context Only**, then a new Debug Transcript Window is opened whenever an error occurs in circumstances when the current environment is different from what it was when evaluation of the current expression began. When such a Debug Transcript Window is opened, error messages are displayed as in the **Following Each Error** case. If, however, no change has occurred in the current environment, then no new Debug Transcript Window is opened, and error messages are displayed in the Primary Transcript Window in accordance with the settings you make in the left half of the lower portion of this dialog box. (The Primary Transcript Window is the window—either the Transcript Window itself or a Debug Transcript Window—that contains the Scheme expression being evaluated when the error occurred.)

- If you click on **Never**, then no Debug Transcript Window is ever opened, and all error messages are directed to the Transcript Window in accordance with the settings you make in the left half of the lower portion of this dialog box. (In addition, the right half of the lower portion is disabled.)

For further information concerning Debug Transcript Windows, refer to Section 3.1: *Transcript Windows* of Chapter 3: *Programming Environment Windows*.

Each of the two lower sections of the dialog box includes an array of three checkboxes that may be set independently of each other and that control the extensiveness of the error messages ***EdScheme*** generates.

In each array that is enabled the checkbox settings have the following effects:

- If you check **Explanation**, then ***EdScheme*** will provide fuller information concerning the immediate cause of each error than is provided in ***EdScheme***'s most basic error messages, which often do little more than alert you to the occurrence of an error without giving much indication as to its possible cause.

- If you check Context, then ***EdScheme*** will tell you the names of all the local variables that have bindings at the time when the current error occurred.

- If you check Stack Trace, ***EdScheme*** will provide information concerning the recent history of the evaluation-in-progress that has given rise to the current error.

To view the default settings of these parameters, click on the Defaults button. To bring any change of settings into immediate effect, click on the OK button. To dismiss this dialog box without changing ***EdScheme***'s behavior with regard to error messages, click on the Cancel button.

Memory

Brings up a Macintosh dialog box that enables you to control a variety of memory allocations and related settings. The areas under your control and the corresponding default settings are as follows:

Files: the maximum number of documents that may be open at the same time (provided sufficient memory is allocated for the **Text** buffer—see later in this section). **Default: 8.**

Recursion Depth: the maximum number of fully recursive (as opposed to tail recursive) calls that are permitted before ***EdScheme*** generates a *Too complex* error. (It is highly unlikely that you will ever need to increase this figure. The most likely cause of a *Too complex* error is that a fully recursive procedure has failed to terminate properly. However, it may sometimes be useful to *decrease* the maximum recursion depth, since this is a convenient means of releasing quite a lot of memory.) **Default: 1024.**

Repeats: the maximum permitted number of nested repeat-loops. **Default: 10.**

Trace Depth: the maximum recursion depth of the history reported when Stack Trace is checked in the Debugging dialog box, described above. **Default: 32.**

Buffer sizes (in kilobytes):

Text: the total amount of memory set aside for all open text and document windows. This setting places no restriction on the *number* of such windows that may be open at once (the setting under **Files**, described above, does that), only the total amount of memory that they utilize. In view of the shared use of this memory space, it is a good idea to close large documents prior to running an application that involves text windows.

The appearance of the error message *Cannot create text window* means that your **Text** Buffer setting is not big enough for the memory requirements of the text and document windows you have tried to open simultaneously. **Default: 80K.**

Graphics: the total amount of memory set aside for all open graphics windows. Once again, the number of open windows is not at issue, only the amount of memory they utilize.

The amount of memory you should set aside for graphics purposes will depend upon how many colors your system has in operation. You should decide how many graphics windows onto default size (that is, 576 by 720) turtle planes you want to be able to have open at one time, and then reserve memory according to the following table:

monochrome systems:	53K per graphics window
color systems:	
4 colors:	105K per graphics window
16 colors:	210K per graphics window
256 colors:	420K per graphics window

If your turtle planes will be any size other than the default, then the amount of memory to be reserved can be calculated by means of simple proportion based on area. Thus, the memory necessary for a 300 by 200 turtle plane on a 16-color system is

$$(300 \times 200 \times 210\,\mathrm{K}) / (576 \times 720),$$

or approximately 31K.

As with text windows, the appearance of the error message *Cannot create graphics window* means that your **Graphics** setting is too small for

the number and size of graphics windows you have tried to have open simultaneously. **Default: 0K.**

WARNING: Graphics and text windows will encroach onto each other's buffer space, if that is the only memory available. So, for example, if your [Preferences] settings are 64K for Text Buffer and 110K for Graphics and you open three graphics windows on a monochrome system in which memory is very tight, then your available Text Buffer will have been reduced below the reserved amount of 64K, possibly to as low as 15K.

Transcript: the amount of memory set aside for the contents of the Transcript Window. When the memory required to show the current contents of this window exceeds the amount reserved in this dialog box, text is discarded from the top of the Transcript; it 'scrolls off the top of the window' and is subsequently unrecoverable. **Default: 10K.**

Debugging: the amount of memory allowed for the contents of each Debug Transcript. As soon as the contents of a Debug Transcript exceed this amount, text begins to be discarded from the top of the window, as in the case of the Transcript Window. **Default: 5K.**

Oblist: the amount of memory set aside for the oblist. You will only need to increase this setting if your program generates enormous quantities of data that are subsequently called upon by the program. Although you may be able to enter a larger number into this dialog, in fact ***EdScheme*** will not increase the oblist memory allocation beyond 64K. *It must never be set to less than 4K.* **Default: 16K.**

Compiler buffer: the size of the compiler workspace. If your program includes some really large Scheme expressions, you may find that an *Expression error* is generated when you try to load or otherwise evaluate them. If this happens, you are advised to increase the compiler buffer setting in increments of 16K until the program (or expression) successfully evaluates. **Default: 16K.**

Note that changes in memory allocation made using this dialog box only come into effect after ***EdScheme*** is restarted.

Specify ...

By default, ***EdScheme*** keeps a record of your current preferences in an inaccessible file called 'EdScheme Preferences' in the 'EdScheme' folder. In some configurations—for example, when the network version of ***EdScheme*** is run from a file server—this may not be a suitable choice of name or location. Choosing the **Specify ...** item from the **Preferences** sub-menu brings up a 'Save File As:' Macintosh File Selector dialog box that enables you to specify some other preferences file name and/or location. Once you have made such a specification, any changes to ***EdScheme***'s preferences made in the current session will be saved to the specified file instead of to the default file.

EdScheme may be invoked by double-clicking on an ***EdScheme*** preferences file (denoted by the plain λ icon). In such a case, the clicked-on file becomes the default preferences file. Thus, for example, network users may use the **Specify ...** sub-menu item to save a preferences file onto a local floppy disk, and may subsequently use this file to startup a customized ***EdScheme*** (even though the ***EdScheme*** application resides on a hard drive or file server).

New Transcript

Closes the current Transcript Window, after inviting you to save its contents (if you have not already done so using either the **Save** or the **Save As ...** items in this menu), and opens a fresh Transcript Window with the title 'Transcript'.

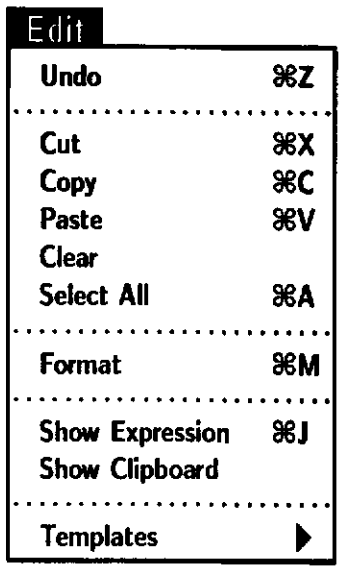
Open As Transcript ...

Brings up a Macintosh File Selector dialog box in which are listed the names under which the contents of previous transcript windows have been saved. If you select one of these 'filed' transcripts, you are invited to save the contents of the current Transcript Window (if you have not already done so using either the **Save** or the **Save As ...** items in this menu), the current Transcript Window is closed, and the 'filed' transcript you have selected is reinstated as the Transcript Window. Such an action does *not*, however, reinstate the environment that was current when the filed transcript was saved. In particular, this means that any derived procedures defined in the filed transcript will not be recognized until you copy their definitions (one by one) to the current prompt and re-evaluate each one.

Quit [Hot Key: COMMAND-Q]

Quits from ***EdScheme***; equivalent to evaluating the Scheme expression **(quit)**.

.2 The Edit Menu



The **Edit** menu contains ten items, as shown on the right. You will notice that many of the descriptions in this section refer to the clipboard. Detailed information, concerning the clipboard is provided in Section 3.5: *The Clipboard Window* in Chapter 3: *Programming Environment Windows* and in the entries in Chapter 11: *Syntax and Semantics* for the procedures whose names begin with **clipboard**.

Undo/Redo [Hot Key: COMMAND-Z]

This menu item lets you to change your mind. In particular, each of the following actions is undoable using this menu item:

typing, cutting, pasting, clearing, formatting.

Having been undone, each of these is also redoable. Note that if a cut is undone, the previous contents of the clipboard are reinstated. The text of the first item in the **Edit** menu changes according to what action, if any, is undoable or redoable and whether it is in the undo or redo phase.

Cut [Hot Key: COMMAND-X]

Removes the currently selected text from its current position, and substitutes it for the current contents of the clipboard, ready for subsequent pasting (see the description below of the Paste item in this menu). In the context of a text window, the procedure text-cut serves the same purpose as this menu item (see the relevant entry on page 224 in Chapter 11: *Syntax and Semantics*.)

See page 24 in Chapter 2: ***An EdScheme Session*** for a description of three ways to select text using the mouse. Text may also be selected in a text window using the procedure **text-set-selection** (see the relevant entry on page 227 in Chapter 11: *Syntax and Semantics*). In addition, text may be selected using the **Select All** item from this menu.

To move text from one place to another, first **Cut** it to the clipboard and then **Paste** it into its new location.

Copy [Hot Key: COMMAND-C]

Replaces the current contents of the clipboard by the currently selected text, while leaving that text in its current position. In the context of a text window, the procedure **text-copy** serves the same purpose as this menu item (see the relevant entry on page 224 in Chapter 11: *Syntax and Semantics*).

To copy text from one place to another, first Copy it to the clipboard, and then Paste it into its intended new location.

Paste [Hot Key: COMMAND-V]

Inserts the current contents of the clipboard beginning at the current location of the caret. In the context of a text window, the procedure **text-paste** serves the same purpose as this menu item (see the relevant entry on page 225 in Chapter 11: *Syntax and Semantics*).

Clear

Deletes the currently selected text from its current position, without changing the contents of the clipboard. In the context of a text window, the procedure **text-clear** serves the same purpose as this menu item (see the relevant entry on page 224 in Chapter 11: *Syntax and Semantics*).

Select All [Hot Key: COMMAND-A]

Selects the entire contents of the current window.

Format [Hot Key: COMMAND-M or OPTION-TAB]

Lays out some or all of the Scheme expressions in the current document or Transcript Window in the standard indented, 'pretty-printed' form. If a portion of text is currently selected, then all lines that are touched by that selection will be correctly laid out, each line relative to the one that precedes it (if such there be). If no text is currently selected, then the entire Scheme expression in which the caret resides is correctly laid out. To select text, you can use the mouse and any of the three methods described on page 24 in Chapter 2: *An EdScheme Session*, or you can select the entire contents of a document window using the Select All item in this menu.

To format a single line relative to the preceding lines, rather than selecting the line and using this menu item, you can alternatively place the caret in the line in question, and hit the TAB key.

Show Expression [Hot Key: COMMAND-J]

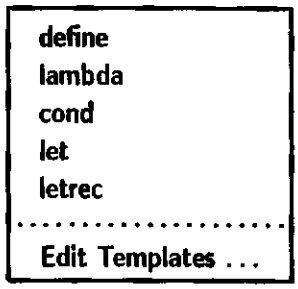
Opens the Expression Window and displays the current expression in it. (See Section 3.4: *The Expression Window* in Chapter 3: *Programming Environment Windows* for fuller information.) If the Expression Window is already open, then this menu item will read Hide Expression, and choosing it will *close* the Expression Window.

Show Clipboard

Opens the Clipboard Window, displaying its current contents. (See Section 3.5: *The Clipboard Window* in Chapter 3: *Programming Environment Windows* for fuller information.) If the Clipboard Window is already open, then this menu item will read Hide Clipboard, and choosing it will close the Clipboard Window.

Templates

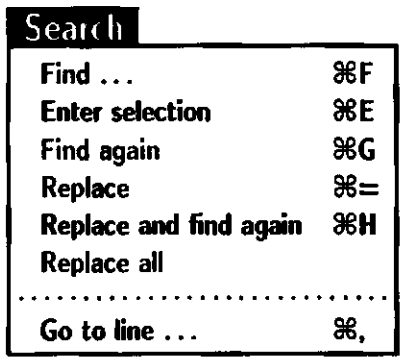
Activates a sub-menu containing a variable number of items, the last of which is Edit Templates ..., the rest being selected ***EdScheme*** keywords (those on the right, for example). Choosing a keyword from this sub-menu inserts the corresponding template at the current caret position in the current transcript or document window, and places the caret in the position specified in the template in question.



The Edit Templates ... menu item allows you to edit existing templates or add new ones. It activates a Macintosh dialog box containing an up-down arrow button that enables you to cycle through the templates already defined. Note that the definition of multi-line templates is *not* 'pretty-printed' in the 'Template' box; such 'pretty-printing' is taken care of—provided you have the appropriate Format & Paren-Matching checkbox checked in the Windows dialog accessed through the Preferences sub-menu in the File menu—at the time that the template is inserted into the current window.

Templates may also be accessed using hot key combinations. Each of these is of the form SHIFT-OPTION-n, where n is the number of the template in question in the Templates sub-menu, counting from the top. (To press SHIFT-OPTION-1, for example, you hold down the SHIFT and OPTION keys, and at the same time press the 1 key.)

.3 The Search Menu



The **Search** menu contains seven items, as follows:

Find ... [Hot Key: COMMAND-F]

Brings up a 'Search and Replace' Macintosh dialog box. In the upper left field of this box (entitled 'Search for:') you may enter words or sequences of characters that you are interested in locating in the current document or transcript window. You can do this simply by typing, or you can enter text into this field without actually activating the dialog box. Two ways to do this are explained below in the description of the **Enter selection** menu item.

In the upper right field (entitled 'Replace with:') you may—if you so desire—enter words or sequences of characters with which you want to replace some or all of the occurrences of whatever you have entered in the 'Search for:' field. To switch between these two fields, either use the mouse or the TAB key.

The dialog box also contains three checkboxes labeled **Ignore Case**, **Match words**, and **Whole file**, and three buttons labeled **Find**, **Don't Find**, and **Cancel**.

- If you exit from this dialog box by clicking on the **Cancel** button, any changes you have made will be jettisoned and the box will be dismissed.
- If you exit by clicking on the **Don't Find** button, any changes you have made will be retained, but the box will be dismissed without any further action taking place.
- If you exit by clicking on the **Find** button, the subsequent behavior depends on the settings of the three checkboxes (which operate independently of each other).
  - If **Whole file** is checked, the first occurrence of the search string in the current document or transcript window will be found. If it is unchecked, the first occurrence beyond the caret's current position will be found.
  - If **Ignore Case** is checked, *EdScheme* will look for occurrences of the search string, paying no attention to the distinction between upper-

and lower-case letters. If it is unchecked, ***EdScheme*** will ignore all occurrences that do not match exactly character by character.

- If **Match words** is checked, ***EdScheme*** will only find occurrences where the search string is *not* part of some longer sequence of characters. For example, if the search string is `form` and **Match words** is checked, then ***EdScheme*** will find the next (or first—depending on the setting of **Whole file**) occurrence of the word 'form', but will not recognize this sequence of four characters in the words 'forms' or 'information'. On the other hand, if **Match words** is unchecked, then ***EdScheme*** will recognize occurrences of the search string that are part of longer sequences of characters.

If the search is successful, the current document or transcript window scrolls so as to make the found occurrence visible, and the occurrence in question is highlighted. If the search is unsuccessful, then the caret is left in the position where it was when the dialog box was activated.

You may enter a special character or control character into either of the two fields of this dialog box by pressing whatever key combination ordinarily produces it. The only exception to this is the newline character. To enter that into one of the fields, use the key combination COMMAND-RETURN. (This may be useful if, for example, you want to comment out a section of a program by searching for newline characters and replacing them by a newline character followed by a semicolon.)

Enter selection [Hot Key: COMMAND-E]

Replaces the current contents of the 'Search for:' field in the 'Search and Replace' Macintosh dialog box by the currently selected text, without activating the dialog box, and without altering the contents of the 'Replace with:' field. An alternative, 'hot key' method for achieving the same result in the case of a single word is to hold down the OPTION key and double click on the word in question. The word will be highlighted in the current document or transcript window, and at the same time will be entered into the 'Search for:' field in the 'Search and Replace' dialog box.

Find again [Hot Key: COMMAND-G]

Finds the next occurrence (if such exists) in the current document or transcript window of whatever is currently entered in the 'Search for:' field of the 'Search and Replace' Macintosh dialog box.

Replace [Hot Key: COMMAND-=]

Replaces the currently selected text in the current document or transcript window by whatever is entered in the 'Replace with:' field of the 'Search and Replace' Macintosh dialog box. On most occasions, the selected text will have been selected as the result of a search operation (initiated either by exiting from the 'Search and Replace' dialog box by clicking on the Find button, or by choosing the **Find again** item or the **Replace and find again** item from the **Search** menu). However, the replacement will take place no matter how the text selection has been made, and whether or not the selected text agrees with what is entered in the 'Search for:' field of the dialog box.

If no text is selected, then the effect of choosing this menu item is to insert whatever is entered in the 'Replace with:' field at the current caret position. Similarly, if the 'Replace with:' field is empty, then choosing this menu item has the effect of deleting the selected text.

Replace and find again [Hot Key: COMMAND-H]

Produces the same result as choosing the **Replace** item and then choosing the **Find again** item from the **Search** menu.

Replace all

Replaces all occurrences of whatever is entered in the 'Search for:' field of the 'Search and Replace' Macintosh dialog box by whatever is entered in the 'Replace with:' field. The replacements include all occurrences in the current document or transcript window if the **Whole file** checkbox is checked; otherwise, they include only those occurrences that come *after* the caret's current position.

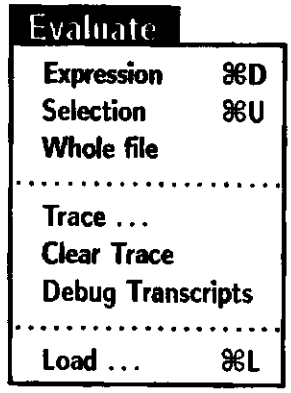
Go to line ... [Hot Key: COMMAND-,]

Brings up a Macintosh dialog box that allows you to specify on which line of the current document or transcript window you would like the caret to be placed. If you dismiss this dialog by clicking on the **OK** button, the caret will be placed at the start of the line in question. If you specify a line number greater than the number of lines in the window, the caret will be placed at the start of the last line in the window.

This feature may also be used to discover the line number of the line that currently contains the caret, because the 'Go To Line:' box displays this information as soon as the dialog box is activated.

.4 The Evaluate Menu

The **Evaluate** menu contains seven menu items, as follows:



Expression [Hot Key: COMMAND-D]

Pastes the current expression into the current Transcript Window, and then evaluates that expression, returning the result in the Transcript Window. For descriptions of how the current expression is determined, refer to Sections 2.3 and 3.4. To view the current expression, choose the **Show Expression** item in the **Edit** menu (described above).

You may achieve the same effect as choosing this menu item by placing the caret immediately after the final right parenthesis of the current expression and hitting the Keypad-ENTER key.

Selection [Hot Key: COMMAND-U]

Evaluates all the Scheme expressions included in the currently selected text, either returning the value of each expression as it is evaluated or returning only the value of the last expression, depending on whether or not you have checked the Print Values when Evaluating Selection or File checkbox in the **Language** dialog box accessed through the **Preferences** sub-menu in the **File** menu. (Three ways to select text are described on page 24 in Chapter 3: *Programming Environment Windows*.)

If an error occurs while these evaluations are taking place, then the evaluation process stops and the error is reported according to your settings in the **Debugging** dialog accessed through the **Preferences** sub-menu in the **File** menu. If, in addition, the selected text is in a document window, then the offending expression will be highlighted so that you can easily identify it the next time the document window in question is the selected window.

Whole file

Evaluates all the Scheme expressions in the current document window, either returning the value of each expression as it is evaluated or returning only the value of the last expression, depending on whether or not you have checked

the Print Values when Evaluating Selection or File checkbox in the Language dialog box referred to in the previous item.

If an error occurs while *EdScheme* is performing some evaluations in response to your having chosen this menu item, then the subsequent behavior is the same as in the case of the Selection item just described.

Trace ...

Brings up a Macintosh dialog box that enables you to control *EdScheme*'s tracing facility. In the upper left corner of the dialog box is a scrollable window containing an alphabetized list of procedure names. If the checkbox entitled Show Only Derived Procedures is checked, then only the derived procedures you have defined or loaded during the current session will be listable. If this checkbox is not checked, then all the primitive and derived procedures will be listable.

Exactly which procedures are actually listed in this window depends upon the trace or untrace action specified in the pop-up menu entitled Action: that appears to the right of the procedure list window. The items in this menu correspond to the six trace and untrace procedures described on pages 230 and 234 in Chapter 11: *Syntax and Semantics*. The six menu items are:

Trace Entry	Trace Exit	Trace Both
Untrace Entry	Untrace Exit	Untrace Both

Only one of these may be chosen at a time, although choosing Trace Both is equivalent to choosing both Trace Entry and Trace Exit, and similarly for Untrace Both.

- If one of the three trace actions is chosen, then the procedure list includes the names of all procedures conforming to the setting of the Show Only Derived Procedures checkbox that are not already being traced in the chosen manner. Thus if a procedure is being traced on entry, its name will be listed if you choose Trace Exit or Trace Both, but not if you choose Trace Entry. On the other hand, if a procedure is already being traced both on entry and on exit, it will *not* be listed no matter which of the three trace actions is chosen.

- If one of the three untrace actions is chosen, then the procedure list includes the names of all procedures currently being traced, at least partially, in the chosen manner. Thus if a procedure is being traced on

entry, its name is listed if you choose Untrace Entry or Untrace Both, but not if you choose Untrace Exit. On the other hand, if a procedure is being traced both on entry and on exit, it is listed no matter which of the three untrace actions is chosen.

The checkbox entitled Output to Trace Window controls whether trace information is directed to a special Trace Window (if the checkbox is checked) or to the current Transcript Window (otherwise).

To initiate some form of trace on one or more procedures, select the name(s) of the procedure(s) in question from the procedure list, choose the desired type of trace action, set the Output to Trace Window checkbox in accordance with where you would like the trace information directed, and click on the OK button.

The trace action you choose for different procedures may be different. Simply select all the procedures you want traced in one way, choose the desired trace action, and then click on the OK button. Then rechoose the Trace ... menu item, select the procedures you want traced in another way, choose that alternative trace action, and click on the OK button again. And so on. The state of the Output to Trace Window button on the most recent occasion when you exited from the Trace ... dialog box by clicking on the OK button determines the destination of all trace information.

To discontinue some aspect of the trace action currently in force, choose one of the three untrace actions, select one or more of the listed procedure names, and click on the OK button. As in the case of tracing, you may selectively discontinue different aspects of tracing in regard to different procedures. To discontinue all tracing of any kind and dismiss this dialog box *without being able to change your mind*, click on the Untrace All button.

To exit from the Trace ... dialog box without changing any aspect of ***EdScheme***'s current tracing behavior, click on the Cancel button.

For your information, the dialog box displays how many procedures are listed in the scrollable window and the number of procedures you have selected (and not subsequently de-selected) since calling up the dialog box.

If the Trace Window is showing, you may hide it in any of the ways you can hide (or 'close') the Transcript Window or any Debug Transcript. It will continue to receive all trace information, even though the window is no longer visible. To show the Trace Window again, use the Trace Window item in the Windows menu (described below) or the alternative COMMAND-T hot key combination.

Clear Trace

Clears the contents from the Trace Window, whether or not it is currently visible.

Debug Transcripts

Lets you temporarily override the settings you have made in the Debugging dialog box accessed through the Preferences sub-menu in the File menu. If you have chosen to have New Debug Transcripts open Following Each Error or In New Context Only, there will be a solid diamond next to this menu item. By choosing Debug Transcripts, you will make this diamond disappear, and—until you choose this menu item again or reset or restart the application—*EdScheme* will behave as if you had clicked on Never in the Debugging dialog box. Rechoosing this menu item resets *EdScheme*'s debugging activity according to your Preferences settings. However, no use of this menu item has any actual effect on those settings.

On the other hand, if you have clicked on Never in the Debugging dialog box, this menu item will be 'grayed out', that is, you will not be able to choose it.

Load ... [Hot Key: COMMAND-L]

Brings up a Macintosh File Selector dialog box from which you may select a file to be loaded into *EdScheme*. If you dismiss this File Selector by clicking on the Cancel button, *EdScheme* returns the boolean `#f`. If you select a file and dismiss the File Selector by clicking on the Open button, *EdScheme* loads the file in question, evaluating all the Scheme expressions it contains. If you have checked the checkbox entitled Print Values when Loading in the Language dialog box accessed through the Preferences sub-menu in the File menu, then the value of each Scheme expression is displayed as it is evaluated, and the value of the last expression is returned. On the other hand, if that checkbox is not checked, then only the value of the final Scheme expression in the file is returned.

If an error occurs while the loading is in progress, then evaluation will stop, and the error will be reported in accordance with the settings you have made in the Debugging dialog accessed through the Preferences sub-menu in the File menu. In addition, *EdScheme* takes steps to preserve the contents of your files by closing all files that are open for loading at the time when the error occurs.

Choosing this menu item is equivalent to evaluating the Scheme expression

```
(load (choose-input-file))
```

Note, however, that the combined use of the **load** and **choose-input-file** procedures is more versatile than choosing this menu item. In fact, evaluating the Scheme expression

```
(load (choose-input-file) #t)
```

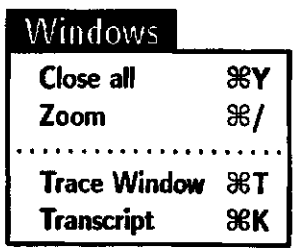
will enable you to select an input file whose contents will then be loaded as if the Print Values when Loading checkbox were checked, *whether or not it actually is*. Similarly, evaluating the Scheme expression

```
(load (choose-input-file) #f)
```

loads your selected file as if that checkbox were unchecked, whether or not it is.

.5 The Windows Menu

The Windows menu contains a variable number of items, the first four of which are as shown on the right.



Close all [Hot Key: COMMAND-Y]

Closes all open document windows, after offering you the opportunity to save those whose contents have been changed since being opened.

Zoom [Hot Key: COMMAND-/]

Either expands the current window to full screen or returns it to its original size, depending on its current state. Choosing this item is equivalent to clicking on the zoom box in the upper right hand corner of the window.

Trace Window [Hot Key: COMMAND-T]

Selects the Trace Window, reshowing it if it is currently hidden.

Transcript [Hot Key: COMMAND-K]

Selects the Transcript Window, reshowing it if it is currently hidden.

Document Windows [Hot Key: COMMAND-<number>]

If you have any document windows open, there will be additional items at the bottom of the Windows menu, each one corresponding to a document window. The listed items will be the names of the files in the windows, and next to each one will be a hot key combination of the form COMMAND-<number>. Using these menu items or the corresponding hot key combinations, you can quickly switch between your open files.

-o O o-

0 Data Expressions

In this chapter we describe *EdScheme*'s data types, whose representations are referred to collectively as 'data expressions'. Note that *all* of *EdScheme*'s data expressions (including procedure objects, streams, continuations, and environments) are first class. That is, they may be presented as arguments to procedures, returned by procedures, and stored in data structures.

The Windows dialog box accessed through the Preferences sub-menu in the File menu allows you to control the colors and/or typefaces in which data expressions, Scheme expressions—the expressions that make up a Scheme program—and system messages are printed on the screen. This facility makes it possible for you to make a clear, visual distinction between evaluated and unevaluated expressions. (For further information, see the entry dealing with this dialog box starting on page 72 in Chapter 9: *The EdScheme Menus*.)

Throughout this chapter there are references made to *EdScheme* procedures. Detailed information about each of them may be found by consulting the alphabetical listing in Section 11.2: *The Syntax of EdScheme* in Chapter 11: *Syntax and Semantics*. The information in this chapter is of necessity more densely-packed and technical in nature than the rest of the Guide. It represents an attempt to express the main features of the formal grammar of Scheme in something approaching 'everyday' language.

10.1 Identifiers: Keywords and Variables

Identifiers (otherwise known as **symbols**) fall into two disjoint categories: **keywords** and **variables**. They are the only *EdScheme* objects that cause

the predicate **symbol?** to return the boolean `#t`. The rules governing the construction of identifiers are as follows:

- The 'stand-alone' tokens `+` `-` and `...` are identifiers.
- Every other identifier is a sequence of one or more tokens, the first of which must be in one of these three lists:

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		
!	$	%	&	*	/	:	<	=	>	?	~	_	^

  If there are any tokens after the first, they may be any of those in the above three lists or any of these additional tokens:

.	+	-	0	1	2	3	4	5	6	7	8	9

- The following identifiers are *EdScheme*'s **keywords**:

```
=>              and             begin           case
cond            define          delay           do
else            if              lambda          let
let*            letrec          or              quasiquote
quote           rec             set!            unquote
unquote-splicing
```

- In *EdScheme*, a **variable** is any identifier that is not a keyword.

Identifiers are *not* self-evaluating; they must be 'quoted' if you want to provide them—as opposed to their *values*—as arguments to procedures, that is, they must be preceded by a single quote-mark (see the relevant entry on page 109 in Chapter 11: *Syntax and Semantics*) or they must be presented in the form of a quote-expression (see the entry for **quote** on page 202).

For example, `$salaries$` is a variable. It can be bound to a value using a define-expression such as the following:

```
(define $salaries$ '(17500 52000 54650))
```

Then the different effects of quoting or not quoting this variable are seen in the following evaluation:

```
(cons '$salaries$ $salaries$)  ↦  ($salaries$ 17500 52000 54650)
```

.2 Booleans

The *EdScheme* boolean objects—denoting *true* and *false*, respectively—are `#t` and `#f`. They are the only *EdScheme* objects that cause the predicate **boolean?** to return `#t`. They are self-evaluating; that is, they do not need to be quoted (as identifiers do).

In Standard Scheme Mode, `#f` is the only data expression recognized as 'false' by *EdScheme*'s predicates; every other data expression (including the empty list—see the next section) is recognized as 'true' by *EdScheme*'s predicates. (For information concerning *EdScheme*'s two Language Modes, see the section—starting on page 74—of Chapter 9: *The EdScheme Menus* that deals with the Language dialog accessed through the Preferences item in the File menu.)

In Schemer's Guide Mode, `#t` is the only data expression recognized as 'true', and `#f` is the only data expression recognized as 'false'.

.3 Pairs and Lists

A **pair** is a data object that has two components, the first of which is accessed by either of the procedures **car** or **first**, the second being accessed by either of the procedures **cdr** or **rest**. Pairs may be constructed using the procedure **cons**. They are the only *EdScheme* objects that cause the predicate **pair?** to return the boolean `#t`.

In Standard Scheme Mode, either component of a pair may be any data expression. (For information concerning *EdScheme*'s two Language Modes, see the section—starting on page 74—of Chapter 9: *The EdScheme Menus* that deals with the Language dialog accessed through the Preferences item in the File menu.)

In Schemer's Guide Mode, the first component of a pair may be any data expression, but its second component must be a list (see the next paragraph).

Lists may be described recursively as follows:

- The empty list ( ) is a list.
- Whenever L is a list, then any pair with second component L is also a list.
- Every list may be analyzed by 'reverse application' of the previous relationship in such a way that after a *finite* number of steps a pair is

reached whose second component is the empty list.

The data expressions that are the first components of the successive pairs revealed in this analysis of a list are the list's **elements**. The number of such pairs is the **length** of the list. The empty list is deemed to have length zero; it has no elements. The elements of a list are **indexed**, the first element having index 0, the second having index 1, and so on. The only valid indexes for a list of length n are the exact integers from 0 through $n - 1$.

EdScheme prints a non-empty list by printing its elements in order, separated by spaces, and enclosed in a pair of parentheses. For example,

```
(a b c d e)
```

As the above description implies, Standard Scheme Mode allows **improper lists**. They are formed in the same way as 'proper' lists, except that the second component of the final pair is some data expression other than the empty list. Such improper lists are printed in the same way as lists, except that the first component of the final pair is separated from its second component by a space-delimited dot. For example,

```
(a b c d . e)
```

In Schemer's Guide Mode, improper lists are not allowed; attempting to create them or use them will generate an error message.

In either mode, *EdScheme* allows a type of pair known as a **circular list**—even though it is not a (proper) list at all. Such a pair is recognizable from the fact that the above recursive definition of a list fails because the 'reverse application' it mentions keeps on revisiting the same element(s) over and over again without ever bringing the empty list to light. Since circular lists cycle endlessly, it is of course difficult to represent them with a finite number of tokens. *EdScheme* does the best it can by making fairly liberal use of ellipses: '...'. For example, the following let-expression sets the second component of the pair *a* to be *a* itself. The result is a circular list that is endlessly 'chasing its tail'.

```
(let ([a '(1 2)])
  (set-rest! a a)
  a)                ↦  (1 1 ...)
```

In either mode, the empty list is the only *EdScheme* object that causes the predicate **null?** to return the boolean **#t**, and (proper) lists are the only

EdScheme objects that cause the predicate **list?** to return the boolean **#t**. Neither pairs nor lists are self-evaluating; they should be 'quoted' when provided as arguments to procedures.

Numbers

EdScheme's treatment of numbers is so extensive and versatile that we have devoted an entire chapter of this User's Guide to explaining it. See Chapter 5: *Numbers and Numeric Functions*, starting on page 35.

Numbers are the only *EdScheme* objects that cause the predicates **number?** and **complex?** to return the boolean **#t**. They are self-evaluating.

10.5 Characters

In *EdScheme*, a **character** is any of the symbols corresponding to the ASCII character codes in the range 0 through 255. Many characters are accessible directly from the keyboard. Those that are not may be generated using the **integer->char** procedure. Characters are the only *EdScheme* objects that cause the predicate **char?** to return the boolean **#t**. They are self-evaluating.

In Standard Scheme mode, *EdScheme* prints (and allows you to type) printable characters by preceding the usual tokens with the combination **#**. For example, the character **P** is represented as **#\P**. In addition, certain characters are represented by descriptive names. For example,

#\space	denotes a space character
#\newline	denotes a newline character

In Schemer's Guide mode, *EdScheme* omits the combination **#** and uses no descriptive names unless it is explicitly given such notation to evaluate (which it does by simply returning the input character in the input notation).

10.6 Strings

In *EdScheme*, a **string** is a (possibly empty) chain of characters. Strings are the only *EdScheme* objects that cause the predicate **string?** to return the boolean **#t**. They are self-evaluating.

When printed or displayed by *EdScheme* or entered at the keyboard, the chain of characters is enclosed between double-quote marks ("). If you want to include a double-quote mark or a backslash (\) as one of the characters in a string, then you must 'slashify' (or 'escape') by preceding it with a backslash, as in the following example:

```
"This double-quote, \", is escaped by a backslash, \\."
```

The characters in a string are **indexed**, the first character having index 0, the second having index 1, and so on. The **length** of a string is the number of characters it contains. The only valid indexes for a string of length n are exact integers from 0 through $n-1$.

10.7 Vectors

A **vector** is a data object that has a finite number (possibly zero) of elements that are **indexed** in order of appearance. Each element may be any *EdScheme* data expression, the first having index 0, the second having index 1, and so on. The **length** of a vector is the number of elements it has, and the only valid indexes for a vector of length n are the exact integers from 0 through $n-1$. Vectors are the only *EdScheme* objects that cause the predicate **vector?** to return the boolean `#t`.

EdScheme prints (and allows you to type) vectors as follows: A hash-sign `#` is followed by a left parenthesis; then come the elements of the vector in order of appearance, separated by spaces; finally there is a right parenthesis. For example,

```
#(23 (a b) "yo" #() #\a)
```

In *EdScheme*, vectors happen to be self-evaluating. However, for compatibility with the requirements of *Revised*^{4} *Report on the Algorithmic Language Scheme* it is recommended that they always be quoted. (We follow this practice in this User's Guide.)

10.8 Procedures

Just like every other Scheme data object, **procedures** are 'first class'. Thus they may be passed as arguments to other procedures, they may be returned

by other procedures, and they may be stored in data structures. Each procedure contains the information it needs in order to perform an evaluation, together—in the case of derived (or compound) procedures—with the environment in which the evaluation is to be performed. Derived procedures may be generated using the **define** or **lambda** special forms. Procedures are the only *EdScheme* objects that cause the procedure **procedure?** to return the boolean `#t`.

Primitive procedures are represented externally as in the following example, which concerns the primitive procedure **first**:

```
<primitive first>   [in Standard Scheme Mode]
(primitive first)   [in Schemer's Guide Mode]
```

As far as derived procedures are concerned, you have some measure of control over their external representation. The relevant details are provided in the section—starting on page 75—of Chapter 9: *The EdScheme Menus* that deals with the checkbox **Extended Procedure Representations** in the [Language] dialog accessed through the [Preferences] item in the [File] menu.

Note that, in Schemer's Guide Mode, if you have checked the **Extended Procedure Representations** checkbox, then the external representations of procedures correspond exactly with what *The Schemer's Guide* calls **function descriptors** (or **FDs**, for short). In this mode only, the (extended) external representation of a procedure may be explicitly 'built' using **cons**, and the result may be applied just as if the external representation were the actual procedure itself. For further details, refer to Chapter 5 of *The Schemer's Guide*.

.9 Continuations

A **continuation** is a data object that represents the default future of a computation. At almost any moment, an *EdScheme* program can take a record of its current state and store this record for future use. Later, if required, the program can jump directly to the recorded state. Continuations are the only *EdScheme* objects that cause the predicate **continuation?** to return the boolean `#t`.

Continuations are sometimes called 'escape procedures', since this is the simplest of their uses; they are the modern equivalent of the less flexible **catch** and **throw** procedures of traditional Lisp systems. They are created using

the procedure **call/cc**—or, to give it its full, official title, **call-with-current-continuation.**

Continuations may *not* be captured during calls to

- **derived procedures whose exit is being traced using either of the procedures trace-exit or trace-both** (see the section—starting on page 90—of Chapter 9: *The EdScheme Menus* that deals with the Trace ... item in the Evaluate menu);

- procedures that have been compiled in-line—most primitive procedures fall into this category—but that have subsequently been redefined. To avoid this restriction, set the variable ***integrate-primitives*** to `#f`, and re-evaluate those definitions that involve redefined in-line procedures.

In *EdScheme*, each continuation is represented externally in the form:

`<continuation` *n*`>`

where *n* is a number assigned individually to a continuation when it is created.

Each captured continuation consumes a little memory (about 100 bytes per continuation) from the text/graphics buffers. Inaccessible continuations are disposed of by a garbage collection, and the memory they use is released. If you plan to store large numbers of continuations in static structures (a globally bound list, for example), you may find it necessary to increase the size of the **Text** buffer reserved in the Memory dialog accessed through the Preferences sub-menu in the File menu (see the relevant section starting on page 79 in Chapter 9: *The EdScheme Menus*).

.10 Atoms

EdScheme groups together identifiers, numeric constants, strings, characters, and boolean objects under the general heading of **atom.** Conversely, the following *EdScheme* data objects are *not* atoms:

> procedures, continuations, environments, ports, windows, menus, bitmaps, streams, lists, pairs, vectors.

Atoms are the only *EdScheme* objects that cause the predicate **atom?** to return the boolean `#t`.

10.11 Streams and Delayed Objects

A **stream** is a sequence of Scheme objects. Unlike a list, however, the objects in a stream are evaluated only when accessed and not when inserted into the sequence. This allows the representation of large, possibly infinite, data structures in a finite space. In ***EdScheme***, a stream is a pair—see Section 10.3 above—whose first component is any data expression, and whose second component is a **delayed object** (such as is created, for example, by either of the special forms **delay** or **freeze**).

The special form **cons-stream** is used to construct streams, just as the procedure **cons** is used to construct lists. Similarly, the role of the empty list is played by the variable ***the-empty-stream***, and the roles of the primitive procedures **first** and **rest** (or **car** and **cdr**) are played by the primitive procedures **head** and **tail**, respectively.

10.12 Environments

An **environment** is a data object that keeps track of variables and their values. Each environment contains a sequence of **frames**, each of which contains a table of **bindings** of variables to their values. In ***EdScheme***, bindings are represented as pairs (see Section 10.3 above); the pair `(a one)`, for example, records the fact that the variable **a** is bound to the atom **one**. ***EdScheme*** prints frames as lists of bindings, and environments as lists of frames. The application usually manipulates its environments out of sight in the background, but there are occasions when it is useful to refer to them directly.

The variable ***user-global-environment*** is bound to an environment that contains a single frame, called the **global frame**. This frame contains, among other things, bindings for ***EdScheme***'s primitive procedures. In addition, the variable ***user-initial-environment*** is bound to an environment that contains two frames: the **initial frame**, containing bindings established by you and/or the Scheme application(s) you are running, and the global frame.

Both the initial environment and the global environment are represented externally by the empty list. The frames within them are printable, however. To display the initial frame, evaluate the Scheme expression

(first *user-initial-environment*)

and to display the global frame, evaluate

(first *user-global-environment*)

At any moment, only one environment is active (for otherwise ***EdScheme*** would soon become confused as to the current values of the active variables!), and this environment is called the **current environment**. Typically, at each point in the course of an evaluation, the current environment contains the global and initial frames, plus various other frames containing, for example, bindings for local variables and procedure parameters. The Scheme expression **the-environment** returns the current environment.

The special form **make-environment** provides a means of constructing a specific environment, and the procedure **eval** allows you to specify the environment within which you wish to evaluate a given Scheme expression.

.13 Ports

Ports are data objects that represent input and output devices. They are the means by which ***EdScheme*** communicates with such devices as disk drives, printers, modems, and robot controllers. In addition, the Transcript Window—which is the principal channel of communication between you and the ***EdScheme*** interpreter (see Section 3.1: *Transcript Windows* in Chapter 3: *Programming Environment Windows*, starting on page 20)—is a port. Ports are the only ***EdScheme*** objects that cause the predicate **port?** to return the boolean `#t`.

Just like every other Scheme data object, ports are 'first class'. Thus, they can be input to procedures, output from procedures, and stored in data structures. ***EdScheme***'s ports fall into three categories: input ports, output ports, and serial ports (which are specialized kinds of input/output ports). The Transcript Window is both an input and an output port.

Ports are represented externally in the form

<port *n*>

where n is a number assigned individually to a port when it is opened. The number n is positive in the case of regular input and output ports, it is -1 for the Transcript Window, and other negative numbers for serial ports.

For much more information concerning ports and how they may be used to communicate with disk files and other parts of the 'outside world', please refer to Chapter 6: ***Files and Ports***, in particular to Section 6.2: ***Ports and File-handling***.

10.14 Miscellaneous Data Expressions

EdScheme supports several other kinds of data expressions, as follows:

Aliases: Produced using the **define–alias** special form (see the relevant entry in Chapter 11: *Syntax and Semantics*). All aliases are represented externally by `<alias>` with no indexing number.

Bitmaps: The only ***EdScheme*** data objects that cause the predicate **bitmap?** to return the boolean `#t`. Represented externally in the form

`<bitmap` *n*`>`

where *n* is a number assigned individually to a bitmap when it is 'made' using the **make–bitmap** procedure. For further information, see Section 7.3: *Bitmaps*.

Graphics Windows: The only ***EdScheme*** data objects that cause the predicate **graphics–window?** to return the boolean `#t`. Represented externally in the form

`<graphics window` *n*`>`

where *n* is a number assigned individually to a graphics window when it is opened. For further information, see Section 7.1: *Graphics Windows*.

Macros: Produced using the **define–macro** special form (see the relevant entry in Chapter 11: *Syntax and Semantics*). All macros are represented externally by `<macro>` with no indexing number.

Menus: The only ***EdScheme*** data objects that cause the predicate **menu? to** return the boolean `#t`. Represented externally in the form

`<menu` *n*`>`

where *n* is a number assigned individually to a user-defined menu when it is 'made' using the **make–menu** procedure. For further information, see Section 8.2: *Menus*.

Text Windows: The only ***EdScheme*** data objects that cause the predicate **text–window?** to return the boolean `#t`. Represented externally in the form

`<text window n>`

where n is a number assigned individually to a text window when it is opened. For further information, see Section 8.1: *Text Windows*.

Transformers: Produced using the **define–transformer** special form (see the relevant entry in Chapter 11: *Syntax and Semantics*). All transformers are represented externally by `<transformer>` with no indexing number.

10.15 Comments

EdScheme ignores all characters between a semicolon and a newline character. You may therefore embed comments within an ***EdScheme*** program by typing remarks at the right-hand end of some or all of the lines of the program and preceding the 'comment tail' of any such program line by a semicolon. Alternatively, or in addition, you can 'comment out' entire lines by ensuring that the first non-space character in the line is a semicolon.

– o O o –

Syntax and Semantics

.1 Abbreviations

The descriptions in the main section of this chapter employ a notational convention and certain abbreviations in a systematic way. The abbreviations are as shown below and on the next page:

alist	an association list, that is, a list of pairs (in the technical sense explained in Section 10.3: *Pairs and Lists*)
atom	an atom
bmap	a bitmap
bool	one of the boolean objects `#t` or `#f`
ch	a character
def	a define-expression
env	an environment
exp	a data expression
fp	a formal parameter
id	an identifier
k	an exact non-negative integer
lam	a lambda expression
list	a list
menu	a menu
n	an integer

pair	a pair, in the technical sense explained in Section 10.3: *Pairs and Lists*
port	a port
proc	a procedure
promise	an object (produced for example by the **delay** special form) that does not evaluate until passed as an argument to the procedure **force**
q	a rational number
rad	a radix, that is, one of the exact integers 2, 8, 10, and 16
rgb	an RGB triple specifying a color (see the more detailed explanation below)
sexp	a Scheme expression
spec	a file specification, either a full file specification, or one of the types of path name or file name described in Chapter 6: *Files and Ports*
str	a string
stm	a stream
sym	a symbol
temp	a quasiquote template expression
vec	a vector
var	a variable
x	a real number
z	a complex number
**win*	where * is replaced by any combination of the letters *t*, *e*, *c*, *g*, and *x*, a window of any of the types indicated by the coded replacement for *

Window Codes		
t – transcript;	*e* – expression;	*c* – clipboard;
g – graphics;	*x* – text	

The notational device we use in our descriptions is the **ellipsis**, '...'. This appears in the parameter list of those procedures described in the next section that accept a variable number of inputs. If it is preceded by just one

identifier, then the procedure takes zero or more inputs; if it is preceded by two identifiers, then the procedure takes at least one input.

The **RGB triples** referred to in the above table are lists of three whole numbers, each in the range from 0 through 65535. The first of these numbers corresponds to the 'amount' of Red in the color mix described by the triple, the second corresponds to the 'amount' of Green, and the third corresponds to the 'amount' of Blue.

Some sample RGB triples			
black	(0 0 0)	purple	(65535 0 65535)
blue	(0 0 65535)	red	(65535 3500 3500)
green	(0 65535 0)	white	(65535 65535 65535)
orange	(65535 33343 1237)	yellow	(65535 65535 0)

.2 The Syntax of *EdScheme*

In this section we describe all the language elements of ***EdScheme***. See the previous section for further information concerning the abbreviations and the notational convention that are used systematically in these descriptions. Be on the alert for references to other parts of this manual. The cross-referenced material often includes more details concerning the conditions under which ***EdScheme*** operates as described here.

#f *Constant*

Evaluates to the boolean `#f`.

#t *Constant*

Evaluates to the boolean `#t`.

'sexp *Syntactic Abbreviation*

Abbreviates the Scheme expression **(quote *sexp*)**. (See the entry for **quote** later in this section.) Example:

'(a b c d) ↦ **(a b c d)**

() *Constant*

Evaluates to the empty list; equivalent to **(quote ())**. Examples:

```
(null? ())    ↦  #t
(cons 'a ())  ↦  (a)
```

(* *z1* ...) *Procedure*

Returns the product of its arguments. (See Chapter 5: *Numbers and Numeric Procedures* for information concerning the numbers that ***EdScheme*** supports.) Examples:

```
(*)                    ↦  1
(* 3)                  ↦  3
(* -5 2.5)             ↦  -12.5
(* 2 -1/2 -4+2/3i)     ↦  4-2/3i
```

(*set-current-input-port* *port*) *Procedure*

Sets the current input port to be *port*, which must be an input port. (See Section 6.2: *Ports and File-handling* in Chapter 6: *Files and Ports* for further information concerning ports. See also the entry for the procedure **current-input-port** later in this section.)

(*set-current-output-port* *port*) *Procedure*

Sets the current output port to be *port*, which must be an output port. (See Section 6.2: *Ports and File-handling* in Chapter 6: *Files and Ports* for further information concerning ports. See also the entry for the procedure **current-output-port** later in this section.)

the-non-printing-object *Variable*

A variable initially bound in the global environment to a non-printing object. This is useful if you want a Scheme expression that you are calling primarily for its side effects to return a predictable value that will not disturb the layout of the rest of your displayed output.

(+ *z1* ...) *Procedure*

Returns the sum of its arguments. (See Chapter 5: *Numbers and Numeric Procedures* for information concerning the numbers that *EdScheme* supports.) Examples:

```
(+)                  ↦  0
(+ 3)                ↦  3
(+ 2 3.0)            ↦  5.0
(+ 2 -1/2 -4+2/3i)   ↦  -5/2+2/3i
```

,*sexp* *Syntactic Abbreviation*

Abbreviates the Scheme expression **(unquote *sexp*)** in the context of a quasiquote-expression. (See the entries later in this section for **quasiquote** and **unquote**. Also see the 'backquote' example on page 114.)

,@*sexp* *Syntactic Abbreviation*

Abbreviates the Scheme expression **(unquote-splicing *sexp*)** in the context of a quasiquote-expression. (See the entries later in this section for **quasiquote** and **unquote-splicing**. Also see the 'backquote' example on page 114.)

(- *z1*) *Procedure*
(- *z1 z2* ...)

With one argument, returns the additive inverse of that argument. With two or more arguments, returns the result of subtracting those arguments, associating to the left. (See Chapter 5: *Numbers and Numeric Procedures* for information concerning the numbers that *EdScheme* supports.) Examples:

```
(- 4)          ↦  -4
(- 1.2 4.2)    ↦  -3.0
(- 2 4 6 8)    ↦  -16
```

the reason for the final result being

$$((2-4)-6)-8 = (-2-6)-8 = -8-8 = -16.$$

(/ *z1*) *Procedure*
(/ *z1 z2* ...)

With one argument (which must be non-zero), returns the multiplicative inverse of that argument. With two or more arguments (all but the first of which must be non-zero), returns the result of dividing those arguments, associating to the left. (See Chapter 5: *Numbers and Numeric Procedures* for information concerning the numbers that *EdScheme* supports.) Examples:

```
(/ -4)                  ↦  -1/4
(/ 1.2 2.4)             ↦  0.5
(/ 24 20+4i 15 10-2i)   ↦  1/130
```

the reason for the final result being

$$\begin{aligned} ((24\ /\ 20+4i)\ /\ 15)\ /\ 10-2i &= ((15/13-3/13i)\ /\ 15)\ /\ 10-2i \\ &= (1/13-1/65i)\ /\ 10-2i \\ &= 1/130 \end{aligned}$$

(< *z1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in strictly increasing order. (The value returned by this procedure is only guaranteed to be accurate if all of its arguments are exact numbers. See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*.) Examples:

```
(< -1 0 2 10)  ↦  #t      (< 5)  ↦  #t
(< 2.58 -4/7)  ↦  #f      (<)    ↦  #t
(< 10 10)      ↦  #f
```

(<= *z1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in non-decreasing order. (The value returned by this procedure is only guaranteed to be accurate if all of its arguments are exact numbers. See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*.) Examples:

```
(<= -1 0 0 20)  ↦  #t      (<= 5)  ↦  #t
(<= 2.58 -4/7)  ↦  #f      (<=)    ↦  #t
(<= 10 10)      ↦  #t
```

(<> *z1 z2*) *Procedure*

A predicate that returns the boolean **#t** if and only if its arguments are unequal. (The value returned by this procedure is only guaranteed to be accurate if both its arguments are exact numbers. See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*.) Examples:

```
(<> -1 (* 0+1i 0+1i))  ↦  #f
(<> 3/4 4/3)           ↦  #t
```

(= *z1* ...) *Procedure*

A predicate that returns the boolean **#t** if and only if all its arguments are equal. (The value returned by this procedure is only guaranteed to be accurate if all of its arguments are exact numbers. See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*. Examples:

```
(= 5 10 (+ 2 3))    ↦  #f      (= 3)  ↦  #t
(= 1-1i (/ 2 1+1i))  ↦  #t      (=)    ↦  #t
```

=> *Keyword*

(Read as 'throw to'.) *In Standard Scheme Mode only*, when **=>** occurs as the second expression in a cond-clause, if the first expression evaluates to something other than **#f**, then the value of that expression is passed to the third expression (which must evaluate to a one-input procedure), and the result is returned as the value of the cond-expression. (See the entry later in this section for the special form **cond**.)

(> *x1* ...) *Procedure*

A predicate that returns the boolean **#t** if and only if its arguments are in strictly decreasing order, that is, in the reverse order to the one tested by the predicate **<**. (The value returned by this procedure is only guaranteed to be accurate if all of its arguments are exact numbers. See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*.)

(>= *x1* ...) *Procedure*

A predicate that returns the boolean #t if and only if its arguments are in non-increasing order, that is, in the reverse order to the one tested by the predicate **<=**. (The value returned by this procedure is only guaranteed to be accurate if all of its arguments are exact numbers. See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*.)

'*temp* *Syntactic Abbreviation*

(Read as 'backquote temp'.) Abbreviates the Scheme expression

(quasiquote *temp*).

(See the entry later in this section for the **quasiquote** special form.) Example:

```
'#(a ,(add1 7) ,@(map add1 '(12 -9)) b)  ↦  #(a 8 13 -8 b)
```

(abs *z*) *Procedure*

Returns the magnitude of the argument. If *z* is complex with an exact zero real part or if it is real, then the result has the same exactness as *z*. For all other values of *z*, the result is inexact. (See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*. See also the entry for the procedure **magnitude** later in this section.) Examples:

```
(abs -3/2)   ↦  3/2        (abs 0+4i)  ↦  4
(abs 5-12i)  ↦  13.0
```

(acos *z*) *Procedure*

Returns the principal angle whose cosine is *z*. If you have chosen Radian Mode in the **Language** dialog accessed through the **Preferences** sub-menu in the **File** menu, and

- if *z* is real and in the interval from −1 to 1 (inclusive), the result is an **inexact number in the interval from 0 to π (inclusive);**

- if z has any other real or non-real value, the result is an inexact number that is calculated as described on page 47 in Chapter 5: *Numbers and Numeric Procedures*.

On the other hand, if you have chosen Degree Mode in the Language dialog, and

- **if z is real and in the interval from −1 to 1 (inclusive), the result is an inexact number in the interval from 0 to 180 (inclusive);**
- if z is real and outside the interval from −1 to 1 (inclusive), an error message is generated;
- if z is not real, the result is an inexact complex number that is calculated as described on page 47 in Chapter 5: *Numbers and Numeric Procedures*.

Examples:

```
(acos 1/2)   ↦  60.0          [in Degree Mode]
(acos 0+1i)  ↦  1.570796326794897-0.881373587019543i
                              [in either Angle Mode]
```

(add1 x) *Procedure*

Returns $x + 1$ with the same exactness as x. Examples:

```
(add1 4)     ↦  5
(add1 -5/3)  ↦  -2/3
(add1 4.3)   ↦  5.3
```

(and *exp1* ...) *Special Form*

Evaluates the arguments in order of appearance, until one is found whose value is the boolean `#f`, whereupon `#f` is returned and the remaining arguments are left unevaluated. If no argument has a false value, the value of the final argument is returned. If there are no arguments, then `#t` is returned.

In Schemer's Guide Mode, all the arguments except the last must evaluate to a boolean object. In Standard Scheme Mode, no such restriction is imposed.

Examples:

```
(and)                    ↦  #t
(and (null? '()) (< 2 1))  ↦  #f
(and 1 2 3)              ↦  3     [Standard Scheme Mode only]
```

(angle *z*) *Procedure*

Returns the polar angle of ***z***, when ***z*** is expressed in 'polar form'. The polar angle will always be in the interval from 0 (included) to 2π (not included). For a complex number in polar form, **r@s**, the polar angle is **s**. (See page 43 in Chapter 5: *Numbers and Numeric Procedures* where the polar form of complex numbers is described.) In geometrical terms, this is the angle between the positive real axis and the ray from the origin of the complex plane through the point representing the complex number ***z***, positive angles being measured counterclockwise. The result is always given in radians, irrespective of the Angle Mode you have chosen in the **Language** dialog accessed through the **Preferences** sub-menu in the **File** menu. ***z*** must be non-zero. Examples:

```
(angle -3)                      ↦  3.141592653589793
(angle 3-4i)                    ↦  -0.9272952180016122
(angle (make-polar 3 (* -17/3 pi)))  ↦  1.047197551196598
```

(append *list1* ...) *Procedure*

Returns a list consisting of the data expressions (in order of appearance) in the first input list, followed by those in all succeeding lists (again, in order of appearance). If there are no inputs, returns the empty list. Examples:

```
(append '(a b) '((a b) (c)) '(d))  ↦  (a b (a b) (c) d)
(append '(1 2 3))                  ↦  (1 2 3)
(append)                           ↦  ()
```

(apply *proc list*) *Procedure*
(apply *proc exp1 ... list*)

In the first form, returns the result of evaluating the procedure ***proc*** with the contents of the input ***list*** as its arguments. In the second form, which generalizes the first, the arguments are the contents of the list made up of

all the input expressions *exp1* and so on, in order of appearance, with the contents of the input *list* appended on the end (see the last example below).

Examples:

```
(apply + '(1 2 3 4))                          ↦  10
(apply (lambda (s) (first (rest s))) '((a b c)))  ↦  b
(apply * 1 2 3 '(4 5 6))                      ↦  720
```

(arc *list x1 x2*) *Procedure*
(arc *list x1 x2 gwin*)

Draws the arc of an ellipse specified by ***list*** in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise. The first three inputs are as follows:

- a list containing two pairs of numbers that give the coordinates of two diagonally opposite corners of the bounding rectangle of the ellipse in question (see the entry for **oval** later in this section);
- **the start angle of the arc, where 0 represents due North and the positive** direction for angle measurement is clockwise.
- the angle of turn, positive for clockwise and negative for counterclockwise.

The two angle arguments are interpreted as being measured in radians or degrees, according to the Angle Mode you have chosen in the Language dialog accessed through the Preferences sub-menu in the File menu.

The resulting drawing is contained entirely within the bounding rectangle, no matter what the size of the pen's 'nib' (see the entry for **pen-state** later in this section).

For example, in Degree Mode, **(arc '((0 0) (50 50)) 180 –90)** causes the south-**east quadrant of the circle bounded by the square with corners (0,0), (0,50),** (50,50), (50,0) to be drawn in the most recently active open graphics window.

(arc-paint *list x1 x2*) *Procedure*
(arc-paint *list x1 x2 gwin*)

Draws a filled sector bounded by the arc that would have been drawn if 'arc-paint' were replaced by 'arc' and the two radii from the center of the arc to the ends of the arc. (See the above entry for arc.) The sector is filled in accordance with the current pen state. (See the entry for pen-state later in this section.) If this does not specify a solid fill but you want the boundary of the filled sector to appear as a solid curve, then the pen-state must be changed and the boundary drawn separately.

(asin *z*) *Procedure*

Returns the principal angle whose sine is z. If you have chosen Radian Mode in the Language dialog accessed through the Preferences sub-menu in the File menu, and

- if z is real and in the interval from -1 to 1 (inclusive), the result is an inexact number in the interval from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$ (inclusive);
- if z has any other real or non-real value, the result is an inexact number that is calculated as described on page 47 in Chapter 5: *Numbers and Numeric Procedures*.

On the other hand, if you have chosen Degree Mode in the Language dialog, and

- if z is real and in the interval from -1 to 1 (inclusive), the result is an inexact number in the interval from -90 to 90 (inclusive);
- if z is real and outside the interval from -1 to 1 (inclusive), an error message is generated;
- if z is not real, the result is an inexact complex number that is calculated as described on page 47 in Chapter 5: *Numbers and Numeric Procedures*.

Examples:

(asin 1/2)	↦	`30.0`	[in Degree Mode]
(asin 0+1i)	↦	`0.0+0.881373587019543i`	[in either Angle Mode]

(assoc *exp alist*) *Procedure*

Returns the first pair—in the technical sense explained in Section 10.3: *Pairs and Lists* in Chapter 10: *Data Expressions*—in *alist* whose first data expression is equivalent to *exp*, where the comparison is made using the predicate equal? (see the relevant entry later in this section). If no such pair is found, the boolean #f is returned. Examples:

```
(assoc 4 '((1 a) (2 b)))              ↦  #f
(assoc '(a b) '(((a) (b)) ((a b))))   ↦  ((a b))
```

(assq *exp alist*) *Procedure*

Similar to assoc (see above), except that the comparison is made using the predicate eq? (see the relevant entry later in this section). Examples:

```
(assq '(a b) '(((b a)) ((a b)) ((a))))      ↦  #f

(let ([y '(a b)])
  (let ([x (list '((b a)) (list y) '((a)))])
    (assq y x)))                            ↦  ((a b))
```

(assv *exp alist*) *Procedure*

Similar to assoc (see above), except that the comparison is made using the predicate eqv? (see the relevant entry later in this section). Examples:

```
(assv 2 '((1 a) (2 b) (3 c)))              ↦  (2 b)
(assv '(a b) '(((b a)) ((a b)) ((a))))     ↦  #f
```

(atan *z*) *Procedure*
(atan *x1 x2*)

In the first form, returns the principal angle whose tangent is z. If z is real, the result is an inexact number of radians (in the interval from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$, neither endpoint included) or degrees (in the interval from -90 to 90, neither endpoint included), depending upon which Angle Mode you have chosen in the Language dialog accessed through the Preferences sub-menu

in the File menu. If *z* is not real, the result is an inexact complex number that is calculated as explained on page 47 in Chapter 5: *Numbers and Numeric Procedures*. *z* must not be either +i or −i.

In the second form, returns the size of the angle between the positive *x*-axis and the ray joining the origin to the point (*x2*,*x1*)—note the order of the coordinates—as an inexact number in the interval from $-\pi$ (not included) to π (included), if you are in Radian Mode, or in the interval from −180 (not included) to 180 (included), if you are in Degree Mode. At least one of *x1* and *x2* must be non-zero.

Examples:

(atan –1)	↦	`-45.0`	[in Degree Mode]
(atan 3–4i)	↦	`1.448306995231465-0.1589971916799992i`	[in either Angle Mode]
(atan 3 –4)	↦	`2.498091544796509`	[in Radian Mode]

(atom? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is an atom. Examples:

(atom? 'Eric)	↦	`#t`	**(atom? '())**	↦	`#f`
(atom? "abc")	↦	`#t`	**(atom? #\a)**	↦	`#t`
(atom? '#(a))	↦	`#f`	**(atom? #t)**	↦	`#t`

(back *x*) *Procedure*
(back *x gwin*)

Makes the turtle back up *x* steps in the graphics window *gwin*, if specified, or in the most recently active open graphics window, otherwise. The nature of the track left by the turtle depends upon the current pen state (see the entry for **pen–state** later in this section).

(begin *exp1* ...) *Special Form*
(begin *def1 def2* ... *exp1*)

In the first form, evaluates its arguments in order of appearance, and returns the value of the last one. The principal use of this type of begin-expression is to sequence side effects such as input, output, and assignment.

Example:

```
(begin
  (display "Et tu, Brute!")
  (newline)
  'Farewell)
Et tu, Brute!
                          ↦  Farewell
```

If evaluated at top-level, the second form of begin-expression, in which the keyword is followed by one or more define-expressions—see the entry for the special form **define** later in this section—followed perhaps by some other Scheme expressions, is equivalent to the sequence of the constituent define-expressions followed by a begin-expression (of the first type) whose body is the sequence of any remaining Scheme expressions. For example, at top-level

```
(begin                    is equivalent to the    (define a 2)
  (define a 2)            sequence of Scheme      (define b 3)
  (define b 3)            expressions on the      (begin
  (- a b)                 right:                    (- a b)
  (expt a b))                                       (expt a b))
```

If evaluated in any environment other than top-level, the second form of begin-expression is equivalent to a nested collection of letrec-expressions—see the entry for the special form **letrec** later in this section—the outer expression binding the first define-expression's variable to that expression's body, the next letrec-expression binding the second define-expression's variable to its body, and so on, with the innermost letrec-body being the sequence of the remaining Scheme expressions. For example, when evaluated in any environment other than top-level, the begin-expression in the previous example is equivalent to the following nested letrec-expression:

```
(letrec ([a 2])
  (letrec ([b 3])
    (- a b)
    (expt a b)))
```

If no arguments are provided, an unspecified value is returned.

(bitmap? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is a bitmap.

(bitmap-close) *Procedure*
(bitmap-close *bmap*)

Closes (that is, destroys) the bitmap ***bmap***, if specified, or all bitmaps currently in existence, otherwise.

(bitmap-fetch *list bmap*) *Procedure*
(bitmap-fetch *list bmap gwin*)

Assigns to the bitmap ***bmap*** the image taken from the rectangle determined by the input ***list*** in the graphics window ***gwin***, if specified, or in the most recently active open graphics window, otherwise.

The input ***list*** must be of one of the following two types:

- if it is a list of two numbers, then the rectangle it determines has the point whose coordinate is the input ***list*** as its bottom left corner, and its dimensions are the same as those of the bitmap ***bmap***;
- if it is a list of two 2-number lists, then the rectangle it determines has opposite corners at the points whose coordinates are the two 2-number lists, and the rectangle image is scaled to fit the dimensions of the bitmap ***bmap***.

Examples:

```
;;; Create a 40 by 50 bitmap, B.
(define B (make-bitmap '(40 50)))

;;; Assign to B the image of the 40 by 50 rectangle with
;;; bottom left corner at the point (0,10).
(bitmap-fetch '(0 10) B)

;;; Assign to B the image of the rectangle with opposite corners
;;; at the points (0,0) and (30,30), scaled to measure 40 by 50.
(bitmap-fetch '((0 0) (30 30)) B)
```

(bitmap-mode *k bmap*) *Procedure*

Specifies the 'stamping mode' that will be used when the bitmap ***bmap*** is subsequently supplied as the *only* bitmap input to the procedure **bitmap-stamp** (see below). The number ***k*** must be an exact integer from 0 through 7, and the 'stamping modes' are as follows:

Mode	*Action*	*Mode*	*Action*
0	copy	4	color-inverted copy
1	or	5	color-inverted or
2	xor	6	color-inverted xor
3	and	7	color-inverted and

(bitmap-set-spec *list*) *Procedure*

Returns a newly-created bitmap whose specification is given by the input ***list***. This ***list*** must, of course, be a valid bitmap specification (such as is generated by the procedure **bitmap-spec**—see the next entry). Example:

```
;;; Open a graphics window and draw a circle.
(define W (make-graphics-window '(200 200)))
(oval '((0 0) (20 20)))

;;; Create a 20 by 20 bitmap B, get the circle image, and
;;; assign it to B. Then close the graphics window.
(define B (make-bitmap '(20 20)))
(bitmap-fetch '(0 0) B)
(window-close W)

;;; Save the bitmap to file, then close the bitmap.
(let ((F (open-output-file "My BitMap")))
  (begin
    (display (bitmap-spec B) F)
    (close-port F)))
(bitmap-close B)

;;; Create a new bitmap whose image is the circle just saved.
(define P (open-input-file "My BitMap"))
(define NewB (bitmap-set-spec (read P)))
(close-port P)
```

(bitmap-spec *bmap*) *Procedure*

Returns a list that completely specifies the bitmap ***bmap***. In conjunction with the procedure **bitmap-set-spec** (see above), this procedure provides a means of saving a bitmap to disk for later use.

(bitmap-stamp *list bmap1*) *Procedure*
(bitmap-stamp *list bmap1 gwin*)
(bitmap-stamp *list bmap1 bmap2*)
(bitmap-stamp *list bmap1 bmap2 gwin*)

Stamps the image of bitmap ***bmap1*** in the rectangle determined by the input ***list*** in the graphics window ***gwin***, if specified, or in the most recently active open graphics window, otherwise. The 'stamping mode' used is mode 0 (copy) unless there is only one bitmap input and some other 'stamping mode' has previously been specified using the procedure **bitmap-mode** (see above).

The nature and meaning of the input ***list*** are as explained in the entry for the procedure **bitmap-fetch** above. If the rectangle determined by the input ***list*** does not have the same dimensions as the bitmap ***bmap1***, then the bitmap image is scaled to fit the rectangle.

If a second bitmap input ***bmap2*** is supplied, then it is used as a mask in the sense that only the points where bitmap ***bmap2*** is non-white are stamped (necessarily in mode 0) with the corresponding points of the image of bitmap ***bmap1***.

On systems with Color Quickdraw™ only, the image of bitmap ***bmap2*** being used as a mask in this process is scaled to fit the 'stamp rectangle', if necessary. (See the entry later in this section for the predicate **color-quickdraw?**.)

For example,

```
(bitmap-stamp '((0 0) (30 30)) BitmapA BitmapA)
```

stamps just the non-white points of ***BitmapA*** (scaled if necessary to fit into a 30 by 30 square) onto the 30 by 30 square whose bottom left corner is at the origin of the current graphics window, leaving all the other points of the turtle plane unchanged.

(boolean? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is one of the boolean objects `#t` and `#f`. Examples:

```
(boolean? #f)  ↦  #t
(boolean? 5)   ↦  #f
```

(call/cc *proc*) *Procedure*

Creates an 'escape procedure' that encapsulates the current continuation and passes it to the one-input procedure ***proc***. The procedure thus created takes a single input, and when passed a value at some future time, it discards the continuation then in effect and instead gives the value to the continuation that was in effect when the escape procedure was created. For example, the following procedure returns the sum of the numbers in the input list. As soon as it detects a non-number in the input list, however, it *immediately* returns a suitable error message.

```
(define sum
  (lambda (list)
    (call/cc
      (lambda (return)
        (letrec ([total
                  (lambda (s)
                    (cond
                      [(null? s) 0]
                      [(not (number? (first s)))
                       (return
                         (string-append
                           (expression->string (first s))
                           " is not a number."))]
                      [else (+ (first s) (total (rest s)))]))])
          (total list))))))      ↦  unspecified

(sum '(1 2 3))                   ↦  6
(sum '(1 x 3))                   ↦  "x is not a number."
```

(See the next entry.) For further details concerning the operation of this procedure, see Section 6.9 of the *Revised^4 Report on the Algorithmic Language Scheme*.

(call-with-current-continuation *proc*) *Procedure*

The full, official name for **call/cc**. See the previous entry.

(call-with-input-file *str proc*) *Initialization File Procedure*

Opens an input port to accept input from the file named by the input ***str***, and evaluates the one-input procedure ***proc***, using the input port as the procedure's argument. Note that the input port is closed just before the procedure returns a value. Example:

```
(call-with-input-file "MyFile" string-read)
```

is equivalent to

```
(let ([in-port (open-input-file "MyFile")])
  (let ([result (string-read in-port)])
    (begin
      (close-input-port in-port)
      result)))
```

For information concerning the naming of files, see Chapter 6: *Files and Ports*.

(call-with-output-file *str proc*) *Initialization File Procedure*

Opens an output port ready to send output to the file named by the input ***str***, and evaluates the one-input procedure ***proc***, using the output port as the procedure's argument. Note that the output port is closed just before the procedure returns a value. For further information concerning the naming of files, see Chapter 6: *Files and Ports*.

(car *pair*) *Procedure*

Returns the first component of the input ***pair***. (See the entry later in this section for the procedure **first**.)

The companion procedure to car is cdr (see the relevant entry below). All 28 combinations of these procedures (such as **caadr**) required by the Scheme standard (see Section 6.3 of the *Revised*^{4} *Report on the Algorithmic Language Scheme*) **are also included as primitive *EdScheme* procedures. Examples:**

```
(cddr '(a ((b c) d) e))   ↦   (e)
(caadr '(a ((b c) d) e))  ↦   (b c)
```

(case *sexp clause1 clause2* ...) *Initialization File Special Form*

Each clause must be of one of the following two forms:

- **[(*exp1* ...) *sexp1 sexp2* ...]**
- **[else *sexp1 sexp2* ...]**

(The convention in *EdScheme* is to enclose case-clauses in brackets. Brackets **and parentheses are completely interchangeable, however.)**

Within each clause of a case-expression and within all its clauses taken together, the expressions ***exp1***, ..., must all be distinct. Case-expressions do **not have to include an else-clause, but if one is included, it should be the last** clause. (In any case, it is the last clause that *EdScheme* pays any attention to!)

A case-expression is evaluated as follows: First, the input Scheme expression ***sexp*** is evaluated, producing a value ***v***. Then, taking the clauses in order of **appearance,**

- if the clause is not an else-clause, the value ***v*** is compared (using the **predicate eqv?—see the relevant entry later in this section) with each** expression in the clause's expression list, and if a match is found, the corresponding Scheme expressions are evaluated in order of appearance, **and the case-expression returns the value of the last one; if no match is** found, the process is repeated with the next clause.
- **if the clause is an else-clause, then the corresponding Scheme expressions** are evaluated in order of appearance, and the case-expression returns **the value of the last one.**

If there is no else-clause, and no match is found in the above evaluation process, an error message is generated in Schemer's Guide Mode, and an unspecified value is returned in Standard Scheme Mode.

Example:

```
(case (list-ref '(Adam Burt Cindy Daniel Ellen) 2)
  [(Adam Burt Daniel Fred Ian Mark) 'boy]
  [(Cindy Ellen Lauren Nancy Olivia) 'girl]
  [(Georgie Hilary Jackie Kim) 'ambiguous])      ↦  girl
```

(caution-alert *str*) *Procedure*

Activates a Macintosh caution alert box containing the contents of the input string ***str***. The size of this box is set by ***EdScheme*** and the text of the alert automatically word-wraps to fit inside the box.

(cdr *pair*) *Procedure*

Returns the second component of the input ***pair***. (See the entry for the procedure **rest** later in this section. Also see the entry above for the procedure **car** for information concerning the various combinations of **car** and **cdr**.)

(ceiling *x*) *Procedure*

Returns the smallest integer greater than or equal to ***x***, the exactness of the result being the same as that of ***x***. (See Section 5.9: ***Exactness*** in Chapter 5: *Numbers and Numeric Procedures.*) Examples:

```
(ceiling 12/5)  ↦  3
(ceiling -3.2)  ↦  -3.0
```

(char? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is a character. Examples:

```
(char? #\a)  ↦  #t      (char? 'a)                    ↦  #f
(char? "a")  ↦  #f      (char? '#(a))                ↦  #f
(char? '(a))  ↦  #f     (char? (string-ref "abc" 1))  ↦  #t
```

(char<? *ch1* ...) *Procedure*

A predicate that returns the boolean #t if and only if its arguments are in strictly increasing ASCII code order. Examples:

```
(char<? #\B #\a #\z)  ↦  #t
(char<? #\a #\a)      ↦  #f
(char<? #\a #\A)      ↦  #f
(char<? #\A)          ↦  #t
(char<?)              ↦  #t
```

(char<=? *ch1* ...) *Procedure*

A predicate that returns the boolean #t if and only if its arguments are in non-strict increasing ASCII code order. Examples:

```
(char<=? #\B #\B #\a)  ↦  #t
(char<=? #\a #\a)      ↦  #t
(char<=? #\a #\A)      ↦  #f
(char<=? #\a)          ↦  #t
(char<=?)              ↦  #t
```

(char=? *ch1* ...) *Procedure*

A predicate that returns the boolean #t if and only if all its arguments are the same character. Examples:

```
(char=? #\a #\A #\a)  ↦  #f
(char=? #\A #\A)      ↦  #t
(char=? #\a)          ↦  #t
(char=?)              ↦  #t
```

(char>? *ch1* ...) *Procedure*

A predicate that returns the boolean #t if and only if its arguments are in strictly decreasing ASCII code order. Examples:

```
(char>? #\B #\a)      ↦  #f
(char>? #\a #\a)      ↦  #f
(char>? #\a #\Z #\A)  ↦  #t
```

```
(char>? #\a)  ↦  #t
(char>?)      ↦  #t
```

(char>=? *ch1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in non-strict decreasing ASCII code order. Examples:

```
(char>=? #\B #\a)       ↦  #f
(char>=? #\a #\a #\Z)   ↦  #t
(char>=? #\a #\A)       ↦  #t
(char>=? #\a)           ↦  #t
(char>=?)               ↦  #t
```

(char-alphabetic? *ch*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is an alphabetic character. Examples:

```
(char-alphabetic? #\A)      ↦  #t
(char-alphabetic? #\space)  ↦  #f
(char-alphabetic? #\5)      ↦  #f
(char-alphabetic? #\*)      ↦  #f
```

(char-ci<? *ch1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in strictly increasing ASCII code order when the distinction between upper- and lower-case is ignored. Examples:

```
(char-ci<? #\B #\a)       ↦  #f
(char-ci<? #\a #\a)       ↦  #f
(char-ci<? #\a #\A)       ↦  #f
(char-ci<? #\a #\B #\z)   ↦  #t
(char-ci<? #\a)           ↦  #t
(char-ci<?)               ↦  #t
```

(char-ci<=? *ch1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in non-strict increasing ASCII code order, when the distinction between upper- and lower-case is ignored. Examples:

```
(char-ci<=? #\B #\a)          ↦  #f
(char-ci<=? #\a #\a)          ↦  #t
(char-ci<=? #\a #\A #\z)      ↦  #t
(char-ci<=? #\a #\B)          ↦  #t
(char-ci<=? #\a)              ↦  #t
(char-ci<=?)                  ↦  #t
```

(char-ci=? *ch1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are the same character when the distinction between upper- and lower-case is ignored. Examples:

```
(char-ci=? #\B #\a #\b)       ↦  #f
(char-ci=? #\a #\a)           ↦  #t
(char-ci=? #\a #\A)           ↦  #t
(char-ci=? #\a)               ↦  #t
(char-ci=?)                   ↦  #t
```

(char-ci>? *ch1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in strictly decreasing ASCII code order, when the distinction between upper- and lower-case is ignored. Examples:

```
(char-ci>? #\Z #\B #\a)       ↦  #t
(char-ci>? #\a #\a)           ↦  #f
(char-ci>? #\a #\A)           ↦  #f
(char-ci>? #\a)               ↦  #t
(char-ci>?)                   ↦  #t
```

(char-ci>=? *ch1* ...) *Procedure*

A predicate that returns the boolean **#t** if and only if its arguments are in non-strict decreasing ASCII code order, when the distinction between upper- and lower-case is ignored. Examples:

```
(char-ci>=? #\Z #\B #\a)  ↦  #t
(char-ci>=? #\a #\a)      ↦  #t
(char-ci>=? #\a #\A)      ↦  #t
(char-ci>=? #\a #\B)      ↦  #f
(char-ci>=? #\a)          ↦  #t
(char-ci>=?)              ↦  #t
```

(char-downcase *ch*) *Procedure*

Returns a character ***ch1*** such that **(char-ci=? *ch ch1*)** returns the boolean **#t**. Furthermore, if ***ch*** is alphabetic, the result is lower-case. Examples:

```
(char-downcase #\*)  ↦  #\*
(char-downcase #\a)  ↦  #\a
(char-downcase #\A)  ↦  #\a
```

(char->integer *ch*) *Procedure*

Returns the ASCII code for the character ***ch*** as an exact integer. Example:

```
(char->integer #\a)  ↦  97
```

In Schemer's Guide Mode, **char->integer** also accepts symbols. Example:

```
(char->integer 'Q)  ↦  81
```

(char-lower-case? *ch*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is a lower-case alphabetic character. Examples:

```
(char-lower-case? #\a)          ↦  #t
(char-lower-case? #\A)          ↦  #f
(char-lower-case? #\5)          ↦  #f
(char-lower-case? #\newline)    ↦  #f
```

(char-numeric? *ch*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a numeric character. Examples:

```
(char-numeric? #\a)          ↦  #f
(char-numeric? #\A)          ↦  #f
(char-numeric? #\5)          ↦  #t
(char-numeric? #\newline)    ↦  #f
```

(char-ready?) *Procedure*
(char-ready? *port*)

A predicate that returns the boolean `#t` if and only if a character is waiting on the input ***port***, if specified, or the current input port, otherwise. (See the entry for the procedure **current-input-port** below.) If ***port*** is not a serial port and is at end-of-file, then **char-ready?** returns the boolean `#t`.

Example (assuming that the current input port is the Transcript Window):

```
(if (char-ready?) (read-char) '(no key on))
;;; If you have pressed a key, then the character corresponding
;;; to that key is returned. Otherwise, the list (no key on)
;;; is returned.
```

If ***port*** is a serial port, then this predicate returns the boolean `#t` if and only if one or more characters are waiting in the input buffer.

(char-upcase *ch*) *Procedure*

Returns a character ***ch1*** such that **(char-ci=? *ch ch1*)** returns the boolean `#t`. Furthermore, if ***ch*** is alphabetic, the result is upper-case. Examples:

```
(char-upcase #\*)  ↦  #\*        (char-upcase #\a)  ↦  #\A
(char-upcase #\A)  ↦  #\A
```

(char-upper-case? *ch*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is an upper-case alphabetic character. Examples:

```
(char-upper-case? #\a)       ↦  #f
(char-upper-case? #\A)       ↦  #t
(char-upper-case? #\5)       ↦  #f
(char-upper-case? #\newline) ↦  #f
```

(char-whitespace? *ch*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a 'whitespace' character, that is, either a space, a tab, a line feed, a form feed, or a carriage return. Examples:

```
(char-whitespace? #\a)       ↦  #f
(char-whitespace? #\A)       ↦  #f
(char-whitespace? #\5)       ↦  #f
(char-whitespace? #\newline) ↦  #t
```

(choose-input-file) *Procedure*

Brings up the Macintosh File Selector dialog, allowing you to select a file from which to accept input. Returns the full file specification of the selected file. For example,

```
(define F (open-input-file (choose-input-file)))
```

defines *F* to be the port through which input from a user-selected file may be read.

(choose-output-file) *Procedure*

Brings up the Macintosh File Selector dialog, allowing you to select a file to which output may be written. Returns the full file specification of the selected file.

(clean) *Procedure*
(clean *gwin*)

Clears the graphics window ***gwin***, if specified, or the most recently active open graphics window, otherwise.

clipboard *Variable*

A global variable bound to (the representation of) the clipboard window.

(clipboard-set-text *str*) *Procedure*

Replaces the contents of the clipboard by the contents of the input ***str*** and updates the Clipboard Window, if visible.

(clipboard-text) *Procedure*

Returns the contents of the clipboard (in the form of a string).

(close-input-port) *Procedure*
(close-input-port *port*)

Closes the file associated with ***port***, if specified, or all files that are open for input, otherwise, thus making it impossible for the associated port(s) to receive any further data from the file(s) in question. If the file(s) is/are already closed, the procedure has no effect. The output from this procedure is unspecified.

(close-output-port) *Procedure*
(close-output-port *port*)

Closes the file associated with ***port***, if specified, or all files that are open for output, otherwise, thus making it impossible for the associated port(s) to send any further data to the file(s) in question. If the file(s) is/are already closed, the procedure has no effect. The output from this procedure is unspecified.

(close–port) *Procedure*
(close–port *port*)

Closes the file associated with ***port***, if specified, or all open files, otherwise, thus making it impossible for the associated port(s) either to send any further data to the file(s) in question or to receive any further data from it/them. If no files are open, the procedure has no effect. The output from this procedure is unspecified.

(color–quickdraw?) *Procedure*
(colour–quickdraw?)

A predicate that returns the boolean `#t` if and only if the Macintosh system on which ***EdScheme*** is mounted has Color QuickDraw™.

(complex? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a complex number, that is, if and only if it is a number. Examples:

```
(complex? 3-4i)   ↦  #t
(complex? -3/4)   ↦  #t
(complex? 'pvc)   ↦  #f
```

(cond *clause1 clause2* ...) *Special Form*

In Standard Scheme Mode, each clause must be of one of these three types:

- **[<*test*> *sexp1* ...]**
- **[else *sexp1 sexp2* ...]**
- **[<*test*> => *sexp*]**

(The convention in ***EdScheme*** is to enclose cond-clauses in brackets. Brackets and parentheses are completely interchangeable, however.)

The second type of clause is known as 'an else-clause'. A cond-expression **does not have to include an else-clause; but if it does, the else-clause should**

be the final clause. (In any case, it is the last clause that ***EdScheme*** pays any attention to!) In the third type of clause, ***sexp*** must evaluate to a one-input procedure. The **<*test*>s** may be Scheme expressions of any kind.

A cond-expression is evaluated by evaluating the **<*test*>s** in order of appearance of the associated clauses until one is found that does *not* evaluate to the boolean **#f**, or until an else-clause is reached.

If the clause thus located is of the first type, and

- if there are *no* Scheme expressions following the **<*test*>**, then the cond-expression returns the value of **<*test*>**;
- otherwise, the Scheme expressions ***sexp1***, ***sexp2***, ... are evaluated in order of appearance, and the cond-expression returns the value of the last one.

If the clause thus located is an else-clause, the Scheme expressions ***sexp1***, ***sexp2***, ... are evaluated in order of appearance, and the cond-expression returns the value of the last one.

If the clause thus located is of the third type, then the cond-expression returns the result of passing the value of **<*test*>** to the one-input procedure that is the value of ***sexp***.

The output when there is no else-clause and all the **<*test*>s** evaluate to the boolean **#f** is unspecified.

Example:

```
(define lookup
  (lambda (exp alist)
    (cond
      [(assoc exp alist) => (lambda (x) (first (rest x)))]
      [else #f])))                    ↦  unspecified
(lookup 'b '((a 1) (b 2) (c 3)))      ↦  2
```

In Standard Scheme Mode only, ***EdScheme*** also accepts a cond-expression that has no clauses. Such an expression returns an unspecified value.

In Schemer's Guide Mode, there are only two acceptable types of clause, namely:

- **[<*test*> *sexp*]**
- **[else *sexp*]**

Each **<*test*>** must be a Scheme expression that evaluates to a boolean object. The manner in which a cond-expression is evaluated is the same as in Standard Scheme Mode—suitably simplified of course to take into account the restricted kinds of acceptable clauses. Furthermore, if there is no else-clause and no **<*test*>** evaluates to the boolean **#t** an error message is generated.

Example (valid in either Language Mode):

```
(cond
  [(null? '(a b)) 1]
  [(zero? 0) 2]
  [else 3])          ↦  2
```

(configure-serial-port *n port*) *Procedure*

Configures the serial port in accordance with the integer *n*. To calculate the value of *n* that corresponds to the configuration you need, add together the code numbers of the desired settings:

Baud Rate	*Code*
300	380
600	189
1200	94
1800	62
2400	46
3600	30
4800	22
7200	14
9600	10
19200	4
57600	0

Stop Bits	*Code*
1	16384
1.5	-32768
2	-16384

Data Bits	*Code*
5	0
6	2048
7	1024
8	3072

Parity	*Code*
None	0
Odd	4096
Even	12288

The serial port must have been opened using the procedure **open-serial-port** (see the relevant entry later in this section as well as Chapter 6: *Files and*

Ports, in which certain restrictions are described concerning the baud rates that may be used with which serial port). Note that the driver configured by this procedure is adequate for communication with simple devices such as Hyperbot™, but in general not for more sophisticated operations such as telecommunications using error-checking protocols.

Example:

```
(define sPort (open-serial-port 'modem))
(configure-serial-port -13302 sPort)

;;; Configures the modem serial port to 9600 baud (code 10),
;;; 8 databits (code 3072), 2 stopbits (code -16384), and
;;; no parity (code 0): 10 + 3072 - 16384 + 0 = -13302.
;;; (In fact, this is the default setting.)
```

(cons *exp1 exp2*) *Procedure*

Returns a pair whose first component is ***exp1*** and whose second component is ***exp2***. If the predicate **eqv?** is used to compare the resulting pair with any Scheme object created earlier in the same ***EdScheme*** session, the result is guaranteed to be **#f**. In Schemer's Guide Mode, ***exp2*** must be a list. Examples:

```
(cons 'a '(b))       ↦  (a b)
(cons '(a) '(a ()))  ↦  ((a) a ())
(cons '(a) 'b)       ↦  ((a) . b)   [Standard Scheme Mode only]
```

(cons-stream *exp sexp*) *Special Form*

Returns a stream whose first component is ***exp*** and whose second component is a 'promise' that, when called by using the procedure **tail** (see the relevant entry later in this section), returns the value of ***sexp***. (See Section 10.11: *Streams and Delayed Objects* in Chapter 10: *Data Expressions* for further information concerning streams.)

The Scheme expression **(cons-stream *a b*)** is equivalent to the Scheme expression **(cons *a* (freeze *b*))**. Examples:

```
(define S (cons-stream 1 2))  ↦  unspecified
(head S)                      ↦  1
(tail S)                      ↦  2
```

```
(define wholes
  (lambda (n)
    (cons-stream n (wholes (add1 n)))))  ↦  unspecified
(define W (wholes 0))                    ↦  unspecified
(head (tail (tail (tail W))))            ↦  3
```

(continuation? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is a continuation. Example:

```
(continuation? (call/cc (lambda (x) x)))  ↦  #t
```

(cos *z*) *Procedure*

Returns the cosine of *z* as an inexact number. If *z* is real, it is interpreted as being in radians or degrees, according to the Angle Mode you have chosen in the Language dialog from the Preferences sub-menu. If *z* is not real, then its cosine is calculated in the manner explained on page 46 in Chapter 5: *Numbers and Numeric Procedures*. Examples:

```
(cos (/ pi 3))  ↦  0.5                          [in Radian Mode]
(cos 90)        ↦  0.0                          [in Degree Mode]
(cos 3-4i)      ↦  -27.03494560307422+3.851153334811778i
                                                [in either Angle Mode]
```

(current-input-port) *Procedure*

Returns the current default input port, that is, the port from which data will be read if the Scheme expression **(read)** is evaluated. (See the entry for the procedure **read** later in this section.)

(current-output-port) *Procedure*

Returns the current default output port, that is, the port to which data will be written if a Scheme expression of the form **(display *exp*)** is evaluated. (See the entry for the procedure **display** later in this section.)

(cursor *k*) *Procedure*

Sets the form of the cursor according to the value of ***k***, as follows:

0: pointer 1: watch 2: I-beam

(The effect of this procedure may be rather short-lived since moving the mouse out of the window it is currently in will automatically cause its form to change.)

(define *var sexp*)	*Special Form*
(define (*var fp1* ...) *sexp*)	
(define (*var fp1 fp2* ... . *fpk*) *sexp*)	[Standard Scheme Mode only]
(define (*var* . *fp*) *sexp*)	[Standard Scheme Mode only]

Establishes a binding in the current environment.

Examples of the first type of define-expression:

```
(define e (exp 1))                          ↦  unspecified
e                                           ↦  2.718281828459045

(define second (lambda (s) (first (rest s)))) ↦  unspecified
(second '(a b c))                           ↦  b
```

The second type of define-expression, in which ***var*** is followed by zero or more formal parameters is equivalent to a define-expression of the first type, as follows:

(define *var* (lambda (*fp1* ...) *sexp*))

Example:

```
(define (second s) (first (rest s)))  ↦  unspecified
(second '(a b c))                     ↦  b
```

The final two types of define-expressions are not available in Schemer's Guide Mode. The third type, in which ***var*** is followed by at least one formal parameter, followed by a space-delimited period, followed by one more formal parameter, is equivalent to a define-expression of the first type, as follows:

(define *var* (lambda (*fp1 fp2 ... . fpk*) *sexp*))

Example:

```
(define (allbut2 x y . z) z)  ↦  unspecified
(allbut2 'a 'b 'c 'd 'e)      ↦  (c d e)
```

The final type of define-expression, in which ***var*** is followed by a space-delimited period and then by a formal parameter, is equivalent to a define-expression of the first type, as follows:

(define *var* (lambda *fp sexp*))

Example:

```
(define (third . s) (first (rest (rest s))))  ↦  unspecified
(third 'a 'b 'c 'd 'e)                        ↦  c
```

For further information concerning the various kinds of lambda-expressions involved in the foregoing examples, see the entry for the special form **lambda** later in this section.

(define-alias *id1 id2*) *Special Form*

Defines ***id1*** to be an identifier whose behavior is identical to that of the identifier ***id2***. Examples:

```
(define-alias make-procedure lambda)  ↦  unspecified
((make-procedure (x) (* x x x)) 5)    ↦  125

(define-alias otherwise else)         ↦  unspecified
(cond
  [(null? '(a b)) 1]
  [otherwise 2])                      ↦  2
```

Aliases must be defined *prior* to the definition of any procedure that uses them. In practice, therefore, it is wise to enter them at the start of an ***EdScheme*** session or place their definitions at the beginning of a Scheme file.

(define-macro (*id fp1 ...*) *sexp*) *Special Form*
(define-macro *id fp sexp*)

Establishes ***id*** as a macro, each use of which is evaluated by (if necessary) expanding ***sexp*** as explained below and then evaluating the resulting Scheme expression.

In the first form, the macro thus defined takes as many arguments (zero or more) as there are formal parameters following the identifier *id* in the define-macro-expression.

If the define-macro-expression involves no formal parameters, no expansion of ***sexp*** is required, and the macro use is evaluated simply by evaluating ***sexp***. Example:

```
(define-macro (patience)
  (letrec ([loop (lambda (n)
                   (if (= n 10)
                     (string->symbol "What's keeping you?")
                     (begin
                       (display "I'm waiting.")
                       (newline)
                       (loop (add1 n))))))])
    (loop 0)))                                    ↦  unspecified
```

The macro use **(patience)** then causes `"I'm waiting."` to be displayed on the screen ten times, each on a new line, before returning the symbol

```
What's keeping you?
```

Note that this could just as effectively have been achieved by defining **patience** as a thunk (that is, a procedure of no arguments); there is no real *need* in this case for the power of macro definition.

Otherwise, ***sexp*** is expanded by replacing each occurrence in ***sexp*** of the first formal parameter, ***fp1***, by the *unevaluated* first argument in the invocation of the macro, replacing each occurrence of the second formal parameter, ***fp2***, if such there be, by the *unevaluated* second argument in the invocation, and so on until all occurrences of formal parameters in ***sexp*** have been replaced.

Examples:

```
;;; The special form cons-stream could be defined as a macro
;;; as follows:
(define-macro (cons-stream exp1 exp2)
  (cons exp1 (lambda ( ) exp2)))

;;; The while construct may be implemented using the
;;; following macro:
(define-macro (while criterion? action)
  (letrec ([loop
            (lambda ( )
              (if criterion?
                  (begin action (loop))))])
    (loop)))

;;; Then, for example,
(while
  (not (char-ready?))
  (begin
    (display "I'm waiting.")
    (newline)))

;;; causes "I'm waiting." to be displayed repeatedly on the
;;; screen until such time as a key is pressed, whereupon an
;;; unspecified value is returned.
```

Note that neither **cons-stream** nor **while** can be defined as procedures.

In the second form, the expansion of ***sexp*** is achieved by first forming a new Scheme expression **F** by deleting the occurrence of ***id*** from the invoking macro-expression, and then replacing each occurrence in ***sexp*** of the formal parameter ***fp*** by the ***unevaluated*** expression **F**. Example:

```
(define-macro special s (length s))

;;; Then we have:

;;; Use:        (special first '((a b) c))
;;; Expansion:  (length (first '((a b) c)))
;;; Value:      2
```

```
;;; Use:        (special (lambda (x) (list x x x)) '(a b))
;;; Expansion:  (length ((lambda (x) (list x x x)) '(a b)))
;;; Value:      3
```

Macros must be defined *prior* to the definition of any procedure that uses them. In practice, therefore, it is wise to enter them at the start of an ***EdScheme*** session or place their definitions at the beginning of a Scheme file.

(define-record *var* (*field1* ...)) *Initialization File Special Form*

When evaluated at top level, defines ***var*** to be a record object whose fields are ***field1***, and so on. (Records are represented externally in ***EdScheme*** as 'tagged' vectors.) In this process, the following procedures are created:

- **make-*var***, which inputs the data expressions that are to fill the fields, and returns a record of type ***var***.
- ***var*?**, a predicate that returns the boolean **#t** if and only if its argument is a record of type ***var***.
- ***var*->*field1***, and so on (one such procedure for each of the fields of the record); each of these procedures inputs a record of type ***var*** and returns the value in the corresponding field of the input record.

Procedures may use the special form **variant-case** to dispatch on a record type (see the entry for **variant-case** later in this section).

The following example implements a binary tree data structure:

```
(define-record leaf (number))
;;; Defines the procedures make-leaf, leaf?, and leaf->number.

(define-record tree (number left-tree right-tree))
;;; Defines the procedures make-tree, tree?, tree->number,
;;; tree->left-tree, and tree->right-tree.

(define tree-a
  (make-tree 1 (make-leaf 2) (make-leaf 3)))  ↦  unspecified
(tree? tree-a)                                ↦  #t
(tree->number tree-a)                         ↦  1
(tree? (tree->left-tree tree-a))              ↦  #f
```

```
(leaf? (tree->left-tree tree-a))          ↦  #t
(leaf->number (tree->left-tree tree-a))   ↦  2
```

(define-transformer *id lam*) *Special Form*

Establishes ***id*** as a transformer, each invocation of which is evaluated by transforming the invocation into a Scheme expression constructed according to the prescription provided by the input ***lam***, and then evaluating that Scheme expression. The input ***lam*** may be any valid lambda-expression, of which there are three types in Standard Scheme Mode, or two in Schemer's Guide Mode (see the entry for **lambda** later in this section for further details).

Calling on the black/red, unevaluated/evaluated metaphor used in *The Schemer's Guide*, the transformed Scheme expression may be 'built' as follows:

1) Define ***id*** as a procedure by replacing **define-transformer** in the transformer definition by **define**.

2) Evaluate the transformer invocation as a procedure application whose arguments are the 'quoted' formal parameters of the transformer being defined. This produces a *red* data expression.

3) Paint the *red* expression *black*. The resulting *black* expression is the desired transformed Scheme expression.

Examples:

```
;;; The special form cons-stream may be defined as a
;;; transformer, as follows:
(define-transformer cons-stream
  (lambda (exp1 exp2)
    (list 'cons exp1 (list 'lambda '() exp2))))

;;; A procedure can be applied to multiple sets of arguments at
;;; once using the transformer multi-apply, defined as follows:
(define-transformer multi-apply
   (lambda s
    (list 'map
        (list 'lambda '(x)
            (list 'apply (first s) 'x))
        (cons 'list (rest s)))))
```

```
;;; [Note that the procedure map is provided in the
;;; EdScheme initialization folder 'Scheme Init Files'.]
(multi-apply + '(1 2) '(3 4 5) '(6 7 8 9))  ↦  (3 12 30)
(multi-apply first '((a b)) '((c d)))       ↦  (a c)
(multi-apply
  (lambda (x) (/ x))
    '(2) '(-5/7) '(1+i))                    ↦  (1/2 -7/5 1/2-1/2i)
```

Transformers must be defined *prior* to the definition of any procedure that uses them. In practice, therefore, it is wise to enter them at the start of an *EdScheme* session or place their definitions at the beginning of a Scheme file.

(delay *sexp*) *Initialization File Special Form*

Returns an object called a 'promise', which may subsequently be asked—using the procedure **force**—to evaluate the Scheme expression ***sexp***, and return its value. (See the entry for **force** later in this section.) The value of the promise is 'memoized' so that, if it is forced more than once, then the value computed in response to the first 'forcing' is returned by all subsequent 'forcings'.

The **delay/force** combination allows the implementation of 'lazy evaluation' (otherwise known as 'delayed evaluation' or 'call by need'). A non-memoized version of **delay** is also available (see the entry for the special form **freeze** later in this section).

(delete! *exp list*) *Procedure*

Returns ***list*** with all occurrences of the data expression ***exp*** removed. (Comparisons of ***exp*** with the entries in ***list*** are performed using the predicate **equal?**—see the relevant entry later in this section.)

This procedure has destructive side-effects that may permanently modify ***list***, so special precautions are required if the original value of ***list*** will be needed later. Furthermore, to be sure to 'capture' a value returned by **delete!**, that value should explicitly be bound to a variable. Examples:

```
(delete! 'a '(a b a c))          ↦  (b c)
(delete! '#(a) '(a #(a) #\a))    ↦  (a #\a)
(delete! 4 '(3  4.0  6/3))       ↦  (3 4.0 2)
```

(denominator *q*) *Procedure*

Returns the denominator of the rational number *q* as an exact positive integer, after having reduced *q* to lowest terms. The denominator of any integer, whether exact or inexact, is 1. (Recall from Chapter 5: *Numbers and Numeric Procedures* that in ***EdScheme*** no non-integer expressed in decimal or exponential form is rational.) Example:

```
(denominator (/ 15 -65)) ↦ 13
```

(derived? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a derived procedure in the current ***EdScheme*** session. Examples:

```
(derived? first)              ↦ #f
(derived? (lambda (x) x))     ↦ #t
```

(desktop) *Procedure*

Returns a pair of number pairs that give the global screen coordinates of the top left and bottom right corners of the desktop, that is, the portion of the screen below the main menu bar.

(display *exp*) *Procedure*
(display *exp port*)
(display *exp gtxwin*)

Prints *exp* to the specified port or window, if provided, or to the current output port, otherwise. (See the entry for **current-output-port** above.) Returns an unspecified value.

Note that slashification and the double-quotes delimiting strings are both suppressed by this procedure. So, if you subsequently use the procedure **read** (see the relevant entry later in this section) to read data that have been written out using the procedure **display**, any space characters belonging to **display**ed strings—whether they are slashified or not—will be interpreted as delimiters between data expressions. To avoid this behavior, use one of the

procedures **string-write**, **write-char**, or **write** (see the relevant entries later in this section).

display also suppresses the **#** combination that identifies characters, displaying only the actual character itself. Similarly, it displays named characters themselves rather than their names. Thus,

```
(display #\space)
```

displays an actual space character—not the symbol **#\space**—on the current output port.

To send more than one character to a serial port using a single display-expression, display the string whose contents are the characters you wish to send, in the desired order.

(do ([*var1 init-val1 update-var1*] ...)
(<*test*> *sexp1* ...)
***sexp* ...)** *Initialization File Special Form*

(The convention in ***EdScheme*** is to use brackets to delineate the variable initialization and update lists that appear in do-expressions. Brackets and parentheses are completely interchangeable, however.)

This special form provides a versatile means of carrying out iterations. It gives the initial values of a collection of variables, specifies how they are to be updated between iterations, and lists what has to be done during each iteration. Finally, there is a terminating condition which, when met, triggers a sequence of instructions, the last of which returns the value of the do-expression.

The presence of the third, 'update' entry in any or all of the variable lists is optional. If it is omitted, the list in question is interpreted as if it had a third entry equal to its first entry. (See the comment on the second example below.)

A do-expression is evaluated as follows:

- The ***init-val***s are evaluated in some unspecified order, and the values assigned to the new local variables, ***var1***, and so on. The iteration then begins.

- **At the start of each iteration, <*test*> is evaluated.**
  - If the value is **#f**, then any final Scheme expression(s) is/are evaluated, for effect, in order of appearance, starting with ***sexp***; the ***update–var*** expressions are evaluated in some unspecified order, and the values assigned to yet more new local variables, ***var1***, and so on, and the next iteration begins.
  - If the value is not **#f**, then the Scheme expressions ***sexp1***, and so on—if any—are evaluated for effect in order of appearance, and the value of the last one is returned as the value of the do-expression. (If there are no Scheme expressions following <*test*>, the value of the do-expression is unspecified.)

Examples:

```
(define factorial
  (lambda (k)
    (do ([n k (- n 1)]
         [product 1 (* product n)])
        ((zero? n) product))))          ↦  unspecified

(factorial 6)                           ↦  720

(do ([a '(0 1 2 3 4) (rest a)]
     [b '()])                  ; this is interpreted as [b '() b]
    ((null? a) b)
  (let ([next (first a)])
    (if (even? next)
        (set! b (cons next b)))))       ↦  (4 2 0)
```

(draw–point *pair*) *Procedure*
(draw–point *pair gwin*)

Plots the point with coordinate ***pair*** in the graphics window ***gwin***, if specified, or the most recently active open graphics window, otherwise, using the current pen color. Returns an unspecified value.

(EdScheme–volume) *Procedure*

Returns the reference number of the volume that was current when *EdScheme*

was started at the beginning of the current session. By including the expression **(set-volume (EdScheme-volume))** just before the end of a program you can ensure that, once execution of the program is complete, the current volume is returned to the value it had when ***EdScheme*** was started.

---

else *Keyword*

Signals the else-clause of a case-expression or a cond-expression. (See the entries above for **case** and **cond**.)

(empty-stream? ***exp*****)** *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is ***the-empty-stream*** (see the relevant entry later in this section).

(eof? ***port*****)** *Procedure*

A predicate that returns the boolean **#t** if and only if ***port*** is at end-of-file. Example:

```
(define copy-file
  (lambda (in-port out-port)
    (if (eof? in-port)
        (begin (close-port) 'done)
        (begin
          (display (read-line in-port) out-port)
          (newline out-port)
          (copy-file in-port out-port)))))   ↦   unspecified

(copy-file
  (open-input-file "MyFile")
  (open-output-file "MyBackUpFile"))          ↦   done

;;; Copies the contents of the file 'MyFile' to the
;;; file 'MyBackUpFile'.
```

(eof-object? ***exp*****)** *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is the end-of-file object (which prints as **#!eof**).

(eq? *exp1 exp2*) *Procedure*

In Standard Scheme Mode, a predicate that returns the boolean **#t** if and only if its arguments are of the same type (symbols, boolean objects, the empty list, pairs, non-empty strings, non-empty vectors, and procedures) and share the same memory location. Its behavior when the arguments are both numbers, both empty strings, or both empty vectors is unspecified. Examples:

```
(eq? (list 'a) (list 'a))  ↦  #f
(let ([x '(a)])
  (eq? x x))               ↦  #t
```

In Schemer's Guide Mode, a predicate that returns the boolean **#t** if and only if its arguments are atoms that print the same. If either or both of the input expressions are lists or vectors, an error message is generated. In this mode, when both its arguments are numbers, **eq?** returns **#t** if and only if the arguments are the same number.

(equal? *exp1 exp2*) *Procedure*

A predicate that returns the boolean **#t** if and only if either

- both arguments are the ***user-global-environment***, or
- both arguments are the ***user-initial-environment***, or
- both arguments print the same, neither of them being the ***user-global-environment*** or the ***user-initial-environment***.

(See the entries for the variables ***user-global-environment*** and ***user-initial-environment*** later in this section.) Examples:

```
(equal? (list 'a) '(a))  ↦  #t
(equal? 2 (sqrt 4))      ↦  #f
```

The reason why the second example returns **#f** is that the procedure **sqrt** always returns an inexact number, and an inexact 2 prints as **2.0** rather than 2. (See the entry for the procedure **sqrt** later in this section, and refer to **Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*** for further information concerning the exactness of numbers.)

(eqv? *exp1 exp2*) *Procedure*

In Standard Scheme Mode, a predicate that returns the boolean **#t** if and only if either

- both arguments are numbers that have the same exactness and are numerically equal according to the predicate =, or
- neither argument is a number, and the arguments are equivalent according to the predicate **eq?**.

(See the entries above for the predicates = and **eq?**, and refer to Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures* for further information concerning the exactness of numbers.) Examples:

```
(eqv? 2 6/3)          ↦  #t      (eqv? 2 2.0)  ↦  #f
(eqv? (list 'a) '(a))  ↦  #f
```

In Schemer's Guide Mode, **eqv?** is indistinguishable in its behavior from the predicate **eq?** (see the 'Schemer's Guide' section of the entry above).

(error *exp sexp1* ...) *Special Form*

Generates an error message. The kind of error is given as ***exp***, and the values of any subsequent Scheme expressions are then listed, thereby providing a means to determine what the source of the error might be. By including **(the-environment)** among these Scheme expressions, all the current values of the active local variables will be displayed. (See the entry for the procedure **the-environment** later in this section.) Example:

```
(define test
  (lambda (x y)
    (if (not (and (number? x) (number? y)))
        (error "Wrong kind of input" (the-environment))
        (< x y)))) ↦  unspecified

(test 5 2)          ↦  #f
(test 'one 3)
```

An error has occurred while evaluating the procedure **test***:*
Error: "Wrong kind of input"
—(((x one) (y 3)))

(eval *exp*) *Procedure*
(eval *exp env*)

Evaluates the expression ***exp*** in the environment ***env***, if specified, or in the current environment, otherwise. Examples:

```
(eval '(first '(a b c))
      (the-environment))      ↦  a
(eval 'b (make-environment
           (define a 1)
           (define b 2)
           (define c 3)))     ↦  2
(let ([x 2] [y 3])
  (eval '(* x y)))            ↦  6
```

See later in this section for entries relating to the procedures **make-environment** and **the-environment**.

(even? *n*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument (which must be an integer) is even. Examples:

```
(even? -4)   ↦  #t      (even? 5e2)  ↦  #t
(even? 6/2)  ↦  #f
```

(event) *Procedure*

Returns the specifications of the next event (mouse click in a graphics window, keypress, or menu selection) in the form of a list of five data expressions, as follows:

1) a symbol specifying the nature of the event, either **key**, **click**, or **menu**.

2) an integer giving the time in 'ticks' since system start-up when the event occurred.

3) a pair of integers giving the location of the mouse at the time when the event occurred. (In the case of a keypress or a menu selection, this is in screen coordinates; in the case of a mouse click in a graphics window, it is in turtle coordinates relative to the current graphics origin of that window.)

4) a list of five booleans giving the states of the OPTION key, the CAPS LOCK key, the SHIFT key, the COMMAND key, and the mouse button (in that order) at the time when the event occurred. In the case of the keys, **#t** means ON and **#f** means OFF; in the case of the mouse button, **#t** means PRESSED and **#f** means NOT PRESSED.

5) a data expression giving some identifying information about the event:

   a) In the case of a keypress, this is a list of two integers, the first of which is the ASCII code of the key and the second of which is the code number of the physical key that was pressed (independent of any modifier key)—see page 251 of *Inside the Macintosh, Volume I*.

   b) In the case of a mouse click in a graphics window, this is (the representation of) the window in which the click occurred.

   c) In the case of a menu selection, this is a string containing the text of the item selected.

For example, invoking **(click-demo)** after the following definitions will cause *EdScheme* to wait for a mouse click in the graphics window that is created by the procedure and return the turtle coordinate of the point clicked on:

```
(define click-demo
  (lambda ()
    (begin
      (make-graphics-window)
      (get-click))))
```

```
(define get-click
  (lambda ()
    (let ([ev (event)])
      (if (eq? (first ev) 'click)      ;;; Is it a click event?
          (list-ref ev 2)              ;;; YES - return the coordinate;
          (get-click)))))              ;;; NO - try again.
```

(event-flush) *Procedure*

Flushes all pending keyboard and mouse events. This procedure is useful, for example, for controlling the flow of user input being obtained using the procedure **event** (see the entry above) so that, effectively, only one event is allowed to be pending at any given moment. It needs to be used judiciously, however, in view of the possibility of a garbage collection occurring while the user produces a desired event that is then immediately flushed in response to a call to this procedure. (See the entry for the procedure **gc** later in this section.)

(event-ready?) *Procedure*

A predicate that returns the boolean `#t` if and only if an event (that is, a mouse click in a graphics window, a keypress, or a menu selection) is waiting to be 'read' using the procedure **event** (see the entry above).

(exact? *z*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is an exact number. Examples:

```
(exact? 3/5)      ↦ #t     (exact? -2.0)      ↦ #f
(exact? 5e2)      ↦ #f     (exact? 2-7/3i)    ↦ #t
(exact? 2.5+17i)  ↦ #f
```

(exact->inexact *z*) *Procedure*

Returns the inexact number that is numerically closest to *z*. Examples:

```
(exact->inexact 2)        ↦  2.0
(exact->inexact 3/5)      ↦  0.6
(exact->inexact -2.5)     ↦  -2.5
(exact->inexact +i)       ↦  0.0+1.0i
(exact->inexact 2.5-17i)  ↦  2.5-17.0i
```

(exact-rationalize *x1 x2*) *Procedure*

Returns the *simplest* exact rational that differs from ***x1*** by no more than the absolute value of ***x2***. (If a, b, c, and d are integers such that b and d are non-zero, a and b are relatively prime, and c and d are relatively prime, then a/b is **simpler** than c/d if and only if $|a| \leq |c|$ and $|b| \leq |d|$, where '$|x|$' denotes the absolute value of x—see the entry earlier in this section for the procedure **abs**) Examples:

```
(exact-rationalize pi 1/1000000)  ↦  355/113
(exact-rationalize 2/5 1/10)      ↦  1/2
(exact-rationalize 6.25 -1/5)     ↦  19/3
```

(exp *z*) *Procedure*

If ***z*** is real, returns e to the power z. If ***z*** is non-real, returns an inexact complex number that is calculated as explained on page 46 in Chapter 5: *Numbers and Numeric Procedures*. Examples:

```
(exp 1)      ↦  2.718281828459045
(exp -2.5)   ↦  0.0820849986238988
(exp 3-4i)   ↦  -13.12878308146216+15.200784463067951
```

(explode *atom*) *Procedure*

Returns a list of the single-character symbols that form the argument. (The entries in the output list are symbols, not characters. If you want the entries to be characters, use the procedure string->list instead—see the relevant entry later in this section.)

If ***atom*** is a string, the output list may include some spaces (as symbols rather than characters). Examples:

```
(explode 'abc)     ↦  (a b c)
(explode "a bc")   ↦  (a   b c)
```

expression *Variable*

A global variable bound to (the representation of) the Expression Window.

(expression-set-text *str*) *Procedure*

Sets the current expression to be the contents of the input ***str*** and updates the Expression Window, if visible. Example:

```
(begin
  (expression-set-text "(first '(a b c))")
  (expression-text))          ↦  "(first '(a b c))"
```

(expression->string *exp*) *Procedure*

Returns the string of the symbols required to print the argument. Examples:

```
(expression->string 'abc)          ↦  "abc"
(expression->string "abc")         ↦  "\"abc\""
(expression->string #\a)           ↦  "#\\a"
(expression->string '#(a b c))     ↦  "#(a b c)"
(expression->string '(a b c))      ↦  "(a b c)"
```

(expression-text) *Procedure*

Returns the current expression (in the form of a string).

(expt *z1 z2*) *Procedure*

Returns ***z1*** to the power ***z2***. If ***z1*** is 0, then ***z2*** must not be a negative real number—if ***z2*** is 0, the result is 1; otherwise, the result is 0. If either or both arguments are non-real (and ***z1*** is not 0), the result is calculated in the manner explained on page 46 in Chapter 5: *Numbers and Numeric Procedures*.

If ***z1*** is an exact complex number and ***z2*** is an exact *integer* (with the one proviso mentioned above in the case when ***z1*** is 0), then the result is exact. Otherwise the result is inexact.

This procedure produces exactly the same results as the procedure **power**—see the relevant entry later in this section. Examples:

```
(expt 2 3)        ↦  8
(expt 243 -1/5)   ↦  0.3333333333333333
(expt 2 5+i)      ↦  24.6156448436471l+20.446760842036311i
(expt 3-4i -3)    ↦  -117/15625+44/15625i
(expt 3-4i 5+i)   ↦  -7847.176113287395-902.88203243601111i
```

(file-exists? *spec*) *Procedure*

A predicate that returns the boolean `#t` if and only if the file specified by ***spec*** exists. In the case when ***spec*** is simply a string or symbol with no separating colons, then the file is only looked for among the non-folder documents on the current volume. (See Chapter 6: *Files and Ports* for information concerning the naming of files.)

(file-length *port*) *Procedure*

Returns an exact non-negative integer that gives the number of characters in the file at the specified port.

(file-margin *port*) *Procedure*

Returns an exact non-negative integer that gives the file margin of the file at the specified port (which must be an open port). A result of 0 corresponds to no file margin being set. (Also see the entry below for the procedure **file-set-margin**.)

(file-position *port*) *Procedure*

Returns an exact non-negative integer that gives the current position of the file pointer of the specified ***port***, that is, the number of bytes that precede the pointer's current location in the file at that port. Example:

```
(file-set-position port (+ (file-position port) 10))

;;; Moves the file pointer 10 bytes further along the file
;;; at the specified port or to the end of the file
;;; (whichever comes first).
```

(file-set-length *port k*) *Procedure*

Sets the length of the specified ***port*** to ***k*** characters by moving the end-of-file marker so that it becomes the character with index ***k*** (which must be non-negative and less than 2 to the power 31, that is, 2147483648).

It is generally undesirable for ***k*** to be greater than the length of the file at the specified ***port***, since reading such a 'lengthened' file might involve reading uninitialized portions of the disk.

(file-set-margin *port k*) *Procedure*

Sets the right margin for the file at the specified ***port*** to ***k*** characters (where ***k*** must be non-negative and less than 2 to the power 31, that is, 2147483648). When this margin is set, ***EdScheme*** sends a newline character to the file immediately following the last character of each atom that first exceeds the right margin. The setting remains in effect until reset by a subsequent use of **file-set-margin** or until the ***port*** is closed.

If ***k*** has the value 0, then the default setting is instated, in which no file margin is set, thus leaving you with the responsibility of explicitly inserting all subsequent newline characters into the file.

Example: After ***EdScheme*** evaluates the following sequence of expressions:

```
(define F (open-output-file "MyFile"))
(file-set-margin F 10)
(repeat 3 (display 'testing F))
(newline F)
(display "How does it look?" F)
(display '(fairly good considering) F)
(close-port F)
```

the contents of the file 'MyFile' will look like this:

```
testingtesting
testing
How does it look?
(fairly good
considering)
```

(file-set-position *port k*) *Procedure*

Places the file pointer of ***port*** at the byte with index ***k*** in the file at that ***port***. (***k*** must be non-negative and less than 2 to the power 31, that is, 2147483648.) If ***k*** is greater than the length of the file, then the file pointer is placed at the end of the file. (With the help of this procedure it is possible to implement random access files.) Example:

```
(file-set-position F 0)

;;; Moves the file pointer to the beginning of the file at
;;; port F.
```

(file-spec *port*) *Procedure*

Returns the full file specification of the file at the specified ***port***. This procedure is helpful if used immediately after opening a file by name, when your intention is subsequently to close that file and then reaccess it later in your program.

(first *pair*) *Procedure*

Returns the first component of the input ***pair***. Examples:

```
(first '(a))      ↦  a
(first '((a) b))  ↦  (a)
```

This procedure behaves in exactly the same way as the procedure **car** (see the relevant entry earlier in this section).

(floor *x*) *Procedure*

Returns the largest integer less than or equal to ***x***, the exactness of the result being the same as that of ***x***. (See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures* for information concerning the exactness of numbers.) Examples:

```
(floor 12/5)  ↦  2
(floor -3.2)  ↦  -4.0
```

(font-set-style *list*) *Procedure*
(font-set-style *list gxwin*)

Sets the style of font in which text will be displayed in the text or graphics window ***gxwin***, if provided, or the most recently active open text or graphics window, otherwise, to be that specified by the input ***list***. This must be a list of four exact positive integers that signify, in order of appearance, the font number, the style number, the text size, and the text mode. Style numbers are as follows:

1:	bold	16:	shadow
2:	italic	32:	condensed
4:	underline	64:	extended
8:	outline		

These may be combined simply by adding. Thus, for example, a style number of 38 corresponds to underlined, condensed, italic text. For details concerning the other numbers, see pages 171 and 219 of *Inside The Macintosh, Volume I*.

(font-style) *Procedure*
(font-style *gxwin*)

Returns a list of four numbers describing the current font style for text displayed in the text or graphics window ***gxwin***, if provided, or the most recently active open text or graphics window, otherwise. For the significance of the numbers, see the above entry for **font-set-style**.

(force *promise*) *Initialization File Procedure*

Forces the value of ***promise*** (which will usually have been generated by the special form **delay**—see the relevant entry earlier in this section). If no value has previously been computed for the ***promise***, a value is calculated and returned. The value calculated in this way is memoized so that if the same ***promise*** is forced again, then the already calculated value will be returned.

For information concerning a special form/procedure pair that behave in a similar fashion to delay and force, but without memoization, see the entries in this section for the special form freeze and the procedure thaw.

(for-each *proc list1 list2* ...) *Initialization File Procedure*

Calls the procedure *proc* (for side-effects only) as many times as there are elements in each input list, each call taking as its arguments similarly-placed elements (for example, the third element from each list). The procedure calls are made in order of appearance of the list elements, that is, on the first elements first, the second elements next, and so on. **The procedure *proc*** must accept as many arguments as there are input lists, and all the input lists must be the same length. The value returned by **for-each** is unspecified.

Example:

```
(define match-up
  (lambda (s1 s2)
    (let ([v (make-vector (length s1))] [ctr 0])
      (for-each (lambda (a b)
                  (begin
                    (if (equal? a b)
                        (vector-set! v ctr a)
                        (vector-set! v ctr '-))
                    (set! ctr (add1 ctr))))
                s1 s2)
      (vector->list v))))        ↦  unspecified

(match-up
  '(2 a #(1 2 3) "str ng" #\z)
  '(1 a #(1 2 3) "string" #\z))  ↦  (- a #(1 2 3) - #\z)
```

(forward *x*) *Procedure*
(forward *x gwin*)

Makes the turtle advance *x* steps in the graphics window ***gwin***, if specified, or in the most recently active open graphics window, otherwise. The nature of the track left by the turtle depends upon the current pen state (see the entry for **pen-state** later in this section).

(freemem) *Procedure*

Returns the size, in bytes, of the largest available contiguous block of free

memory available to ***EdScheme***. This gives you some indication of, for example, the largest graphics window you could open in the current state of your system.

(freesp) *Procedure*

Returns the number of unused Scheme pairs currently available in your system. When this number reaches 0, ***EdScheme*** automatically triggers a garbage collection, thereby (usually) releasing new pairs. Thus, the value returned by this procedure gives you some indication of how long it might be before the next automatic garbage collection. (See the entry for the procedure **gc** later in this section.)

(freeze *sexp*) *Special Form*

Creates a thunk which, when passed to the procedure **thaw** (see the relevant entry later in this section), yields the value of the input ***sexp***.

A special form/procedure pair that behaves in a similar fashion to **freeze** and **thaw**, but which uses memoization, is also available. See the entries in this section for the procedure **force** and the special form **delay**.

(freshline) *Procedure*
(freshline *port*)
(freshline *txwin*)

Sends a newline character to the specified port or window, if provided, or the current output port (usually the Transcript Window), otherwise, *unless* the character most recently sent to that port or window was also a newline character. In that case, no action is taken. (See the entry for the procedure **current-output-port** earlier in this section.) Returns an unspecified value.

To send a newline character *unconditionally*, use the procedure **newline**, described later in this section.

(gc) *Procedure*
(gc *bool*)

Invokes a garbage collection, releasing as many Scheme pairs as possible. If

an argument of **#t** is provided, then ***EdScheme***'s oblist is also purged and compacted. Otherwise (if an argument of **#f** or no argument at all is provided), a 'partial' garbage collection is performed in which no such purging and compaction of the oblist take place.

(gcd *n1* ...) *Procedure*

Returns the greatest common divisor of its arguments, if any, or 0, if there are none. The result is always non-negative, and is exact if and only if *all* the arguments are exact. Examples:

```
(gcd)              ↦  0
(gcd -4)           ↦  4
(gcd 6 -15.0 54)   ↦  3.0
```

(graphics-origin) *Procedure*
(graphics-origin *gwin*)

Returns a pair of inexact numbers indicating the location of the origin of the turtle plane corresponding to the graphics window *gwin*, if provided, or the most recently active open graphics window, otherwise, measured in turtle steps relative to the bottom left corner of the plane. By default, the origin of a turtle plane is at its center.

(graphics-set-origin *pair*) *Procedure*
(graphics-set-origin *pair gwin*)

Sets the graphics origin in the graphics window *gwin*, if provided, or the most recently active open graphics window, otherwise, as indicated by the input *pair*. This input contains two numbers that give the coordinate of the origin's desired position in turtle steps relative to the bottom left corner of the corresponding turtle plane. The graphics origin is the turtle's 'home' (see the entry for the procedure home later in this section).

(graphics-window) *Procedure*

Returns (the representation of) the most recently active open graphics window, or the boolean #f if no graphics window is open.

(graphics-window? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is a graphics window.

(head *stm*) *Procedure*

Returns the first component of the input stream. Example:

```
(define S (cons-stream 1 2))  ↦  unspecified
(head S)                      ↦  1
```

(home) *Procedure*
(home *gwin*)

Sends the turtle in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, to the graphics origin and changes its heading to zero (that is, due North), while leaving the pen state unchanged (see the entry for the procedure **pen-state** later in this section).

(if <*test*> *sexp1 sexp2*) *Special Form*
(if <*test*> *sexp1*)

If **<*test*>** evaluates to a true value, then the Scheme expression ***sexp1*** is evaluated and its value returned. On the other hand, if **<*test*>** evaluates to the boolean **#f**, then the 'alternative' Scheme expression ***sexp2*** is evaluated (if it is provided) and its value returned. If no alternative Scheme expression is provided and **<*test*>** evaluates to **#f**, the value of the if-expression is unspecified.

In Schemer's Guide mode only, **<*test*>** is required to evaluate to a boolean object.

Example:

```
(if (< 2/5 3/7)
    (- 2/5 3/7)
    (- 3/7 2/5))  ↦  -1/35
```

(imag-part *z*) *Procedure*

Returns the imaginary part of the complex number *z*. Examples:

```
(imag-part 1.2)   ↦  0      (imag-part 2+3.5i)  ↦  3.5
(imag-part 1-i)   ↦  -1     (imag-part (/ 1-i))  ↦  1/2
```

(implode *list*) *Procedure*

Concatenates the atoms in the input ***list*** and returns the result as a symbol. (The input ***list*** must be non-empty and contain only atoms.) Since the output from this procedure is always a symbol, if that symbol is interpretable as a number and you wish to use it as such, you must pass the output through a combination of the procedures **symbol->string** and **string->number** before you can do so (see the entries for these two procedures later in this section, and the second example below). Examples:

```
(implode '(abra "cad" abr #\a))  ↦  abracadabra
(add1 (string->number
        (symbol->string
          (implode '(0 0 7)))))  ↦  8
```

(inexact? *z*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is inexact. (See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures* for further information concerning the exactness of numbers.) Examples:

```
(inexact? 1/2)      ↦  #f
(inexact? 2-0.5i)   ↦  #t
(inexact? (sqrt 4))  ↦  #t
```

(inexact->exact *z*) *Procedure*

Returns the exact number that is numerically closest to the input complex number. (See page 45 in Chapter 5: *Numbers and Numeric Procedures* for details of why this procedure sometimes behaves in a surprising fashion.) Examples:

(inexact->exact 2) ↦ 2
(inexact->exact *pi*)
↦ 3141592653589793239/1000000000000000000
(inexact->exact 2-0.6i) ↦ 2-3/5i
(inexact->exact (/ 5e2))
↦ 1999999999999999/1000000000000000000

(input-port? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is either the Transcript Window, a serial port, a port at which there is a file that has been opened for input, or an 'input-string' (created by a call to the procedure **open-input-string**—see the relevant entry later in this section).

(integer? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is an integer. Examples:

(integer? 'ten) ↦ **#f** **(integer? 5e2)** ↦ **#t**
(integer? 6/3) ↦ **#t** **(integer? 1/2)** ↦ **#f**
(integer? 5-0.0i) ↦ **#t** **(integer? *pi*)** ↦ **#f**

(integer->char *k*) *Procedure*

In Standard Scheme mode, returns the character whose ASCII code is the remainder when ***k*** is divided by 256. In Schemer's Guide mode, returns the corresponding symbol. Examples:

(integer->char 112) ↦ **#\p** [in Standard Scheme mode]
(integer->char 575) ↦ **#\?** [in Standard Scheme mode]
(integer->char 36) ↦ **$** [in Schemer's Guide mode]

---

integrate-primitives *Variable*

When set to the boolean **#t**, ***EdScheme*** reserves the right to compile certain primitive procedures (such as **first** and **add1**) in-line, thus speeding up ***EdScheme***'s performance. When set to the boolean **#f**, all identifiers in the operator position of a Scheme expression are treated as variables. Initially **bound to the boolean #t.**

(lambda <*formals*> *sexp1 sexp2* ...) *Special Form*

Returns a procedure object that incorporates a record of the environment in effect when the lambda-expression is evaluated. If the procedure so described is later called on some actual argument(s), then the variable(s) that feature(s) in **<*formals*>** is/are bound to the corresponding argument(s), the resulting binding(s) is/are added to the recorded environment, the Scheme expressions ***sexp1***, ***sexp2***, ... are evaluated in order of appearance in the extended environment, and the values of the final Scheme expression is returned.

In Standard Scheme Mode, **<*formals*>** must be of one of the following forms:

(*var1* ...): The procedure takes a fixed number of arguments (as many as there are variables); when it is called, the first argument is bound to the first variable, the second argument is bound to the second variable, and so on.

var: The procedure takes a variable number of arguments; when it is called, the actual arguments (in order of appearance) are made into a list which is then bound to the variable ***var***.

(*var1* ... *vark–1* . *vark*): (Note that a space-delimited dot precedes the final variable.) The procedure takes $k - 1$ arguments *or more*; when it is called, the first argument is bound to the first variable, and so on up through the $(k - 1)$st argument, and any remaining arguments (in order of appearance) are made into a list which is then bound to the variable ***vark***.

In Schemer's Guide Mode, only the first two of these forms are allowable. In neither mode may any variable appear more than once in **<*formals*>**.

Examples:

```
((lambda (x y) (cons x y)) 'in '(sert))  ↦  (in sert)

((lambda s
   (display "And the total is: ")
   (apply + s))
 2 4 6 8)
And the total is:                        ↦  20
```

```
;;; The next example is only possible in Standard Scheme Mode.
((lambda (x y . z)
   (cond
     [(> x y) (first z)]
     [(< x y) (last z)]
     [else (list-ref z 1)]))
 5 7 'a 'ab 'b)                    ↦  b
```

(language-preferences) *Procedure*

Returns a list of five booleans that report the settings in the Language dialog accessed through the Preferences sub-menu in the File menu. Their meanings, in order of appearance, are as follows:

1) The Language Mode selection: `#t` corresponding to Standard Scheme Mode and `#f` corresponding to Schemer's Guide Mode.

2) The Angle Mode selection: `#t` corresponding to Radian Mode and `#f` corresponding to Degree Mode.

3) The first Error Mode selection: `#t` corresponding to extended error messages being activated and `#f` corresponding to their being switched off.

4) The second Error Mode selection: `#t` corresponding to a (partial) garbage collection being invoked following each error—see the entry for the procedure **gc** earlier in this section—and `#f` corresponding to no such garbage collection being invoked.

5) The third Error Mode selection: `#t` corresponding to stack tracing error messages being activated and `#f` corresponding to their being switched off.

(language-set-preferences *list*) *Procedure*

Sets features in the Language dialog accessed through the Preferences sub-menu in the File menu that can be adjusted while an ***EdScheme*** session is in progress so that they conform to the input ***list***. This must be a list of five boolean objects. The features in question and the settings that correspond

to **#t** and **#f**, respectively, are explained in the above entry for the procedure **language-preferences**. The new settings take effect immediately, and the report of the current settings is automatically updated in the **Language** dialog from the **Preferences** sub-menu. Example:

```
(language-set-preferences '(#t #t #t #f #f))

;;; Sets EdScheme to Standard Scheme Mode, Radian Mode,
;;; with extended error messages, but without a forced
;;; garbage collection after each error and without stack
;;; tracing error messages.
```

(last *list*) *Procedure*

Returns the last data expression in its argument, which must be a non-empty list. Examples:

```
(last '(a b c))          ↦  c
(last '(a (b (c (d)))))  ↦  (b (c (d)))
```

(last-pair *list*) *Procedure*

Returns the last pair of its argument, which must be a non-empty list. Examples:

```
(last-pair '(a b c))              ↦  (c)

(define A (list 'a 'b 'c))        ↦  unspecified
(set-rest! (last-pair A) '(d e))  ↦  unspecified
A                                 ↦  (a b c d e)
```

(last-volume) *Procedure*

Returns the reference number of the most recently accessed volume. ('Accessing' includes opening and loading *EdScheme* files.) This procedure is useful if you are writing an *EdScheme* program that will cause specified files to be read from or written to. Consider, for example, the Scheme expression

```
(set-volume (last-volume))
```

at the beginning of the file 'game startup.s' in the 'Game' folder on the *EdScheme* implementation disk. As explained in the 'Game ReadMe' file, that file is loaded from within *EdScheme*, the order of operations being

1. launch the *EdScheme* application;
2. open and evaluate all the data expressions in the file 'game startup.s'.

Since this file is located in the 'Game' folder, the above expression sets the current volume to the one that corresponds to the 'Game' folder. This means that files referred to explicitly later in the program—as, for example, by the expression **(load "game graphics.s")**—are looked for in the 'Game' folder.

(lcm *n1* ...) *Procedure*

Returns the least common multiple of its arguments, if any, or 1, if there are none. The result is always non-negative, and is exact if and only if all the arguments are exact. (See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures* for more information about the exactness of numbers.) If one or more of the arguments is/are 0, then the result is also 0. Examples:

```
(lcm)      ↦  1      (lcm -2 0 24)  ↦  0
(lcm -4)   ↦  4      (lcm -15 6.0)  ↦  30.0
```

(left *x*) *Procedure*
(left *x gwin*)

Subtracts *x* radians or degrees (according to the Angle Mode specified in the Language dialog accessed through the Preferences sub-menu in the File menu) from the current heading of the turtle in graphics window *gwin*, if provided, or the most recently active open graphics window, otherwise, adjusting the direction of the turtle on screen if it is currently begin shown.

(length *list*) *Procedure*

Returns the length of—that is, the number of data expressions in—its argument as an exact integer. Examples:

```
(length '())       ↦  0      (length '(a (b (c (d (e))))))  ↦  2
(length '(b b b))  ↦  3
```

(let ([*var1 sexp1*] ...) *sexpk sexpk+1* ...) *Special Form*
(let *var* ([*var1 sexp1*] ...) *sexpk sexpk+1* ...)

In the first form, the Scheme expressions ***sexp1*** through ***sexpk–1*** are evaluated in the current environment in some unspecified order, the variables ***var1***, ..., are bound to the results of the corresponding Scheme expressions, the Scheme expressions ***sexpk***, ..., are evaluated in the current environment extended by these new bindings, and the value of the final Scheme expression is returned. The sequence of Scheme expressions from ***sexpk*** onward is known as 'the let-body'. The bindings of the variables ***var1***, ..., are local to the let-body. Example:

```
(let ([a 2] [b 3])
  (let ([a 4] [c (- a b)])
    (* c a)))                ↦  -4

;;; The a appearing in the binding for c takes its value from
;;; the outside let-expression (so c is bound to 2 - 3 = -1),
;;; whereas the a appearing in the body of the inside let-
;;; expression takes its value from that inner expression.
```

In the second form—known as 'a named **let**'—the method of evaluation is the same as in the first form, except that within the let-body the variable ***var*** is bound to a procedure whose formal arguments are the variables ***var1***, ..., and whose body is the let-body. This introduces the possibility of versatile looping procedures. Example:

```
(let loop ([factors1 '(2 3 3 7)]
           [factors2 '(2 2 2 3 5 7 7 11)])
  (cond
    [(or (null? factors1) (null? factors2)) '()]
    [(< (first factors1) (first factors2))
     (loop (rest factors1) factors2)]
    [(> (first factors1) (first factors2))
     (loop factors1 (rest factors2))]
    [else (cons (first factors1)
                (loop (rest factors1)
                      (rest factors2)))]))   ↦   (2 3 7)
```

For both forms, there must be no repetitions among the variables bound by the let-expression in the context of the let-body. It is convenient to highlight the bindings being established by delineating them with square brackets (see, for instance, the bindings **[*a* 2]** and **[*b* 3]** in the first example above). As far as ***EdScheme*** is concerned, however, brackets and parentheses are entirely interchangeable in this context.

(let* ([*var1 sexp1*] ...) *Initialization File Special Form*
** *sexpk sexpk+1* ...)**

Evaluates in the same way as the corresponding let-expression would evaluate, except that the Scheme expressions ***sexp1*** through ***sexpk−1*** are evaluated in order of appearance, and the binding established for each variable is visible not only to the let-body, but also to all the Scheme expressions appearing later in the variable-binding list. Example:

```
(let ([a 2] [b 3])
  (let* ([a 4] [c (- a b)])
    (* c a)))                    ↦  4

;;; In the (inner) let*-expression, the binding of the value 4
;;; to the variable a is visible when the binding for c is
;;; established. Consequently, c is bound to 4 − 3 = 1.
```

As in the case of let-expressions, ***EdScheme*** views brackets and parentheses interchangeably.

(letrec ([*var1 sexp1*] ...) *sexpk sexpk+1* ...) *Special Form*

Bindings for the variables ***var1***, ..., to unspecified values are added to the current environment, the Scheme expressions ***sexp1*** through ***sexpk−1*** are evaluated in this extended environment in some unspecified order, the resulting values being assigned to the corresponding variables, the Scheme expressions ***sexpk***, ..., are evaluated in the new environment, and the value of the final Scheme expression is returned.

The bindings of the variables extend over the entire letrec-expression, not just the body (comprising the Scheme expressions *sexpk*, ...). This makes it possible for the variable-bindings to include mutually recursive procedures.

There must be no repetitions among the variables ***var1*** through ***vark–1***, and it must be possible to evaluate each of the Scheme expressions ***sexp1*** through ***sexpk–1*** without assigning or referring to the value of any of the variables. (In most normal uses of **letrec**, this last restriction is automatically satisfied because the Scheme expressions in question are all lambda-expressions.)

Example:

```
(letrec ([move1
          (lambda (n) (if (> n 0) (move2 (+ n 15)) n))]
         [move2
          (lambda (n) (if (< n 20) (move1 (- n 7)) n))])
  (move1 3))                                    ↦  26
```

(list *exp1* ...) *Procedure*

Returns the list of its arguments, in order of appearance. Example:

```
(list 'two 'and 'two 'is (+ 2 2))  ↦  (two and two is 4)
```

(list? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a list. Note that 'circular lists', such as the one in the final example below, are not 'proper' lists; they generate a result of `#f` when input to **list?**. Examples:

```
(list? '())          ↦  #t
(list? '(a b))       ↦  #t
(list? 'a)           ↦  #f
(list? '(a . b))     ↦  #f    [Standard Scheme Mode only]
(let ([x '(a)])
  (set-rest! x x)
    (list? x))       ↦  #f
```

(list-ref *list k*) *Procedure*

Returns the data expression with index ***k*** in the input ***list***, where the first expression has index 0, the second has index 1, and so on. ***k*** must be an exact, non-negative integer that is strictly less than the length of ***list***. Example:

```
(list-ref '(a ((a) b) c) 1)  ↦  ((a) b)
```

(list->string *list*) *Procedure*

Returns the string that results when the contents of its argument (which must be a list of characters) are concatenated. Example:

```
(list->string '(#\a #\b #\c))  ↦  "abc"
```

(list-tail *list k*) *Procedure*

Returns what remains of the input ***list*** once the first ***k*** data expressions have been removed. (If ***k*** is greater than the length of the ***list***, then the empty list is returned.) Example:

```
(list-tail '(a b c d e) 2)  ↦  (c d e)
```

(list->vector *list*) *Procedure*

Returns a vector whose entries are the data expressions in the input ***list*** in order of appearance. Example:

```
(list->vector '(a b c))  ↦  #(a b c)
```

(load *spec*) *Procedure*
(load *spec bool*)

Loads the (existing) file given by the input specification, and sequentially evaluates the Scheme expressions it contains in the current environment. If ***bool*** is the boolean `#t`, then as each Scheme expression is evaluated its value is displayed on the screen. If ***bool*** is the boolean `#f`, then the values of the Scheme expressions are not displayed. If no boolean argument is provided, then ***EdScheme*** operates as specified by your setting of the **Print Values when Loading** checkbox in the Language dialog accessed through the Preferences sub-menu in the File menu: If the checkbox is checked, ***EdScheme*** behaves as if a boolean argument of `#t` had been provided; if the checkbox is not checked, it behaves as if a boolean argument of `#f` had been provided. In all cases, the value of the final Scheme expression is returned. (Note that the current input and output ports are not altered by the evaluation of a load-expression.)

If you do not know the correct file specification for the file you want to load, use one of the following three Scheme expressions:

```
(load (choose-input-file) #f)
(load (choose-input-file) #t)
(load (choose-input-file))
```

and indicate the desired file using the File Selector. As an alternative to the last of these, you could choose the Load ... option from the Evaluate menu.

If an error occurs while the loading is in progress, then evaluation will stop, and the error will be reported in accordance with the settings you have made in the Debugging dialog accessed through the Preferences sub-menu in the File menu. In addition, *EdScheme* takes steps to preserve the contents of your files by closing all files that are open for loading at the time when the error occurs.

(log *z1*) *Procedure*
(log *z1 z2*)

Returns the logarithm of *z1* to the base *z2*, if provided, or to the base *e* (that is, approximately 2.718281828459045), otherwise. Neither argument may be 0, and the second argument (when provided) must not be 1. Examples:

```
(log 5)          ↦  1.6094379124341
(log 2 10)       ↦  0.3010299956639812
(log 3-4i)       ↦  1.6094379124341-0.9272952180016122i
(log 1-i 2+5i)   ↦  -0.08264069265990966-0.40806143099836561
```

For information on how these values are calculated (especially in the case when complex numbers are involved), see page 47 in Chapter 5: *Numbers and Numeric Procedures*.

(magnitude *z*) *Procedure*

Returns the magnitude of the argument. If *z* is complex with an exact zero real part or if it is real, then the result has the same exactness as *z*. For all other values of *z*, the result is inexact. (See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures* for information concerning the

exactness of numbers. Also see the entry for the procedure **abs** earlier in this section.) Examples:

```
(magnitude -3/2)   ↦  3/2        (magnitude 0+4i)  ↦  4
(magnitude 5-12i)  ↦  13.0
```

(make-bitmap *pair*) *Procedure*
(make-bitmap *pair k*)

Returns a bitmap whose size is given by the input ***pair***, which must be a list of two positive exact integers. The first of these two numbers specifies the width of the bitmap, and the second specifies its height. The optional second input specifies the 'color depth' of the bitmap; it must be one of the following four numbers:

1:	black and white	2:	4 colors
4:	16 colors	8:	256 colors

If you omit the second input or if you specify a value that exceeds the greatest 'color depth' of any device connected to your computer, then ***EdScheme*** sets a depth equal to that maximum 'color depth'.

See the entries for the procedures **bitmap-fetch** and **bitmap-set-spec** earlier in this section for examples of this procedure being used in context.

(make-environment *sexp1 sexp2* ...) *Special Form*

Returns the environment that results from evaluating the given Scheme expressions in order of appearance in the current environment. See the entry for the procedure **eval** earlier in this section for an example of this procedure being used in context.

(make-graphics-window) *Procedure*
(make-graphics-window *pair1*)
(make-graphics-window *pair2*)
(make-graphics-window *pair1 pair2*)

Creates a graphics window and returns (a representation of) that window.

If the optional input ***pair1*** (a pair of exact whole numbers) is provided, then the width of the turtle plane onto which the graphics window looks is given by the first number and its height is given by the second. (If either of the numbers in this pair is less than 20, then ***EdScheme*** replaces it by 20. Furthermore, the smallest graphics window ***EdScheme*** will open measures 50 by 50, even if the turtle plane onto which it looks has smaller dimensions.) If ***pair1*** is not provided, then the default width of the turtle plane is 576 turtle steps and its default height is 720 turtle steps.

If the optional input ***pair2*** (a pair consisting of an exact whole number ***n*** from 0 through 6 and a boolean ***bool***) is provided, then the nature of the window that is created is determined according to the following tables:

n	*Close box*	*Scroll bars*	*Title*
0	Yes	Yes	Yes
1	No	Yes	Yes
2	Yes	No	Yes
3	No	No	Yes
4	No	No	No (single border)
5	No	No	No (shadow border)
6	No	No	No (double border)

bool	
`#t`	The window is created in a 'shown' state.
`#f`	The window is created in a 'hidden' state.

If ***pair2*** is not provided, then ***EdScheme*** behaves as if an input of `(0 #t)` had been provided.

It is advisable to create a window in a hidden state if you have to size it, position it, and/or give it a title before any further activity takes place in it. Then, once the window is in a 'presentable' state, you can 'show' it using the procedure **window-show**.

Calling the procedure **window-set-title** on a graphics window that has been created with no title will have no effect (and no error will be generated). The only way to scroll a window that has been created with no scroll bars is to use the procedure **window-set-position**. (The various graphics window procedures are catalogued in Section 12.7: *Graphics and Text Windows* in Chapter 12: *Language Elements* and described in detail in this section, where they appear in alphabetical order.)

(make-menu *str list*) *Procedure*

Space permitting, adds a menu to the menu bar whose title is the contents of the string ***str*** and whose items are the contents of the strings in the input ***list***, in order of appearance from top to bottom. A separator line may be included in the menu by using the string **"-"** in the appropriate position of the input ***list***. Returns (a representation of) the menu.

The items in user-defined menus are deselected and unchecked at all times except during an invocation of the thunk **(event)**—see the relevant entry earlier in this section—when the status of the items is determined by the most recent call to **menu-set-item** (see below), if there has been such a call, or the default status (all selected, all unchecked), otherwise. Example:

```
(define M (make-menu
            "React"
            '("Laugh" "Cry" "Sniff" "-" "Cancel")))
```

creates a menu (denoted by the identifier ***M***) whose title is React and whose items, from top to bottom, are Laugh, Cry, Sniff, and Cancel, with a separator line between the last two items.

(make-polar *x1 x2*) *Procedure*

Returns the complex number ***x1*** $*[(\cos$ ***x2***$)+(i*(\sin$ ***x2***$))]$. The second input is interpreted as being in radians, irrespective of the Angle Mode you have chosen in the Language dialog accessed through the Preferences sub-menu in the File menu. The result is an inexact complex number unless either the first input is an exact zero or both inputs are exact with the second being zero. (See Chapter 5: *Numbers and Numeric Procedures* for further information concerning ***EdScheme***'s treatment of complex numbers.) Example:

```
(make-polar 4 (/ pi 6))  ↦  3.464101615137755+2.0i
(make-polar 0 5.7)       ↦  0
(make-polar 5 0)         ↦  5
```

(make-rectangular *x1 x2*) *Procedure*

Returns the complex number *x1* + (i * *x2*). (See Chapter 5: *Numbers and Numeric Procedures* for further information concerning ***EdScheme***'s treatment of complex numbers.) Example:

```
(make-rectangular 2 -3.5)  ↦  2-3.5i
```

(make-string *k*) *Procedure*
(make-string *k ch*)

Returns a string that is *k* characters long. The string is filled with the input character *ch*, if provided; otherwise, the contents of the string are unspecified. (The optional input may be provided in the form of a symbol, a character, or a string.) *k* must be less than 2 to the power of 31, that is, 2147483648. Example:

```
(define S (make-string 10 "A"))  ↦  unspecified
S                                ↦  "AAAAAAAAAA"
```

(make-text-window) *Procedure*
(make-text-window *k*)
(make-text-window *pair*)
(make-text-window *k pair*)

Creates a text window whose line width (that is, the maximum length before text wraps onto the next line) is *k* pixels, if the optional input is provided. In that case, the line width will not change if the window is subsequently resized by dragging or using the procedure **window-set-position** (see the relevant entry later in this section). On the other hand, if no width is specified, then the line width is the same as the width of the window, and will change if the window is subsequently resized. Returns (a representation of) the text window thus created.

k must be non-negative and less than 2 to the power 15, that is, 32768.

The provision of the optional input *pair*, consisting of a whole number from 0 through 6 and a boolean, has the same effect as for graphics windows (see the entry for **make-graphics-window** above). Its default value is again `(0 #t)`.

The comments made under **make-graphics-window** regarding the use of the procedure **window-set-title** and the possibility of scrolling the window being created also apply in the case of text windows.

(make-vector *k*) *Procedure*
(make-vector *k exp*)

Returns a vector that has ***k*** elements. All the elements are set to the input data expression, if provided; otherwise, the elements of the vector are unspecified. ***k*** must be non-negative and less than 2 to the power of 31, that is, 2147483648. Example:

```
(define V (make-vector 3 '#(a "B")))
                    ↦  unspecified
V                   ↦  #(#(a "B") #(a "B") #(a "B"))
```

(map *proc list1 list2* ...) *Initialization File Procedure*

Returns a list whose first element is the result of applying the input ***proc*** to the first elements of the input lists, whose second element is the result of applying ***proc*** to the second elements of the lists, and so on. The procedure ***proc*** must accept as many arguments as there are input lists, and these lists must all have the same length. (The order in which the applications of ***proc*** take place is not specified, but the order of the elements in the output list is the same as that of the elements in the input lists.) Example:

```
(map (lambda (x y)
       (cons (last x) (rest y)))
     '((a b) (2 7) (3 3) (y z))
     '((3 2) (h) (4 2 1) (x y)))   ↦   ((b 2) (7) (3 2 1) (z y))
```

(max *x1 x2* ...) *Procedure*

Returns the greatest of the inputs as an exact number (if all the inputs are exact) or as an inexact number (if not all the inputs are exact). See Section 5.9: ***Exactness*** in Chapter 5: ***Numbers and Numeric Procedures*** for further information concerning the exactness of numbers. Examples:

```
(max -3)            ↦  -3
(max 4 -5.0 79)     ↦  79.0
(max -999 -999.0)   ↦  -999.0
```

(member *exp list*) *Procedure*

Returns the boolean **#f** if the input ***list*** does not contain the data expression ***exp***. Otherwise, the sublist of ***list*** from the first occurrence of ***exp*** to the end of ***list*** is returned. (Comparisons are made using the predicate **equal?**—see the relevant entry earlier in this section.) Examples:

```
(member 'a '(b c d e))          ↦  #f
(member '(a) '(b (a) (b a)))    ↦  ((a) (b a))
```

(member? *exp list*) *Procedure*

A predicate that returns the boolean **#t** if and only if its first argument is contained in its second. (Comparisons are made using the predicate **equal?**—see the relevant entry earlier in this section.) Examples:

```
(member? 'a '(b c d e))          ↦  #f
(member? '(a) '(b (a) (b a)))    ↦  #t
```

(memq *exp list*) *Procedure*

Behaves just like the procedure **member** (see above), except that comparisons are made using the predicate **eq?** (see the relevant entry earlier in this section). Examples:

```
(memq 'a '(b a c))                ↦  (a c)
(memq '(a) '((b) (a) (c)))        ↦  #f
(let ([X '(a)])
  (memq X (list '(b) X '(c))))    ↦  ((a) (c))
```

(memv *exp list*) *Procedure*

Behaves just like the procedure **member** (see above), except that comparisons are made using the predicate **eqv?** (see the relevant entry earlier in this section). Examples:

```
(memv 3 '(1 3 5))                  ↦  (3 5)
(memv '(2 b) '((1 a) (2 b) (3 c))) ↦  #f
```

(menu? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if the input ***exp*** is a menu.

(menu-close) *Procedure*
(menu-close *menu*)

Closes the input user ***menu***, if provided, or all user-defined menus, otherwise, and removes it/them from the menu bar, repositioning any remaining menus on the menu bar so that they are as far to the left as possible.

(menu-item *menu k*) *Procedure*

Returns a list describing the current state of the item with index ***k*** in the input user ***menu***. (Note that, unlike most of Scheme's indexed structures, menu items are 1-referenced, that is, the first item in the menu has index 1, rather than 0.) ***k*** must be greater than 0 and less than or equal to the number of items in the menu (see the entry below for the procedure **menu-number-of-items**).

The list contains three data expressions, the first being a string containing the text of the item in question, the second being a boolean that is **#t** if the item is deselected (that is, dimmed) and **#f** if it is selected (that is, undimmed), and the third being a boolean that is **#t** if the item is checked and **#f** if it is not checked.

(menu-number-of-items *menu*) *Procedure*

Returns the number of items in the input user ***menu***. Note that a separator line counts as an item.

(menu-set-item *menu k list*) *Procedure*

Sets the state of the item with index *k* in the input user *menu* (the first item having index 1) so that it conforms to the specifications in the input

list, which must contain three data expressions of the type and having the meanings explained in the above entry for the procedure **menu-item**.

Note that it is permissible for the item text in the final argument to differ from what it was at the time when the menu was created; this enables you to make dynamic changes in the contents of the menus you create. *k* must be positive and less than or equal to the number of items in the menu (see the entry above for the procedure **menu-number-of-items**).

The state of the menu item does not come into effect until the next invocation of **(event)**—see the relevant entry earlier in this section—and remains in effect until the next call to **menu-set-item** with the same inputs *menu* and *k*.

It is usual for matters to be so arranged that separator lines in menus are always dimmed and unchecked; to achieve this, use **menu-set-item** with the appropriate inputs *menu* and *k* and a final input of `("-" #t #f)`.

There are many examples of the use of this procedure in the files relating to the demonstration game provided on the *EdScheme* implementation disk.

(min *x1 x2* ...) *Procedure*

Returns the smallest of its inputs as an exact number (if all the inputs are exact) or as an inexact number (if not all the inputs are exact). See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures* for further information concerning the exactness of numbers. Examples:

```
(min -3)          ↦  -3
(min 4.03 -5 79)  ↦  -5.0
(min -999 -999.0) ↦  -999.0
```

(modifiers) *Procedure*

Returns a list of five booleans giving the states of the OPTION key, the CAPS LOCK key, the SHIFT key, the COMMAND key, and the mouse button (in that order) at the time of the most recent invocation of **(read-char)** (see the relevant entry later in this section). In the case of the keys, `#t` means ON and `#f` means OFF; in the case of the mouse button, `#t` means PRESSED and `#f` means NOT PRESSED.

(modulo *n1 n2*) *Procedure*

Returns the unique integer r strictly between $n2$ and $-n2$ such that $r * n2$ is positive and, for some integer q,

$$n1 = (q * n2) + r,$$

that is, the remainder—with the same sign as $n2$—when $n1$ is divided by $n2$. The second input must not be zero. The result is exact if both inputs are exact, but inexact otherwise. (See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures* for further information concerning the exactness of numbers.) Examples:

```
(modulo 14 3)     ↦  2      (modulo 14 -3.0)  ↦  -1.0
(modulo -14 -3)   ↦  -2     (modulo -14 3.0)  ↦  1.0
```

(See also the procedure **remainder**, which is described later in this section, and behaves slightly differently.)

(mouse-state) *Procedure*
(mouse-state *gxwin*)

Returns a list of two data expressions indicating the state of the mouse in the context of the text or graphics window ***gxwin***, if provided, or the most recently active open text or graphics window, otherwise. The first data expression is a pair of numbers giving the coordinate of the mouse in the window in question (relative to the current graphics origin in the case of a graphics window, and relative to the top left corner of the window in the case of a text window), and the second is a boolean, which is `#t` if the mouse button is pressed and `#f` otherwise.

(negative? *x*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is negative. (The result may be unreliable if the argument is an inexact number that is close to zero. See Section 5.9: in Chapter 5: *Numbers and Numeric Procedures* for further information concerning the exactness of numbers.) Examples:

```
(negative? 5)  ↦  #f      (negative? -6.25)  ↦  #t
```

(newline) *Procedure*
(newline *port*)
(newline *txwin*)

Sends a newline character to the specified port or window, if provided, or the current output port (usually the Transcript Window), otherwise. (See the entry for the procedure **current-output-port** earlier in this section.) Returns an unspecified value.

(not *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is the boolean **#f**. In Schemer's Guide Mode, the input must be a boolean object; in Standard Scheme Mode, no such restriction applies. Examples:

```
(not (list? 3))     ↦  #t
(not 3)             ↦  #f      [Standard Scheme Mode only]
(not (null? '()))   ↦  #f
```

(note-alert *str*) *Procedure*

Activates a Macintosh note alert box containing the contents of the input string. The size of this box is set by ***EdScheme*** and the text of the alert automatically word-wraps to fit inside the box.

(nth *k list*) *Procedure*

Returns the ***k***th data expression in the input ***list***, where the expressions are numbered in the natural, intuitive way starting from 1. (By contrast, the procedure **list-ref**, described earlier in this section, does a similar job, but numbers the expressions in the list starting from 0.) ***k*** must be positive and less than or equal to the number of data expressions in the input ***list***. Example:

```
(nth 2 '(a b c d))  ↦  b
```

(null? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is the empty list. In Schemer's Guide Mode, the input must be a list; in Standard Scheme Mode, no such restriction applies. Examples:

```
(null? (rest '(a)))  ↦  #t
(null? '(a))         ↦  #f
(null? 'a)           ↦  #f      [Standard Scheme Mode only]
```

(number? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is a number. Examples:

```
(number? 3)   ↦  #t      (number? '())  ↦  #f
(number? 'a)  ↦  #f
```

(number->string *z*) *Procedure*
(number->string *z rad*)

If the input number *z* is exact, returns—as a string—a representation of *z*, using the radix *rad*, if provided, or 10, otherwise. If *z* is inexact, ignores the input radix, if one is provided, and returns—as a string—a representation of *z* in decimal or exponential form. The resulting string contains no explicit reference to the radix.

See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures* for further information concerning the exactness of numbers.

Examples:

```
(number->string (/ 15 3))          ↦  "5"
(number->string 1000 2)            ↦  "1111101000"
(number->string 63/115 16)         ↦  "3f/73"
(number->string #x-1c+21/a2i 8)    ↦  "-34+13/66i"
(number->string (sqrt 6.25))       ↦  "2.5"
(number->string (sqrt 6.25) 2)     ↦  "2.5"
```

(numerator *q*) *Procedure*

Returns the numerator of the rational number *q* as an integer (whose exactness matches that of *q*), after having reduced *q* to lowest terms with a positive denominator. (Recall from Chapter 5: *Numbers and Numeric Procedures* that in *EdScheme* no non-integer expressed in decimal or exponential form is rational. See Section 5.9: *Exactness* in that chapter for further information concerning the exactness of numbers.) Examples:

```
(numerator (/ 15 -65))  ↦  -5
(numerator 2e3)         ↦  2000.0
```

(odd? *n*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument (which must be an integer) is odd. Examples:

```
(odd? -4)   ↦  #f      (odd? 5e2)  ↦  #f
(odd? 6/2)  ↦  #t
```

(open-extend-file *spec*) *Procedure*

Opens the existing file specified by the argument, and places the file pointer at the end of the file, ready for output from *EdScheme*. Returns an output port. If you are unsure of the correct file specification, use the Scheme expression

```
(open-extend-file (choose-input-file))
```

and use the File Selector to indicate which file you have in mind. See the entry for the procedure **choose-input-file** earlier in this section.

It may seem strange to see an 'input' procedure being used in this patently 'output' situation. The reason is that the 'natural' choice of **choose-output-file** involves you in more work. This procedure brings up the 'Save As:' File Selector with all the current file names 'grayed out'. You then have to type in the name of the file you wish to extend in the form that it appears in the File Selector, and click on the Yes button when asked 'Replace existing <filename>?'.

Example:

```
(define F (open-extend-file "MyFile"))
                        ↦  unspecified

(begin
  (display "some more info" F)
  (close-port F)
  'done)                ↦  done

;;; Writes the contents of the string "some more info"
;;; to the end of 'MyFile'.
```

(open-input-file *spec*) *Procedure*

Opens the existing file specified by the argument, and places the file pointer at the beginning of the file, ready for input to *EdScheme*. Returns an input port.

If you are unsure of the correct file specification, use the Scheme expression

```
(open-input-file (choose-input-file))
```

and use the File Selector to indicate which file you have in mind. (See the entry for the procedure **choose-input-file** earlier in this section.)

Example:

```
(define F (open-input-file "MyFile"))
                        ↦  unspecified

(let ([contents (string-read F)])
  (close-port F)
  contents)             ↦  a string containing the
                           entire contents of 'MyFile'
```

(open-input-string *str*) *Procedure*

Returns an input port whose contents are those of the input string. This allows data expressions to be read individually from the string.

Used in conjunction with **string-read** (see the relevant entry later in this section), this provides a means of reading the contents of a disk file all at once, making those contents available for subsequent inspection without any further need for accessing the disk.

Examples:

```
(define T (open-input-string "abc 5 (A B C)"))
                                        ↦  unspecified
(read T)                                ↦  abc
(read T)                                ↦  5
(read T)                                ↦  (A B C)
(eof? T)                                ↦  #t

(define S
  (let ([in-port (open-input-file "MyFile")])
    (let ([contents (open-input-string (string-read in-port))])
      (close-port in-port)
      contents)))                       ↦  unspecified

;;; Defines S to be an input port (in RAM) containing the entire
;;; contents of the file 'MyFile'. These contents may then be
;;; accessed as in the first example above, using the procedure
;;; read.
```

(open-output-file *spec*) *Procedure*

Opens a new file specified by the argument, deleting any existing file with the same specification, and places the file pointer at the beginning of the file, ready for output from *EdScheme*. Returns an output port.

If you are unsure of how to specify the file so that it is stored in the desired folder, use the Scheme expression

```
(open-output-file (choose-output-file))
```

and use the File Selector to provide a suitable name in the correct folder. (See the entry for the procedure **choose-output-file** earlier in this section.)

(open-serial-port *sym*) *Procedure*

Opens the serial port specified by the input symbol, which must be either `printer` or `modem`, and returns the port in question. The serial port thus opened is both an input port and an output port, and it may be configured using the procedure **configure-serial-port** (see the relevant entry earlier in this

section). Note, however, that the printer port should only be used for baud rates of 300 or less; there is no such restriction on the modem port.

Both serial ports may be open at the same time, and, once a serial port is open, there is no need to close it. In fact, ***EdScheme*** does nothing in response to the Scheme expression **(close-port *S*)** when ***S*** is a serial port. To flush serial port ***S***, evaluate the Scheme expression **(read *S*)**.

(or *sexp1* ...) *Special Form*

Evaluates the arguments in order of appearance, until one is found whose value is *not* the boolean **#f**, whereupon the value of the Scheme expression in question is returned, leaving any remaining arguments unevaluated. If all the arguments have false values, **#f** is returned.

In Schemer's Guide Mode, all the arguments except the last must evaluate to a boolean object. In Standard Scheme Mode, no such restriction is imposed.

If there are no arguments, then `#f` is returned. Examples:

```
(or)                          ↦  #f
(or (< 2 1) (null? '()))      ↦  #t
(or 1 2 3)                    ↦  1      [Standard Scheme Mode only]
```

(output-port? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is either the Transcript Window, a serial port, or a port at which there is a file that has been opened for output or extension.

(oval *list*) *Procedure*
(oval *list gwin*)

Draws the largest possible ellipse in the rectangle specified by the input ***list*** in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise. The list must be a pair of pairs, specifying the coordinates of either pair of diagonally opposite corners of the rectangle in question. The resulting drawing will be contained entirely within the bounding rectangle, no matter what the size of the pen's 'nib' (see the entry for the procedure **pen-state** later in this section), even if the 'nib' is more extensive than the bounding rectangle.

For example, **(oval '((0 0) (50 50)))** will cause the circle of radius 25 centered at the point (25,25) to be drawn in the most recently active open graphics window.

(oval-paint *list*) *Procedure*
(oval-paint *list gwin*)

Draws a filled ellipse specified by ***list*** and ***gwin*** (if provided) in the manner described under **oval** above. Refer also to the entry earlier in this section for the procedure **arc-paint**; the comments made there concerning the **pen-state** and the appearance of the boundary of the figure apply in this case also.

(pair? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a pair in the technical sense explained in Section 10.3: *Pairs and Lists* in Chapter 10: *Data Expressions*. Examples:

```
(pair? '())       ↦  #f     (pair? 'a)       ↦  #f
(pair? '#(a b))   ↦  #f     (pair? '(a b))   ↦  #t
(pair? '(a . b))  ↦  #t
```

The final example holds in either mode, since

- in Standard Scheme mode, `(a . b)` is an improper list;
- in Schemer's Guide mode, the dot is *not* a reserved character, and in consequence `(a . b)` is viewed as a list containing three data expressions.

(peek-char) *Procedure*
(peek-char *port*)

Returns the next character ready at the specified input ***port***, if provided, or the current input port, otherwise. (See the entry for the procedure **current-input-port** earlier in this section.) The position of the file pointer is *not* altered in any way. If the port is associated with a file that is at end-of-file, or if it is the Transcript Window and no characters are pending, then the end-of-file object `#!eof` is returned.

The input port must *not* be a serial port.

(pen-color) *Procedure*
(pen-color *gwin*)
(pen-colour)
(pen-colour *gwin*)

Returns an RGB triple denoting the current color of the turtle's pen in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise. (See the first section of this chapter for information concerning RGB triples.)

(pen-down) *Procedure*
(pen-down *gwin*)

Causes subsequent turtle movements in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, to leave their trace in the current color of that turtle's pen. (See the above entry for the procedure **pen-color**.)

For greater control over the state of the turtle's pen, use the procedure **pen-set-state**, described later in this section.

(pen-down?) *Procedure*
(pen-down? *gwin*)

A predicate that returns the boolean **#t** if and only if the turtle in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, has its pen down.

For more detailed information concerning the state of the turtle's pen, use the procedure **pen-state**, described later in this section.

(pen-erase) *Procedure*
(pen-erase *gwin*)

Causes subsequent turtle movements in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, to leave

their trace in the background color, thus in effect erasing any portions of existing drawings that the turtle's pen passes over.

(pen-reverse) *Procedure*
(pen-reverse *gwin*)

Causes the color of each point the turtle's pen subsequently passes over in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, to be changed to its logical color complement.

(pen-set-color *rgb*) *Procedure*
(pen-set-color *rgb gwin*)
(pen-set-colour *rgb*)
(pen-set-colour *rgb gwin*)

Sets the color of the turtle's pen in graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, to that given by the triple ***rgb***, with effect from the next movement of the turtle in question. (See the first section of this chapter for information about RGB triples.)

(pen-set-state *list*) *Procedure*
(pen-set-state *list gwin*)

Sets the drawing state of the pen in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, to be as specified by the input ***list***. This list must be of the form described in the entry below for the procedure **pen-state**.

(pen-state) *Procedure*
(pen-state *gwin*)

Returns a list describing the drawing state of the pen in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise. This list contains four data expressions, as follows:

1. an integer in the range −15 through 15 indicating the pen writemode (numbers that differ by 16 have the same significance):

0: pen-erase mode—the turtle colors its track in the background color, thus effectively erasing anything it passes over.

8: pen-down mode—the turtle colors its track in the current pen color.

10: pen-reverse mode—the turtle changes the color of each point it passes over to its reverse (or logical color complement).

−1 (or 15): pen-up mode—the turtle leaves no trace.

2. a number indicating the QuickDraw™ writemode. See page 170 of *Inside The Macintosh, Volume I* for further details.

3. a pair of whole numbers, the first giving the pen width and the second giving the pen height. Together, these two numbers determine the thickness of lines drawn by the turtle. (If the width and height are unequal, then line thickness will vary according to the direction of the line in question.) The pen's 'nib', whose dimensions are specified by this pair, extends to the right and downward from the current turtle position.

4. a list of eight integers in the range −255 through 255 specifying the pen pattern in which regions will be filled. (Numbers that differ by 256 have the same significance.) The numbers represent a binary coding of an 8 by 8 fill pattern. Thus, the 8 by 8 fill pattern at the top of the next page is represented by the list `(60 102 6 12 24 48 126 0)`, whose elements describe the rows of the pattern in order from top to bottom.

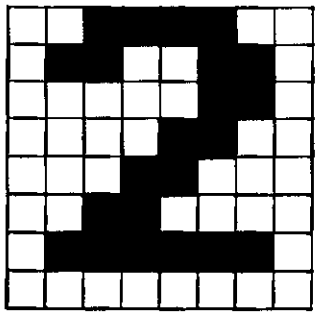
The default pen-state corresponds to the list

```
(8 8 (1 1) (-1 -1 -1 -1 -1 -1 -1 -1))
```

that is, the turtle's pen is down and is one turtle step wide and one turtle step high, and the fill pattern is a solid fill (all the dots in the 8 by 8 fill pattern are colored, giving a binary code for each row of 11111111 = 255 or, alternatively, −1.

(pen-up) *Procedure*
(pen-up *gwin*)

Causes subsequent turtle movements in graphics window *gwin*, if provided,



```
⟶  00111100 =  60
⟶  01100110 = 102
⟶  00000110 =   6
⟶  00001100 =  12
⟶  00011000 =  24
⟶  00110000 =  48
⟶  01111110 = 126
⟶  00000000 =   0
```

Figure 11.1: Coding a Fill Pattern

or the most recently active open graphics window, otherwise, to leave no trace. For greater control over the state of the turtle's pen, use the procedure **pen-set-state**, described above.

---

pi *Variable*

A global variable, bound initially to the value 3.141592653589793, that is, (approximately) the ratio of the circumference of a circle to its diameter. Example:

```
(define degrees->radians
  (lambda (x)
    (/ (* x pi) 180)))   ↦  unspecified

(degrees->radians 150)   ↦  2.617993877991494
```

(pick-color ***str rgb*****)** *Procedure*
(pick-colour ***str rgb*****)**

If Color QuickDraw™ is not present in the system, returns the boolean `#f`.

If Color QuickDraw is present, brings up the **Control Panel** color-picker dialog with the contents of the input string displayed as a prompt in its top left corner, and the pointer initially on the color specified by the input RGB triple. Returns the RGB triple corresponding to the chosen color, if the OK button is pressed, or the boolean `#f`, if the Cancel button is pressed. (For further information about RGB triples, see the first section of this chapter.)

Example (on a Color QuickDraw system):

```
(pick-color "Choose a color ..." '(65535 33343 1237))

;;; Brings up the color-picker dialog with the prompt 'Choose a
;;; color ...' displayed in the top left corner and the pointer
;;; initially on orange; returns an RGB triple or a boolean
;;; object as described above.
```

(point-color *pair*) *Procedure*
(point-color *pair gwin*)
(point-colour *pair*)
(point-colour *pair gwin*)

Returns an RGB triple specifying the color of the point in graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, whose coordinate relative to the current graphics origin in that window is given by the input ***pair***. (For further information about RGB triples, see the first section of this chapter.)

(point-on? *pair*) *Procedure*
(point-on? *pair gwin*)

A predicate that returns the boolean `#t` if and only if the point in graphics window ***gwin***, if provided, or in the most recently active open graphics window, otherwise, whose coordinate relative to the current graphics origin in that window is given by the input ***pair*** has some color other than the background color.

(polygon *list*) *Procedure*
(polygon *list gwin*)

Draws the rectilinear figure in graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, that is obtained when the turtle's 'head' joins the dots whose coordinates are given in the input ***list*** (which must be a list of pairs of numbers). The size and position of the resulting figure's boundary will depend on the current state of the turtle's 'pen' (see the above entry for the procedure **pen-state**), which extends to the

right and down from the turtle's 'head'. To draw a *closed* polygon, make the final pair in the input ***list*** the same as the first.

Example:

```
(polygon '((0 -50) (-50 -50) (-50 50) (0 50)) G)
;;; Draws a squared-off 'C'-shaped figure in graphics window G.
```

(polygon-paint *list*) *Procedure*
(polygon-paint *list gwin*)

Draws the filled polygon in graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, that is specified by the input ***list*** in the manner described under **polygon** above. With **polygon-paint**, however, there is no need for the last coordinate in the input list to be the same as the first; the missing edge of the boundary is supplied automatically before the filling occurs.

The comments concerning the state of the turtle's pen and the appearance of the boundary of the figure, made in the entry earlier in this section for the procedure **arc-paint**, apply in this case also.

(port? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is an open port. (Note that the Transcript Window is a port.)

(positive? *x*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is positive. (The result may be unreliable if the argument is an inexact number that is close to zero. For further information concerning the exactness of numbers, see Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*.) Examples:

```
(positive? 5)  ↦  #t      (positive? -6.25)  ↦  #f
```

(power *z1 z2*) *Procedure*

Returns *z1* to the power *z2*. If *z1* is 0, then *z2* must not be a negative real number—if *z2* is 0, the result is 1; otherwise, the result is 0. If either or both arguments are non-real (and *z1* is not 0), the result is calculated in the manner explained on page 46 in Chapter 5: *Numbers and Numeric Procedures.*

If *z1* is an exact complex number and *z2* is an exact *integer* (with the one proviso mentioned above in the case when *z1* is 0), then the result is exact. Otherwise the result is inexact.

This procedure produces exactly the same results as the procedure **expt**—see the relevant entry earlier in this section. Examples:

```
(power 2 3)       ↦  8
(power 243 -1/5)  ↦  0.3333333333333333
(power 2 5+i)     ↦  24.6156448436471+20.44676084203631i
(power 3-4i -3)   ↦  -117/15625+44/15625i
(power 3-4i 5+i)  ↦  -7847.176113287395-902.8820324360111i
```

(primitive? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is a primitive procedure. Examples:

```
(primitive? first)  ↦  #t      (primitive? '#(a b))  ↦  #f
```

(print-length *exp*) *Procedure*

Returns the number of characters needed in order to print the input expression on the screen. Example:

```
(print-length '(a (b (c (d)))))  ↦  15
```

(procedure? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is a procedure in the current *EdScheme* session. Examples:

```
(procedure? first)                ↦  #t
(procedure? (lambda (x) x))       ↦  #t
(procedure? "modus operandi")     ↦  #f
```

(procedure-definition *proc*) *Procedure*

Returns a lambda-expression equivalent to the input ***proc***, which must be a derived procedure. Example:

```
(define second
  (lambda (s)
    (first (rest s))))            ↦  unspecified

(procedure-definition second)     ↦  (lambda (s) (first (rest s)))
```

(procedure-environment *proc*) *Procedure*

Returns the local environment of the input ***proc***, which must be a derived procedure. Example:

```
(define label
  (let ([x 'tag] [y 'Monday])
    (lambda (s)
      (cons x s))))               ↦  unspecified

(label '(a b))                    ↦  (tag a b)

(procedure-environment label)     ↦  (((x tag) (y Monday)))
```

(See the note on page 103 in Chapter 10, Section 10.12: *Environments* concerning the printing of environments.)

(quasiquote *temp*) *Special Form*

If no commas or occurrences of **unquote** or **unquote-splicing**—see the relevant entries later in this section—occur in the input ***temp***, returns the same value as the Scheme expression **(quote *temp*)**.

If a **,@**-combination (or the **unquote-splicing** special form) occurs in ***temp***, the expression that follows that combination must evaluate to a list, the

elements of which (without the enclosing delimiting parentheses) are inserted into ***temp*** in place of the ,@-combination and the following expression.

If a comma—without an @-sign—(or the **unquote** special form) occurs in ***temp***, then the value of the expression that follows is inserted into ***temp*** in place of the comma and the following expression.

Once all these insertions have been completed, the resulting data expression is returned.

Example:

```
(quasiquote (a ,(cons 'm '()) ,@(rest '(x y z)) end))
                          ↦  (a (m) y z end)
```

The Scheme expression (**quasiquote** ***temp***) may be abbreviated to '***temp***, using a 'backquote' (see the relevant entry on page 114, earlier in this section).

(quit) *Procedure*

Terminates the current ***EdScheme*** session and exits from the application. (This is equivalent to choosing Quit from the File menu.)

(quote *sexp*) *Special Form*

Returns the Scheme expression ***sexp*** without evaluating it. The Scheme expression (**quote** ***sexp***) may be abbreviated by '***sexp*** (see the relevant entry on page 109, earlier in this section). Examples:

```
(quote (* 2 pi))  ↦  (* 2 pi)
(cons 1 '(4 all))  ↦  (1 4 all)
```

(quotient *n1 n2*) *Procedure*

Returns the unique integer q with the same sign as $n1 * n2$ which is such that, for some non-negative integer r strictly less than the absolute value of $n2$,

$$|n1| = (|q| * |n2|) + r,$$

where '$|x|$' denotes the absolute value of x (see the entry for the procedure **abs** earlier in this section). **The second input must not be zero.**

If both inputs are exact, then the result is exact; otherwise, the result is inexact. (See Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures* for further information concerning the exactness of numbers.) Examples:

```
(quotient 17 3)   ↦  5      (quotient -17 3.0)   ↦  -5.0
(quotient 17 -3)  ↦  -5     (quotient -17.0 -3)  ↦  5.0
```

(random *k*) *Procedure*

Returns a pseudo-random integer from 0 through *k*−1; *k* must be an exact positive integer no greater than 32767. Example:

```
(random 4)

;;; Returns one of the numbers 0, 1, 2, or 3.
```

(randomise) *Procedure*
(randomize)

Seeds the random number generator with an integer that depends on the state of the computer's internal clock. This should be called before the procedure random—see above—is used to generate a sequence of random numbers.

(rational? *exp*) *Procedure*

A predicate that returns the boolean #t if and only if its argument is a rational number. Examples:

```
(rational? -3/7)   ↦  #t     (rational? (/ 1+i))      ↦  #f
(rational? 3.5)    ↦  #f     (rational? '#(1 2 3))    ↦  #f
(rational? -6.0)   ↦  #t     (rational? 'fraction)    ↦  #f
```

(rationalize *x1 x2*) *Procedure*

Returns a representation of the **simplest** rational that differs from ***x1*** by no more than the absolute value of ***x2***. (If a, b, c, and d are integers such that b and d are non-zero, a and b are relatively prime, and c and d are relatively prime, then a/b is **simpler** than c/d if and only if $|a| \leq |c|$ and $|b| \leq |d|$,

where $|x|$ denotes the absolute value of x—see the entry for the procedure **abs** earlier in this section.)

If both inputs are exact numbers, the result will be an exact rational; otherwise, the result will be inexact. If you require the result to be an exact rational independently of the exactness of the inputs, then you should use the procedure **exact-rationalize**, described earlier in this section. (For further information about the exactness of numbers, see Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*.)

Examples:

```
(rationalize pi 1/1000000) ↦ 3.141592920353982
(rationalize 2/5 1/10)     ↦ 1/2
(rationalize 6.25 -1/5)    ↦ 6.333333333333333
```

(read) *Procedure*
(read *port*)

Returns the next data expression from the input ***port***, if provided, or the current input port, otherwise—see the entry for the procedure **current-input-port** earlier in this section—updating the port to the first character following that data expression.

If an end-of-file marker is reached before a character is located that could begin a data expression, or if the ***port*** is already at end-of-file, then the end-of-file object **#!eof** is returned.

If an end-of-file marker is reached before a data expression is complete, an error is reported.

If ***port*** is a serial port, ***EdScheme*** returns the entire contents of the serial port's input buffer in the form of a string. If the input buffer is empty, then the empty string is returned. Sometimes, however, **read** may appear to return the empty string even though the input buffer is not empty. This is because the buffer may contain some non-printing characters, a zero, for example. It is therefore advisable when using **read** to access input from a serial port to pass the resulting string through the procedure **string->list** (see the relevant entry later in this section).

If ***port*** is specified as the Transcript Window, or if no port is specified and the current input port is the Transcript Window, then ***EdScheme*** waits while

you enter characters at the keyboard. As soon as you hit a carriage return after having completed the data expression that began with the first non-whitespace character you entered, ***EdScheme*** returns the completed data expression. Example:

```
(read)

(a b "c
  d"
  e) f #(g h)   ↦   (a b "c
                       d" e)
```

(read-char) *Procedure*
(read-char *port*)

Returns the next character from the input ***port***, if provided, or the current input port, otherwise—see the entry for the procedure **current-input-port** earlier in this section—updating the port to the following character.

If the ***port*** is at end-of-file, then the end-of-file object **#!eof** is returned, and the port remains at end-of-file.

If ***port*** is a serial port, then ***EdScheme*** returns the next character in the input buffer or, if there are no characters in the buffer, a (non-printing) zero character.

If ***port*** is specified as the Transcript Window, or if no port is specified and the current input port is the Transcript Window, then ***EdScheme*** waits for you to enter a character at the keyboard. As soon as you do so, ***EdScheme*** returns the character in question, without displaying your keypress on the screen.

(read-line) *Procedure*
(read-line *port*)

If ***port*** is associated with an input file, or if no port is specified but the current input port is associated with an input file—see the entry for the procedure **current-input-port** earlier in this section—then the portion of the contents of that file from the current file position up to (but not including) the next newline character or the end-of-file marker (whichever comes first) is returned as a string, and the port is updated to the first character of the next

line or to the end-of-file, respectively. If ***port*** is already at end-of-file, then the end-of-file object `#!eof` is returned, and the port remains at end-of-file.

If ***port*** is a serial port, then **read-line** behaves exactly as the procedure **read** in similar circumstances (see the relevant entry above).

If ***port*** is specified as the Transcript Window, or if no port is specified and the current input port is the Transcript Window, then ***EdScheme*** waits while you enter characters at the keyboard. As soon as you hit a carriage return, ***EdScheme*** returns a string containing everything you have typed (not including the carriage return).

(real? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a real number. (See Chapter 5: *Numbers and Numeric Procedures* for information concerning the types of numbers ***EdScheme*** supports.) Examples:

```
(real? -3/7)      ↦  #t      (real? #o25)     ↦  #t
(real? 3@1.57)    ↦  #f      (real? 1+i)      ↦  #f
(real? '(a b))    ↦  #f      (real? #\5)      ↦  #f
(real? 15##)      ↦  #t
```

(real-part *z*) *Procedure*

Returns the real part of the complex number ***z***. (For information concerning ***EdScheme***'s treatment of complex numbers, see Chapter 5: *Numbers and Numeric Procedures*.) Examples:

```
(real-part 1.2)                      ↦  1.2
(real-part 1-i)                      ↦  1
(real-part (/ 3-4i))                 ↦  3/25
(real-part (make-polar 3 (/ pi 3)))  ↦  1.5
```

(rec *var sexp*) *Special Form*

A binding for the variable ***var*** to an unspecified value is added to the current environment, the Scheme expression ***sexp*** is evaluated in this extended environment, the value to which ***var*** is bound is changed to the resulting value, and the value of ***var*** in the resulting new environment is returned.

Usually, ***sexp*** evaluates to a procedure that is recursive. The environment of such a procedure contains a self-reference, and in consequence the procedure will execute more rapidly than would otherwise be the case, since the recursive calls do not require a search of the global environment. Furthermore, procedures defined in this way may be 'renamed' using the **define** special form.

The Scheme expression **(rec *var sexp*)** is equivalent to the Scheme expression **(letrec ([*var sexp*]) *var*)**. (See the entry for the **letrec** special form earlier in this section.)

(remainder *n1 n2*) *Procedure*

Returns the unique integer r strictly between $n2$ and $-n2$ such that $r * n1$ is positive and, for some integer q,

$$n1 = (q * n2) + r,$$

that is, the remainder—with the same sign as $n1$—when $n1$ is divided by $n2$. The second input must not be zero. The result is exact if both inputs are exact, but inexact otherwise. Examples:

```
(remainder 14 3)    ↦  2      (remainder 14 -3.0)  ↦  2.0
(remainder -14 -3)  ↦  -2     (remainder -14 3.0)  ↦  -2.0
```

(See also the procedure **modulo**, which is described earlier in this section, and behaves slightly differently.)

(repeat *k sexp1* ...) *Special Form*

Evaluates the Scheme expressions ***sexp1***, ..., in order of appearance ***k*** times, and returns the value of the last Scheme expression (or some unspecified value, if there are no Scheme expressions).

Example (in Degree Mode):

```
(repeat 3 (forward 100 G) (right 120 G) 'done)    ↦  done

;;; Draws an equilateral triangle with sides 100 turtle steps
;;; long in graphics window G.
```

(reset) *Procedure*

Resets ***EdScheme***, returning the initial and global environments to their original states, and re-loading the file 'EdScheme Init.s', if it is present.

(rest *pair*) *Procedure*

Returns the second component of the input ***pair***. Examples:

```
(rest '(a b c))          ↦  (b c)
(rest '(((a) b) (c d)))  ↦  ((c d))
```

This procedure behaves in exactly the same way as the procedure **cdr** (see the relevant entry earlier in this section).

(reverse *list*) *Procedure*

Returns a list containing the data expressions in the input ***list*** in the reverse order. Example:

```
(reverse '(a (b c) (d (e))))  ↦  ((d (e)) (b c) a)
```

(right *x*) *Procedure*
(right *x gwin*)

Adds ***x*** radians or degrees (depending on Angle Mode you have chosen in the **Language** dialog accessed through the **Preferences** sub-menu in the **File** menu) to the current heading of the turtle in graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, adjusting the direction of the turtle on screen if it is currently being shown.

(round *x*) *Procedure*

Returns the integer that is closest to the real number ***x***, choosing the even possibility if two integers are equally close. The result is exact if and only if the input is exact. (For further information about the exactness of numbers, see Section 5.9: *Exactness* in Chapter 5: *Numbers and Numeric Procedures*.) Examples:

(round -17/2)	↦	-8	**(round 3.5)**	↦	4.0
(round *pi*)	↦	3.0	**(round -355/113)**	↦	-3

(runtime) *Procedure*

Returns the current value of the system clock as an inexact number (rounded to the nearest hundredth of a second).

(seed *k*) *Procedure*

Seeds the ***EdScheme*** random number generator with the integer ***k***, which must be non-negative and less than 65536.

(set! *var sexp*) *Special Form*

Evaluates the Scheme expression ***sexp*** in the current environment, assigns the resulting value to the variable ***var*** (for which there must already be a binding in the current environment) and returns an unspecified value. Example:

(define *x* 9)	↦	*unspecified*
x	↦	9
(set! *x* (* *x* *x*))	↦	*unspecified*
x	↦	81

(set-car! *pair exp*) *Procedure*

Sets the first component of the input ***pair*** to be ***exp***, returning an unspecified value. (See the entry for the procedure **set-first!** below.)

(set-cdr! *pair exp*) *Procedure*

Sets the second component of the input ***pair*** to be ***exp***, returning an unspecified value. (See the entry for the procedure **set-rest!** below.)

(set-first! *pair exp*) *Procedure*

Sets the first component of the input ***pair*** to be ***exp***, returning an unspecified value. Example:

```
(define L '(a b c))      ↦  unspecified
L                        ↦  (a b c)
(set-first! L '(a b c))  ↦  unspecified
L                        ↦  ((a b c) b c)
```

(See also the procedure **set-car!**, described above.)

(set-rest! *pair exp*) *Procedure*

Sets the second component of the input ***pair*** to be ***exp***, returning an unspecified value. Example:

```
(define L '(a b c))      ↦  unspecified
L                        ↦  (a b c)
(set-rest! L '(1 2 3))   ↦  unspecified
L                        ↦  (a 1 2 3)
```

(See also the procedure **set-cdr!**, described above.)

(set-volume *k*) *Procedure*

Sets the current volume to the one whose reference number is ***k***. If ***k*** is not the reference number of one of the volumes recognized by the operating system, then an operating system error message is generated. To avoid this happening, use a Scheme expression such as the following:

```
(let ([port (open-input-file (choose-input-file))])
  (set-volume (last-volume))
  (close-port port)
  'done)
```

and, using the File Selector, select any file that is in the desired volume.

(sin *z*) *Procedure*

Returns the sine of ***z*** as an inexact number. If ***z*** is real, it is interpreted as being in radians or degrees, according to the Angle Mode you have chosen in the **Language** dialog accessed through the **Preferences** sub-menu in the **File** menu. If ***z*** is not real, then its sine is calculated in the manner explained on **page 46 in Chapter 5: *Numbers and Numeric Procedures*. Examples:**

```
(sin 90)          ↦  1.0                          [in Degree Mode]
(sin (/ pi 6))    ↦  0.5                          [in Radian Mode]
(sin 3-4i)        ↦  3.853738037919377+27.01681325800393i
                                                  [in either mode]
```

(sqrt *z*) *Procedure*

If *z* is real, returns the positive square root of *z*. If *z* is not real, returns the square root such that either its real part is positive or its real part is zero and its imaginary part is non-negative. In all cases, the result is inexact. (For further information about *EdScheme*'s handling of complex numbers, see Chapter 5: *Numbers and Numeric Procedures*; in particular, Section 5.9: *Exactness* provides more information about the exactness of numbers.) Examples:

```
(sqrt 4)      ↦  2.0          (sqrt -6.25)  ↦  0+2.5i
(sqrt 3-4i)   ↦  2.0-1.0i
```

(stop-alert *str*) *Procedure*

Activates a Macintosh stop alert box containing the contents of the input string. The size of this box is set by *EdScheme* and the text of the alert automatically word-wraps to fit inside the box.

(string *ch1* ...) *Procedure*

Returns a string whose contents comprise all the arguments in order of appearance. Example:

```
(string #\O #\space #\h #\i #\!)  ↦  "O hi!"
```

(string? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a string. Examples:

```
(string? "twine and thread")  ↦  #t
(string? 'twine-and-thread)   ↦  #f
```

(string<? *str1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in strict lexicographic order based on the **char<?** comparison (see the relevant entry earlier in this section). Examples:

```
(string<? "left" "margin" "wide")  ↦  #t
(string<? "Ray" "Raymond")         ↦  #t
(string<? "down" "UP")             ↦  #f
(string<? "vest")                  ↦  #t
(string<?)                         ↦  #t
```

(string<=? *str1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in non-strict lexicographic order based on the **char<?** comparison (see the relevant entry earlier in this section). Examples:

```
(string<=? "teeny" "teeny" "weeny")  ↦  #t
(string<=? "polka-dot" "bikini")     ↦  #f
(string<=? "Bean")                   ↦  #t
(string<=?)                          ↦  #t
```

(string=? *str1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if all its arguments are, character for character, equal strings. Examples:

```
(string=? "abc" "abc" (string #\a #\b #\c))  ↦  #t
(string=? "Abc" "abc")                       ↦  #f
(string=? "abc")                             ↦  #t
(string=?)                                   ↦  #t
```

(string>? *str1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in strict lexicographic order based on the **char>?** comparison (see the relevant entry earlier in this section). Examples:

```
(string>? "right" "margin" "fat")  ↦  #t
(string>? "Jo" "Josephine")        ↦  #f
(string>? "back" "DOWN")           ↦  #t
(string>? "Gee!")                  ↦  #t
(string>?)                         ↦  #t
```

(string>=? *str1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in non-strict lexicographic order based on the **char>?** comparison (see the relevant entry earlier in this section). Examples:

```
(string>=? "that" "binds" "Together")  ↦  #t
(string>=? "him" "up")                 ↦  #f
(string>=? "orchestra")                ↦  #t
(string>=?)                            ↦  #t
```

(string-append *str1* ...) *Procedure*

Returns the string obtained by concatenating all the input strings, in order. Example:

```
(string-append "back" "to" "front")  ↦  "backtofront"
```

(string-ci<? *str1* ...) *Procedure*

A predicate that returns the boolean `#t` if and only if its arguments are in strict lexicographic order based on the **char-ci<?** comparison (see the relevant entry earlier in this section). Examples:

```
(string-ci<? "Ab" "ab" "xyz")  ↦  #f
(string-ci<? "abc" "AbcD")     ↦  #t
(string-ci<? "DOG")            ↦  #t
(string-ci<?)                  ↦  #t
```

(string-ci<=? *str1* ...) *Procedure*

A predicate that returns the boolean **#t** if and only if its arguments are in non-strict lexicographic order based on the **char-ci<?** comparison (see the relevant entry earlier in this section). Examples:

```
(string-ci<=? "What" "what" "WHAT?")  ↦  #t
(string-ci<=? "No" "GO")              ↦  #f
(string-ci<=? "K")                    ↦  #t
(string-ci<=?)                        ↦  #t
```

(string-ci=? *str1* ...) *Procedure*

A predicate that returns the boolean **#t** if and only if its arguments are, character for character, equal strings, when the distinction between upper- and lower-case letters is ignored. Examples:

```
(string-ci=? "Abc" "abc" (string #\a #\B #\C))  ↦  #t
(string-ci=? "AbcD" "abc")                      ↦  #f
(string-ci=? "solo")                            ↦  #t
(string-ci=?)                                   ↦  #t
```

(string-ci>? *str1* ...) *Procedure*

A predicate that returns the boolean **#t** if and only if its arguments are in strict lexicographic order based on the **char-ci>?** comparison (see the relevant entry earlier in this section). Examples:

```
(string-ci>? "sloppy" "Organization" "EVERYWHERE")  ↦  #t
(string-ci>? "except" "HERE")                       ↦  #f
(string-ci>? "tie")                                 ↦  #t
(string-ci>?)                                       ↦  #t
```

(string-ci>=? *str1* ...)

A predicate that returns the boolean **#t** if and only if its arguments are in non-strict lexicographic order based on the **char-ci>?** comparison (see the relevant entry earlier in this section). Examples:

```
(string-ci>=? "Slide" "downhill" "BACKwards")  ↦  #t
(string-ci>=? "Scared" "stiff!")               ↦  #f
(string-ci>=? "CRASH")                         ↦  #t
(string-ci>=?)                                 ↦  #t
```

(string-copy *str*) *Procedure*

Returns a copy of the input string. This is useful for preserving a copy of a string that may subsequently be changed permanently by a modifier procedure such as **string-set!** (see the relevant entry later in this section). Example:

```
(define S "abcde")          ↦  unspecified
(define T (string-copy S))  ↦  unspecified
(string-set! S 2 "z")       ↦  unspecified
S                           ↦  "abzde"
T                           ↦  "abcde"
```

(string->expression *str*) *Procedure*

Returns the first data expression in the input string. If the string ends before the data expression that begins with the first of the string's characters is completed, an error is signalled.

Examples:

```
(string->expression "(1 list) 2 many")  ↦  (1 list)
(string->expression "#(a \"b\") c")     ↦  #(a "b")
```

(string-fill! *str ch*) *Procedure*

Sets all the characters of the input string to be ***ch***. Returns an unspecified value. Example:

```
(define S "hog tied")  ↦  unspecified
(string-fill! S #\x)   ↦  unspecified
S                      ↦  "xxxxxxxx"
```

(string-length *str*) *Procedure*

Returns the number of characters in the input string. Example:

```
(string-length "one two")  ↦  7
```

(string->list *str*) *Procedure*

Returns a list of the characters that make up the input string.

If you would prefer the output list to contain symbols rather than characters, then use the procedure **explode** instead (see the relevant entry earlier in this section).

Example:

```
(string->list "Split up")
          ↦  (#\S #\p #\l #\i #\t #\space #\u #\p)
```

(string->number *str*) *Procedure*
(string->number *str rad*)

If the contents of the input string may be interpreted as an exact number with an explicit radix prefix (see the first two examples below), or as an inexact number, returns a number equal in value to the contents of ***str***, ignoring the ***rad*** input, if provided. (The result is displayed on the screen in radix 10 notation.)

If the contents of the input string may be interpreted as an exact number without an explicit radix prefix, returns a number equal in value to the contents of the input string, when interpreted as a number in radix ***rad***, if provided, or radix 10, otherwise. (Again, the result is displayed on the screen in radix 10 notation.)

In the potentially ambiguous case when the input string's contents includes no explicit radix prefix, but features one of the exponent markers `d`, `e`, `f` that double as hexadecimal digits, the string's contents are interpreted as a hexadecimal number if ***rad*** is 16 and as an inexact decimal number if ***rad*** is either 2, 8, 10, or not provided. (See the sixth through eighth examples below.)

If the string's contents are not interpretable as a number, or they are not interpretable as a number in the specified radix, then the boolean **#f** is returned.

For further information concerning *EdScheme*'s representation of numbers, see Chapter 5: *Numbers and Numeric Procedures*.

Examples:

```
(string->number "#o123")      ↦  83
(string->number "#o123" 16)   ↦  83
(string->number "123" 8)      ↦  83
(string->number "123" 16)     ↦  291
(string->number "4.9e3")      ↦  4900.0
(string->number "2f3" 16)     ↦  755
(string->number "2f3" 8)      ↦  2000.0
(string->number "2f3")        ↦  2000.0
(string->number "15##" 8)     ↦  1500.0
(string->number "ten")        ↦  #f
(string->number "29" 8)       ↦  #f
```

(string-read *port*) *Procedure*
(string-read *port k*)
(string-read *xwin*)
(string-read *xwin k*)

If the first input is a port, reads the contents of the file at that port, starting at the current file position, and

- if an optional argument ***k*** is provided and its value is less than the number of characters to the end of the file, ending ***k*** characters later;
- otherwise, ending with the final character before the end-of-file marker.

If the file position before **string-read** is applied is ***k1***, then the file position afterward will be at end-of-file (if the optional second input is not provided), or whichever is the earlier of ***k1*** + ***k*** and the end of the file (if it is).

If the first input is a text window, reads the contents of that window, beginning from the start of the current text selection, and

- if an optional argument ***k*** is provided and its value is less than the number of characters to the end of the window's contents, ending ***k*** characters later;
- otherwise, ending with the final character in the window.

When **string–read** is used to read from a text window, the text selection in that window is left unchanged (see Chapter 8: *Text Windows and User Menus* and the entry for the procedure **text–set–selection** later in this section).

Returns the result as a string. The optional input ***k*** must be positive and less than 2 to the power 31 (that is, 2147483648).

For example, to read the entire contents of a text window ***T***, use

```
(text–set–selection '(0 0) T)
(define contents (string–read T))
```

(string–ref *str k*) *Procedure*

Returns the character with index ***k*** in the input string, where the first character has index 0, the second has index 1, and so on. The exact integer ***k*** must be non-negative and less than the length of the string. Example:

```
(string–ref "abcde" 3)  ↦  #\d
```

(string–set! *str k ch*) *Procedure*

Sets the character with index ***k*** in the input string to be ***ch***, returning an unspecified value. Note that the first character of a string has index 0, the second has index 1, and so on. The exact integer ***k*** must be non-negative and less than the length of the string. Example:

```
(define S "abcde")          ↦  unspecified
S                           ↦  "abcde"
(string–set! S 2 #\newline) ↦  unspecified
S                           ↦  "ab
                               de"
```

(string->symbol *str*) *Procedure*

Returns the contents of the input string as a symbol. This procedure allows you to create symbols containing special characters or, if you have not set the Case Sensitive Mode in the Language dialog accessed through the Preferences sub-menu in the File menu, containing characters not in the default case. Example:

```
(string->symbol "one two")  ↦  one two
                               ;;; Note that this symbol contains
                               ;;; a space character.
```

(string-width *str*) *Procedure*
(string-width *str gwin*)

Returns, as an exact number of pixels, the width of the input string if it were to be printed in the current font style in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise. The information provided by this procedure is useful if you want to center text relative to a given vertical line in a graphics window.

(string-write *str port*) *Procedure*
(string-write *str xwin*)

Writes the input string ***str*** to the specified port or text window.

In the case of writing to a text window, the string is *inserted* at the start of the current text selection, and the text selection is reset by adding ***k*** to both start and finish, where ***str*** is ***k*** characters long. (See Chapter 8: *Text Windows and User Menus* for information about text selections in text windows, and later in this section for an entry describing the procedure **text-set-selection**.)

In the case of writing to a port, the string is placed in the file starting at the current file position, *overwriting* the next ***k*** characters (where ***str*** is ***k*** characters long) and extending the file if necessary.

(sub1 *x*) *Procedure*

Returns $x - 1$ with the same exactness as x. (See Section 5.9: *Exactness* in

Chapter 5: *Numbers and Numeric Procedures* for information concerning the exactness of numbers.) Examples:

```
(sub1 10)                          ↦  9
(sub1 -8/7)                        ↦  -15/7
(sub1 3.4)                         ↦  2.4
(number->string (sub1 65536) 16)   ↦  "ffff"
```

(substring *str k1 k2*) *Procedure*

Returns the substring of ***str*** starting at the character with index ***k1*** and ending at the character with index ***k2***−1. (Strings are 'zero-indexed'; that is, the first character has index 0, the second has index 1, and so on.)

k1 and ***k2*** must be non-negative integers, each no greater than the length of ***str***, and such that ***k1*** is less than or equal to ***k2***.

Examples:

```
(substring "abcde" 2 4)  ↦  "cd"
(substring "abcde" 3 3)  ↦  ""
```

(substring-copy! *str1 k str2*) *Procedure*

Sets the substring of ***str1*** starting at the character with index to be the string ***str2***. The number of characters replaced is equal to the lesser of the length of ***str2*** and the number of characters there are in ***str1*** from the character with index ***k*** to the end of the string. (Strings are 'zero-indexed'; that is, the first character has index 0, the second has index 1, and so on.)

k must be a non-negative integer strictly less than the length of ***str1***.

Examples:

```
(define S "abcdefgh")          ↦  unspecified
S                              ↦  "abcdefgh"
(substring-copy! S 3 "zyx")    ↦  unspecified
S                              ↦  "abczyxgh"
(substring-copy! S 6 "new")    ↦  unspecified
S                              ↦  "abczyxne"
```

(substring-fill! *str k1 k2 ch*) *Procedure*

Sets all the characters of ***str*** from the one with index ***k1*** through the one with index ***k2***−1 to be ***ch***. (Strings are 'zero-indexed'; that is, the first character has index 0, the second has index 1, and so on.)

k1 and ***k2*** must be non-negative integers, each no greater than the length of ***str***, and such that ***k1*** is less than or equal to ***k2***.

Examples:

```
(define S "abcdefgh")          ↦  unspecified
S                              ↦  "abcdefgh"
(substring-fill! S 2 6 "w")    ↦  unspecified
S                              ↦  "abwwwwgh"
(substring-fill! S 5 8 #\z)    ↦  unspecified
S                              ↦  "abwwwzzz"
```

(substring-find *str1 k1 k2 str2*) *Procedure*
(substring-find *str1 k1 k2 str2 bool*)

If the boolean input is not provided or is `#t`, returns a number giving the index of the leftmost character of ***str1*** from the one with index ***k1*** through the one with index ***k2***−1 that matches a character in ***str2***, or returns the boolean `#f` if no such match is found.

If the boolean input is `#f`, returns the index of the rightmost character of ***str1*** from the one with index ***k1*** through the one with index ***k2***−1 that matches a character in ***str2***, or returns the boolean `#f` if no such match is found.

(Strings are 'zero-indexed'; that is, the first character has index 0, the second has index 1, and so on.)

k1 and ***k2*** must be non-negative integers, each no greater than the length of ***str1***, and such that ***k1*** is less than or equal to ***k2***.

Examples:

```
(substring-find "abcdabcdabcd" 3 9 "brat")      ↦  4
(substring-find "abcdabcdabcd" 3 9 "brat" #f)   ↦  8
```

(symbol? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a symbol. Examples:

```
(symbol? 'word)  ↦  #t      (symbol? "abc")      ↦  #f
(symbol? 123)    ↦  #f      (symbol? #\a)        ↦  #f
(symbol? '(a))   ↦  #f      (symbol? '#(1 2 3))  ↦  #f
(symbol? '())    ↦  #f
```

(symbol->string *sym*) *Procedure*

Returns the name of the input symbol as a string, using—with one exception that is explained below—the default standard case for alphabetic characters if that name includes any such characters and you have *not* chosen the Case Sensitive Mode in the Language dialog accessed through the Preferences submenu in the File menu.

The one exception to the case-modification just described is as follows: If the input symbol has been returned by the procedure **string->symbol** (see the relevant entry earlier in this section), then **symbol->string** returns the string using the original cases, irrespective of the Case Sensitivity setting. Examples:

```
(symbol->string 'UnEvEn)
          ↦  "UnEvEn"    [with Case Sensitivity set]
          ↦  "uneven"    [with Lower Case set]
          ↦  "UNEVEN"    [with Upper Case set]
(symbol->string
  (string->symbol "UnEvEn"))
          ↦  "UnEvEn"    [in every case]
```

(tail *stm*) *Procedure*

Returns the (evaluated) second component of the input stream. Example:

```
(define A$
  (cons-stream 1 (cons-stream 2 3)))  ↦  unspecified
(head (tail A$))                      ↦  2
(tail (tail A$))                      ↦  3
```

(tan *z*) *Procedure*

Returns the tangent of *z* as an inexact number. If *z* is real, it is interpreted as being in radians or degrees, according to the Angle Mode you have chosen in the Language dialog accessed through the Preferences sub-menu in the File menu. If *z* is not real, then its tangent is calculated in the manner explained on page 46 in Chapter 5: *Numbers and Numeric Procedures*. (There are restrictions on which numbers are suitable inputs to this procedure. See Chapter 5 for more details.) Examples:

```
(tan 135)          ↦  -1.0                          [in Degree Mode]
(tan (/ pi -3))    ↦  -1.732050807568877            [in Radian Mode]
(tan -1+2i)        ↦  -0.03381282607989669+1.014793616146634i
                                                    [in either mode]
```

(text-alignment) *Procedure*
(text-alignment *xwin*)

Returns an integer specifying the type of text alignment currently in force in the text window *xwin*, if provided, or the most recently active open text window, otherwise. The resulting integer is to be interpreted as follows:

0: flush left text 1: centered text −1: flush right text

(text-char-closest *pair*) *Procedure*
(text-char-closest *pair xwin*)

Returns the index of the character in the text window *xwin*, if provided, or the most recently active open text window, otherwise, that is closest (to the left and on the same line) to the point whose coordinate in that window is given by the specified pair of whole numbers.

The input *pair* must be a list of two numbers; it will usually be generated by the Scheme expression **(first (mouse-state))**—see the entry for the procedure **mouse-state** earlier in this section.

The contents of text windows are 'zero-indexed'; that is, the first character has index 0, the second has index 1, and so on.

(text-clean) *Procedure*
(text-clean *xwin*)

Wipes all the contents from the text window ***xwin***, if provided, or the most recently active open text window, otherwise, and sets the text selection to `(0 0)`. (See the entry below for the procedure **text-set-selection**.)

(text-clear) *Procedure*
(text-clear *xwin*)

Deletes the current text selection from the text window ***xwin***, if provided, or the most recently active text window, otherwise. (See the entry below for the procedure **text-set-selection**.) If the text selection prior to carrying out this operation was (***k1 k2***), then it is reset to (***k1 k1***).

(text-contents) *Procedure*
(text-contents *xwin*)

Returns the entire contents of the text window ***xwin***, if provided, or the most recently active text window, otherwise, as a string. The current text selection—see the entry below for the procedure **text-set-selection**—remains unchanged.

(text-copy) *Procedure*
(text-copy *xwin*)

Replaces the current contents of the clipboard by a copy of the current text selection from the text window ***xwin***, if provided, or the most recently active open text window, otherwise. (See the entry below for the procedure **text-set-selection**.) The current text selection remains unchanged.

(text-cut) *Procedure*
(text-cut *xwin*)

Cuts the current text selection from the text window ***xwin***, if provided, or the most recently active text window, otherwise, replacing the contents of the clipboard by a copy of the selected text and deleting it from the window.

(See the entry below for the procedure **text-set-selection**.) If the text selection prior to carrying out this operation was (***k1 k2***), then it is reset to (***k1 k1***).

(text-display *exp*) *Procedure*
(text-display *exp xwin*)

Displays the data expression ***exp*** in the text window ***xwin***, if provided, or the most recently active open text window, otherwise, inserting it immediately before the first character of the current text selection (see the entry below for the procedure **text-set-selection**).

If the text selection prior to carrying out this operation was (***k1 k2***), then it is reset to (***k1+n k2+n***), where ***n*** is the print length of ***exp*** (see the entry earlier in this section for the procedure **print-length**).

(text-length) *Procedure*
(text-length *xwin*)

Returns the number of characters of text in the text window ***xwin***, if provided, or the most recently active open text window, otherwise.

(text-lines) *Procedure*
(text-lines *xwin*)

Returns the number of lines of text in the text window ***xwin***, if provided, or the most recently active open text window, otherwise.

If the final character in the window is a newline character, then, irrespective of the current text selection (see the entry below for the procedure **text-set-selection**), the output from this procedure will be greater than the number of 'visible' lines by the number of final newline characters.

(text-paste) *Procedure*
(text-paste *xwin*)

Pastes the contents of the clipboard into the text window ***xwin***, if provided, or the most recently active open text window, otherwise.

If the print length of the inserted text (that is, the contents of the clipboard) is ***n*** and the text selection prior to carrying out this operation is (***k1 k2***),

then the selected text—irrespective of its length—is *replaced* by the inserted text, and the text selection becomes (***k1+n k1+n***), that is, the caret is placed at the end of the newly-inserted text. (See the entry earlier in this section for the procedure **print-length** and the entry below for the procedure **text-set-selection**.)

(text-readline) *Procedure*
(text-readline *xwin*)

Selects text window ***xwin***, if provided, or the most recently active text window, otherwise—see the entry for the procedure **window-select** later in this section—and waits for you to type something. As you type, the characters you are entering appear in the text window, starting after the last character of the current text selection—see the entry below for the procedure **text-set-selection**—and overprinting anything that is there already. If, however, you reach the right margin of the text window, what you type stops being echoed into the window. As soon as you enter a newline character, ***EdScheme*** returns a string containing everything you have typed, except for the final newline character. Furthermore, what you have typed disappears from the text window, anything that had been overprinted is reinstated, and the text selection remains the same as it was before the call to **text-readline**.

(text-scroll-to-selection) *Procedure*
(text-scroll-to-selection *xwin*)

Scrolls the text window ***xwin***, if provided, or the most recently active open text window, otherwise, so that the current text selection—or caret, if the text selection is of the form (*n n*)—is visible. (See the entry below for the procedure **text-set-selection**.)

(text-selection) *Procedure*
(text-selection *xwin*)

Returns the current text selection in the text window ***xwin***, if provided, or the most recently active open text window, otherwise. (See the entry below for the procedure **text-set-selection**.)

(text-set-alignment *k*) *Procedure*
(text-set-alignment *k xwin*)

Sets the type of text alignment in the text window ***xwin***, if provided, or the most recently active open text window, otherwise. The alignment setting goes into immediate effect on the entire contents of the window, including any material entered prior to the change of setting.

The input ***k*** must be one of −1, 0, or 1, the significance of each value being as described under **text-alignment** above. The default alignment is flush left, which corresponds to a **text-alignment** setting of 0.

(text-set-contents *str*) *Procedure*
(text-set-contents *str xwin*)

Sets the contents of the text window ***xwin***, if provided, or the most recently active open text window, otherwise, to be the contents of the given string. Changes the text selection (see the entry below for the procedure **text-set-selection**) to (***k k***), where ***k*** is the number of characters in the given string, that is, the caret in the text window is placed at the end of the text.

(text-set-selection *pair*) *Procedure*
(text-set-selection *pair xwin*)

Selects the section of text indicated by the input ***pair*** in the text window ***xwin***, if provided, or the most recently active open text window, otherwise.

The input ***pair*** must be a list of two whole numbers, each no greater than the total number of characters in the text window in question, and such that the first is less than or equal to the second.

If this ***pair*** is (***k1 k2***), then the selected text comprises all the characters in the window from the one with index ***k1*** through the one with index ***k2*** − 1. (The contents of text windows are 'zero-indexed'; that is, the first character has index 0, the second has index 1, and so on.)

If the text window is the selected window—see the entry for the procedure **window-select** later in this section—and its current scroll position permits, the selected text will be highlighted in the window as soon as ***EdScheme*** returns to its top-level or following the next call to the procedure **event**—see the relevant

entry earlier in this section—whichever comes sooner. Whether visible or not, it may be cut, copied, cleared, and so on, under program control using **text-cut**, **text-copy**, **text-clear** (see the relevant entries above), or by the standard mouse-and-menu methods.

If ***k1*** and ***k2*** are *equal*, then the effect is to position the caret immediately *before* the character with index ***k1***. Thus, you may position the caret at the start of text window ***T*** by evaluating the Scheme expression

(text-set-selection '(0 0) *T*).

(text-set-spacing *k*) *Procedure*
(text-set-spacing *k xwin*)

Sets the line spacing in the text window ***xwin***, if provided, or the most recently active open text window, otherwise, to that indicated by the input ***k***. This input must be one of the numbers 1, 2, or 3, which are to be interpreted as described under **text-spacing** below.

(text-spacing) *Procedure*
(text-spacing *xwin*)

Returns an integer specifying the current line spacing in the text window ***xwin***, if provided, or the most recently active open text window, otherwise. The significance of this integer is as follows:

1: single spacing 2: double spacing 3: triple spacing

(text-window) *Procedure*

Returns (the representation of) the most recently active open text window, or the boolean **#f** if no text window is open.

(text-window? *exp*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is a text window.

(thaw *proc*) *Procedure*

Evaluates its argument, which must be a thunk—usually this will have been generated using the procedure **freeze** (see the relevant entry earlier in this section)—and returns the result. Examples:

```
(thaw (lambda () 'yo!))                ↦  yo!
(define F (freeze (cons 'a '(1 2 3)))) ↦  unspecified
(thaw F)                               ↦  (a 1 2 3)
```

the-empty-stream *Variable*

A variable that is bound initially to the empty list.

(the-environment) *Procedure*

Returns a representation of the current environment. (See the note in Chapter 10, Section 10.12: *Environments*, concerning the printing of environments.) Example:

```
(let ([a 2] [b 7]) (the-environment))  ↦  (((a 2) (b 7)))
```

(towards *pair*) *Procedure*
(towards *pair gwin*)

Returns an inexact number that gives the heading (in radians or degrees, according to which Angle Mode you have chosen in the Language dialog accessed through the Preferences sub-menu in the File menu) required to aim the turtle in graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, at the point with coordinate indicated by the input ***pair***—which must be a list of two numbers. (See the entry for the procedure **turtle-set-heading** later in this section.) Example:

```
(turtle-set-heading (towards '(100 -50) G) G)
;;; Aims the turtle in graphics window G at the
;;; point (100,-50).
```

(trace *proc*) *Procedures*
(trace–both *proc*)
(trace–entry *proc*)
(trace–exit *proc*)

Cause messages to be printed either in the Trace Window or the Transcript Window each time the procedure ***proc*** is invoked. The target of these messages is determined by whether or not you have checked the Output to Trace Window checkbox in the dialog accessed through the Trace ... item in the Evaluate menu. The procedures **trace** and **trace–entry** generate messages on entry to ***proc***; the procedure **trace–exit** generates messages on exit from ***proc***; and the procedure **trace–both** does so both on entry to and on exit from ***proc***. (See the entries for the corresponding **untrace** procedures later in this section.)

transcript *Variable*

A global variable bound to (the representation of) the Transcript Window.

(transcript–off) *Procedure*

Cancels the current call to **transcript–on** (see the next entry), if one is in effect, by closing the transcript file. If no such call is currently in effect, then no action is taken. Returns an unspecified value.

(transcript–on *spec*) *Procedure*

Causes an echo of the screen to be written to the file given by the input specification. This echoing continues until such time as it is terminated by a call to the procedure **transcript–off** (see the previous entry). The simplest way to provide **transcript–on** with an accurate file specification is to enter a suitable file name into the File Selector that is brought up in response to the Scheme expression

```
(transcript–on (choose–output–file))
```

(truncate *x*) *Procedure*

Returns the integer closest to *x* whose absolute value is less than or equal to the absolute value of *x*. The exactness of the result matches that of the input. See Section 5.9: em Exactness in Chapter 5: *Numbers and Numeric Procedures* for information concerning the exactness of numbers. Examples:

```
(truncate -13/5)  ↦  -2        (truncate pi)  ↦  3.0
(truncate -10.5)  ↦  -10.0
```

(turtle-color) *Procedure*
(turtle-color *gwin*)
(turtle-colour)
(turtle-colour *gwin*)

Returns an RGB triple denoting the current turtle color in the graphics window ***gwin***, if specified, or the most recently active open graphics window, otherwise. (See the first section of this chapter for further information concerning RGB triples.)

(turtle-display *exp*) *Procedure*
(turtle-display *exp gwin*)

Prints the data expression *exp* using the current pen color—see the entry earlier in this section for the procedure pen-color—in the graphics window *gwin*, if provided, or the most recently active open graphics window, otherwise. The printing starts with the lower left corner of the data expression's first character at the current turtle position—see the entry below for the procedure turtle-position—and, when the printing has been completed, the turtle's position has been moved to the lower right corner of the data expressions's last character. This procedure suppresses the double quotes delimiting strings, 'slashification' within strings, and the #\-combination that identifies characters.

(turtle-heading) *Procedure*
(turtle-heading *gwin*)

Returns an inexact number indicating the heading of the turtle (in graphics

window ***gwin***, if provided, or the most recently active open graphics window, otherwise) in radians or degrees—depending on which Angle Mode you have chosen in the Language dialog accessed through the Preferences sub-menu in the File menu—measured clockwise from due North.

(turtle-hide) *Procedure*
(turtle-hide *gwin*)

With immediate effect, hides the turtle in graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise. (This has the useful side effect of considerably speeding up most graphical activity.) If the turtle in question is already hidden, no action is taken.

(turtle-paint *bool*) *Procedure*
(turtle-paint *bool gwin*)

If ***bool*** is `#t`, the turtle in graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, will be shown (with immediate effect) as a solid-filled triangle drawn in the current turtle color (see the entry above for the procedure **turtle-color**). Otherwise, the turtle in question is shown as a triangular outline drawn in the current turtle color. The default condition is for the turtle to be shown in outline only.

(turtle-plane) *Procedure*
(turtle-plane *gwin*)

Returns a list of two pairs of inexact numbers, the first being the coordinate (in turtle steps and relative to the current graphics origin) of the top left corner of the turtle plane corresponding to the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, and the second being the coordinate of its bottom right corner. The difference between the first components of these coordinates is the width of the turtle plane, and the difference between the second components is its height.

(turtle-position) *Procedure*
(turtle-position *gwin*)

Returns a list of two inexact numbers, indicating the coordinate (in turtle

steps and relative to the current graphics origin) of the turtle's current position in the graphics window *gwin*, if provided, or the most recently active open graphics window, otherwise.

(turtle-set-color *rgb*) *Procedure*
(turtle-set-color *rgb gwin*)
(turtle-set-colour *rgb*)
(turtle-set-colour *rgb gwin*)

Sets the color in which the turtle is drawn in graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, to that specified by the input RGB triple. (See the first section of this chapter for information concerning RGB triples.)

(turtle-set-heading *x*) *Procedure*
(turtle-set-heading *x gwin*)

Sets the heading of the turtle in graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, to ***x***, measured—in radians or degrees, according to the Angle Mode you have chosen in the Language dialog accessed through the Preferences sub-menu in the File menu—clockwise from due North.

(turtle-set-position *pair*) *Procedure*
(turtle-set-position *pair gwin*)

Moves the turtle in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, from its current location to the point whose coordinate is given by ***pair*** (in turtle steps relative to the current graphics origin), *drawing the intervening segment* in accordance with the current pen state in that window. (See the entries for the procedures **turtle-position** and **pen-state** earlier in this section.) ***pair*** must be a list of two numbers.

(turtle-show) *Procedure*
(turtle-show *gwin*)

With immediate effect, reveals the turtle in the graphics window ***gwin***, if

provided, or the most recently active open graphics window, otherwise. If the turtle in question is not hidden, then no action is taken.

(turtle-shown?) *Procedure*
(turtle-shown? ***gwin*****)**

A predicate that returns the boolean `#t` if and only if the turtle in the graphics window ***gwin***, if provided, or the most recently active open graphics window, otherwise, is currently not hidden. (Note that a turtle that generates a `#t` output from this predicate may not actually be visible, because it might be located outside the boundaries of its graphics window or in a graphics window that is currently either hidden or overlaid by other windows.)

(unquote ***sexp*****)** *Special Form*

Evaluates ***sexp*** and inserts the result at the position in the quasiquote-expression in which this unquote-expression occurs. See the entry for the special form **quasiquote** earlier in this section. Unquote-expressions may also be abbreviated using a comma. See the entry on page 111 earlier in this section.

(unquote-splicing ***sexp*****)** *Special Form*

Evaluates ***sexp***, which must evaluate to a list, and inserts the elements of that list, in order of appearance, at the position in the quasiquote-expression in which this unquote-splicing-expression occurs. See the entry for the special form **quasiquote** earlier in this section. Unquote-splicing-expressions may also be abbreviated using a **,@**-combination. See the entry on page 111 earlier in this section.

(untrace ***proc*****)** *Procedure*
(untrace-entry ***proc*****)**
(untrace-exit ***proc*****)**

These procedures terminate the generation of messages that are being produced in response to calls to one or more of the corresponding **trace** procedures (see the relevant entries earlier in this section). The procedure **untrace-entry** terminates messages generated on entry to the procedure ***proc***; **untrace-exit**

terminates those generated on exit from the procedure; and **untrace** terminates all messages generated when the procedure ***proc*** is invoked.

---

user-global-environment *Variable*

A variable bound to the user global environment. This contains a single frame: the global frame (which contains bindings for all the ***EdScheme*** primitive procedures, for example). It prints as the empty list. Nevertheless, the contents of the global frame may be printed out by evaluating the Scheme expression **(first *user-global-environment*)** at ***EdScheme***'s top-level.

---

user-initial-environment *Variable*

A variable bound to the user initial environment. This contains two frames: the initial frame (containing bindings you have defined at ***EdScheme***'s top-level), and the global frame (described in the previous entry). It prints as the empty list, but the initial frame may be printed out by evaluating the Scheme expression **(first *user-initial-environment*)** at ***EdScheme***'s top-level.

(variant-case *sexp clause1 clause2* ...) *Initialization File Special Form*

Each clause must be of one of the following two forms:

- **[*rec1* (*field1* ...) *sexp1 sexp2* ...]**
- **[else *sexp1 sexp2* ...]**

(The convention in ***EdScheme*** is to enclose variant-case-clauses in brackets. Brackets and parentheses are completely interchangeable, however.)

Taking all the clauses of a variant-case-expression together, the 'dispatch' expressions ***rec1***, ..., must all be distinct. Variant-case-expressions do not have to include an else-clause, but if one is included, it should be the last clause. (In any case, it is the last clause that ***EdScheme*** pays any attention to!)

A variant-case-expression is evaluated as follows: First, the input Scheme expression ***sexp*** is evaluated, producing a value ***v***. Then, taking the clauses in order of appearance,

- if the clause is not an else-clause, the value ***v*** is tested (using the predicate ***rec1?***) to see if it is a record of type ***rec1***. If it is, then the corresponding Scheme expressions, ***sexp1***, ***sexp2***, ..., are evaluated in order of appearance in an environment in which the variables ***field1***, ..., are bound to the values of the corresponding—that is, same-named—fields of ***v***, and the variant-case-expression returns the value of the last one. If ***v*** is not a record of type ***rec1***, the process is repeated with the next clause.
- if the clause is an else-clause, then the corresponding Scheme expressions, ***sexp1***, ***sexp2***, ..., are evaluated in order of appearance, and the variant-case-expression returns the value of the last one.

It there is no else-clause, and ***v*** is not a record object of any of the types ***rec1***, ..., then an unspecified value is returned.

For example, assuming **tree-a** is defined as in the example given for the special form **define-record** earlier in this section,

```
(variant-case tree-a
  [leaf (number) number]
  [tree (number) number]
  [else "error"])                          ↦  1

(define tree-sum
  (lambda (tr)
    (variant-case tr
      [leaf (number) number]
      [tree (number left-tree right-tree)
            (+ number
               (tree-sum left-tree)
               (tree-sum right-tree))])))  ↦  unspecified
(tree-sum tree-a)                          ↦  6
```

(vector *exp1* ...) *Procedure*

Returns a vector whose entries are the given data expressions, in order of appearance. Examples:

```
(vector 1 "ab" #\c '(a b c))  ↦  #(1 "ab" #\c (a b c))
(vector)                      ↦  #()
```

(vector? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a vector. Examples:

```
(vector? '(a b))  ↦  #f      (vector? '#(a b))  ↦  #t
(vector? "a b")   ↦  #f      (vector? 'ab)      ↦  #f
```

(vector-fill! *vec exp*) *Procedure*

Sets each entry in the vector ***vec*** to be the data expression ***exp***, and returns an unspecified value. Example:

```
(define V (vector 'a 1 'c))  ↦  unspecified
V                            ↦  #(a 1 c)
(vector-fill! V "pad")       ↦  unspecified
V                            ↦  #("pad" "pad" "pad")
```

(vector-length *vec*) *Procedure*

Returns the number of entries in its argument as an exact integer. Example:

```
(vector-length '#(a #(a b) "ab" (a b)))  ↦  4
```

(vector->list *vec*) *Procedure*

Returns a list whose elements are the entries of the input vector, in order of appearance. Examples:

```
(vector->list '#(a (b c) "de" #\z))  ↦  (a (b c) "de" #\z)
(vector->list '#())                  ↦  ()
```

(vector-ref *vec k*) *Procedure*

Returns the entry of the vector ***vec*** with index ***k***. Vectors are 'zero-indexed'; that is, the first entry has index 0, the second has index 1, and so on. ***k*** must be non-negative and strictly less than the length of the vector. Example:

```
(vector-ref '#(a b c d) 2)  ↦  c
```

(vector-set! *vec k exp*) *Procedure*

Sets the entry of the vector ***vec*** with index ***k*** to be the data expression ***exp***, and returns an unspecified value. Vectors are 'zero-indexed'; that is, the first entry has index 0, the second has index 1, and so on. ***k*** must be non-negative and strictly less than the length of the vector. Example:

```
(define V (vector 0 'a "b" '(a b c)))  ↦  unspecified
V                                      ↦  #(0 a "b" (a b c))
(vector-set! V 3 #\c)                  ↦  unspecified
V                                      ↦  #(0 a "b" #\c)
```

(volume) *Procedure*

Returns the volume reference number of the current volume.

(wait *x*) *Procedure*
(wait *x bool*)

Causes ***EdScheme*** to pause for ***x*** seconds. (***x*** must be a non-negative real number less than or equal to 3600.) This procedure can be useful when sending data to or reading information from a serial port.

If an optional boolean argument of `#t` is provided, ***EdScheme*** continues to process events in (almost) the usual way. For example, you can open, close, edit, and save documents, the Transcript Window, the Trace Window, and so on. However, you cannot quit or edit the preferences or do anything that requires action on the part of the compiler. So, you cannot evaluate Scheme expressions during such a pause by typing them into the Transcript Window or using the Keypad-Enter method. A pause may be interrupted using the COMMAND-PERIOD hot key combination.

At the end of the pause, the program that was executing when the call to the procedure **wait** was made continues as before.

Repeated calls using the Scheme expression **(wait 0 #t)** give the effect of being in the ***EdScheme*** program development environment while a Scheme program is actually running. However, the only 'development' possible under such circumstances is simple editing in text or document windows.

(window? *exp*) *Procedure*

A predicate that returns the boolean `#t` if and only if its argument is a window.

(window-close) *Procedure*
(window-close *tecgxwin*)

Closes the window designated by the input, if provided. Otherwise, *all* open text and graphics windows are closed. When an input is provided, this is equivalent to choosing Close from the File menu.

(window-hide) *Procedure*
(window-hide *tecgxwin*)

Hides (that is, makes invisible) the window designated by the input, if provided. Otherwise, the most recently active open text or graphics window is hidden. Hiding a window simply removes it from view without changing its status as an open window or its place in the 'stack' of active windows. (Note that evaluating either

(window-hide *expression*) or **(window-hide *clipboard*)**

is equivalent to choosing either Hide expression or Hide clipboard, respectively, from the Edit menu.) This procedure has no effect if the window in question is already hidden.

(window-position) *Procedure*
(window-position *tecgxwin*)

Returns a list of three pairs of numbers. The first pair gives the scroll position of the window designated by the input, if provided, or the most recently active text or graphics window, otherwise. The second pair gives the position of the window's top left corner in screen coordinates (if the window is currently hidden, this pair will be `(0 0)`, and if the window has a title bar, the corner in question is the bottom left corner of the title bar), and the third pair gives the width and height of the window in screen units (in the case of a window with a title bar, the height does *not* include the depth of the title bar).

(window-print) *Procedure*
(window-print *tgxwin*)

Provided a suitable printer driver has been chosen using the Chooser item in the Apple menu, prints the window designated by the input, if provided, or the most recently active open text or graphics window, otherwise.

(window-select *tecgxwin*) *Procedure*

Selects the designated window, making it the most recently active window to which all (appropriate) subsequently evaluated procedures are targeted by default. This is equivalent to clicking on the window.

(window-set-position *list*) *Procedure*
(window-set-position *list tecgxwin*)

Positions the window designated by the second input, if provided, or the most recently active text or graphics window, otherwise, so as to conform to the positional information provided in the input ***list***. This must be a list of three pairs of numbers, whose significance is explained in the above entry for the procedure **window-position**.

Note that, if the optional input is either ***expression*** or ***clipboard***, then specifying any scroll position other than `(0 0)` will have no effect on the actual scroll position of the window in question, which remains `(0 0)`.

The smallest window that ***EdScheme*** will open measures 50 by 50. If you attempt to specify a smaller size, then ***EdScheme*** will replace any smaller dimension by 50.

(window-set-title *str*) *Procedure*
(window-set-title *str tgxwin*)

Changes the title of the window designated by the input, if provided, or the most recently active text or graphics window, otherwise, to the contents of the input string.

(window-show) *Procedure*
(window-show *tecgzwin*)

Shows (that is, makes visible) the window designated by the input, if provided. Otherwise, the most recently active open (but hidden) text or graphics window is shown. Note that evaluating either

(window-show *expression*) or **(window-show *clipboard*)**

is equivalent to choosing Show expression or Show clipboard, respectively, from the Edit menu. Nothing happens if the window in question is already shown. Furthermore, the window's position in the 'stack' of open windows is not changed; it is *not* necessarily revealed as the topmost window.

(windows-preferences) *Procedure*

Returns a list of five lists of exact non-negative integers that report the current settings in the Windows dialog accessed through the Preferences sub-menu in the File menu.

The first three lists, in order of appearance, give details of the fonts in use for system messages, Scheme expressions, and data expressions, respectively. Each of these lists contains three numbers, the first being the font number, the second being the style number (calculated as explained in the entry for **font-set-style** earlier in this section), and the third indicating the color according to the following code:

0 – black 1 – red 2 – green 3 – blue
4 – cyan 5 – magenta 6 – yellow

The fourth list contains two numbers, the first being the point size of the characters used in Transcript Windows, and the second being the point size of those used in document windows.

The fifth list also contains two numbers, the first being the right margin of the Transcript Window, and the second being the line length of the Transcript Window. As expressions are displayed in the Transcript Window, *EdScheme* starts a new line immediately after each atom (or 'word') that surpasses the right margin, or as soon as the line length is reached (even if that means splitting an atom or 'word'), if this occurs sooner.

The default settings for these (which, in the case of the fonts in use, depend on whether you are running ***EdScheme*** under Color Quickdraw™ or not) may be found by clicking on the Defaults button in the Windows dialog accessed through the Preferences sub-menu in the File menu.

(windows-set-preferences *list*) *Procedure*

Sets all the features in the Windows dialog accessed through the Preferences sub-menu in the File menu so that they conform to the input *list*. This must be a list of five lists whose form and meaning are described in the above entry for the procedure windows-preferences. Example:

```
(windows-set-preferences
  '((2 2 6) (1 32 0) (3 1 5) (10 10) (70 75)))

;;; Configures EdScheme to print system messages in yellow, italic
;;; characters from Font #2, Scheme expressions in black, condensed
;;; characters from Font #1, and data expressions in magenta, bold
;;; characters from Font #3. Also, both the transcript and document
;;; windows are set to use 10-point characters, and the Transcript
;;; Window's right margin and line length are set to 70 and 75
;;; characters, respectively.
```

(with-input-from-file *spec proc*) *Initialization File Procedure*

Opens the existing file identified by the given specification ready for input to ***EdScheme***, establishing an associated port which is temporarily made the current input port (see the entry earlier in this section for the procedure **current-input-port**), calls the given procedure ***proc*** (which must be a thunk, that is, a procedure of no arguments), and returns the value returned by the thunk, after closing the temporary input port and restoring the input port that was current when this procedure was called.

(with-output-to-file *spec proc*) *Initialization File Procedure*

Opens the specified file ready for output from *EdScheme*, establishing an associated port which is temporarily made the current output port (see the entry earlier in this section for the procedure current-output-port), calls the given procedure *proc* (which must be a thunk, that is, a procedure of no

arguments), and returns the value returned by the thunk, after closing the temporary output port and restoring the output port that was current when this procedure was called. If the specified file already exists, its contents will be overwritten during the course of the evaluation of this procedure.

(write *exp*) *Procedure*
(write *exp port*)
(write *exp gtxwin*)

Prints ***exp*** to the specified port or window, if provided, or to the current output port, otherwise. (See the entry earlier in this section for **current-output-port**.) Returns an unspecified value.

Unlike the procedure **display**—see the relevant entry earlier in this section—**write** does *not* suppress slashification, the double-quotes delimiting strings, or the **#**-combination identifying characters. In particular, this means that you should only use this procedure to send data to a serial port if you really intend double-quotes and special characters to be sent intact.

(write-char *ch*) *Procedure*
(write-char *ch port*)
(write-char *ch gtxwin*)

Writes the given character—*not* its external representation—to the specified port or window, if provided, or the current output port, otherwise. (See the entry earlier in this section for the procedure **current-output-port**.) Returns an unspecified value.

(zero? *z*) *Procedure*

A predicate that returns the boolean **#t** if and only if its argument is zero. If *z* is inexact and close to zero, output from this procedure may be unreliable. For further information about the exactness of numbers, see Section 5.9: *Exactness* in Chapter 5: *Number and Numeric Procedures*. Examples:

```
(zero? (- 5 5))  ↦  #t      (zero? 2.3)  ↦  #f
(zero? -5e-10)   ↦  #f
```

-o O o-

12 Language Elements

This chapter catalogues the language elements—the procedures, variables, constants, and special forms—of ***EdScheme*** 4.0 for the Macintosh. They are grouped into sections according to their type, and are cross-referenced to the detailed descriptions provided in Section 11.2: ***The Syntax of EdScheme***, where these same language elements are dealt with in alphabetical order. In addition, to the left of each entry in this chapter there appears a symbol indicating its status relative to the Scheme language, as described in the *Revised*^{4} *Report on the Algorithmic Language Scheme*. *R4RS* categorizes language elements as **essential** or—by implication—**non-essential**. We therefore partition ***EdScheme***'s language elements into three categories, denoted by the following symbols:

• – *R4RS* essential; ∘ – *R4RS* non-essential; ⋆ – ***EdScheme*** special

This coding will be useful if you are writing Scheme programs that you want to be portable to other Scheme implementations.

12.1 Constants, Booleans, and Equivalence Predicates

	Element	*Page*		*Element*	*Page*
•	#f	109	•	#t	109
•	()	110	⋆	*the-non-printing-object*	110
•	boolean?	125	⋆	empty-stream?	151
•	eq?	152	•	equal?	152

	Element	Page		Element	Page
•	eqv?	153	•	not	187
⋆	*pi*	197	⋆	the-empty-stream	229

12.2 Characters

	Element	Page		Element	Page
•	char?	128	•	char<?	129
•	char<=?	129	•	char=?	129
•	char>?	129	•	char>=?	130
•	char-alphabetic?	130	•	char-ci<?	130
•	char-ci<=?	131	•	char-ci=?	131
•	char-ci>?	131	•	char-ci>=?	132
•	char-downcase	132	•	char->integer	132
•	char-lower-case?	132	•	char-numeric?	133
•	char-upcase	133	•	char-upper-case?	134
•	char-whitespace?	134	•	integer->char	168

12.3 Strings

	Element	Page		Element	Page
⋆	expression->string	158	•	list->string	176
•	make-string	181	•	number->string	188
⋆	open-input-string	190	•	string	211
•	string?	211	•	string<?	212
•	string<=?	212	•	string=?	212
•	string>?	212	•	string>=?	213
•	string-append	213	•	string-ci<?	213
•	string-ci<=?	214	•	string-ci=?	214
•	string-ci>?	214	•	string-ci>=?	214
○	string-copy	215	⋆	string->expression	215
○	string-fill!	215	•	string-length	216
•	string->list	216	•	string->number	216
⋆	string-read	217	•	string-ref	218
•	string-set!	218	•	string->symbol	219
⋆	string-width	219	⋆	string-write	219
•	substring	220	⋆	substring-copy!	220

	Element	Page		Element	Page
⋆	substring-fill!	221	⋆	substring-find	221
•	symbol->string	222			

12.4 Vectors

	Element	Page		Element	Page
•	list->vector	176	•	make-vector	182
•	vector	236	•	vector?	237
o	vector-fill!	237	•	vector-length	237
•	vector->list	237	•	vector-ref	237
•	vector-set!	238			

12.5 Numbers

	Element	Page		Element	Page
•	*	110	•	+	111
•	-	111	•	/	112
•	<	112	•	<=	112
⋆	<>	113	•	=	113
•	>	113	•	>=	114
•	abs	114	o	acos	114
⋆	add1	115	o	angle	116
o	asin	118	o	atan	119
•	ceiling	128	•	complex?	136
o	cos	140	o	denominator	148
•	even?	154	•	exact?	156
o	exact->inexact	156	⋆	exact-rationalize	157
o	exp	157	o	expt	158
•	floor	161	•	gcd	165
o	imag-part	167	•	inexact?	167
o	inexact->exact	167	•	integer?	168
•	lcm	172	o	log	177
o	magnitude	177	o	make-polar	180
o	make-rectangular	181	•	max	182
•	min	185	•	modulo	186
•	negative?	186	•	number?	188

	Element	Page		Element	Page
o	numerator	189	•	odd?	189
•	positive?	199	★	power	200
•	quotient	202	★	random	203
★	randomise	203	★	randomize	203
•	rational?	203	o	rationalize	203
•	real?	206	o	real-part	206
•	remainder	207	•	round	208
★	seed	209	o	sin	210
o	sqrt	211	★	sub1	219
o	tan	223	•	truncate	231
•	zero?	243			

12.6 Symbols and Lists

	Element	Page		Element	Page
•	append	116	•	assoc	119
•	assq	119	•	assv	119
•	car	126	•	cdr	128
•	cons	139	★	delete!	147
★	first	161	•	for-each	163
★	last	171	★	last-pair	171
•	length	172	•	list	175
•	list?	175	•	list-ref	175
o	list-tail	176	•	map	182
•	member	183	★	member?	183
•	memq	183	•	memv	183
★	nth	187	•	null?	188
•	pair?	193	★	rest	208
•	reverse	208	•	set-car!	209
•	set-cdr!	209	★	set-first!	209
★	set-rest!	210	•	symbol?	222

12.7 Graphics and Text Windows

	Element	Page		Element	Page
★	arc	117	★	arc-paint	118

	Element	*Page*		*Element*
★	back	120	★	bitmap?
★	bitmap–close	122	★	bitmap–fetch
★	bitmap–mode	123	★	bitmap–set–spec
★	bitmap–spec	124	★	bitmap–stamp
★	clean	135	★	draw–point
★	font–set–style	162	★	font–style
★	forward	163	★	graphics–origin
★	graphics–set–origin	165	★	graphics–window
★	graphics–window?	166	★	home
★	left	172	★	make–bitmap
★	make–graphics–window	178	★	make–text–window
★	oval	192	★	oval–paint
★	pen–color	194	★	pen–colour
★	pen–down	194	★	pen–down?
★	pen–erase	194	★	pen–reverse
★	pen–set–color	195	★	pen–set–colour
★	pen–set–state	195	★	pen–state
★	pen–up	196	★	point–color
★	point–colour	198	★	point–on?
★	polygon	198	★	polygon–paint
★	right	208	★	text–alignment
★	text–char–closest	223	★	text–clean
★	text–clear	224	★	text–contents
★	text–copy	224	★	text–cut
★	text–display	225	★	text–length
★	text–lines	225	★	text–paste
★	text–readline	226	★	text–scroll–to–selection
★	text–selection	226	★	text–set–alignment
★	text–set–contents	227	★	text–set–selection
★	text–set–spacing	228	★	text–spacing
★	text–window	228	★	text–window?
★	towards	229	★	turtle–color
★	turtle–colour	231	★	turtle–display
★	turtle–heading	231	★	turtle–hide
★	turtle–paint	232	★	turtle–plane
★	turtle–position	232	★	turtle–set–color
★	turtle–set–colour	233	★	turtle–set–heading

	Element	*Page*		*Element*	*Page*
⋆	turtle–set–position	233	⋆	turtle–show	233
⋆	turtle–shown?	234	⋆	window?	239
⋆	window–close	239	⋆	window–hide	239
⋆	window–position	239	⋆	window–print	240
⋆	window–select	240	⋆	window–set–position	240
⋆	window–set–title	240	⋆	window–show	241

12.8 Keyboard and Ports

... not to mention the Clipboard, Expression, and Transcript Windows.

	Element	*Page*		*Element*	*Page*
⋆	*set–current–input–port*	110	⋆	*set–current–output–port*	110
•	call–with–input–file	126	•	call–with–output–file	126
o	char–ready?	133	⋆	choose–input–file	134
⋆	choose–output–file	134	⋆	***clipboard***	135
⋆	clipboard–set–text	135	⋆	clipboard–text	135
•	close–input–port	135	•	close–output–port	135
⋆	close–port	136	⋆	configure–serial–port	138
•	current–input–port	140	•	current–output–port	140
•	display	148	⋆	EdScheme–volume	150
⋆	eof?	151	•	eof–object?	151
⋆	***expression***	158	⋆	expression–set–text	158
⋆	expression–text	158	⋆	file–exists?	159
⋆	file–length	159	⋆	file–margin	159
⋆	file–position	159	⋆	file–set–length	160
⋆	file–set–margin	160	⋆	file–set–position	161
⋆	file–spec	161	⋆	freshline	164
•	input–port?	168	⋆	last–volume	171
•	load	176	•	newline	187
⋆	open–extend–file	189	•	open–input–file	190
•	open–output–file	191	⋆	open–serial–port	191
•	output–port?	192	•	peek–char	193
⋆	port?	199	•	read	204
•	read–char	205	⋆	read–line	205
⋆	set–volume	210	⋆	***transcript***	230
o	transcript–off	230	o	transcript–on	230

	Element	*Page*		*Element*	*Page*
⋆	volume	238	o	with-input-from-file	242
o	with-output-to-file	242	•	write	243
•	write-char	243			

12.9 Events, Menus, and the Mouse

	Element	*Page*		*Element*	*Page*
⋆	event	154	⋆	event-flush	156
⋆	event-ready?	156	⋆	make-menu	180
⋆	menu?	184	⋆	menu-close	184
⋆	menu-item	184	⋆	menu-number-of-items	184
⋆	menu-set-item	184	⋆	modifiers	185
⋆	mouse-state	186			

12.10 Debugging

	Element	*Page*		*Element*	*Page*
⋆	trace	230	⋆	trace-both	230
⋆	trace-entry	230	⋆	trace-exit	230
⋆	untrace	234	⋆	untrace-entry	234
⋆	untrace-exit	234			

12.11 Keywords and Special Forms

	Element	*Page*		*Element*	*Page*
o	=>	113	•	and	115
•	begin	120	•	case	127
•	cond	136	⋆	cons-stream	139
•	define	141	⋆	define-alias	142
⋆	define-macro	143	⋆	define-record	145
⋆	define-transformer	146	o	delay	147
o	do	149	•	else	151
⋆	error	153	⋆	freeze	164
•	if	166	•	lambda	169
•	let	173	o	let*	174

	Element	*Page*		*Element*	*Page*
•	letrec	174	★	make–environment	178
•	or	192	•	quasiquote	201
•	quote	202	★	rec	206
★	repeat	207	★	set!	209
•	unquote	234	•	unquote–splicing	234
★	variant–case	235			

12.12 Miscellaneous

	Element	*Page*		*Element*	*Page*
•	' (single right quote)	109	•	, (comma)	111
•	,@ (comma–at)	111	•	` (single left quote)	114
•	apply	116	★	atom?	120
o	call/cc	125	★	caution–alert	128
★	color–quickdraw?	136	★	colour–quickdraw?	136
★	continuation?	140	★	cursor	141
★	derived?	148	★	desktop	148
★	eval	154	★	explode	157
o	force	162	★	freemem	163
★	freesp	164	★	gc	164
★	head	166	★	implode	167
★	*integrate–primitives*	168	★	language–preferences	170
★	language–set–preferences	170	★	note–alert	187
★	pick–color	197	★	pick–colour	197
★	primitive?	200	★	print–length	200
•	procedure?	200	★	procedure–definition	201
★	procedure–environment	201	★	quit	202
★	reset	208	★	runtime	209
★	stop–alert	211	★	tail	222
★	thaw	229	★	the–environment	229
★	wait	238	★	windows–preferences	241
★	windows–set–preferences	242			
•	call–with–current–continuation	126			
★	*user–global–environment*	235			
★	*user–initial–environment*	235			

Index

" (double-quote), 100
(hash symbol), 32
#b, 37
#d, 37
#!eof, 151
#f, 97
#f, 109
#o, 37
#t, 97
#t, 109
#x, 37
' (single right quote), 96, 109
(), 29
(), 110
*, 110
set-current-input-port, 110
set-current-output-port, 110
the-non-printing-object, 110
+, 111
, (comma), 111
,@ (comma-at), 111
-, 111
..., 108
/, 112
<, 112

Angle Mode, 75, 170
    default, 75
angles in mathematics, 54
angles in turtle geometry, 54
append, 116
applications, Macintosh, 6
apply, 116
arc, 117
arc–paint, 118, 193, 199
arithmetic,
    bignum, 5
    complex number, 5
    floating point, 5
    integer, 5
    rational, 5
artificial intelligence, 3
asin, 46, 118
assoc, 119
association list, 107
assq, 119
assv, 119
atan, 46, 119
atom?, 102, 120
atoms, 102

back, 120
backquote, 114, 202
backslash, 100
bang, 33 (footnote)
begin, 120
bignum arithmetic, 5, 39
binary, 36
binary tree, 145
bindings, 103
    in Debug Transcripts, 22
bitmap?, 105, 122
bitmap–close, 59, 122
bitmap–fetch, 58, 122, 178
bitmap–mode, 58, 123
bitmaps, 6, 58, 105
    color depth of, 178
    external representation of, 105
bitmap–set–spec, 58, 123, 178
bitmap–spec, 58, 124
bitmap–stamp, 58, 124
bitmap stamping mode, 58, 123
boolean?, 97, 125
boolean objects, 35, 97
boolean values, 97
booleans, 97
breaking lines, 72
buffers,
    debugging, 81
    graphics, 9, 80
    text, 25, 62, 80
    transcript, 21, 81

call by need, 147
call/cc, 102, 125
call–with–current–continuation, 102, 126
call–with–input–file, 126
call–with–output–file, 126
car, 30, 97, 103, 126, 161
caret, 11
    moving, 13 (footnote)
carriage return, 134
case, 127, 151
case-sensitivity, 74
    default setting, 75
catch, 101
caution–alert, 128
cdr, 30, 97, 103, 128, 208
ceiling, 128
centered text in text windows, 61
changing exact to inexact, 45
changing inexact to exact, 45

changing menu items, 185
char?, 99, 128
char<?, 129, 212
char<=?, 129
char=?, 129
char>?, 129, 212
char>=?, 130
char-alphabetic?, 130
char-ci<?, 130, 213
char-ci<=?, 131
char-ci=?, 131
char-ci>?, 131, 214
char-ci>=?, 132
char-downcase, 132
char->integer, 132
char-lower-case?, 132
char-numeric?, 133
char-ready?, 51, 133
char-upcase, 133
char-upper-case?, 134
char-whitespace?, 134
characters, 35, 99
    printing, 99
checked menu items, 184
choose-input-file, 51, 93, 134, 177, 189
choose-output-file, 51, 134, 189, 191
CHURCH, Alonzo, 4 (footnote)
circular list, 98, 175
clean, 135
Clear, 84
Clear Trace, 92
clearing text, 84
clearing the Trace Window, 92
clipboard, 83, 135
clipboard-set-text, 135
clipboard-text, 135
clipboard window, 26, 83, 85
    hiding, 85
    showing, 85
Close, 70
Close All, 93
close-input-port, 51, 135
close-output-port, 51, 135
close-port, 51, 136
closing all document windows, 93
closing Debug Transcripts, 23
closing serial ports, 192
closing the Transcript Window, 21
closing windows, 70
codes for windows, 108
color depth of bitmaps, 178
Color QuickDraw™, 197
color-quickdraw?, 136
colors for text, 20, 72
colour-quickdraw?, 136
comma, 111
comma-at, 111
comments, 106
compiler workspace, 81
complex?, 99, 136
complex number arithmetic, 5
complex numbers, 6, 35, 43
    polar form, 116
cond, 136, 151
configuration codes for serial ports, 138
configure-serial-port, 52, 138, 191
cons, 30, 97, 103, 139
cons-stream, 103, 139, 144, 146
continuation?, 101, 140
continuations, 5, 7, 101
    external representation of, 102
    inaccessible, 102
    memory used by, 102

control character, searching for, 87
control panel, 197
convention, notational, 107
Copy, 84
copying text, 26, 84
cos, 46, 140
cosh, 47
cosine function, 46
    hyperbolic, 47
    inverse, 46
creating templates, 85
current environment, 104
current expression, 14, 25, 73, 85
    evaluating, 15, 89
current-input-port, 140, 193, 204, 205, 242
current-output-port, 140, 164, 187, 242
current volume, 49
cursor, 141
customizing document windows, 72
customizing transcript windows, 72
Cut, 83
cutting text, 26, 83

data expressions, 12, 29, 95
Debugging, 77
debugging, 77
debugging buffer setting, 81
debugging window, 5
debug preference setting, overriding, 92
Debug Transcripts, 92
Debug Transcripts, 20, 21, 78
    bindings in, 22
    closing, 23
    colors in, 22
    disabling, 22
Debug Transcripts,
    fonts in, 22
    memory allocation for, 22
    mutator procedures in, 22
    numbering of, 22
    procedures in, 22
    temporarily disabling, 92
    type styles in, 22
decimal, 36
decimal form, 41
decimal notations, 38
default Angle Mode, 75
default case-sensitivity setting, 75
default Language Mode, 75
default line length, 72
default partition size, 8
default pen state, 196
default right margin, 72
default text alignment, 227
default text size
    in document windows, 74
    in transcript windows, 72
define, 101, 141, 146, 207
define-alias, 105, 142
define-macro, 105, 143
define-record, 145, 236
define-transformer, 106, 146
definition formatting, 16
degrees->radians, 197
delay, 103, 147, 162, 164
delayed evaluation, 5, 147
delayed objects, 103
delete!, 147
deleting text, 84, 88
denominator, 148
derived?, 148
deselected menu items, 184
desktop, 148

digits, 37
disabling Debug Transcripts, 22
    temporarily, 92
display, 52, 61, 140, 148
displaying the global frame, 103
displaying the initial frame, 103
divisor, greatest common, 165
do, 149
document,
    creating, 23
document windows, 13, 23
    closing, 23
    closing all, 93
    customizing, 72
    default text size in, 74
    hot key switching, 17, 24, 94
    indexing, 16
    memory allocation, 80
    opening, 17, 23, 69
    saving, 17, 23, 70
    text size in, 74
dot notation, 30
double spacing, 228
double-quote, 100
draw-point, 150

Edit, 83
Edit Templates ..., 85
editing templates, 5, 85
editor, 5
EdScheme, 4
    features, 4
    initializing, 208
    launching, 11
    on a network, 82
    quitting from, 82
EdScheme session,
    history of, 12
EdScheme session,
    transcript of, 12
EdScheme-volume, 150
elements of a list, 98
elements of a vector, 100
elements of the empty list, 98
ellipsis, 108
else, 151
else-clause, 127, 136, 236
empty list, 29, 35, 97, 110
    elements of, 98
    length of, 98
empty-stream?, 151
Enter Selection, 87
ENTER (on keypad), 13
entering selected text into search and replace, 87
entering selected word into search and replace, 87
environments, 5, 103
    current, 104
    global, 208
    initial, 208
eof?, 51, 151
eof-object?, 51, 151
eq?, 152, 183
equal?, 152, 183
eqv?, 153, 183
erasing text, 84
error, 153
error messages,
    target of, 78
    types of, 78
Error Modes, 170
error while evaluating selections, 89
escape procedure, 101, 125
escaping special characters, 100
eval, 104, 154, 178

Evaluate, 89
evaluated expressions, 95
evaluating a selection, 24, 77
evaluating a whole file, 77, 90
evaluating current expression, 15, 89
evaluating expressions, 25
evaluating selected expressions, 89
  error while, 89
evaluation, 12
  delayed, 5, 147
  hot key combination, 23
  lazy, 147
even?, 154
event, 66, 154, 180, 185
event-flush, 156
event-ready?, 156
exact?, 44, 156
exact->inexact, 45, 156
exact integer form, 37
exactness, 44
exactness prefix, 37
exact numbers, 44
exact-rationalize, 45, 157
exact-root, 40
exact to inexact, changing, 45
exiting from search and replace dialog, 86
exiting from trace dialog, 91
exp, 45, 157
explode, 157, 216
exponent markers, 41, 216
exponential form, 41
exponential function,
  general, 45
  natural, 45
Expression, 89
expression, 158
expression,
  current, 14
    evaluating, 15
  data, 12, 29
  evaluation of, 25
  formatting, font selection for, 72
  Scheme, 12, 29
expressions,
  evaluated, 95
  unevaluated, 95
expression-set-text, 158
expression->string, 158
expression-text, 158
Expression Window, 14, 25, 85
  hiding, 85
  showing, 85
expt, 45, 158, 200
extension of the language, 5
external representation of
  bitmaps, 105
  continuations, 102
  graphics windows, 105
  macros, 105
  menus, 105
  ports, 104
  procedures, 76, 101
  records, 145
  text windows, 106
  transformers, 106
  vectors, 32

#f, 109
factorial, 150
false, 97
FD, 101
fetching a bitmap image, 58
fields of a record, 145

File, 69
file-exists?, 159
file-handling, 5
file indexing, 5
file-length, 51, 159
file-margin, 51, 159
file name, 49
file-position, 51, 159
files, 49
    loading, 77, 92
    random access, 161
File Selector, 51
file-set-length, 51, 160
file-set-margin, 51, 159
file-set-position, 51, 161
file-spec, 161
file specification, full, 50
Find ..., 86
Find again, 87
Finder™, 49
finding the line number, 88
first, 30, 97, 103, 161
first class objects, 95
floating point arithmetic, 5
floor, 161
floppy disk, starting *EdScheme* from, 82
flush left text, 61
flush right text, 61
flushing a serial port, 192
fonts, 20, 72
font selection for expression formatting, 72
font-set-style, 162, 241
font-style, 72, 162
font style for graphics, 219
for-each, 163
force, 147, 162, 164
form feed, 134
Format, 84
formatting definitions, 5, 16
formatting, hot key combinations, 16
formatting Scheme expressions, 72, 84
forward, 163
forward slash in rational numbers, 40
fractions, 6
frame, 103
    global, 103
    initial, 103
freemem, 163
freesp, 164
freeze, 103, 147, 162, 164, 229
freshline, 52, 164
full file specification, 50
function descriptor, 101

garbage collection, 74, 164, 170
    partial, 75, 165, 170
gc, 75, 156, 164, 170
gcd, 165
general exponential function, 45
general logarithm function, 46
global environment, 208
global frame, 103, 235
    displaying the, 103
Go to line ..., 88
go to line number, 88
graphics buffer, 9, 80
graphics font style, 219
graphics-origin, 165
graphics origin, 165
graphics-set-origin, 165
graphics toolbox, 5

graphics, turtle, 5
graphics-window, 165
graphics-window?, 105, 166
graphics windows, 6, 53, 105
    external representation of, 105
    memory requirements, 54
    minimum size, 54
greatest common divisor, 165

hash f, 97
hash symbol, 32
hash t, 97
head, 103, 166
hexadecimal, 36
Hide Clipboard, 85
Hide Expression, 85
hiding the Clipboard Window, 85
hiding the Expression Window, 85
hiding the Trace Window, 91
history of *EdScheme* session, 12
home, 165
hot key combinations for
        formatting, 16
hot key document-switching, 94
hot key evaluation, 23
hot keys for templates, 85
hyperbolic cosine function, 47
hyperbolic sine function, 47
hyperbolic tangent function, 47
Hyperbot™, 5, 139

identifiers, 95
    quoting, 96
    rules for, 96
if, 166
imag-part, 167
implode, 167
improper fractions, 41
improper lists, 30, 98
    printing, 98
inaccessible continuations, 102
indentation, 15
index in a list, 31, 98
index in a string, 100
index in a vector, 32, 100
indexing document windows, 16
indexing files, 5
inexact?, 44, 167
inexact->exact, 45, 167
inexact numbers, 44
inexact to exact, changing, 45
initial environment, 208
initial frame, 103, 235
    displaying the, 103
initializing *EdScheme*, 208
input port, 51, 104, 140
input-port?-com, 51, 168
input-string, 168
inserting a template, 85
inserting text, 88
integer?, 168
integer arithmetic, 5
integer->char, 99, 168
integers, 35, 37
integrated editor, 5
integrate-primitives, 102, 168
inverse cosine function, 46
inverse sine function, 46
inverse tangent function, 46
iteration, 149

Keypad-ENTER, 13
keywords, 95

lambda, 101, 142, 146, 169
Language, 74
language extension, 5

Language Mode, 9, 74, 170
  default setting, 75
language preferences, 74
language-preferences, 77, 170
language-set-preferences, 77, 170
last, 31, 171
last-pair, 171
last-volume, 171
launching *EdScheme*, 11
lazy evaluation, 147
lcm, 172
least common multiple, 172
left, 172
length, 31, 172
length of a list, 98
length of a string, 100
length of a vector, 100
length of the empty list, 98
let, 173
  named, 173
let*, 174
let-body, 173
letrec, 174, 207
lexicographic order, 212
line feed, 134
line length, 72
  default setting, 72
line number,
  finding, 88
  go to, 88
line width in text windows, 61
line-breaking, 72
Lisp, 3, 29
list, 175
list?, 30, 99, 175
list, 6, 29, 97
  association, 107
  circular, 98, 175
list,
  elements of, 98
  empty, 29, 35, 97, 110
  improper, 30, 98
  index in, 31, 98
  length of, 98
  one-referenced, 31
  printing, 98
  zero-referenced, 31
list-ref, 31, 175, 187
list->string, 176
list-tail, 176
list->vector, 176
ln, 47
load, 77, 176
Load ..., 92
loading a file, 77, 92
log, 46, 177
logarithm function,
  general, 46
  natural, 46
lookup, 137

macros, 5, 105
  external representation of, 105
magnitude, 114, 177
make-bitmap, 58, 105, 178
make-environment, 104, 154, 178
make-graphics-window, 178, 181
make-menu, 65, 105, 180
make-polar, 75, 180
make-rectangular, 181
make-string, 181
make-text-window, 61, 181
make-vector, 33, 182
map, 182
marker, exponent, 41
masking a bitmap image, 58

matching parentheses, 5, 11, 25, 72
mathematical information, 45
max, 182
member, 183
member?, 183
memoization, 147, 162
Memory, 79
memory allocations, 79
    for Debug Transcripts, 22
    for text, 25
    for the Trace Window, 23
memory encroachment, 81
memory requirements,
    for document windows, 80
    for graphics windows, 54
    for text windows, 62
memory used by continuations, 102
memq, 183
memv, 183
menu?, 105, 184
menu-close, 66, 184
menu-item, 65, 184
menu-items, 66
menu items,
    changing, 185
    checked, 184
    deselected, 184
    selected, 184
    unchecked, 184
menu-number-of-items, 66, 184
menus, 65, 105
    external representation of, 105
    user-generated, 6
menus:
    Edit, 83
    Evaluate, 89
    File, 69
    Search, 86
menus:
    Windows, 93
menu separator line, 66, 180, 184
menu-set-item, 65, 180, 184
min, 185
minimum turtle plane, 179
mixed numbers, 41
mode,
    angle, 75, 170
    bitmap-stamping, 58
    case-sensitivity, 74
    language, 9, 74, 170
modem, 191
modifiers, 185
modulo, 186, 207
Monaco, 72
mouse-state, 186
moving text, 83
moving the caret, 13 (footnote)
multi-apply, 146
Multifinder™, 8
multiple, least common, 172
mutator procedures in Debug Transcripts, 22

name of a file, 49
named **let**, 173
natural exponential function, 45
natural logarithm function, 46
negative?, 186
network, running *EdScheme* on, 82
New, 69
New Transcript, 82
newline, 52, 164, 187
newline character, searching for, 87
Newton's Method, 39
nib of turtle pen, 196
not, 187

notation for decimals, 38
notational convention, 107
note-alert, 187
nth, 31, 187
null?, 29, 98, 188
number?, 35, 99, 188
number bases, 36
numbering of debug transcripts, 22
numbers, 99
    complex, 6, 35, 43
    exact, 44
    inexact, 44
    rational, 6, 35, 39
    real, 35, 41
number->string, 36, 41, 188
numerator, 189
numeric constants, 36
numeric prefixes, 36

object-oriented programming, 7
objects, first class, 95
oblist, 165
oblist size setting, 81
octal, 36
odd?, 189
one-referenced list, 31
Open ..., 69
Open As Transcript ..., 82
open-extend-file, 51, 189
open files setting, 79
opening a new transcript, 82
opening document windows, 17, 69
open-input-file, 51, 190
open-input-string, 168, 190
open-output-file, 51, 191
open-serial-port, 52, 191
or, 192
origin, graphics, 165
output from tracing, 91
output port, 51, 104
output-port?, 51, 192
oval, 192
oval-paint, 193
overriding debug preference setting, 92

Page Setup ..., 71
pair?, 30, 97, 193
pairs, 30, 35, 97, 108
parenthesis-matching, 5, 11, 25, 72
partial garbage collection, 75, 165, 170
partition size, 8
    default, 8
Paste, 84
pasting text, 26, 84
path name, 50
patience, 143
peek-char, 51, 193
pen nib, turtle, 196
pen-color, 194
pen-colour, 194
pen-down, 194
pen-down?, 194
pen-erase, 194
pen-reverse, 195
pen-set-color, 195
pen-set-colour, 195
pen-set-state, 194, 195
pen-state, 117, 118, 163, 166, 192, 194, 195, 198, 233
pen state, default, 196
pen-up, 196
pi, 197
pick-color, 197
pick-colour, 197

point-color, 198
point-colour, 198
point-on?, 198
polar angle, 116
polar form of complex numbers, 43, 116
polygon, 198
polygon-paint, 199
port?, 104, 199
ports, 50, 104
  external representation of, 104
  input, 51, 104
  output, 51, 104
  serial, 5, 49, 104
positive?, 199
positive sense for angle measurement, 54
power, 46, 159, 200
Preferences, 71
preferences,
  language, 74
  windows, 72
preferences file, 82
prefixes, 36
pretty-printing, 72, 84
primary Transcript Window, 78
primitive?, 200
Print ..., 71
print-length, 200, 225
printer, 191
printer driver, 71
printer setup, 71
printing,
  characters, 99
  from windows, 71
  improper lists, 98
  lists, 98
  strings, 100
printing,
  vectors, 100
procedure?, 101, 200
procedure-definition, 201
procedure-environment, 201
procedures, 35, 100
  external representation of, 76, 101
procedures in Debug Transcripts, 22
program formatting, 5
programming interface, 5
promise, 139, 147, 162
prompt, 11

quasiquote, 201, 234
Quit, 82
quit, 82, 202
quitting from *EdScheme*, 82
quote, 96, 202
quote mark, 109
quotient, 202
quoting identifiers, 96

radix, 36, 108
radix prefixes, 36, 216
random, 203
random access files, 161
randomise, 203
randomize, 203
rational?, 203
rational arithmetic, 5
rational form, 42
rationalize, 203
rational numbers, 6, 35, 39
rational, simpler, 204
re-opening a Transcript Window, 21
read, 51, 140, 204

read-char, 51, 185, 205
reading from serial ports, 204
read-line, 51, 205
real?, 206
real numbers, 35, 41
real-part, 206
rec, 206
record, external representation of, 145
record fields, 145
record object, 145
rectangular form of complex numbers, 44
recursion, 4
recursion depth setting, 79
recursion, tail-, 4
Redo, 83
redoing, 83
reinstating a filed transcript, 20, 82
remainder, 186, 207
repeat, 207
repeat depth setting, 79
Replace, 88
Replace all, 88
Replace and find again, 88
replace, search and, 86
reset, 208
rest, 30, 97, 103, 208
reverse, 208
RGB triples, 109, 194
right, 208
right margin, 72
  default, 72
round, 208
rules for identifiers, 96
runtime, 209

sans serif typeface, 29
Save, 70
Save As ..., 70
Save As Text ..., 71
saving document windows, 17, 70
saving text windows, 70
saving Transcript Windows, 20, 70
Scheme, 3
Scheme expression, 12, 29
  evaluating selected, 89
  formatting, 72, 84
Schemer's Guide Mode, 74
Search, 86
search and replace dialog, 86
  entering selected text, 87
  entering selected word, 87
  exiting from, 86
searching for,
  control character, 87
  newline character, 87
  special character, 87
seed, 209
Select all, 84
selected menu items, 184
selecting text, 24, 83, 84
selecting the Trace Window, 93
selecting the Transcript Window, 93
Selection, 89
selection, evaluating, 24, 77
separator line in menus, 66, 180, 184
serial interface, 6
serial ports, 5, 49, 104, 138, 238
  closing, 192
  configuration codes, 138
  flushing, 192
  reading from, 204
set!, 209

set-car!, 209
set-cdr!, 209
set-first!, 209
set-rest!, 210
set-volume, 151, 171, 210
settings:
    compiler workspace, 81
    debugging buffer, 81
    graphics buffer, 80
    oblist size, 81
    open files, 79
    recursion depth, 79
    repeat depth, 79
    text buffer, 80
    trace depth, 79
    transcript buffer, 81
Show Clipboard, 85
Show Expression, 85
showing,
    the Clipboard Window, 85
    the Expression Window, 85
    the Trace Window, 91
SICP, 7
simpler rational number, 204
sin, 46, 210
sine function, 46
    hyperbolic, 47
    inverse, 46
single spacing, 228
sinh, 47
slash, forward, 40, 42
slashification, 100
solid fill, 196
sorting, 7
space, 134
special characters, searching for, 87
specification, full file, 50
Specify ..., 82
sqrt, 39, 46, 152, 211
square root function, 46
stack tracing, 170
stamping a bitmap image, 58
stamping mode for bitmaps, 123
Standard Scheme Mode, 74
starting *EdScheme* from a floppy disk, 82
STEELE, Guy Lewis, Jr. 3
stop-alert, 211
stopping tracing, 91
streams, 5, 103
string, 211
string?, 99, 211
string<?, 212
string<=?, 212
string=?, 212
string>?, 212
string>=?, 213
string-append, 213
string-ci<?, 213
string-ci<=?, 214
string-ci=?, 214
string-ci>?, 214
string-ci>=?, 214
string-copy, 215
string->expression, 215
string-fill!, 215
string-length, 216
string->list, 157, 204, 216
string->number, 167, 216, 219
string-read, 51, 62, 190, 217
string-ref, 218
strings, 35, 99
    indexing, 100
    printing, 100
string-set!, 215, 218
string->symbol, 219

string-width, 219
string-write, 52, 61, 219
style numbers, 162
styles for text, 20, 72
sub1, 219
substring, 220
substring-copy!, 220
substring-fill!, 221
substring-find, 221
SUSSMAN, Gerald Jay, 3
switching between documents, 17, 24
symbol?, 96, 222
symbols, 35, 95
symbol->string, 167, 22
System 7™, 8

#t, 109
Tab, 134
tail, 103, 139, 222
tail-recursion, 4
tan, 46, 223
tangent function, 46
  hyperbolic, 47
  inverse, 46
tanh, 47
target of error messages, 78
Templates, 85
templates, 85
  creating, 85
  editing, 5, 85
  hot keys for, 85
  inserting, 85
text-alignment, 61, 223
text alignment in text windows, default, 227
text buffer, 25, 62, 80
text-char-closest, 223
text-clean, 224
text-clear, 62, 84, 224
text colors, 20, 72
text-contents, 62, 224
text-copy, 62, 84, 224
text-cut, 62, 83, 224
text-display, 61, 225
text fonts, 20
text-length, 225
text-lines, 225
text-paste, 62, 84, 225
text-readline, 62, 226
text resource files, 62
text-scroll-to-selection, 226
text, selecting, 24
text-selection, 226
text-set-alignment, 61, 227
text-set-contents, 61, 227
text-set-selection, 62, 83, 219, 224, 226, 227
text-set-spacing, 228
text size
  in document windows, 74
  in transcript windows, 72, 73
text-spacing, 228
text styles, 20, 72
text-window, 228
text-window?, 105, 228
text windows, 6, 61, 105
  alignment in, 61
  external representation of, 106
  line width in, 61
  memory requirements of, 62
  minimum size of, 61
  saving, 70
  word wrap in, 61
thaw, 162, 164, 229
the-empty-stream, 103, 151, 229

the-environment, 104, 153, 229
throw, 101
thunk, 143
title bar of window, 239
Too complex error, 79
toolbox, graphics, 5
towards, 229
trace, 230, 234
Trace ..., 90
trace-both, 102, 230
trace depth setting, 79
trace dialog, exiting from, 91
trace-entry, 230
trace-exit, 102, 230
trace output, 91
Trace Window, 93
Trace Window,
clearing, 23, 92
hiding, 91
memory allocation for, 23
selecting, 93
showing, 23, 91
tracing programs, 90
stopping, 91
Transcript, 93
transcript, 230
transcript buffer, 21, 81
adjusting, 21
transcript of an *EdScheme* session, 12
transcript-off, 230
transcript-on, 230
Transcript Window, 5, 11, 20, 104
closing, 21
customizing, 72
Debug, 20, 78
default text size in, 72
default text size in, 73
Transcript Window,
opening a new, 82
primary, 78
reinstating, 20, 82
re-opening, 21
saving, 20, 70
selecting, 93
text size in, 72
transformers, 5, 106
external representation of, 106
tree, binary, 145
triple spacing, 228
true, 97
truncate, 231
turtle-color, 231
turtle-colour, 231
turtle-display, 231
turtle graphics interface, 5
turtle-heading, 54, 231
turtle-hide, 232
turtle-paint, 232
turtle pen nib, 196
turtle-plane, 232
turtle plane, 54, 179
default size, 54
minimum size, 54, 179
turtle-position, 231, 232
turtle-set-color, 233
turtle-set-colour, 233
turtle-set-heading, 54, 229, 233
turtle-set-position, 233
turtle-show, 233
turtle-shown?, 234
types of error messages, 78
typewriter typeface, 29

unchecked menu items, 184
Undo, 83

undoing, 83
unevaluated expressions, 95
unknown digit form, 38, 41
unquote, 201, 234
unquote-splicing, 201, 234
untrace, 230, 234
untrace-entry, 234
untrace-exit, 234
user-global-environment, 103, 152, 235
user-initial-environment, 103, 152, 235

variables, 95, 96
variant-case, 145, 235
vector, 32, 236
vector?, 100, 237
vector elements, 100
  indexing, 100
vector-fill!, 237
vector-length, 33, 237
vector->list, 237
vector-ref, 32, 237
vectors, 6, 29, 32, 35, 100
  external representation of, 32
  indexes for, 32
  length of, 100
  printing, 100
  zero-referenced, 32
vector-set!, 33, 238
volume, 49
  current, 49
volume, 238

W, 140
wait, 238
while, 144
whitespace, 134
Whole file, 89
whole file,
  evaluating, 77, 90
wholes, 140
window?, 239
window-close, 239
window codes, 108
window-hide, 239
window-position, 239
window-print, 240
Windows, 72, 93
windows,
  Clipboard, 85
  closing, 70
  Debug Transcript, 5
  document, 13, 23
    indexing, 16
  Expression, 14
  graphics, 6, 53
  printing from, 71
  text, 6, 61
  Transcript, 5, 11, 20
  zooming, 93
window-select, 227, 240
window-set-position, 61, 179, 181, 240
window-set-title, 179, 240
window-show, 179, 241
windows preferences, 72
windows-preferences, 74, 241
windows-set-preferences, 74, 242
window title bar, 239
with-input-from-file, 242
with-output-to-file, 242
word wrap in text windows, 61
write, 52, 61, 243
write-char, 52, 243

zero?, 44, 243
zero-referenced list, 31

zero-referenced vectors, 32
Zoom, 93
zooming a window, 93

EdScheme for the Macintosh

Explore the art of programming with *EdScheme!*

EdScheme for the Macintosh is an incremental optimizing compiler for the Scheme language designed specifically with the learner in mind, but providing a complete implementation of the Scheme specification for the intermediate and advanced user. The programming environment takes advantage of the capabilities of the Macintosh computer, and includes:

- A full-featured integrated editor, with special capabilities such as parenthesis-matching, program formatting, file indexing, and template editing.
- Customized transcript and debugging windows featuring colored and styled text in addition to all the facilities provided by the integrated editor.
- A powerful and comprehensive turtle graphics interface, providing users with access to the Macintosh's Color QuickDraw graphics toolbox and many additional graphics capabilities.

At the same time, *EdScheme* is a powerful implementation, with:

- Unlimited precision 'bignum' integral and rational arithmetic, double-precision floating point arithmetic to approximately 16 significant digits, and complex number arithmetic.
- Comprehensive file-handling facilities, as well as access through the Macintosh's serial ports to external devices such as the Hyperbot robotic controller.
- Language extension using macros and transformers, support for advanced programming techniques such as object-oriented programming, delayed evaluation, and streams, and first-class continuations and environments.

Requirements
- Mac Plus or newer
- System 6.0.4 or better
- One megabyte RAM

ISBN 0-9628745-6-6