Calc Manual

GNU Emacs Calc Version 2.02

January 1992

Dave Gillespie
daveg@synaptics.com

Copyright © 1990, 1991 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the section entitled "GNU General Public License" is included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the section entitled "GNU General Public License" may be included in a translation approved by the author instead of in the original English.

GNU GENERAL PUBLIC LICENSE

Version 1, February 1989

Copyright © 1989 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

1. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.
3. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
   - cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
   - cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).
   - If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.
   - You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.
4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,

- accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

   Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.
6. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.
8. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.
9. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

10. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE

DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

11. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

1 Getting Started

This chapter provides a general overview of Calc, the GNU Emacs Calculator: What it is, how to start it and how to exit from it, and what are the various ways that it can be used.

1.1 What is Calc?

Calc is an advanced calculator and mathematical tool that runs as part of the GNU Emacs environment. Very roughly based on the HP-28/48 series of calculators, its many features include:

- Choice of algebraic or RPN (stack-based) entry of calculations.
- Arbitrary precision integers and floating-point numbers.
- Arithmetic on rational numbers, complex numbers (rectangular and polar), error forms with standard deviations, open and closed intervals, vectors and matrices, dates and times, infinities, sets, quantities with units, and algebraic formulas.
- Mathematical operations such as logarithms and trigonometric functions.
- Programmer's features (bitwise operations, non-decimal numbers).
- Financial functions such as future value and internal rate of return.
- Number theoretical features such as prime factorization and arithmetic modulo M for any M.
- Algebraic manipulation features, including symbolic calculus.
- Moving data to and from regular editing buffers.
- "Embedded mode" for manipulating Calc formulas and data directly inside any editing buffer.
- Graphics using GNUPLOT, a versatile (and free) plotting program.
- Easy programming using keyboard macros, algebraic formulas, algebraic rewrite rules, or extended Emacs Lisp.

Calc tries to include a little something for everyone; as a result it is large and might be intimidating to the first-time user. If you plan to use Calc only as a traditional desk calculator, all you really need to read is the "Getting Started" chapter of this manual and possibly the first few sections of the tutorial. As you become more comfortable with the program you can learn its additional features. In terms of efficiency, scope and depth, Calc cannot replace a powerful tool like Mathematica. But Calc has the advantages of convenience, portability, and availability of the source code. And, of course, it's free!

1.2 About This Manual

This document serves as a complete description of the GNU Emacs Calculator. It works both as an introduction for novices, and as a reference for experienced users. While it helps to have some experience with GNU Emacs in order to get the most out of Calc, this manual ought to be readable even if you don't know or use Emacs regularly.

The manual is divided into three major parts: the "Getting Started" chapter you are reading now, the Calc tutorial (chapter 2), and the Calc reference manual (the remaining chapters and appendices).

If you are in a hurry to use Calc, there is a brief "demonstration" below which illustrates the major features of Calc in just a couple of pages. If you don't have time to go through the full tutorial, this will show you everything you need to know to begin. See *XXX* [Demonstration of Calc], page *XXX*.

The tutorial chapter walks you through the various parts of Calc with lots of hands-on examples and explanations. If you are new to Calc and you have some time, try going through at least the beginning of the tutorial. The tutorial includes about 70 exercises with answers. These exercises give you some guided practice with Calc, as well as pointing out some interesting and unusual ways to use its features.

The reference section discusses Calc in complete depth. You can read the reference from start to finish if you want to learn every aspect of Calc. Or, you can look in the table of contents or the Concept Index to find the parts of the manual that discuss the things you need to know.

Every Calc keyboard command is listed in the Calc Summary, and also in the Key Index. Algebraic functions, *M-x* commands, and variables also have their own indices. Each paragraph that is referenced in the Key or Function Index is marked in the margin with its index entry.

You can access this manual on-line at any time within Calc by pressing the *h i* key sequence. Outside of the Calc window, you can press *M-# i* to read the manual on-line. Also, you can jump directly to the Tutorial by pressing *h t* or *M-# t*, or to the Summary by pressing *h s* or *M-# s*. Within Calc, you can also go to the part of the manual describing any Calc key, function, or variable using *h k*, *h f*, or *h v*, respectively. See *XXX* [Help Commands], page *XXX*.

Printed copies of this manual are also available from the Free Software Foundation.

1.3 Notations Used in This Manual

This section describes the various notations that are used throughout the Calc manual.

In keystroke sequences, uppercase letters mean you must hold down the shift key while typing the letter. Keys pressed with Control held down are shown as *C-x*. Keys pressed with Meta held down are shown as *M-x*. Other notations are *RET* for the Return key, *SPC* for the space bar, *TAB* for the Tab key, *DEL* for the Delete key, and *LFD* for the Line-Feed key.

(If you don't have the *LFD* or *TAB* keys on your keyboard, the *C-j* and *C-i* keys are equivalent to them, respectively. If you don't have a Meta key, look for Alt or Extend Char. You can also press *ESC* or *C-[* first to get the same effect, so that *M-x*, *ESC x*, and *C-[x* are all equivalent.)

Sometimes the *RET* key is not shown when it is "obvious" that you must press *RET* to proceed. For example, the *RET* is usually omitted in key sequences like *M-x calc-keypad RET*.

Commands are generally shown like this: *p* (`calc-precision`) or *M-# k* (`calc-keypad`). This means that the command is normally used by pressing the *p* key or *M-# k* key sequence, but it also has the full-name equivalent shown, e.g., *M-x calc-precision*.

Commands that correspond to functions in algebraic notation are written: *C* (`calc-cos`) [`cos`]. This means the *C* key is equivalent to *M-x calc-cos*, and that the corresponding function in an algebraic-style formula would be '`cos(x)`'.

A few commands don't have key equivalents: `calc-sincos` [`sincos`].

1.4 A Demonstration of Calc

This section will show some typical small problems being solved with Calc. The focus is more on demonstration than explanation, but everything you see here will be covered more thoroughly in the Tutorial.

To begin, start Emacs if necessary (usually the command `emacs` does this), and type *M-# c* (or *ESC # c*) to start the Calculator. (See *XXX* [Starting Calc], page *XXX*, if this doesn't work for you.)

Be sure to type all the sample input exactly, especially noting the difference between lower-case and upper-case letters. Remember, *RET*, *TAB*, *DEL*, and *SPC* are the Return, Tab, Delete, and Space keys.

RPN calculation. In RPN, you type the input number(s) first, then the command to operate on the numbers.

Type *2 RET 3 + Q* to compute $\sqrt{2+3} = 2.2360679775$.

Type *P 2 ^* to compute $\pi^2 = 9.86960440109$.

Type *TAB* to exchange the order of these two results.

Type *- I H S* to subtract these results and compute the Inverse Hyperbolic sine of the difference, 2.72996136574.

Type *DEL* to erase this result.

Algebraic calculation. You can also enter calculations using conventional "algebraic" notation. To enter an algebraic formula, use the apostrophe key.

Type *' sqrt(2+3) RET* to compute $\sqrt{2+3}$.

Type *' pi^2 RET* to enter π^2. To evaluate this symbolic formula as a number, type *=*.

Type *' arcsinh($ - $$) RET* to subtract the second-most-recent result from the most-recent and compute the Inverse Hyperbolic sine.

Keypad mode. If you are using the X window system, press *M-# k* to get Keypad mode. (If you don't use X, skip to the next section.)

Click on the *2*, *ENTER*, *3*, *+*, and *SQRT* "buttons" using your left mouse button.

Click on *PI*, *2*, and `y^x`.

Click on *INV*, then *ENTER* to swap the two results.

Click on *-*, *INV*, *HYP*, and *SIN*.

Click on `<-` to erase the result, then click *OFF* to turn the Keypad Calculator off.

Grabbing data. Type *M-# x* if necessary to exit Calc. Now select the following numbers as an Emacs region: "Mark" the front of the list by typing control-*SPC* or control-*@* there, then move to the other end of the list. (Either get this list from the on-line copy of this manual, accessed by *M-# i*, or just type these numbers into a scratch file.) Now type *M-# g* to "grab" these numbers into Calc.

```
1.23  1.97
1.6   2
1.19  1.08
```

The result '`[1.23, 1.97, 1.6, 2, 1.19, 1.08]`' is a Calc "vector." Type *V R +* to compute the sum of these numbers.

Type *U* to Undo this command, then type *V R ** to compute the product of the numbers.

You can also grab data as a rectangular matrix. Place the cursor on the upper-leftmost '`1`' and set the mark, then move to just after the lower-right '`8`' and press *M-# r*.

Type *v t* to transpose this 3×2 matrix into a 2×3 matrix. Type *v u* to unpack the rows into two separate vectors. Now type *V R + TAB V R +* to compute the sums of the two original columns. (There is also a special grab-and-sum-columns command, *M-# :*.)

Units conversion. Units are entered algebraically. Type *' 43 mi/hr RET* to enter the quantity 43 miles-per-hour. Type *u c km/hr RET*. Type *u c m/s RET*.

Date arithmetic. Type *t N* to get the current date and time. Type *90 +* to find the date 90 days from now. Type *' <25 dec 87> RET* to enter a date, then *- 7 /* to see how many weeks have passed since then.

Algebra. Algebraic entries can also include formulas or equations involving variables. Type *' [x + y = a, x y = 1] RET* to enter a pair of equations involving three variables. (Note the leading apostrophe in this example; also, note that the space between '`x y`' is required.) Type *a S x,y RET* to solve these equations for the variables x and y.

Type *d B* to view the solutions in more readable notation. Type *d C* to view them in C language notation, and *d T* to view them in the notation for the TeX typesetting system. Type *d N* to return to normal notation.

Type *7.5*, then *s l a RET* to let $a = 7.5$ in these formulas. (That's a letter *l*, not a numeral *1*.)

Help functions. You can read about any command in the on-line manual. Type *M-# c* to return to Calc after each of these commands: *h k t N* to read about the *t N* command, *h f sqrt RET* to read about the `sqrt` function, and *h s* to read the Calc summary.

Press *DEL* repeatedly to remove any leftover results from the stack. To exit from Calc, press *q* or *M-# c* again.

1.5 Using Calc

Calc has several user interfaces that are specialized for different kinds of tasks. As well as Calc's standard interface, there are Quick Mode, Keypad Mode, and Embedded Mode.

Calc must be *installed* before it can be used. See *XXX* [Installation], page *XXX*, for instructions on setting up and installing Calc. We will assume you or someone on your system has already installed Calc as described there.

1.5.1 Starting Calc

On most systems, you can type `M-#` to start the Calculator. The notation `M-#` is short for Meta-`#`. On most keyboards this means holding down the Meta (or Alt) and Shift keys while typing `3`.

Once again, if you don't have a Meta key on your keyboard you can type `ESC` first, then `#`, to accomplish the same thing. If you don't even have an `ESC` key, you can fake it by holding down Control or `CTRL` while typing a left square bracket (that's `C-[` in Emacs notation).

`M-#` is a *prefix key*; when you press it, Emacs waits for you to press a second key to complete the command. In this case, you will follow `M-#` with a letter (upper- or lower-case, it doesn't matter for `M-#`) that says which Calc interface you want to use.

To get Calc's standard interface, type `M-# c`. To get Keypad Mode, type `M-# k`. Type `M-# ?` to get a brief list of the available options, and type a second `?` to get a complete list.

To ease typing, `M-# M-#` (or `M-# #` if that's easier) also works to start Calc. It starts the same interface (either `M-# c` or `M-# k`) that you last used, selecting the `M-# c` interface by default. (If your installation has a special function key set up to act like `M-#`, hitting that function key twice is just like hitting `M-# M-#`.)

If `M-#` doesn't work for you, you can always type explicit commands like `M-x calc` (for the standard user interface) or `M-x calc-keypad` (for Keypad Mode). First type `M-x` (that's Meta with the letter `x`), then, at the prompt, type the full command (like `calc-keypad`) and press Return.

If you type `M-x calc` and Emacs still doesn't recognize the command (it will say '`[No match]`' when you try to press `RET`), then Calc has not been properly installed.

The same commands (like `M-# c` or `M-# M-#`) that start the Calculator also turn it off if it is already on.

1.5.2 The Standard Calc Interface

Calc's standard interface acts like a traditional RPN calculator, operated by the normal Emacs keyboard. When you type `M-# c` to start the Calculator, the Emacs screen splits into two windows with the file you were editing on top and Calc on the bottom.

```
...
--**-Emacs: myfile          (Fundamental)----All----------------------
--- Emacs Calculator Mode ---                   |Emacs Calc Mode v2.00...
2:  17.3                                        |    17.3
1:  -5                                          |    3
    .                                           |    2
                                                |    4
                                                |  * 8
                                                |  ->-5
                                                |
--%%-Calc: 12 Deg       (Calculator)----All----- --%%-Emacs: *Calc Trail*
```

In this figure, the mode-line for '`myfile`' has moved up and the "Calculator" window has appeared below it. As you can see, Calc actually makes two windows side-by-side. The lefthand one is called the *stack window* and the righthand one is called the *trail window*. The stack holds the numbers involved in the calculation you are currently performing. The trail holds a complete record of all calculations you have done. In a desk calculator with a printer, the trail corresponds to the paper tape that records what you do.

In this case, the trail shows that four numbers (17.3, 3, 2, and 4) were first entered into the Calculator, then the 2 and 4 were multiplied to get 8, then the 3 and 8 were subtracted to get −5. (The '`>`' symbol shows that this was the most recent calculation.) The net result is the two numbers 17.3 and −5 sitting on the stack.

Most Calculator commands deal explicitly with the stack only, but there is a set of commands that allow you to search back through the trail and retrieve any previous result.

Calc commands use the digits, letters, and punctuation keys. Shifted (i.e., upper-case) letters are different from lowercase letters. Some letters are *prefix* keys that begin two-letter commands. For example, *e* means "enter exponent" and shifted *E* means e^x. With the *d* ("display modes") prefix the letter "e" takes on very different meanings: *d e* means "engineering notation" and *d E* means "*eqn* language mode."

There is nothing stopping you from switching out of the Calc window and back into your editing window, say by using the Emacs *C-x o* (`other-window`) command. When the cursor is inside a regular window, Emacs acts just like normal. When the cursor is in the Calc stack or trail windows, keys are interpreted as Calc commands.

When you quit by pressing *M-# c* a second time, the Calculator windows go away but the actual Stack and Trail are not gone, just hidden. When you press *M-# c* once again you will get the same stack and trail contents you had when you last used the Calculator.

The Calculator does not remember its state between Emacs sessions. Thus if you quit Emacs and start it again, *M-# c* will give you a fresh stack and trail. There is a command (*m m*) that lets you save your favorite mode settings between sessions, though. One of the things it saves is which user interface (standard or Keypad) you last used; otherwise, a freshly started Emacs will always treat *M-# M-#* the same as *M-# c*.

The *q* key is another equivalent way to turn the Calculator off.

If you type *M-# b* first and then *M-# c*, you get a full-screen version of Calc (`full-calc`) in which the stack and trail windows are still side-by-side but are now as tall as the whole Emacs screen. When you press *q* or *M-# c* again to quit, the file you were editing before reappears. The *M-# b* key switches back and forth between "big" full-screen mode and the normal partial-screen mode.

Finally, *M-# o* (`calc-other-window`) is like *M-# c* except that the Calc window is not selected. The buffer you were editing before remains selected instead. *M-# o* is a handy way to switch out of Calc momentarily to edit your file; type *M-# c* to switch back into Calc when you are done.

1.5.3 Quick Mode (Overview)

Quick Mode is a quick way to use Calc when you don't need the full complexity of the stack and trail. To use it, type *M-# q* (`quick-calc`) in any regular editing buffer.

Quick Mode is very simple: It prompts you to type any formula in standard algebraic notation (like '`4 - 2/3`') and then displays the result at the bottom of the Emacs screen (3.33333333333 in this case). You are then back in the same editing buffer you were in before, ready to continue editing or to type *M-# q* again to do another quick calculation. The result of the calculation will also be in the Emacs "kill ring" so that a *C-y* command at this point will yank the result into your editing buffer.

Calc mode settings affect Quick Mode, too, though you will have to go into regular Calc (with *M-# c*) to change the mode settings.

See *XXX* [Quick Calculator], page *XXX*, for further information.

1.5.4 Keypad Mode (Overview)

Keypad Mode is a mouse-based interface to the Calculator. It is designed for use with the X window system. If you don't have X, you will have to operate keypad mode with your arrow keys (which is probably more trouble than it's worth). Keypad mode is currently not supported under Emacs 19.

Type *M-# k* to turn Keypad Mode on or off. Once again you get two new windows, this time on the righthand side of the screen instead of at the bottom. The upper window is the familiar Calc Stack; the lower window is a picture of a typical calculator keypad.

Keypad Mode is much easier for beginners to learn, because there is no need to memorize lots of obscure key sequences. But not all commands in regular Calc are available on the Keypad. You can always switch the cursor into the Calc stack window to use standard Calc commands if you need. Serious Calc users, though, often find they prefer the standard interface over Keypad Mode.

To operate the Calculator, just click on the "buttons" of the keypad using your left mouse button. To enter the two numbers shown here you would click *1 7 . 3 ENTER 5 +/- ENTER*; to add them together you would then click `+` (to get 12.3 on the stack).

If you click the right mouse button, the top three rows of the keypad change to show other sets of commands, such as advanced math functions, vector operations, and operations on binary numbers.

```
|--- Emacs Calculator Mode ---
|2:  17.3
|1:  -5
|    .
|--%%-Calc: 12 Deg       (Calcul
|----+-----Calc 2.00-----+----1
|FLR |CEIL|RND |TRNC|CLN2|FLT |
|----+----+----+----+----+----|
| LN |EXP |    |ABS |IDIV|MOD |
|----+----+----+----+----+----|
|SIN |COS |TAN |SQRT|y^x |1/x |
|----+----+----+----+----+----|
|  ENTER  |+/- |EEX |UNDO| <- |
|-----+---+-+--+--+-+---++----|
| INV |  7  |  8  |  9  |  /  |
|-----+-----+-----+-----+-----|
| HYP |  4  |  5  |  6  |  *  |
|-----+-----+-----+-----+-----|
|EXEC |  1  |  2  |  3  |  -  |
|-----+-----+-----+-----+-----|
| OFF |  0  |  .  | PI  |  +  |
|-----+-----+-----+-----+-----+
```

Because Keypad Mode doesn't use the regular keyboard, Calc leaves the cursor in your original editing buffer. You can type in this buffer in the usual way while also clicking on the Calculator keypad. One advantage of Keypad Mode is that you don't need an explicit command to switch between editing and calculating.

If you press *M-# b* first, you get a full-screen Keypad Mode (`full-calc-keypad`) with three windows: The keypad in the lower left, the stack in the lower right, and the trail on top.

See *XXX* [Keypad Mode], page *XXX*, for further information.

1.5.5 Standalone Operation

If you are not in Emacs at the moment but you wish to use Calc, you must start Emacs first. If all you want is to run Calc, you can give the commands:

```
emacs -f full-calc
```

or

```
emacs -f full-calc-keypad
```

which run a full-screen Calculator (as if by *M-# b M-# c*) or a full-screen X-based Calculator (as if by *M-# b M-# k*). In standalone operation, quitting the Calculator (by pressing *q* or clicking on the keypad *EXIT* button) quits Emacs itself.

1.5.6 Embedded Mode (Overview)

Embedded Mode is a way to use Calc directly from inside an editing buffer. Suppose you have a formula written as part of a document like this:

```
The derivative of

                                   ln(ln(x))

is
```

and you wish to have Calc compute and format the derivative for you and store this derivative in the buffer automatically. To do this with Embedded Mode, first copy the formula down to where you want the result to be:

```
The derivative of

                                   ln(ln(x))

is

                                   ln(ln(x))
```

Now, move the cursor onto this new formula and press *M-# e*. Calc will read the formula (using the surrounding blank lines to tell how much text to read), then push this formula (invisibly) onto the Calc stack. The cursor will stay on the formula in the editing buffer, but the buffer's mode line will change to look like the Calc mode line (with mode indicators like '`12 Deg`' and so on). Even though you are still in your editing buffer, the keyboard now acts like the Calc keyboard, and any new result you get is copied from the stack back into the buffer. To take the derivative, you would type *a d x RET*.

```
The derivative of

                                   ln(ln(x))

is

1 / ln(x) x
```

To make this look nicer, you might want to press *d =* to center the formula, and even *d B* to use "big" display mode.

```
The derivative of

                                  ln(ln(x))

is
% [calc-mode: justify: center]
% [calc-mode: language: big]

                                      1
                                   -------
                                   ln(x) x
```

Calc has added annotations to the file to help it remember the modes that were used for this formula. They are formatted like comments in the TEX typesetting language, just in case you are using TEX. (In this example TEX is not being used, so you might want to move these comments up to the top of the file or otherwise put them out of the way.)

As an extra flourish, we can add an equation number using a righthand label: Type *d } (1) RET*.

```
% [calc-mode: justify: center]
% [calc-mode: language: big]
% [calc-mode: right-label: " (1)"]

                                      1
                                   -------                          (1)
                                   ln(x) x
```

To leave Embedded Mode, type *M-# e* again. The mode line and keyboard will revert to the way they were before. (If you have actually been trying this as you read along, you'll want to press *M-# 0* [with the digit zero] now to reset the modes you changed.)

The related command *M-# w* operates on a single word, which generally means a single number, inside text. It uses any non-numeric characters rather than blank lines to delimit the formula it reads. Here's an example of its use:

```
A slope of one-third corresponds to an angle of 1 degrees.
```

Place the cursor on the '1', then type *M-# w* to enable Embedded Mode on that number. Now type *3 /* (to get one-third), and *I T* (the Inverse Tangent converts a slope into an angle), then *M-# w* again to exit Embedded mode.

```
A slope of one-third corresponds to an angle of 18.4349488229 degrees.
```

See *XXX* [Embedded Mode], page *XXX*, for full details.

1.5.7 Other `M-#` Commands

Two more Calc-related commands are `M-# g` and `M-# r`, which "grab" data from a selected region of a buffer into the Calculator. The region is defined in the usual Emacs way, by a "mark" placed at one end of the region, and the Emacs cursor or "point" placed at the other.

The `M-# g` command reads the region in the usual left-to-right, top-to-bottom order. The result is packaged into a Calc vector of numbers and placed on the stack. Calc (in its standard user interface) is then started. Type `v u` if you want to unpack this vector into separate numbers on the stack. Also, `C-u M-# g` interprets the region as a single number or formula.

The `M-# r` command reads a rectangle, with the point and mark defining opposite corners of the rectangle. The result is a matrix of numbers on the Calculator stack.

Complementary to these is `M-# y`, which "yanks" the value at the top of the Calc stack back into an editing buffer. If you type `M-# y` while in such a buffer, the value is yanked at the current position. If you type `M-# y` while in the Calc buffer, Calc makes an educated guess as to which editing buffer you want to use. The Calc window does not have to be visible in order to use this command, as long as there is something on the Calc stack.

Here, for reference, is the complete list of `M-#` commands. The shift, control, and meta keys are ignored for the keystroke following `M-#`.

Commands for turning Calc on and off:

`#`	Turn Calc on or off, employing the same user interface as last time.
`C`	Turn Calc on or off using its standard bottom-of-the-screen interface. If Calc is already turned on but the cursor is not in the Calc window, move the cursor into the window.
`O`	Same as `C`, but don't select the new Calc window. If Calc is already turned on and the cursor is in the Calc window, move it out of that window.
`B`	Control whether `M-# c` and `M-# k` use the full screen.
`Q`	Use Quick Mode for a single short calculation.
`K`	Turn Calc Keypad mode on or off.
`E`	Turn Calc Embedded mode on or off at the current formula.
`J`	Turn Calc Embedded mode on or off, select the interesting part.
`W`	Turn Calc Embedded mode on or off at the current word (number).
`Z`	Turn Calc on in a user-defined way, as defined by a `Z I` command.
`X`	Quit Calc; turn off standard, Keypad, or Embedded mode if on. (This is like `q` or `OFF` inside of Calc.)

Commands for moving data into and out of the Calculator:

G	Grab the region into the Calculator as a vector.
R	Grab the rectangular region into the Calculator as a matrix.
:	Grab the rectangular region and compute the sums of its columns.
_	Grab the rectangular region and compute the sums of its rows.
Y	Yank a value from the Calculator into the current editing buffer.

Commands for use with Embedded Mode:

A	"Activate" the current buffer. Locate all formulas that contain '`:=`' or '`=>`' symbols and record their locations so that they can be updated automatically as variables are changed.
D	Duplicate the current formula immediately below and select the duplicate.
F	Insert a new formula at the current point.
N	Move the cursor to the next active formula in the buffer.
P	Move the cursor to the previous active formula in the buffer.
U	Update (i.e., as if by the `=` key) the formula at the current point.
`	Edit (as if by `calc-edit`) the formula at the current point.

Miscellaneous commands:

I	Run the Emacs Info system to read the Calc manual. (This is the same as *h i* inside of Calc.)
T	Run the Emacs Info system to read the Calc Tutorial.
S	Run the Emacs Info system to read the Calc Summary.
L	Load Calc entirely into memory. (Normally the various parts are loaded only as they are needed.)
M	Read a region of written keystroke names (like '`C-n a b c RET`') and record them as the current keyboard macro.
0	(This is the "zero" digit key.) Reset the Calculator to its default state: Empty stack, and default mode settings. With any prefix argument, reset everything but the stack.

1.6 History and Acknowledgements

Calc was originally started as a two-week project to occupy a lull in the author's schedule. Basically, a friend asked if I remembered the value of 2^{32}. I didn't offhand, but I said, "that's easy, just call up an `xcalc`." `Xcalc` duly reported that the answer to our question was '`4.294967e+09`'—with no way to see the full ten digits even though we knew they were there in the program's memory! I was so annoyed, I vowed to write a calculator of my own, once and for all.

I chose Emacs Lisp, a) because I had always been curious about it and b) because, being only a text editor extension language after all, Emacs Lisp would surely reach its limits long before the project got too far out of hand.

To make a long story short, Emacs Lisp turned out to be a distressingly solid implementation of Lisp, and the humble task of calculating turned out to be more open-ended than one might have expected.

Emacs Lisp doesn't have built-in floating point math, so it had to be simulated in software. In fact, Emacs integers will only comfortably fit six decimal digits or so—not enough for a decent calculator. So I had to write my own high-precision integer code as well, and once I had this I figured that arbitrary-size integers were just as easy as large integers. Arbitrary floating-point precision was the logical next step. Also, since the large integer arithmetic was there anyway it seemed only fair to give the user direct access to it, which in turn made it practical to support fractions as well as floats. All these features inspired me to look around for other data types that might be worth having.

Around this time, my friend Rick Koshi showed me his nifty new HP-28 calculator. It allowed the user to manipulate formulas as well as numerical quantities, and it could also operate on matrices. I decided that these would be good for Calc to have, too. And once things had gone this far, I figured I might as well take a look at serious algebra systems like Mathematica, Macsyma, and Maple for further ideas. Since these systems did far more than I could ever hope to implement, I decided to focus on rewrite rules and other programming features so that users could implement what they needed for themselves.

Rick complained that matrices were hard to read, so I put in code to format them in a 2D style. Once these routines were in place, Big mode was obligatory. Gee, what other language modes would be useful?

Scott Hemphill and Allen Knutson, two friends with a strong mathematical bent, contributed ideas and algorithms for a number of Calc features including modulo forms, primality testing, and float-to-fraction conversion.

Units were added at the eager insistence of Mass Sivilotti. Later, Ulrich Mueller at CERN and Przemek Klosowski at NIST provided invaluable expert assistance with the units table. As far as I can remember, the idea of using algebraic formulas and variables to represent units dates back to an ancient article in Byte magazine about muMath, an early algebra system for microcomputers.

Many people have contributed to Calc by reporting bugs and suggesting features, large and small. A few deserve special mention: Tim Peters, who helped develop the ideas that led to the selection commands, rewrite rules, and many other algebra features; François Pinard, who contributed an early prototype of the Calc Summary appendix as well as providing valuable suggestions in many other areas of Calc; Carl Witty, whose eagle eyes discovered many typographical and factual errors in the Calc manual; Tim Kay, who drove the development of Embedded mode; Ove Ewerlid, who made many suggestions relating to the algebra commands and contributed some code for polynomial operations; Randal Schwartz, who suggested the `calc-eval` function; Robert J. Chassell, who suggested the Calc Tutorial and exercises; and Juha Sarlin, who first worked out how to split Calc into quickly-loading parts. Bob Weiner helped immensely with the Lucid Emacs port.

Among the books used in the development of Calc were Knuth's *Art of Computer Programming* (especially volume II, *Seminumerical Algorithms*); *Numerical Recipes* by Press, Flannery, Teukolsky, and Vetterling; Bevington's *Data Reduction and Error Analysis for the Physical Sciences*; *Concrete Mathematics* by Graham, Knuth, and Patashnik; Steele's *Common Lisp, the Language*; the *CRC Standard Math Tables* (William H. Beyer, ed.); and Abramowitz and Stegun's venerable *Handbook of Mathematical Functions*. I consulted the user's manuals for the HP-28 and HP-48 calculators, as well as for the programs Mathematica, SMP, Macsyma, Maple, MathCAD, Gnuplot, and others. Also, of course, Calc could not have been written without the excellent *GNU Emacs Lisp Reference Manual*, by Bil Lewis and Dan LaLiberte.

Final thanks go to Richard Stallman, without whose fine implementations of the Emacs editor, language, and environment, Calc would have been finished in two weeks.

2 Tutorial

This chapter explains how to use Calc and its many features, in a step-by-step, tutorial way. You are encouraged to run Calc and work along with the examples as you read (see *XXX* [Starting Calc], page *XXX*). If you are already familiar with advanced calculators, you may wish to skip on to the rest of this manual.

This tutorial describes the standard user interface of Calc only. The "Quick Mode" and "Keypad Mode" interfaces are fairly self-explanatory. See *XXX* [Embedded Mode], page *XXX*, for a description of the "Embedded Mode" interface.

The easiest way to read this tutorial on-line is to have two windows on your Emacs screen, one with Calc and one with the Info system. (If you have a printed copy of the manual you can use that instead.) Press *`M-# c`* to turn Calc on or to switch into the Calc window, and press *`M-# i`* to start the Info system or to switch into its window.

This tutorial is designed to be done in sequence. But the rest of this manual does not assume you have gone through the tutorial. The tutorial does not cover everything in the Calculator, but it touches on most general areas.

The Calc Summary at the end of the reference manual includes some blank space for your own use. You may wish to keep notes there as you learn Calc.

2.1 Basic Tutorial

In this section, we learn how RPN and algebraic-style calculations work, how to undo and redo an operation done by mistake, and how to control various modes of the Calculator.

2.1.1 RPN Calculations and the Stack

Calc normally uses RPN notation. You may be familiar with the RPN system from Hewlett-Packard calculators, FORTH, or PostScript. (Reverse Polish Notation, RPN, is named after the Polish mathematician Jan Łukasiewicz.)

The central component of an RPN calculator is the *stack*. A calculator stack is like a stack of dishes. New dishes (numbers) are added at the top of the stack, and numbers are normally only removed from the top of the stack.

In an operation like $2 + 3$, the 2 and 3 are called the *operands* and the + is the *operator*. In an RPN calculator you always enter the operands first, then the operator. Each time you type a number, Calc adds or *pushes* it onto the top of the Stack. When you press an operator key like `+`, Calc *pops* the appropriate number of operands from the stack and pushes back the result.

Thus we could add the numbers 2 and 3 in an RPN calculator by typing: *`2 RET 3 RET +`*. (The *`RET`* key, Return, corresponds to the *`ENTER`* key on traditional RPN calculators.) Try this now if you wish; type *`M-# c`* to switch into the Calc window (you can type *`M-# c`* again or *`M-# o`* to switch back to the Tutorial window). The first four keystrokes "push" the numbers 2 and 3 onto the stack.

The + key "pops" the top two numbers from the stack, adds them, and pushes the result (5) back onto the stack. Here's how the stack will look at various points throughout the calculation:

```
          .              1:  2          2:  2          1:  5              .
                             .          1:  3              .
                                            .

       M-# c              2 RET          3 RET           +              DEL
```

The '.' symbol is a marker that represents the top of the stack. Note that the "top" of the stack is really shown at the bottom of the Stack window. This may seem backwards, but it turns out to be less distracting in regular use.

The numbers '1:' and '2:' on the left are *stack level numbers*. Old RPN calculators always had four stack levels called x, y, z, and t. Calc's stack can grow as large as you like, so it uses numbers instead of letters. Some stack-manipulation commands accept a numeric argument that says which stack level to work on. Normal commands like + always work on the top few levels of the stack.

The Stack buffer is just an Emacs buffer, and you can move around in it using the regular Emacs motion commands. But no matter where the cursor is, even if you have scrolled the '.' marker out of view, most Calc commands always move the cursor back down to level 1 before doing anything. It is possible to move the '.' marker upwards through the stack, temporarily "hiding" some numbers from commands like +. This is called *stack truncation* and we will not cover it in this tutorial; see *XXX* [Truncating the Stack], page *XXX*, if you are interested.

You don't really need the second *RET* in *2 RET 3 RET +*. That's because if you type any operator name or other non-numeric key when you are entering a number, the Calculator automatically enters that number and then does the requested command. Thus *2 RET 3 +* will work just as well.

Examples in this tutorial will often omit *RET* even when the stack displays shown would only happen if you did press *RET*:

```
   1:  2          2:  2          1:  5
       .          1:  3              .
                      .

    2 RET             3              +
```

Here, after pressing *3* the stack would really show '1: 2' with 'Calc: 3' in the minibuffer. In these situations, you can press the optional *RET* to see the stack as the figure shows.

(•) **Exercise 1.** (This tutorial will include exercises at various points. Try them if you wish. Answers to all the exercises are located at the end of the Tutorial chapter. Each exercise will include a cross-reference to its particular answer. If you are reading with the Emacs Info system, press *f* and the exercise number to go to the answer, then the letter *l* to return to where you were.)

Here's the first exercise: What will the keystrokes *1 RET 2 RET 3 RET 4 + * -* compute? ('*' is the symbol for multiplication.) Figure it out by hand, then try it with Calc to see if you're right. See *XXX* [RPN Answer 1], page *XXX*. (•)

(•) **Exercise 2.** Compute $(2 \times 4) + (7 \times 9.4) + \frac{5}{4}$ using the stack. See *XXX* [RPN Answer 2], page *XXX*. (•)

The *DEL* key is called Backspace on some keyboards. It is whatever key you would use to correct a simple typing error when regularly using Emacs. The *DEL* key pops and throws away the top value on the stack. (You can still get that value back from the Trail if you should need it later on.) There are many places in this tutorial where we assume you have used *DEL* to erase the results of the previous example at the beginning of a new example. In the few places where it is really important to use *DEL* to clear away old results, the text will remind you to do so.

(It won't hurt to let things accumulate on the stack, except that whenever you give a display-mode-changing command Calc will have to spend a long time reformatting such a large stack.)

Since the `-` key is also an operator (it subtracts the top two stack elements), how does one enter a negative number? Calc uses the `_` (underscore) key to act like the minus sign in a number. So, typing *-5 RET* won't work because the `-` key will try to do a subtraction, but *_5 RET* works just fine.

You can also press *n*, which means "change sign." It changes the number at the top of the stack (or the number being entered) from positive to negative or vice-versa: *5 n RET*.

If you press *RET* when you're not entering a number, the effect is to duplicate the top number on the stack. Consider this calculation:

```
1:  3              2:  3              1:  9              2:  9              1:  81
    .              1:  3                  .              1:  9                  .
                       .                                     .

   3 RET              RET                 *                 RET                 *
```

(Of course, an easier way to do this would be *3 RET 4 ^*, to raise 3 to the fourth power.)

The space-bar key (denoted *SPC* here) performs the same function as *RET*; you could replace all three occurrences of *RET* in the above example with *SPC* and the effect would be the same.

Another stack manipulation key is *TAB*. This exchanges the top two stack entries. Suppose you have computed *2 RET 3 +* to get 5, and then you realize what you really wanted to compute was $20/(2+3)$.

```
1:  5              2:  5              2:  20             1:  4
    .              1:  20             1:  5                  .
                       .                  .

  2 RET 3 +           20                 TAB                /
```

Planning ahead, the calculation would have gone like this:

```
1:  20             2:  20             3:  20             2:  20             1:  4
    .              1:  2              2:  2              1:  5                  .
                       .              1:  3                  .
                                          .

   20 RET            2 RET                3                  +                  /
```

A related stack command is *M-TAB* (hold *META* and type *TAB*). It rotates the top three elements of the stack upward, bringing the object in level 3 to the top.

```
1:  10             2:  10             3:  10             3:  20             3:  30
    .              1:  20             2:  20             2:  30             2:  10
                       .              1:  30             1:  10             1:  20
                                          .                  .                  .

   10 RET            20 RET             30 RET             M-TAB              M-TAB
```

(•) **Exercise 3.** Suppose the numbers 10, 20, and 30 are on the stack. Figure out how to add one to the number in level 2 without affecting the rest of the stack. Also figure out how to add one to the number in level 3. See *XXX* [RPN Answer 3], page *XXX*. (•)

Operations like `+`, `-`, `*`, `/`, and `^` pop two arguments from the stack and push a result. Operations like `n` and `Q` (square root) pop a single number and push the result. You can think of them as simply operating on the top element of the stack.

```
1:  3           1:  9           2:  9           1:  25          1:  5
    .               .           1:  16              .               .
                                    .

  3 RET           RET *         4 RET RET *         +               Q
```

(Note that capital `Q` means to hold down the Shift key while typing `q`. Remember, plain unshifted `q` is the Quit command.)

Here we've used the Pythagorean Theorem to determine the hypotenuse of a right triangle. Calc actually has a built-in command for that called `f h`, but let's suppose we can't remember the necessary keystrokes. We can still enter it by its full name using `M-x` notation:

```
1:  3           2:  3           1:  5
    .           1:  4               .
                    .

  3 RET           4 RET       M-x calc-hypot
```

All Calculator commands begin with the word '`calc-`'. Since it gets tiring to type this, Calc provides an `x` key which is just like the regular Emacs `M-x` key except that it types the '`calc-`' prefix for you:

```
1:  3           2:  3           1:  5
    .           1:  4               .
                    .

  3 RET           4 RET          x hypot
```

What happens if you take the square root of a negative number?

```
1:  4           1:  -4          1:  (0, 2)
    .               .               .

  4 RET             n               Q
```

The notation (a, b) represents a complex number. Complex numbers are more traditionally written $a + bi$; Calc can display in this format, too, but for now we'll stick to the (a, b) notation.

If you don't know how complex numbers work, you can safely ignore this feature. Complex numbers only arise from operations that would be errors in a calculator that didn't have complex numbers. (For example, taking the square root or logarithm of a negative number produces a complex result.)

Complex numbers are entered in the notation shown. The `(` and `,` and `)` keys manipulate "incomplete complex numbers."

```
1:  ( ...       2:  ( ...       1:  (2, ...     1:  (2, ...     1:  (2, 3)
    .           1:  2               .               3               .
                    .                               .

    (               2               ,               3               )
```

You can perform calculations while entering parts of incomplete objects. However, an incomplete object cannot actually participate in a calculation:

```
1:  ( ...       2:  ( ...       3:  ( ...       1:  ( ...       1:  ( ...
    .           1:  2           2:  2               5               5
                    .           1:  3               .               .
                                    .
                                                                 (error)
    (              2 RET            3               +               +
```

Adding 5 to an incomplete object makes no sense, so the last command produces an error message and leaves the stack the same.

Incomplete objects can't participate in arithmetic, but they can be moved around by the regular stack commands.

```
2:  2           3:  2           3:  3           1:  ( ...       1:  (2, 3)
1:  3           2:  3           2:  ( ...           2               .
    .           1:  ( ...       1:  2               3
                    .               .               .

2 RET 3 RET         (             M-TAB           M-TAB             )
```

Note that the `,` (comma) key did not have to be used here. When you press `)` all the stack entries between the incomplete entry and the top are collected, so there's never really a reason to use the comma. It's up to you.

(•) **Exercise 4.** To enter the complex number (2, 3), your friend Joe typed `( 2 , SPC 3 )`. What happened? (Joe thought of a clever way to correct his mistake in only two keystrokes, but it didn't quite work. Try it to find out why.) See *XXX* [RPN Answer 4], page *XXX*. (•)

Vectors are entered the same way as complex numbers, but with square brackets in place of parentheses. We'll meet vectors again later in the tutorial.

Any Emacs command can be given a *numeric prefix argument* by typing a series of `META`-digits beforehand. If `META` is awkward for you, you can instead type `C-u` followed by the necessary digits. Numeric prefix arguments can be negative, as in `M-- M-3 M-5` or `C-u - 3 5`. Calc commands use numeric prefix arguments in a variety of ways. For example, a numeric prefix on the `+` operator adds any number of stack entries at once:

```
1:  10          2:  10          3:  10          3:  10          1:  60
    .           1:  20          2:  20          2:  20              .
                    .           1:  30          1:  30
                                    .               .

  10 RET          20 RET          30 RET          C-u 3             +
```

For stack manipulation commands like `RET`, a positive numeric prefix argument operates on the top *n* stack entries at once. A negative argument operates on the entry in level *n* only. An argument of zero operates on the entire stack. In this example, we copy the second-to-top element of the stack:

```
1:  10          2:  10          3:  10          3:  10          4:  10
    .           1:  20          2:  20          2:  20          3:  20
                    .           1:  30          1:  30          2:  30
                                    .               .           1:  20
                                                                    .

  10 RET          20 RET          30 RET          C-u -2          RET
```

Another common idiom is *M-0 DEL*, which clears the stack. (The *M-0* numeric prefix tells *DEL* to operate on the entire stack.)

2.1.2 Algebraic-Style Calculations

If you are not used to RPN notation, you may prefer to operate the Calculator in "algebraic mode," which is closer to the way non-RPN calculators work. In algebraic mode, you enter formulas in traditional $2+3$ notation.

You don't really need any special "mode" to enter algebraic formulas. You can enter a formula at any time by pressing the apostrophe (*'*) key. Answer the prompt with the desired formula, then press *RET*. The formula is evaluated and the result is pushed onto the RPN stack. If you don't want to think in RPN at all, you can enter your whole computation as a formula, read the result from the stack, then press *DEL* to delete it from the stack.

Try pressing the apostrophe key, then *2+3+4*, then *RET*. The result should be the number 9.

Algebraic formulas use the operators '`+`', '`-`', '`*`', '`/`', and '`^`'. You can use parentheses to make the order of evaluation clear. In the absence of parentheses, '`^`' is evaluated first, then '`*`', then '`/`', then finally '`+`' and '`-`'. For example, the expression

```
2 + 3*4*5 / 6*7^8 - 9
```

is equivalent to

```
2 + ((3*4*5) / (6*(7^8)) - 9
```

or, in large mathematical notation,

$$2+\frac{3\times4\times5}{6\times7^8}-9$$

The result of this expression will be the number -6.99999826533.

Calc's order of evaluation is the same as for most computer languages, except that '`*`' binds more strongly than '`/`', as the above example shows. As in normal mathematical notation, the '`*`' symbol can often be omitted: '`2 a`' is the same as '`2*a`'.

Operators at the same level are evaluated from left to right, except that '`^`' is evaluated from right to left. Thus, '`2-3-4`' is equivalent to '`(2-3)-4`' or -5, whereas '`2^3^4`' is equivalent to '`2^(3^4)`' (a very large integer; try it!).

If you tire of typing the apostrophe all the time, there is an "algebraic mode" you can select in which Calc automatically senses when you are about to type an algebraic expression. To enter this mode, press the two letters *m a*. (An '`Alg`' indicator should appear in the Calc window's mode line.)

Press *m a*, then *2+3+4* with no apostrophe, then *RET*.

In algebraic mode, when you press any key that would normally begin entering a number (such as a digit, a decimal point, or the _ key), or if you press *(* or *[*, Calc automatically begins an algebraic entry.

Functions which do not have operator symbols like '`+`' and '`*`' must be entered in formulas using function-call notation. For example, the function name corresponding to the square-root key *Q* is `sqrt`. To compute a square root in a formula, you would use the notation '`sqrt(`*x*`)`'.

Press the apostrophe, then type *sqrt(5*2) - 3*. The result should be 0.16227766017.

Note that if the formula begins with a function name, you need to use the apostrophe even if you are in algebraic mode. If you type *arcsin* out of the blue, the *a r* will be taken as an Algebraic Rewrite command, and the *csin* will be taken as the name of the rewrite rule to use!

Some people prefer to enter complex numbers and vectors in algebraic form because they find RPN entry with incomplete objects to be too distracting, even though they otherwise use Calc as an RPN calculator.

Still in algebraic mode, type:

```
1:  (2, 3)      2:  (2, 3)      1:  (8, -1)     2:  (8, -1)     1:  (9, -1)
    .           1:  (1, -2)         .           1:  1               .
                    .                               .

  (2,3) RET       (1,-2) RET        *              1 RET            +
```

Algebraic mode allows us to enter complex numbers without pressing an apostrophe first, but it also means we need to press *RET* after every entry, even for a simple number like 1.

(You can type *C-u m a* to enable a special "incomplete algebraic mode" in which the *(* and *[* keys use algebraic entry even though regular numeric keys still use RPN numeric entry. There is also a "total algebraic mode," started by typing *m t*, in which all normal keys begin algebraic entry. You must then use the *META* key to type Calc commands: *M-m t* to get back out of total algebraic mode, *M-q* to quit, etc. Total algebraic mode is not supported under Emacs 19.)

If you're still in algebraic mode, press *m a* again to turn it off.

Actual non-RPN calculators use a mixture of algebraic and RPN styles. In general, operators of two numbers (like `+` and `*`) use algebraic form, but operators of one number (like *n* and *Q*) use RPN form. Also, a non-RPN calculator allows you to see the intermediate results of a calculation as you go along. You can accomplish this in Calc by performing your calculation as a series of algebraic entries, using the *$* sign to tie them together. In an algebraic formula, *$* represents the number on the top of the stack. Here, we perform the calculation $\sqrt{2 \times 4 + 1}$, which on a traditional calculator would be done by pressing *2 * 4 + 1 =* and then the square-root key.

```
1:  8           1:  9           1:  3
    .               .               .

   ' 2*4 RET       $+1 RET         Q
```

Notice that we didn't need to press an apostrophe for the *$+1*, because the dollar sign always begins an algebraic entry.

(•) **Exercise 1.** How could you get the same effect as pressing *Q* but using an algebraic entry instead? How about if the *Q* key on your keyboard were broken? See *XXX* [Algebraic Answer 1], page *XXX*. (•)

The notations *$$*, *$$$*, and so on stand for higher stack entries. For example, *' $$+$ RET* is just like typing `+`.

Algebraic formulas can include *variables*. To store in a variable, press *s s*, then type the variable name, then press *RET*. (There are actually two flavors of store command: *s s* stores a number in a variable but also leaves the number on the stack, while *s t* removes a number from the stack and stores it in the variable.) A variable name should consist of one or more letters or digits, beginning with a letter.

```
1:  17                .              1:  a + a^2     1:  306
    .                                    .               .

    17            s t a RET       ' a+a^2 RET          =
```

The `=` key *evaluates* a formula by replacing all its variables by the values that were stored in them.

For RPN calculations, you can recall a variable's value on the stack either by entering its name as a formula and pressing `=`, or by using the *s r* command.

```
1:  17            2:  17          3:  17          2:  17          1:  306
    .             1:  17          2:  17          1:  289             .
                      .           1:  2               .
                                      .

  s r a RET       ' a RET =           2               ^               +
```

If you press a single digit for a variable name (as in *s t 3*, you get one of ten *quick variables* `q0` through `q9`. They are "quick" simply because you don't have to type the letter `q` or the *RET* after their names. In fact, you can type simply *s 3* as a shorthand for *s s 3*, and likewise for *t 3* and *r 3*.

Any variables in an algebraic formula for which you have not stored values are left alone, even when you evaluate the formula.

```
1:  2 a + 2 b     1:  34 + 2 b
    .                 .

  ' 2a+2b RET             =
```

Calls to function names which are undefined in Calc are also left alone, as are calls for which the value is undefined.

```
1:  2 + log10(0) + log10(x) + log10(5, 6) + foo(3)
    .

  ' log10(100) + log10(0) + log10(x) + log10(5,6) + foo(3) RET
```

In this example, the first call to `log10` works, but the other calls are not evaluated. In the second call, the logarithm is undefined for that value of the argument; in the third, the argument is symbolic, and in the fourth, there are too many arguments. In the fifth case, there is no function called `foo`. You will see a "Wrong number of arguments" message referring to '`log10(5,6)`'. Press the *w* ("why") key to see any other messages that may have arisen from the last calculation. In this case you will get "logarithm of zero," then "number expected: `x`". Calc automatically displays the first message only if the message is sufficiently important; for example, Calc considers "wrong number of arguments" and "logarithm of zero" to be important enough to report automatically, while a message like "number expected: `x`" will only show up if you explicitly press the *w* key.

(•) **Exercise 2.** Joe entered the formula '`2 x y`', stored 5 in `x`, pressed `=`, and got the expected result, '`10 y`'. He then tried the same for the formula '`2 x (1+y)`', expecting '`10 (1+y)`', but it didn't work. Why not? See *XXX* [Algebraic Answer 2], page *XXX*. (•)

(•) **Exercise 3.** What result would you expect *1 RET 0 /* to give? What if you then type *0 **? See *XXX* [Algebraic Answer 3], page *XXX*. (•)

One interesting way to work with variables is to use the *evaluates-to* ('`=>`') operator. It works like this: Enter a formula algebraically in the usual way, but follow the formula with an '`=>`' symbol. (There is also an *s =* command which builds an '`=>`' formula using the stack.) On the stack, you will see two copies of the formula with an '`=>`' between them. The lefthand formula is exactly like you typed it; the righthand formula has been evaluated as if by typing `=`.

```
2:  2 + 3 => 5                    2:  2 + 3 => 5
1:  2 a + 2 b => 34 + 2 b         1:  2 a + 2 b => 20 + 2 b
    .                                 .

' 2+3 => RET  ' 2a+2b RET s =         10 s t a RET
```

Notice that the instant we stored a new value in `a`, all '`=>`' operators already on the stack that referred to *a* were updated to use the new value. With '`=>`', you can push a set of formulas on the stack, then change the variables experimentally to see the effects on the formulas' values.

You can also "unstore" a variable when you are through with it:

```
2:  2 + 5 => 5
1:  2 a + 2 b => 2 a + 2 b
    .

    s u a RET
```

We will encounter formulas involving variables and functions again when we discuss the algebra and calculus features of the Calculator.

2.1.3 Undo and Redo

If you make a mistake, you can usually correct it by pressing shift-*U*, the "undo" command. First, clear the stack (*M-0 DEL*) and exit and restart Calc (*M-# M-# M-# M-#*) to make sure things start off with a clean slate. Now:

```
1:  2          2:  2          1:  8          2:  2          1:  6
    .          1:  3              .          1:  3              .
                   .                             .

   2 RET           3              ^              U              *
```

You can undo any number of times. Calc keeps a complete record of all you have done since you last opened the Calc window. After the above example, you could type:

```
1:  6          2:  2          1:  2              .              .
    .          1:  3              .
                   .
                                                             (error)
                   U              U              U              U
```

You can also type *D* to “redo” a command that you have undone mistakenly.

```
    .              1:  2           2:  2           1:  6           1:  6
                       .           1:  3               .               .
                                       .
                                                                  (error)
                       D               D               D               D
```

It was not possible to redo past the 6, since that was placed there by something other than an undo command.

You can think of undo and redo as a sort of “time machine.” Press *U* to go backward in time, *D* to go forward. If you go backward and do something (like `*`) then, as any science fiction reader knows, you have changed your future and you cannot go forward again. Thus, the inability to redo past the 6 even though there was an earlier undo command.

You can always recall an earlier result using the Trail. We’ve ignored the trail so far, but it has been faithfully recording everything we did since we loaded the Calculator. If the Trail is not displayed, press *t d* now to turn it on.

Let’s try grabbing an earlier result. The 8 we computed was undone by a *U* command, and was lost even to Redo when we pressed `*`, but it’s still there in the trail. There should be a little ‘`>`’ arrow (the *trail pointer*) resting on the last trail entry. If there isn’t, press *t]* to reset the trail pointer. Now, press *t p* to move the arrow onto the line containing 8, and press *t y* to “yank” that number back onto the stack.

If you press *t]* again, you will see that even our Yank command went into the trail.

Let’s go further back in time. Earlier in the tutorial we computed a huge integer using the formula ‘`2^3^4`’. We don’t remember what it was, but the first digits were “241”. Press *t r* (which stands for trail-search-reverse), then type *241*. The trail cursor will jump back to the next previous occurrence of the string “241” in the trail. This is just a regular Emacs incremental search; you can now press *C-s* or *C-r* to continue the search forwards or backwards as you like.

To finish the search, press *RET*. This halts the incremental search and leaves the trail pointer at the thing we found. Now we can type *t y* to yank that number onto the stack. If we hadn’t remembered the “241”, we could simply have searched for *2^3^4*, then pressed *RET t n* to halt and then move to the next item.

You may have noticed that all the trail-related commands begin with the letter *t*. (The store-and-recall commands, on the other hand, all began with *s*.) Calc has so many commands that there aren’t enough keys for all of them, so various commands are grouped into two-letter sequences where the first letter is called the *prefix* key. If you type a prefix key by accident, you can press *C-g* to cancel it. (In fact, you can press *C-g* to cancel almost anything in Emacs.) To get help on a prefix key, press that key followed by *?*. Some prefixes have several lines of help, so you need to press *?* repeatedly to see them all. This may not work under Lucid Emacs, but you can also type *h h* to see all the help at once.

Try pressing *t ?* now. You will see a line of the form,

```
trail/time: Display; Fwd, Back; Next, Prev, Here, [, ]; Yank:  [MORE]  t-
```

The word “trail” indicates that the *t* prefix key contains trail-related commands. Each entry on the line shows one command, with a single capital letter showing which letter you press to get that command. We have used *t n*, *t p*, *t]*, and *t y* so far. The ‘`[MORE]`’ means you can press *?* again to see more *t*-prefix comands. Notice that the commands are roughly divided (by semicolons) into related groups.

When you are in the help display for a prefix key, the prefix is still active. If you press another key, like *y* for example, it will be interpreted as a *t y* command. If all you wanted was to look at the help messages, press *C-g* afterwards to cancel the prefix.

One more way to correct an error is by editing the stack entries. The actual Stack buffer is marked read-only and must not be edited directly, but you can press *`* (the backquote or accent grave) to edit a stack entry.

Try entering '`3.141439`' now. If this is supposed to represent π, it's got several errors. Press *`* to edit this number. Now use the normal Emacs cursor motion and editing keys to change the second 4 to a 5, and to transpose the 3 and the 9. When you press *RET*, the number on the stack will be replaced by your new number. This works for formulas, vectors, and all other types of values you can put on the stack. The *`* key also works during entry of a number or algebraic formula.

2.1.4 Mode-Setting Commands

Calc has many types of *modes* that affect the way it interprets your commands or the way it displays data. We have already seen one mode, namely algebraic mode. There are many others, too; we'll try some of the most common ones here.

Perhaps the most fundamental mode in Calc is the current *precision*. Notice the '`12`' on the Calc window's mode line:

```
    --%%-Calc: 12 Deg       (Calculator)----All------
```

Most of the symbols there are Emacs things you don't need to worry about, but the '`12`' and the '`Deg`' are mode indicators. The '`12`' means that calculations should always be carried to 12 significant figures. That is why, when we type *1 RET 7 /*, we get 0.142857142857 with exactly 12 digits, not counting leading and trailing zeros.

You can set the precision to anything you like by pressing *p*, then entering a suitable number. Try pressing *p 30 RET*, then doing *1 RET 7 /* again:

```
    1:  0.142857142857
    2:  0.142857142857142857142857142857
        .
```

Although the precision can be set arbitrarily high, Calc always has to have *some* value for the current precision. After all, the true value 1/7 is an infinitely repeating decimal; Calc has to stop somewhere.

Of course, calculations are slower the more digits you request. Press *p 12* now to set the precision back down to the default.

Calculations always use the current precision. For example, even though we have a 30-digit value for 1/7 on the stack, if we use it in a calculation in 12-digit mode it will be rounded down to 12 digits before it is used. Try it; press *RET* to duplicate the number, then *1 +*. Notice that the *RET* key didn't round the number, because it doesn't do any calculation. But the instant we pressed *+*, the number was rounded down.

```
    1:  0.142857142857
    2:  0.142857142857142857142857142857
    3:  1.14285714286
        .
```

In fact, since we added a digit on the left, we had to lose one digit on the right from even the 12-digit value of 1/7.

How did we get more than 12 digits when we computed '`2^3^4`'? The answer is that Calc makes a distinction between *integers* and *floating-point* numbers, or *floats*. An integer is a number that does not contain a decimal point. There is no such thing as an "infinitely repeating fraction integer," so Calc doesn't have to limit itself. If you asked for '`2^10000`' (don't try this!), you would have to wait a long time but you would eventually get an exact answer. If you ask for '`2.^10000`', you will quickly get an answer which is correct only to 12 places. The decimal point tells Calc that it should use floating-point arithmetic to get the answer, not exact integer arithmetic.

You can use the `F` (`calc-floor`) command to convert a floating-point value to an integer, and `c f` (`calc-float`) to convert an integer to floating-point form.

Let's try entering that last calculation:

```
1:  2.          2:  2.          1:  1.99506311689e3010
    .           1:  10000           .
                    .


    2.0 RET         10000 RET       ^
```

Notice the letter '`e`' in there. It represents "times ten to the power of," and is used by Calc automatically whenever writing the number out fully would introduce more extra zeros than you probably want to see. You can enter numbers in this notation, too.

```
1:  2.          2:  2.          1:  1.99506311678e3010
    .           1:  10000.          .
                    .


    2.0 RET         1e4 RET         ^
```

Hey, the answer is different! Look closely at the middle columns of the two examples. In the first, the stack contained the exact integer 10000, but in the second it contained a floating-point value with a decimal point. When you raise a number to an integer power, Calc uses repeated squaring and multiplication to get the answer. When you use a floating-point power, Calc uses logarithms and exponentials. As you can see, a slight error crept in during one of these methods. Which one should we trust? Let's raise the precision a bit and find out:

```
    .           1:  2.          2:  2.          1:  1.995063116880828e3010
                    .           1:  10000.          .
                                    .


  p 16 RET        2. RET          1e4             ^    p 12 RET
```

Presumably, it doesn't matter whether we do this higher-precision calculation using an integer or floating-point power, since we have added enough "guard digits" to trust the first 12 digits no matter what. And the verdict is... Integer powers were more accurate; in fact, the result was only off by one unit in the last place.

Calc does many of its internal calculations to a slightly higher precision, but it doesn't always bump the precision up enough. In each case, Calc added about two digits of precision during its calculation and then rounded back down to 12 digits afterward. In one case, it was enough; in the the other, it wasn't. If you really need x digits of precision, it never hurts to do the calculation with a few extra guard digits.

What if we want guard digits but don't want to look at them? We can set the *float format*. Calc supports four major formats for floating-point numbers, called *normal*, *fixed-point*, *scientific notation*, and *engineering notation*. You get them by pressing `d n`, `d f`, `d s`, and `d e`, respectively. In each case, you can supply a numeric prefix argument which says how many digits should be displayed. As an example, let's put a few numbers onto the stack and try some different display modes. First, use `M-0 DEL` to clear the stack, then enter the four numbers shown here:

```
4:  12345        4:  12345        4:  12345        4:  12345        4:  12345
3:  12345.       3:  12300.       3:  1.2345e4     3:  1.23e4       3:  12345.000
2:  123.45       2:  123.         2:  1.2345e2     2:  1.23e2       2:  123.450
1:  12.345       1:  12.3         1:  1.2345e1     1:  1.23e1       1:  12.345
    .                .                .                .                .

    d n              M-3 d n          d s              M-3 d s          M-3 d f
```

Notice that when we typed `M-3 d n`, the numbers were rounded down to three significant digits, but then when we typed `d s` all five significant figures reappeared. The float format does not affect how numbers are stored, it only affects how they are displayed. Only the current precision governs the actual rounding of numbers in the Calculator's memory.

Engineering notation, not shown here, is like scientific notation except the exponent (the power-of-ten part) is always adjusted to be a multiple of three (as in "kilo," "micro," etc.). As a result there will be one, two, or three digits before the decimal point.

Whenever you change a display-related mode, Calc redraws everything in the stack. This may be slow if there are many things on the stack, so Calc allows you to type shift-`H` before any mode command to prevent it from updating the stack. Anything Calc displays after the mode-changing command will appear in the new format.

```
4:  12345        4:  12345        4:  12345        4:  12345        4:  12345
3:  12345.000    3:  12345.000    3:  12345.000    3:  1.2345e4     3:  12345.
2:  123.450      2:  123.450      2:  1.2345e1     2:  1.2345e1     2:  123.45
1:  12.345       1:  1.2345e1     1:  1.2345e2     1:  1.2345e2     1:  12.345
    .                .                .                .                .

    H d s            DEL U            TAB              d SPC            d n
```

Here the `H d s` command changes to scientific notation but without updating the screen. Deleting the top stack entry and undoing it back causes it to show up in the new format; swapping the top two stack entries reformats both entries. The `d SPC` command refreshes the whole stack. The `d n` command changes back to the normal float format; since it doesn't have an `H` prefix, it also updates all the stack entries to be in `d n` format.

Notice that the integer 12345 was not affected by any of the float formats. Integers are integers, and are always displayed exactly.

Large integers have their own problems. Let's look back at the result of `2^3^4`.

```
2417851639229258349412352
```

Quick—how many digits does this have? Try typing `d g`:

```
2,417,851,639,229,258,349,412,352
```

Now how many digits does this have? It's much easier to tell! We can actually group digits into clumps of any size. Some people prefer `M-5 d g`:

```
24178,51639,22925,83494,12352
```

Let's see what happens to floating-point numbers when they are grouped. First, type *p 25 RET* to make sure we have enough precision to get ourselves into trouble. Now, type *1e13 /*:

```
24,17851,63922.9258349412352
```

The integer part is grouped but the fractional part isn't. Now try *M-- M-5 d g* (that's meta-minus-sign, meta-five):

```
24,17851,63922.92583,49412,352
```

If you find it hard to tell the decimal point from the commas, try changing the grouping character to a space with *d , SPC*:

```
24 17851 63922.92583 49412 352
```

Type *d , ,* to restore the normal grouping character, then *d g* again to turn grouping off. Also, press *p 12* to restore the default precision.

Press *U* enough times to get the original big integer back. (Notice that *U* does not undo each mode-setting command; if you want to undo a mode-setting command, you have to do it yourself.) Now, type *d r 16 RET*:

```
16#200000000000000000000
```

The number is now displayed in *hexadecimal*, or "base-16" form. Suddenly it looks pretty simple; this should be no surprise, since we got this number by computing a power of two, and 16 is a power of 2. In fact, we can use *d r 2 RET* to see it in actual binary form:

```
2#1000000000000000000000000000000000000000000000000000000 ...
```

We don't have enough space here to show all the zeros! They won't fit on a typical screen, either, so you will have to use horizontal scrolling to see them all. Press < and > to scroll the stack window left and right by half its width. Another way to view something large is to press *`* (back-quote) to edit the top of stack in a separate window. (Press *M-# M-#* when you are done.)

You can enter non-decimal numbers using the # symbol, too. Let's see what the hexadecimal number '5FE' looks like in binary. Type *16#5FE* (the letters can be typed in upper or lower case; they will always appear in upper case). It will also help to turn grouping on with *d g*:

```
2#101,1111,1110
```

Notice that *d g* groups by fours by default if the display radix is binary or hexadecimal, but by threes if it is decimal, octal, or any other radix.

Now let's see that number in decimal; type *d r 10*:

```
1,534
```

Numbers are not *stored* with any particular radix attached. They're just numbers; they can be entered in any radix, and are always displayed in whatever radix you've chosen with *d r*. The current radix applies to integers, fractions, and floats.

(•) **Exercise 1.** Your friend Joe tried to enter one-third as '3#0.1' in *d r 3* mode with a precision of 12. He got '3#0.0222222...' (with 25 2's) in the display. When he multiplied that by three, he got '3#0.222222...' instead of the expected '3#1'. Next, Joe entered '3#0.2' and, to his great relief, saw '3#0.2' on the screen. But when he typed *2 /*, he got '3#0.10000001' (some zeros omitted). What's going on here? See *XXX* [Modes Answer 1], page *XXX*. (•)

(•) **Exercise 2.** Scientific notation works in non-decimal modes in the natural way (the exponent is a power of the radix instead of a power of ten, although the exponent itself is always written in decimal). Thus '8#1.23e3 = 8#1230.0'. Suppose we have the hexadecimal number 'f.e8f' times

16 to the 15th power: We write '`16#f.e8fe15`'. What is wrong with this picture? What could we write instead that would work better? See *XXX* [Modes Answer 2], page *XXX*. (•)

The *m* prefix key has another set of modes, relating to the way Calc interprets your inputs and does computations. Whereas *d*-prefix modes generally affect the way things look, *m*-prefix modes affect the way they are actually computed.

The most popular *m*-prefix mode is the *angular mode*. Notice the '`Deg`' indicator in the mode line. This means that if you use a command that interprets a number as an angle, it will assume the angle is measured in degrees. For example,

```
1:  45           1:  0.707106781187    1:  0.500000000001     1:  0.5
    .                .                     .                      .

    45               S                     2 ^                    c 1
```

The shift-*S* command computes the sine of an angle. The sine of 45 degrees is $\sqrt{2}/2$; squaring this yields $2/4 = 0.5$. However, there has been a slight roundoff error because the representation of $\sqrt{2}/2$ wasn't exact. The *c 1* command is a handy way to clean up numbers in this case; it temporarily reduces the precision by one digit while it re-rounds the number on the top of the stack.

(•) **Exercise 3.** Your friend Joe computed the sine of 45 degrees as shown above, then, hoping to avoid an inexact result, he increased the precision to 16 digits before squaring. What happened? See *XXX* [Modes Answer 3], page *XXX*. (•)

To do this calculation in radians, we would type *m r* first. (The indicator changes to '`Rad`'.) 45 degrees corresponds to $\frac{\pi}{4}$ radians. To get π, press the *P* key. (Once again, this is a shifted capital *P*. Remember, unshifted *p* sets the precision.)

```
1:  3.14159265359    1:  0.785398163398    1:  0.707106781187
    .                    .                     .

    P                    4 /        m r    S
```

Likewise, inverse trigonometric functions generate results in either radians or degrees, depending on the current angular mode.

```
1:  0.707106781187    1:  0.785398163398    1:  45.
    .                     .                     .

    .5 Q        m r       I S        m d        U I S
```

Here we compute the Inverse Sine of $\sqrt{0.5}$, first in radians, then in degrees.

Use *c d* and *c r* to convert a number from radians to degrees and vice-versa.

```
1:  45           1:  0.785398163397     1:  45.
    .                .                      .

    45               c r                    c d
```

Another interesting mode is *fraction mode*. Normally, dividing two integers produces a floating-point result if the quotient can't be expressed as an exact integer. Fraction mode causes integer division to produce a fraction, i.e., a rational number, instead.

```
2:  12           1:  1.33333333333    1:  4:3
1:  9                .                    .
    .

 12 RET 9            /            m f        U /      m f
```

In the first case, we get an approximate floating-point result. In the second case, we get an exact fractional result (four-thirds).

You can enter a fraction at any time using *:* notation. (Calc uses *:* instead of */* as the fraction separator because */* is already used to divide the top two stack elements.) Calculations involving fractions will always produce exact fractional results; fraction mode only says what to do when dividing two integers.

(•) **Exercise 4.** If fractional arithmetic is exact, why would you ever use floating-point numbers instead? See *XXX* [Modes Answer 4], page *XXX*. (•)

Typing *m f* doesn't change any existing values in the stack. In the above example, we had to Undo the division and do it over again when we changed to fraction mode. But if you use the evaluates-to operator you can get commands like *m f* to recompute for you.

```
1:  12 / 9 => 1.33333333333    1:  12 / 9 => 1.333    1:  12 / 9 => 4:3
    .                              .                      .

   ' 12/9 => RET                  p 4 RET                m f
```

In this example, the righthand side of the '=>' operator on the stack is recomputed when we change the precision, then again when we change to fraction mode. All '=>' expressions on the stack are recomputed every time you change any mode that might affect their values.

2.2 Arithmetic Tutorial

In this section, we explore the arithmetic and scientific functions available in the Calculator.

The standard arithmetic commands are +, -, *, /, and ^. Each normally takes two numbers from the top of the stack and pushes back a result. The *n* and *&* keys perform change-sign and reciprocal operations, respectively.

```
1:  5            1:  0.2          1:  5.           1:  -5.          1:  5.
    .                .                .                .                .

    5                &                &                n                n
```

You can apply a "binary operator" like + across any number of stack entries by giving it a numeric prefix. You can also apply it pairwise to several stack elements along with the top one if you use a negative prefix.

```
3:  2            1:  9            3:  2            4:  2            3:  12
2:  3                .            2:  3            3:  3            2:  13
1:  4                             1:  4            2:  4            1:  14
    .                                 .            1:  10               .
                                                       .

2 RET 3 RET 4      M-3 +              U                10             M-- M-3 +
```

You can apply a "unary operator" like *&* to the top *n* stack entries with a numeric prefix, too.

```
3:  2             3:  0.5                3:  0.5
2:  3             2:  0.333333333333     2:  3.
1:  4             1:  0.25               1:  4.
    .                 .                      .

2 RET 3 RET 4     M-3 &                  M-2 &
```

Notice that the results here are left in floating-point form. We can convert them back to integers by pressing *F*, the "floor" function. This function rounds down to the next lower integer. There is also *R*, which rounds to the nearest integer.

```
7:  2.            7:  2             7:  2
6:  2.4           6:  2             6:  2
5:  2.5           5:  2             5:  3
4:  2.6           4:  2             4:  3
3:  -2.           3:  -2            3:  -2
2:  -2.4          2:  -3            2:  -2
1:  -2.6          1:  -3            1:  -3
    .                 .                 .

                M-7 F           U M-7 R
```

Since dividing-and-flooring (i.e., "integer quotient") is such a common operation, Calc provides a special command for that purpose, the backslash **. Another common arithmetic operator is *%*, which computes the remainder that would arise from a ** operation, i.e., the "modulo" of two numbers. For example,

```
2:  1234          1:  12            2:  1234          1:  34
1:  100               .             1:  100               .
    .                                   .

1234 RET 100          \                 U                 %
```

These commands actually work for any real numbers, not just integers.

```
2:  3.1415        1:  3             2:  3.1415        1:  0.1415
1:  1                 .             1:  1                 .
    .                                   .

3.1415 RET 1          \                 U                 %
```

(•) **Exercise 1.** The ** command would appear to be a frill, since you could always do the same thing with */ F*. Think of a situation where this is not true—*/ F* would be inadequate. Now think of a way you could get around the problem if Calc didn't provide a ** command. See *XXX* [Arithmetic Answer 1], page *XXX*. (•)

We've already seen the *Q* (square root) and *S* (sine) commands. Other commands along those lines are *C* (cosine), *T* (tangent), *E* (e^x) and *L* (natural logarithm). These can be modified by the *I* (inverse) and *H* (hyperbolic) prefix keys.

Let's compute the sine and cosine of an angle, and verify the identity $\sin^2 x + \cos^2 x = 1$. We'll arbitrarily pick -64 degrees as a good value for x. With the angular mode set to degrees (type *m d*), do:

```
2:  -64          2:  -64          2:  -0.89879   2:  -0.89879   1:  1.
1:  -64          1:  -0.89879     1:  -64        1:  0.43837        .
    .                .                .              .

  64 n RET RET       S                TAB            C              f h
```

(For brevity, we're showing only five digits of the results here. You can of course do these calculations to any precision you like.)

Remember, *f h* is the `calc-hypot`, or square-root of sum of squares, command.

Another identity is $\tan x = \dfrac{\sin x}{\cos x}$.

```
2:  -0.89879   1:  -2.0503    1:  -64.
1:  0.43837        .              .
    .

    U              /              I T
```

A physical interpretation of this calculation is that if you move 0.89879 units downward and 0.43837 units to the right, your direction of motion is −64 degrees from horizontal. Suppose we move in the opposite direction, up and to the left:

```
2:  -0.89879   2:  0.89879    1:  -2.0503    1:  -64.
1:  0.43837    1:  -0.43837       .              .
    .              .

    U U            M-2 n          /              I T
```

How can the angle be the same? The answer is that the / operation loses information about the signs of its inputs. Because the quotient is negative, we know exactly one of the inputs was negative, but we can't tell which one. There is an *f T* [`arctan2`] function which computes the inverse tangent of the quotient of a pair of numbers. Since you feed it the two original numbers, it has enough information to give you a full 360-degree answer.

```
2:  0.89879    1:  116.       3:  116.       2:  116.       1:  180.
1:  -0.43837       .          2:  -0.89879   1:  -64.           .
    .                         1:  0.43837        .
                                  .

    U U            f T        M-RET M-2 n        f T            -
```

The resulting angles differ by 180 degrees; in other words, they point in opposite directions, just as we would expect.

The *META-RET* we used in the third step is the "last-arguments" command. It is sort of like Undo, except that it restores the arguments of the last command to the stack without removing the command's result. It is useful in situations like this one, where we need to do several operations on the same inputs. We could have accomplished the same thing by using *M-2 RET* to duplicate the top two stack elements right after the *U U*, then a pair of *M-TAB* commands to cycle the 116 up around the duplicates.

A similar identity is supposed to hold for hyperbolic sines and cosines, except that it is the *difference* $\cosh^2 x - \sinh^2 x$ that always equals one. Let's try to verify this identity.

```
2:  -64          2:  -64          2:  -64          2:  9.7192e54  2:  9.7192e54
1:  -64          1:  -3.1175e27 1:  9.7192e54  1:  -64          1:  9.7192e54
    .                .                .                .                .

 64 n RET RET        H C              2 ^              TAB              H S 2 ^
```

Something's obviously wrong, because when we subtract these numbers the answer will clearly be zero! But if you think about it, if these numbers *did* differ by one, it would be in the 55th decimal place. The difference we seek has been lost entirely to roundoff error.

We could verify this hypothesis by doing the actual calculation with, say, 60 decimal places of precision. This will be slow, but not enormously so. Try it if you wish; sure enough, the answer is 0.99999, reasonably close to 1.

Of course, a more reasonable way to verify the identity is to use a more reasonable value for x!

Some Calculator commands use the Hyperbolic prefix for other purposes. The logarithm and exponential functions, for example, work to the base e normally but use base-10 instead if you use the Hyperbolic prefix.

```
1:  1000         1:  6.9077       1:  1000         1:  3
    .                .                .                .

    1000             L                U                H L
```

First, we mistakenly compute a natural logarithm. Then we undo and compute a common logarithm instead.

The *B* key computes a general base-b logarithm for any value of b.

```
2:  1000         1:  3            1:  1000.        2:  1000.        1:  6.9077
1:  10               .                .            1:  2.71828          .
    .                                                  .

 1000 RET 10         B                H E              H P              B
```

Here we first use *B* to compute the base-10 logarithm, then use the "hyperbolic" exponential as a cheap hack to recover the number 1000, then use *B* again to compute the natural logarithm. Note that *P* with the hyperbolic prefix pushes the constant e onto the stack.

You may have noticed that both times we took the base-10 logarithm of 1000, we got an exact integer result. Calc always tries to give an exact rational result for calculations involving rational numbers where possible. But when we used *H E*, the result was a floating-point number for no apparent reason. In fact, if we had computed *10 RET 3 ^* we *would* have gotten an exact integer 1000. But the *H E* command is rigged to generate a floating-point result all of the time so that *1000 H E* will not waste time computing a thousand-digit integer when all you probably wanted was '1e1000'.

(•) **Exercise 2.** Find a pair of integer inputs to the *B* command for which Calc could find an exact rational result but doesn't. See *XXX* [Arithmetic Answer 2], page *XXX*. (•)

The Calculator also has a set of functions relating to combinatorics and statistics. You may be familiar with the *factorial* function, which computes the product of all the integers up to a given number.

```
1:  100          1:  93326215443...    1:  100.         1:  9.3326e157
    .                .                     .                .

    100              !                     U c f            !
```

Recall, the *c f* command converts the integer or fraction at the top of the stack to floating-point format. If you take the factorial of a floating-point number, you get a floating-point result accurate to the current precision. But if you give *!* an exact integer, you get an exact integer result (158 digits long in this case).

If you take the factorial of a non-integer, Calc uses a generalized factorial function defined in terms of Euler's Gamma function $\Gamma(n)$ (which is itself available as the *f g* command).

```
3:  4.          3:  24.               1:  5.5          1:  52.342777847
2:  4.5         2:  52.3427777847         .                .
1:  5.          1:  120.
    .               .

                    M-3 !                 M-0 DEL 5.5      f g
```

Here we verify the identity $n! = \Gamma(n+1)$.

The binomial coefficient n-choose-m or $\binom{n}{m}$ is defined by $\frac{n!}{m!\,(n-m)!}$ for all reals n and m. The intermediate results in this formula can become quite large even if the final result is small; the *k c* command computes a binomial coefficient in a way that avoids large intermediate values.

The *k* prefix key defines several common functions out of combinatorics and number theory. Here we compute the binomial coefficient 30-choose-20, then determine its prime factorization.

```
2:  30          1:  30045015   1:  [3, 3, 5, 7, 11, 13, 23, 29]
1:  20              .              .
    .

  30 RET 20         k c            k f
```

You can verify these prime factors by using *v u* to "unpack" this vector into 8 separate stack entries, then *M-8 ** to multiply them back together. The result is the original number, 30045015.

Suppose a program you are writing needs a hash table with at least 10000 entries. It's best to use a prime number as the actual size of a hash table. Calc can compute the next prime number after 10000:

```
1:  10000       1:  10007      1:  9973
    .               .              .

    10000           k n            I k n
```

Just for kicks we've also computed the next prime *less* than 10000.

See *XXX* [Financial Functions], page *XXX*, for a description of the Calculator commands that deal with business and financial calculations (functions like `pv`, `rate`, and `sln`).

See *XXX* [Binary Functions], page *XXX*, to read about the commands for operating on binary numbers (like `and`, `xor`, and `lsh`).

2.3 Vector/Matrix Tutorial

A *vector* is a list of numbers or other Calc data objects. Calc provides a large set of commands that operate on vectors. Some are familiar operations from vector analysis. Others simply treat a vector as a list of objects.

2.3.1 Vector Analysis

If you add two vectors, the result is a vector of the sums of the elements, taken pairwise.

```
1:  [1, 2, 3]     2:  [1, 2, 3]     1:  [8, 8, 3]
    .             1:  [7, 6, 0]         .
                      .

    [1,2,3]  s 1      [7 6 0]  s 2      +
```

Note that we can separate the vector elements with either commas or spaces. This is true whether we are using incomplete vectors or algebraic entry. The *s 1* and *s 2* commands save these vectors so we can easily reuse them later.

If you multiply two vectors, the result is the sum of the products of the elements taken pairwise. This is called the *dot product* of the vectors.

```
2:  [1, 2, 3]     1:  19
1:  [7, 6, 0]         .
    .

    r 1 r 2           *
```

The dot product of two vectors is equal to the product of their lengths times the cosine of the angle between them. (Here the vector is interpreted as a line from the origin $(0, 0, 0)$ to the specified point in three-dimensional space.) The *A* (absolute value) command can be used to compute the length of a vector.

```
3:  19            3:  19            1:  0.550782    1:  56.579
2:  [1, 2, 3]     2:  3.741657          .               .
1:  [7, 6, 0]     1:  9.219544
    .                 .

    M-RET             M-2 A             * /             I C
```

First we recall the arguments to the dot product command, then we compute the absolute values of the top two stack entries to obtain the lengths of the vectors, then we divide the dot product by the product of the lengths to get the cosine of the angle. The inverse cosine finds that the angle between the vectors is about 56 degrees.

The *cross product* of two vectors is a vector whose length is the product of the lengths of the inputs times the sine of the angle between them, and whose direction is perpendicular to both input vectors. Unlike the dot product, the cross product is defined only for three-dimensional vectors. Let's double-check our computation of the angle using the cross product.

```
    2:  [1, 2, 3]  3:  [-18, 21, -8]  1:  [-0.52, 0.61, -0.23]  1:  56.579
    1:  [7, 6, 0]  2:  [1, 2, 3]          .                         .
        .          1:  [7, 6, 0]
                       .

        r 1 r 2        V C  s 3  M-RET    M-2 A * /                  A I S
```

First we recall the original vectors and compute their cross product, which we also store for later reference. Now we divide the vector by the product of the lengths of the original vectors. The length of this vector should be the sine of the angle; sure enough, it is!

Vector-related commands generally begin with the *v* prefix key. Some are uppercase letters and some are lowercase. To make it easier to type these commands, the shift-*V* prefix key acts the same as the *v* key. (See *XXX* [General Mode Commands], page *XXX*, for a way to make all prefix keys have this property.)

If we take the dot product of two perpendicular vectors we expect to get zero, since the cosine of 90 degrees is zero. Let's check that the cross product is indeed perpendicular to both inputs:

```
    2:  [1, 2, 3]      1:  0          2:  [7, 6, 0]      1:  0
    1:  [-18, 21, -8]      .          1:  [-18, 21, -8]      .
        .                                 .

        r 1 r 3            *              DEL r 2 r 3        *
```

(•) **Exercise 1.** Given a vector on the top of the stack, what keystrokes would you use to *normalize* the vector, i.e., to reduce its length to one without changing its direction? See *XXX* [Vector Answer 1], page *XXX*. (•)

(•) **Exercise 2.** Suppose a certain particle can be at any of several positions along a ruler. You have a list of those positions in the form of a vector, and another list of the probabilities for the particle to be at the corresponding positions. Find the average position of the particle. See *XXX* [Vector Answer 2], page *XXX*. (•)

2.3.2 Matrices

A *matrix* is just a vector of vectors, all the same length. This means you can enter a matrix using nested brackets. You can also use the semicolon character to enter a matrix. We'll show both methods here:

```
    1:  [ [ 1, 2, 3 ]              1:  [ [ 1, 2, 3 ]
          [ 4, 5, 6 ] ]                  [ 4, 5, 6 ] ]
        .                              .

      [[1 2 3] [4 5 6]]             ' [1 2 3; 4 5 6] RET
```

We'll be using this matrix again, so type *s 4* to save it now.

Note that semicolons work with incomplete vectors, but they work better in algebraic entry. That's why we use the apostrophe in the second example.

When two matrices are multiplied, the lefthand matrix must have the same number of columns as the righthand matrix has rows. Row i, column j of the result is effectively the dot product of row i of the left matrix by column j of the right matrix.

If we try to duplicate this matrix and multiply it by itself, the dimensions are wrong and the multiplication cannot take place:

```
1:  [ [ 1, 2, 3 ]   * [ [ 1, 2, 3 ]
      [ 4, 5, 6 ] ]     [ 4, 5, 6 ] ]
    .

    RET *
```

Though rather hard to read, this is a formula which shows the product of two matrices. The '*' function, having invalid arguments, has been left in symbolic form.

We can multiply the matrices if we *transpose* one of them first.

```
2:  [ [ 1, 2, 3 ]       1:  [ [ 14, 32 ]      1:  [ [ 17, 22, 27 ]
      [ 4, 5, 6 ] ]           [ 32, 77 ] ]          [ 22, 29, 36 ]
1:  [ [ 1, 4 ]              .                       [ 27, 36, 45 ] ]
      [ 2, 5 ]                                    .
      [ 3, 6 ] ]
    .

    U v t                   *                     U TAB *
```

Matrix multiplication is not commutative; indeed, switching the order of the operands can even change the dimensions of the result matrix, as happened here!

If you multiply a plain vector by a matrix, it is treated as a single row or column depending on which side of the matrix it is on. The result is a plain vector which should also be interpreted as a row or column as appropriate.

```
2:  [ [ 1, 2, 3 ]      1:  [14, 32]
      [ 4, 5, 6 ] ]        .
1:  [1, 2, 3]
    .

    r 4 r 1                *
```

Multiplying in the other order wouldn't work because the number of rows in the matrix is different from the number of elements in the vector.

(•) **Exercise 1.** Use '*' to sum along the rows of the above 2×3 matrix to get $[6, 15]$. Now use '*' to sum along the columns to get $[5, 7, 9]$. See *XXX* [Matrix Answer 1], page *XXX*. (•)

An *identity matrix* is a square matrix with ones along the diagonal and zeros elsewhere. It has the property that multiplication by an identity matrix, on the left or on the right, always produces the original matrix.

```
1:  [ [ 1, 2, 3 ]      2:  [ [ 1, 2, 3 ]      1:  [ [ 1, 2, 3 ]
      [ 4, 5, 6 ] ]          [ 4, 5, 6 ] ]          [ 4, 5, 6 ] ]
    .                  1:  [ [ 1, 0, 0 ]          .
                             [ 0, 1, 0 ]
                             [ 0, 0, 1 ] ]
                           .

    r 4                    v i 3 RET              *
```

If a matrix is square, it is often possible to find its *inverse*, that is, a matrix which, when multiplied by the original matrix, yields an identity matrix. The `&` (reciprocal) key also computes the inverse of a matrix.

```
1:  [ [ 1, 2, 3 ]       1:  [ [   -2.4,      1.2,   -0.2 ]
      [ 4, 5, 6 ]             [    2.8,     -1.4,    0.4 ]
      [ 7, 6, 0 ] ]           [ -0.73333, 0.53333, -0.2 ] ]
    .                       .

    r 4 r 2 |  s 5          &
```

The vertical bar `|` *concatenates* numbers, vectors, and matrices together. Here we have used it to add a new row onto our matrix to make it square.

We can multiply these two matrices in either order to get an identity.

```
1:  [ [ 1., 0., 0. ]      1:  [ [ 1., 0., 0. ]
      [ 0., 1., 0. ]            [ 0., 1., 0. ]
      [ 0., 0., 1. ] ]          [ 0., 0., 1. ] ]
    .                         .

    M-RET  *                  U TAB *
```

Matrix inverses are related to systems of linear equations in algebra. Suppose we had the following set of equations:

$$\begin{aligned} a + 2b + 3c &= 6 \\ 4a + 5b + 6c &= 2 \\ 7a + 6b \qquad &= 3 \end{aligned}$$

This can be cast into the matrix equation,

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 6 & 0 \end{pmatrix} \times \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 6 \\ 2 \\ 3 \end{pmatrix}$$

We can solve this system of equations by multiplying both sides by the inverse of the matrix. Calc can do this all in one step:

```
2:  [6, 2, 3]           1:  [-12.6, 15.2, -3.93333]
1:  [ [ 1, 2, 3 ]           .
      [ 4, 5, 6 ]
      [ 7, 6, 0 ] ]
    .

    [6,2,3] r 5             /
```

The result is the $[a, b, c]$ vector that solves the equations. (Dividing by a square matrix is equivalent to multiplying by its inverse.)

Let's verify this solution:

```
2:  [ [ 1, 2, 3 ]                1:  [6., 2., 3.]
      [ 4, 5, 6 ]                    .
      [ 7, 6, 0 ] ]
1:  [-12.6, 15.2, -3.93333]
    .

    r 5  TAB                         *
```

Note that we had to be careful about the order in which we multiplied the matrix and vector. If we multiplied in the other order, Calc would assume the vector was a row vector in order to make the dimensions come out right, and the answer would be incorrect. If you don't feel safe letting Calc take either interpretation of your vectors, use explicit $N \times 1$ or $1 \times N$ matrices instead. In this case, you would enter the original column vector as '`[[6], [2], [3]]`' or '`[6; 2; 3]`'.

(•) **Exercise 2.** Algebraic entry allows you to make vectors and matrices that include variables. Solve the following system of equations to get expressions for x and y in terms of a and b.

$$\begin{aligned} x + ay &= 6 \\ x + by &= 10 \end{aligned}$$

See *XXX* [Matrix Answer 2], page *XXX*. (•)

(•) **Exercise 3.** A system of equations is "over-determined" if it has more equations than variables. It is often the case that there are no values for the variables that will satisfy all the equations at once, but it is still useful to find a set of values which "nearly" satisfy all the equations. In terms of matrix equations, you can't solve $AX = B$ directly because the matrix A is not square for an over-determined system. Matrix inversion works only for square matrices. One common trick is to multiply both sides on the left by the transpose of A: $A^T A\,X = A^T B$, where A^T is the transpose '`trn(A)`'. Now $A^T A$ is a square matrix so a solution is possible. It turns out that the X vector you compute in this way will be a "least-squares" solution, which can be regarded as the "closest" solution to the set of equations. Use Calc to solve the following over-determined system:

$$\begin{aligned} a + 2b + 3c &= 6 \\ 4a + 5b + 6c &= 2 \\ 7a + 6b \qquad &= 3 \\ 2a + 4b + 6c &= 11 \end{aligned}$$

See *XXX* [Matrix Answer 3], page *XXX*. (•)

2.3.3 Vectors as Lists

Although Calc has a number of features for manipulating vectors and matrices as mathematical objects, you can also treat vectors as simple lists of values. For example, we saw that the `k f` command returns a vector which is a list of the prime factors of a number.

You can pack and unpack stack entries into vectors:

```
3:  10           1:  [10, 20, 30]     3:  10
2:  20               .                2:  20
1:  30                                1:  30
    .                                     .

                     M-3 v p              v u
```

You can also build vectors out of consecutive integers, or out of many copies of a given value:

```
1:  [1, 2, 3, 4]    2:  [1, 2, 3, 4]    2:  [1, 2, 3, 4]
    .               1:  17              1:  [17, 17, 17, 17]
                        .                   .

    v x 4 RET           17                  v b 4 RET
```

You can apply an operator to every element of a vector using the *map* command.

```
1:  [17, 34, 51, 68]   1:  [289, 1156, 2601, 4624]  1:  [17, 34, 51, 68]
    .                      .                            .

    V M *                  2 V M ^                      V M Q
```

In the first step, we multiply the vector of integers by the vector of 17's elementwise. In the second step, we raise each element to the power two. (The general rule is that both operands must be vectors of the same length, or else one must be a vector and the other a plain number.) In the final step, we take the square root of each element.

(•) **Exercise 1.** Compute a vector of powers of two from 2^{-4} to 2^4. See *XXX* [List Answer 1], page *XXX*. (•)

You can also *reduce* a binary operator across a vector. For example, reducing '`*`' computes the product of all the elements in the vector:

```
1:  123123      1:  [3, 7, 11, 13, 41]      1:  123123
    .               .                           .

    123123          k f                         V R *
```

In this example, we decompose 123123 into its prime factors, then multiply those factors together again to yield the original number.

We could compute a dot product "by hand" using mapping and reduction:

```
2:  [1, 2, 3]      1:  [7, 12, 0]      1:  19
1:  [7, 6, 0]          .                   .
    .

    r 1 r 2            V M *               V R +
```

Recalling two vectors from the previous section, we compute the sum of pairwise products of the elements to get the same answer for the dot product as before.

A slight variant of vector reduction is the *accumulate* operation, *V U*. This produces a vector of the intermediate results from a corresponding reduction. Here we compute a table of factorials:

```
1:  [1, 2, 3, 4, 5, 6]    1:  [1, 2, 6, 24, 120, 720]
    .                         .

    v x 6 RET                 V U *
```

Calc allows vectors to grow as large as you like, although it gets rather slow if vectors have more than about a hundred elements. Actually, most of the time is spent formatting these large vectors for display, not calculating on them. Try the following experiment (if your computer is very fast you may need to substitute a larger vector size).

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

    v x 500 RET                1 V M +
```

Now press *v .* (the letter *v*, then a period) and try the experiment again. In *v .* mode, long vectors are displayed "abbreviated" like this:

```
1:  [1, 2, 3, ..., 500]   1:  [2, 3, 4, ..., 501]
    .                         .

    v x 500 RET               1 V M +
```

(where now the '`...`' is actually part of the Calc display). You will find both operations are now much faster. But notice that even in *v .* mode, the full vectors are still shown in the Trail. Type *t .* to cause the trail to abbreviate as well, and try the experiment one more time. Operations on long vectors are now quite fast! (But of course if you use *t .* you will lose the ability to get old vectors back using the *t y* command.)

An easy way to view a full vector when *v .* mode is active is to press *`* (back-quote) to edit the vector; editing always works with the full, unabbreviated value.

As a larger example, let's try to fit a straight line to some data, using the method of least squares. (Calc has a built-in command for least-squares curve fitting, but we'll do it by hand here just to practice working with vectors.) Suppose we have the following list of values in a file we have loaded into Emacs:

```
  x          y
 ---        ---
 1.34       0.234
 1.41       0.298
 1.49       0.402
 1.56       0.412
 1.64       0.466
 1.73       0.473
 1.82       0.601
 1.91       0.519
 2.01       0.603
 2.11       0.637
 2.22       0.645
 2.33       0.705
 2.45       0.917
 2.58       1.009
 2.71       0.971
 2.85       1.062
 3.00       1.148
 3.15       1.157
 3.32       1.354
```

If you are reading this tutorial in printed form, you will find it easiest to press *M-# i* to enter the on-line Info version of the manual and find this table there. (Press *g*, then type *List Tutorial*, to jump straight to this section.)

Position the cursor at the upper-left corner of this table, just to the left of the 1.34. Press *C-@* to set the mark. (On your system this may be *C-2*, *C-SPC*, or *NUL*.) Now position the cursor to the lower-right, just after the 1.354. You have now defined this region as an Emacs "rectangle." Still in the Info buffer, type *M-# r*. This command (`calc-grab-rectangle`) will pop you back into the Calculator, with the contents of the rectangle you specified in the form of a matrix.

```
1:  [ [ 1.34, 0.234 ]
      [ 1.41, 0.298 ]
      ...
```

(You may wish to use `v .` mode to abbreviate the display of this large matrix.)

We want to treat this as a pair of lists. The first step is to transpose this matrix into a pair of rows. Remember, a matrix is just a vector of vectors. So we can unpack the matrix into a pair of row vectors on the stack.

```
1:  [ [ 1.34,  1.41,  1.49,  ... ]     2:  [1.34, 1.41, 1.49, ... ]
      [ 0.234, 0.298, 0.402, ... ] ]   1:  [0.234, 0.298, 0.402, ... ]
    .                                      .


    v t                                    v u
```

Let's store these in quick variables 1 and 2, respectively.

```
1:  [1.34, 1.41, 1.49, ... ]        .
    .


    t 2                             t 1
```

(Recall that `t 2` is a variant of `s 2` that removes the stored value from the stack.)

In a least squares fit, the slope m is given by the formula

$$m = \frac{N \sum xy - \sum x \sum y}{N \sum x^2 - \left(\sum x\right)^2}$$

where $\sum x$ represents the sum of all the values of x. While there is an actual `sum` function in Calc, it's easier to sum a vector using a simple reduction. First, let's compute the four different sums that this formula uses.

```
1:  41.63                 1:  98.0003
    .                         .


 r 1 V R +   t 3           r 1 2 V M ^ V R +   t 4


1:  13.613                1:  33.36554
    .                         .


 r 2 V R +   t 5           r 1 r 2 V M * V R +   t 6
```

These are $\sum x$, $\sum x^2$, $\sum y$, and $\sum xy$, respectively. (We could have used `*` to compute $\sum x^2$ and $\sum xy$.)

Finally, we also need N, the number of data points. This is just the length of either of our lists.

```
1:  19
    .


 r 1 v l   t 7
```

(That's `v` followed by a lower-case `l`.)

Now we grind through the formula:

```
1:  633.94526  2:  633.94526  1:  67.23607
    .          1:  566.70919      .
                   .


 r 7 r 6 *      r 3 r 5 *          -
```

```
2:  67.23607    3:  67.23607    2:  67.23607    1:  0.52141679
1:  1862.0057   2:  1862.0057   1:  128.9488        .
    .           1:  1733.0569       .
                    .

 r 7 r 4 *       r 3 2 ^            -                /   t 8
```

That gives us the slope m. The y-intercept b can now be found with the simple formula,

$$b = \frac{\sum y - m \sum x}{N}$$

```
1:  13.613      2:  13.613      1:  -8.09358    1:  -0.425978
    .           1:  21.70658        .               .
                    .

    r 5             r 8 r 3 *       -               r 7 /   t 9
```

Let's "plot" this straight line approximation, $y \approx mx + b$, and compare it with the original data.

```
1:  [0.699, 0.735, ... ]    1:  [0.273, 0.309, ... ]
    .                           .

    r 1 r 8 *                   r 9 +    s 0
```

Notice that multiplying a vector by a constant, and adding a constant to a vector, can be done without mapping commands since these are common operations from vector algebra. As far as Calc is concerned, we've just been doing geometry in 19-dimensional space!

We can subtract this vector from our original y vector to get a feel for the error of our fit. Let's find the maximum error:

```
1:  [0.0387, 0.0112, ... ]   1:  [0.0387, 0.0112, ... ]   1:  0.0897
    .                            .                            .

    r 2 -                        V M A                        V R X
```

First we compute a vector of differences, then we take the absolute values of these differences, then we reduce the `max` function across the vector. (The `max` function is on the two-key sequence *`f x`*; because it is so common to use `max` in a vector operation, the letters *`X`* and *`N`* are also accepted for `max` and `min` in this context. In general, you answer the *`V M`* or *`V R`* prompt with the actual key sequence that invokes the function you want. You could have typed *`V R f x`* or even *`V R x max RET`* if you had preferred.)

If your system has the GNUPLOT program, you can see graphs of your data and your straight line to see how well they match. (If you have GNUPLOT 3.0, the following instructions will work regardless of the kind of display you have. Some GNUPLOT 2.0, non-X-windows systems may require additional steps to view the graphs.)

Let's start by plotting the original data. Recall the "x" and "y" vectors onto the stack and press *`g f`*. This "fast" graphing command does everything you need to do for simple, straightforward plotting of data.

```
2:  [1.34, 1.41, 1.49, ... ]
1:  [0.234, 0.298, 0.402, ... ]
    .

    r 1 r 2   g f
```

If all goes well, you will shortly get a new window containing a graph of the data. (If not, contact your GNUPLOT or Calc installer to find out what went wrong.) In the X window system, this will be a separate graphics window. For other kinds of displays, the default is to display the graph in Emacs itself using rough character graphics. Press *q* when you are done viewing the character graphics.

Next, let's add the line we got from our least-squares fit:

```
2:  [1.34, 1.41, 1.49, ... ]
1:  [0.273, 0.309, 0.351, ... ]
    .

    DEL r 0   g a  g p
```

It's not very useful to get symbols to mark the data points on this second curve; you can type *g S g p* to remove them. Type *g q* when you are done to remove the X graphics window and terminate GNUPLOT.

(•) **Exercise 2.** An earlier exercise showed how to do least squares fitting to a general system of equations. Our 19 data points are really 19 equations of the form $y_i = mx_i + b$ for different pairs of (x_i, y_i). Use the matrix-transpose method to solve for m and b, duplicating the above result. See *XXX* [List Answer 2], page *XXX*. (•)

(•) **Exercise 3.** If the input data do not form a rectangle, you can use *M-# g* (`calc-grab-region`) to grab the data the way Emacs normally works with regions—it reads left-to-right, top-to-bottom, treating line breaks the same as spaces. Use this command to find the geometric mean of the following numbers. (The geometric mean is the nth root of the product of n numbers.)

```
2.3  6  22  15.1  7
  15  14  7.5
  2.5
```

The *M-# g* command accepts numbers separated by spaces or commas, with or without surrounding vector brackets. See *XXX* [List Answer 3], page *XXX*. (•)

As another example, a theorem about binomial coefficients tells us that the alternating sum of binomial coefficients $\binom{n}{0} - \binom{n}{1} + \binom{n}{2} - \cdots \pm \binom{n}{n}$ always comes out to zero. Let's verify this for $n = 6$.

```
1:  [1, 2, 3, 4, 5, 6, 7]     1:  [0, 1, 2, 3, 4, 5, 6]
    .                             .

    v x 7 RET                     1 -


1:  [1, -6, 15, -20, 15, -6, 1]          1:  0
    .                                        .

    V M ' (-1)^$ choose(6,$) RET             V R +
```

The *V M '* command prompts you to enter any algebraic expression to define the function to map over the vector. The symbol '`$`' inside this expression represents the argument to the function.

The Calculator applies this formula to each element of the vector, substituting each element's value for the '$' sign(s) in turn.

To define a two-argument function, use '$$' for the first argument and '$' for the second: *V M ' $$-$ RET* is equivalent to *V M -*. This is analogous to regular algebraic entry, where '$$' would refer to the next-to-top stack entry and '$' would refer to the top stack entry, and *' $$-$ RET* would act exactly like -.

Notice that the *V M '* command has recorded two things in the trail: The result, as usual, and also a funny-looking thing marked 'oper' that represents the operator function you typed in. The function is enclosed in '< >' brackets, and the argument is denoted by a '#' sign. If there were several arguments, they would be shown as '#1', '#2', and so on. (For example, *V M ' $$-$* will put the function '<#1 - #2>' on the trail.) This object is a "nameless function"; you can use nameless '< >' notation to answer the *V M '* prompt if you like. Nameless function notation has the interesting, occasionally useful property that a nameless function is not actually evaluated until it is used. For example, *V M ' $+random(2.0)* evaluates 'random(2.0)' once and adds that random number to all elements of the vector, but *V M ' <#+random(2.0)>* evaluates the 'random(2.0)' separately for each vector element.

Another group of operators that are often useful with *V M* are the relational operators: a =, for example, compares two numbers and gives the result 1 if they are equal, or 0 if not. Similarly, a < checks for one number being less than another.

Other useful vector operations include *v v*, to reverse a vector end-for-end; *V S*, to sort the elements of a vector into increasing order; and *v r* and *v c*, to extract one row or column of a matrix, or (in both cases) to extract one element of a plain vector. With a negative argument, *v r* and *v c* instead delete one row, column, or vector element.

(•) **Exercise 4.** The kth *divisor function* $\sigma_k(n)$ is the sum of the kth powers of all the divisors of an integer n. Figure out a method for computing the divisor function for reasonably small values of n. As a test, the 0th and 1st divisor functions of 30 are 8 and 72, respectively. See *XXX* [List Answer 4], page *XXX*. (•)

(•) **Exercise 5.** The *k f* command produces a list of prime factors for a number. Sometimes it is important to know that a number is *square-free*, i.e., that no prime occurs more than once in its list of prime factors. Find a sequence of keystrokes to tell if a number is square-free; your method should leave 1 on the stack if it is, or 0 if it isn't. See *XXX* [List Answer 5], page *XXX*. (•)

(•) **Exercise 6.** Build a list of lists that looks like the following diagram. (You may wish to use the *v /* command to enable multi-line display of vectors.)

```
1:  [ [1],
      [1, 2],
      [1, 2, 3],
      [1, 2, 3, 4],
      [1, 2, 3, 4, 5],
      [1, 2, 3, 4, 5, 6] ]
```

See *XXX* [List Answer 6], page *XXX*. (•)

(•) **Exercise 7.** Build the following list of lists.

```
    1:  [ [0],
          [1, 2],
          [3, 4, 5],
          [6, 7, 8, 9],
          [10, 11, 12, 13, 14],
          [15, 16, 17, 18, 19, 20] ]
```

See *XXX* [List Answer 7], page *XXX*. (•)

(•) **Exercise 8.** Compute a list of values of Bessel's $J_1(x)$ function '`besJ(1,x)`' for x from 0 to 5 in steps of 0.25. Find the value of x (from among the above set of values) for which '`besJ(1,x)`' is a maximum. Use an "automatic" method, i.e., just reading along the list by hand to find the largest value is not allowed! (There is an `a X` command which does this kind of thing automatically; see *XXX* [Numerical Solutions], page *XXX*.) See *XXX* [List Answer 8], page *XXX*. (•)

(•) **Exercise 9.** You are given an integer in the range $0 \le N < 10^m$ for $m = 12$ (i.e., an integer of less than twelve digits). Convert this integer into a vector of m digits, each in the range from 0 to 9. In vector-of-digits notation, add one to this integer to produce a vector of $m + 1$ digits (since there could be a carry out of the most significant digit). Convert this vector back into a regular integer. A good integer to try is 25129925999. See *XXX* [List Answer 9], page *XXX*. (•)

(•) **Exercise 10.** Your friend Joe tried to use `V R a =` to test if all numbers in a list were equal. What happened? How would you do this test? See *XXX* [List Answer 10], page *XXX*. (•)

(•) **Exercise 11.** The area of a circle of radius one is π. The area of the 2×2 square that encloses that circle is 4. So if we throw N darts at random points in the square, about $\pi/4$ of them will land inside the circle. This gives us an entertaining way to estimate the value of π. The `k r` command picks a random number between zero and the value on the stack. We could get a random floating-point number between -1 and 1 by typing `2.0 k r 1 -`. Build a vector of 100 random (x, y) points in this square, then use vector mapping and reduction to count how many points lie inside the unit circle. Hint: Use the `v b` command. See *XXX* [List Answer 11], page *XXX*. (•)

(•) **Exercise 12.** The *matchstick problem* provides another way to calculate π. Say you have an infinite field of vertical lines with a spacing of one inch. Toss a one-inch matchstick onto the field. The probability that the matchstick will land crossing a line turns out to be $2/\pi$. Toss 100 matchsticks to estimate π. (If you want still more fun, the probability that the GCD (`k g`) of two large integers is one turns out to be $6/\pi^2$. That provides yet another way to estimate π.) See *XXX* [List Answer 12], page *XXX*. (•)

(•) **Exercise 13.** An algebraic entry of a string in double-quote marks, '`"hello"`', creates a vector of the numerical (ASCII) codes of the characters (here, $[104, 101, 108, 108, 111]$). Sometimes it is convenient to compute a *hash code* of a string, which is just an integer that represents the value of that string. Two equal strings have the same hash code; two different strings *probably* have different hash codes. (For example, Calc has over 400 function names, but Emacs can quickly find the definition for any given name because it has sorted the functions into "buckets" by their hash codes. Sometimes a few names will hash into the same bucket, but it is easier to search among a few names than among all the names.) One popular hash function is computed as follows: First set $h = 0$. Then, for each character from the string in turn, set $h = 3h + c_i$ where c_i is the character's ASCII code. If we have 511 buckets, we then take the hash code modulo 511 to get the bucket number. Develop a simple command or commands for converting string vectors into hash codes. The hash code for '`"Testing, 1, 2, 3"`' is 1960915098, which modulo 511 is 121. See *XXX* [List Answer 13], page *XXX*. (•)

(•) **Exercise 14.** The *H V R* and *H V U* commands do nested function evaluations. *H V U* takes a starting value and a number of steps *n* from the stack; it then applies the function you give to the starting value 0, 1, 2, up to *n* times and returns a vector of the results. Use this command to create a "random walk" of 50 steps. Start with the two-dimensional point $(0, 0)$; then take one step a random distance between -1 and 1 in both x and y; then take another step, and so on. Use the *g f* command to display this random walk. Now modify your random walk to walk a unit distance, but in a random direction, at each step. (Hint: The `sincos` function returns a vector of the cosine and sine of an angle.) See *XXX* [List Answer 14], page *XXX*. (•)

2.4 Types Tutorial

Calc understands a variety of data types as well as simple numbers. In this section, we'll experiment with each of these types in turn.

The numbers we've been using so far have mainly been either *integers* or *floats*. We saw that floats are usually a good approximation to the mathematical concept of real numbers, but they are only approximations and are susceptible to roundoff error. Calc also supports *fractions*, which can exactly represent any rational number.

```
1:  3628800    2:  3628800    1:  518400:7   1:  518414:7   1:  7:518414
    .          1:  49             .              .              .
                   .

    10 !           49 RET         :              2 +            &
```

The *:* command divides two integers to get a fraction; */* would normally divide integers to get a floating-point result. Notice we had to type *RET* between the *49* and the *:* since the *:* would otherwise be interpreted as part of a fraction beginning with 49.

You can convert between floating-point and fractional format using *c f* and *c F*:

```
1:  1.35027217629e-5    1:  7:518414
    .                       .

    c f                     c F
```

The *c F* command replaces a floating-point number with the "simplest" fraction whose floating-point representation is the same, to within the current precision.

```
1:  3.14159265359   1:  1146408:364913   1:  3.1416   1:  355:113
    .                   .                    .            .

    P                   c F      DEL         p 5 RET P    c F
```

(•) **Exercise 1.** A calculation has produced the result 1.26508260337. You suspect it is the square root of the product of π and some rational number. Is it? (Be sure to allow for roundoff error!) See *XXX* [Types Answer 1], page *XXX*. (•)

Complex numbers can be stored in both rectangular and polar form.

```
1:  -9      1:  (0, 3)    1:  (3; 90.)    1:  (6; 90.)    1:  (2.4495; 45.)
    .           .             .               .               .

    9 n         Q             c p             2 *             Q
```

The square root of -9 is by default rendered in rectangular form $(0+3i)$, but we can convert it to polar form (3 with a phase angle of 90 degrees). All the usual arithmetic and scientific operations are defined on both types of complex numbers.

Another generalized kind of number is *infinity*. Infinity isn't really a number, but it can sometimes be treated like one. Calc uses the symbol `inf` to represent positive infinity, i.e., a value greater than any real number. Naturally, you can also write '`-inf`' for minus infinity, a value less than any real number. The word `inf` can only be input using algebraic entry.

```
2:  inf          2:  -inf         2:  -inf         2:  -inf         1:  nan
1:  -17          1:  -inf         1:  -inf         1:  inf              .
    .                .                .                .

 ' inf RET 17 n      *  RET           72 +             A                +
```

Since infinity is infinitely large, multiplying it by any finite number (like -17) has no effect, except that since -17 is negative, it changes a plus infinity to a minus infinity. ("A huge positive number, multiplied by -17, yields a huge negative number.") Adding any finite number to infinity also leaves it unchanged. Taking an absolute value gives us plus infinity again. Finally, we add this plus infinity to the minus infinity we had earlier. If you work it out, you might expect the answer to be -72 for this. But the 72 has been completely lost next to the infinities; by the time we compute '`inf - inf`' the finite difference between them, if any, is indetectable. So we say the result is *indeterminate*, which Calc writes with the symbol `nan` (for Not A Number).

Dividing by zero is normally treated as an error, but you can get Calc to write an answer in terms of infinity by pressing *m i* to turn on "infinite mode."

```
3:  nan          2:  nan          2:  nan          2:  nan          1:  nan
2:  1            1:  1 / 0        1:  uinf         1:  uinf             .
1:  0                .                .                .
    .

  1 RET 0            /       m i      U /              17 n *           +
```

Dividing by zero normally is left unevaluated, but after *m i* it instead gives an infinite result. The answer is actually `uinf`, "undirected infinity." If you look at a graph of $1/x$ around $x=0$, you'll see that it goes toward plus infinity as you approach zero from above, but toward minus infinity as you approach from below. Since we said only 1/0, Calc knows that the answer is infinite but not in which direction. That's what `uinf` means. Notice that multiplying `uinf` by a negative number still leaves plain `uinf`; there's no point in saying '`-uinf`' because the sign of `uinf` is unknown anyway. Finally, we add `uinf` to our `nan`, yielding `nan` again. It's easy to see that, because `nan` means "totally unknown" while `uinf` means "unknown sign but known to be infinite," the more mysterious `nan` wins out when it is combined with `uinf`, or, for that matter, with anything else.

(•) **Exercise 2.** Predict what Calc will answer for each of these formulas: '`inf / inf`', '`exp(inf)`', '`exp(-inf)`', '`sqrt(-inf)`', '`sqrt(uinf)`', '`abs(uinf)`', '`ln(0)`'. See *XXX* [Types Answer 2], page *XXX*. (•)

(•) **Exercise 3.** We saw that '`inf - inf = nan`', which stands for an unknown value. Can `nan` stand for a complex number? Can it stand for infinity? See *XXX* [Types Answer 3], page *XXX*. (•)

HMS forms represent a value in terms of hours, minutes, and seconds.

```
1:  2@ 30' 0"      1:  3@ 30' 0"      2:  3@ 30' 0"      1:  2.
    .                  .              1:  1@ 45' 0."         .
                                          .

   2@ 30' RET          1 +                RET 2 /            /
```

HMS forms can also be used to hold angles in degrees, minutes, and seconds.

```
1:  0.5         1:  26.56505   1:  26@ 33' 54.18"    1:  0.44721
    .               .              .                     .

    0.5             I T            c h                   S
```

First we convert the inverse tangent of 0.5 to degrees-minutes-seconds form, then we take the sine of that angle. Note that the trigonometric functions will accept HMS forms directly as input.

(•) **Exercise 4.** The Beatles' *Abbey Road* is 47 minutes and 26 seconds long, and contains 17 songs. What is the average length of a song on *Abbey Road*? If the Extended Disco Version of *Abbey Road* added 20 seconds to the length of each song, how long would the album be? See *XXX* [Types Answer 4], page *XXX*. (•)

A *date form* represents a date, or a date and time. Dates must be entered using algebraic entry. Date forms are surrounded by '`< >`' symbols; most standard formats for dates are recognized.

```
2:  <Sun Jan 13, 1991>                    1:  2.25
1:  <6:00pm Thu Jan 10, 1991>                 .
    .

' <13 Jan 1991>, <1/10/91, 6pm> RET           -
```

In this example, we enter two dates, then subtract to find the number of days between them. It is also possible to add an HMS form or a number (of days) to a date form to get another date form.

```
1:  <4:45:59pm Mon Jan 14, 1991>     1:  <2:50:59am Thu Jan 17, 1991>
    .                                    .

    t N                                  2 + 10@ 5' +
```

The *t N* ("now") command pushes the current date and time on the stack; then we add two days, ten hours and five minutes to the date and time. Other date-and-time related commands include *t J*, which does Julian day conversions, *t W*, which finds the beginning of the week in which a date form lies, and *t I*, which increments a date by one or several months. See *XXX* [Date Arithmetic], page *XXX*, for more.

(•) **Exercise 5.** How many days until the next Friday the 13th? See *XXX* [Types Answer 5], page *XXX*. (•)

(•) **Exercise 6.** How many leap years will there be between now and the year 10001 A.D.? See *XXX* [Types Answer 6], page *XXX*. (•)

An *error form* represents a mean value with an attached standard deviation, or error estimate. Suppose our measurements indicate that a certain telephone pole is about 30 meters away, with an estimated error of 1 meter, and 8 meters tall, with an estimated error of 0.2 meters. What is the slope of a line from here to the top of the pole, and what is the equivalent angle in degrees?

```
1:  8 +/- 0.2    2:  8 +/- 0.2   1:  0.266 +/- 0.011   1:  14.93 +/- 0.594
    .            1:  30 +/- 1        .                     .
                     .

    8 p .2 RET       30 p 1          /                     I T
```

This means that the angle is about 15 degrees, and, assuming our original error estimates were valid standard deviations, there is about a 60% chance that the result is correct within 0.59 degrees.

(•) **Exercise 7.** The volume of a torus (a donut shape) is $2\pi^2 R r^2$ where R is the radius of the circle that defines the center of the tube and r is the radius of the tube itself. Suppose R is 20 cm and r is 4 cm, each known to within 5 percent. What is the volume and the relative uncertainty of the volume? See *XXX* [Types Answer 7], page *XXX*. (•)

An *interval form* represents a range of values. While an error form is best for making statistical estimates, intervals give you exact bounds on an answer. Suppose we additionally know that our telephone pole is definitely between 28 and 31 meters away, and that it is between 7.7 and 8.1 meters tall.

```
1:  [7.7 .. 8.1]  2:  [7.7 .. 8.1]  1:  [0.24 .. 0.28]  1:  [13.9 .. 16.1]
    .             1:  [28 .. 31]        .                   .
                      .

  [ 7.7 .. 8.1 ]    [ 28 .. 31 ]        /                     I T
```

If our bounds were correct, then the angle to the top of the pole is sure to lie in the range shown.

The square brackets around these intervals indicate that the endpoints themselves are allowable values. In other words, the distance to the telephone pole is between 28 and 31, *inclusive*. You can also make an interval that is exclusive of its endpoints by writing parentheses instead of square brackets. You can even make an interval which is inclusive ("closed") on one end and exclusive ("open") on the other.

```
1:  [1 .. 10)    1:  (0.1 .. 1]   2:  (0.1 .. 1]   1:  (0.2 .. 3)
    .                .            1:  [2 .. 3)         .
                                      .

  [ 1 .. 10 )        &              [ 2 .. 3 )         *
```

The Calculator automatically keeps track of which end values should be open and which should be closed. You can also make infinite or semi-infinite intervals by using '`-inf`' or '`inf`' for one or both endpoints.

(•) **Exercise 8.** What answer would you expect from '`1 / (0 .. 10)`'? What about '`1 / (-10 .. 0)`'? What about '`1 / [0 .. 10]`' (where the interval actually includes zero)? What about '`1 / (-10 .. 10)`'? See *XXX* [Types Answer 8], page *XXX*. (•)

(•) **Exercise 9.** Two easy ways of squaring a number are *RET* `*` and `2 ^`. Normally these produce the same answer. Would you expect this still to hold true for interval forms? If not, which of these will result in a larger interval? See *XXX* [Types Answer 9], page *XXX*. (•)

A *modulo form* is used for performing arithmetic modulo M. For example, arithmetic involving time is generally done modulo 12 or 24 hours.

```
1:  17 mod 24    1:  3 mod 24     1:  21 mod 24    1:  9 mod 24
    .                .                .                .

    17 M 24 RET      10 +             n                5 /
```

In this last step, Calc has found a new number which, when multiplied by 5 modulo 24, produces the original number, 21. If M is prime it is always possible to find such a number. For non-prime M like 24, it is only sometimes possible.

```
1:  10 mod 24     1:  16 mod 24     1:  1000000...    1:  16
    .                 .                 .                 .

    10 M 24 RET       100 ^             10 RET 100 ^      24 %
```

These two calculations get the same answer, but the first one is much more efficient because it avoids the huge intermediate value that arises in the second one.

(•) **Exercise 10.** A theorem of Pierre de Fermat says that $x^{n-1} \bmod n = 1$ if n is a prime number and x is an integer less than n. If n is *not* a prime number, this will *not* be true for most values of x. Thus we can test informally if a number is prime by trying this formula for several values of x. Use this test to tell whether the following numbers are prime: 811749613, 15485863. See *XXX* [Types Answer 10], page *XXX*. (•)

It is possible to use HMS forms as parts of error forms, intervals, modulo forms, or as the phase part of a polar complex number. For example, the `calc-time` command pushes the current time of day on the stack as an HMS/modulo form.

```
1:  17@ 34' 45" mod 24@ 0' 0"      1:  6@ 22' 15" mod 24@ 0' 0"
    .                                  .

    x time RET                         n
```

This calculation tells me it is six hours and 22 minutes until midnight.

(•) **Exercise 11.** A rule of thumb is that one year is about $\pi \times 10^7$ seconds. What time will it be that many seconds from right now? See *XXX* [Types Answer 11], page *XXX*. (•)

(•) **Exercise 12.** You are preparing to order packaging for the CD release of the Extended Disco Version of *Abbey Road*. You are told that the songs will actually be anywhere from 20 to 60 seconds longer than the originals. One CD can hold about 75 minutes of music. Should you order single or double packages? See *XXX* [Types Answer 12], page *XXX*. (•)

Another kind of data the Calculator can manipulate is numbers with *units*. This isn't strictly a new data type; it's simply an application of algebraic expressions, where we use variables with suggestive names like '`cm`' and '`in`' to represent units like centimeters and inches.

```
1:  2 in         1:  5.08 cm      1:  0.027778 fath   1:  0.0508 m
    .                .                .                   .

    ' 2in RET        u c cm RET       u c fath RET        u b
```

We enter the quantity "2 inches" (actually an algebraic expression which means two times the variable '`in`'), then we convert it first to centimeters, then to fathoms, then finally to "base" units, which in this case means meters.

```
1:  9 acre     1:  3 sqrt(acre)   1:  190.84 m   1:  190.84 m + 30 cm
    .              .                  .              .

  ' 9 acre RET     Q                  u s            ' $+30 cm RET
```

```
1:  191.14 m      1:  36536.3046 m^2     1:  365363046 cm^2
    .                 .                      .

    u s               2 ^                    u c cgs
```

Since units expressions are really just formulas, taking the square root of '`acre`' is undefined. After all, `acre` might be an algebraic variable that you will someday assign a value. We use the "units-simplify" command to simplify the expression with variables being interpreted as unit names.

In the final step, we have converted not to a particular unit, but to a units system. The "cgs" system uses centimeters instead of meters as its standard unit of length.

There is a wide variety of units defined in the Calculator.

```
1:  55 mph     1:  88.5139 kph   1:   88.5139 km / hr   1:  8.201407e-8 c
    .              .                  .                     .

  ' 55 mph RET     u c kph RET        u c km/hr RET         u c c RET
```

We express a speed first in miles per hour, then in kilometers per hour, then again using a slightly more explicit notation, then finally in terms of fractions of the speed of light.

Temperature conversions are a bit more tricky. There are two ways to interpret "20 degrees Fahrenheit"—it could mean an actual temperature, or it could mean a change in temperature. For normal units there is no difference, but temperature units have an offset as well as a scale factor and so there must be two explicit commands for them.

```
1:  20 degF        1:  11.1111 degC     1:  -20:3 degC     1:  -6.666 degC
    .                  .                    .                  .

  ' 20 degF RET        u c degC RET         U u t degC RET     c f
```

First we convert a change of 20 degrees Fahrenheit into an equivalent change in degrees Celsius (or Centigrade). Then, we convert the absolute temperature 20 degrees Fahrenheit into Celsius. Since this comes out as an exact fraction, we then convert to floating-point for easier comparison with the other result.

For simple unit conversions, you can put a plain number on the stack. Then *u c* and *u t* will prompt for both old and new units. When you use this method, you're responsible for remembering which numbers are in which units:

```
1:  55         1:  88.5139              1:  8.201407e-8
    .              .                        .

    55             u c mph RET kph RET      u c km/hr RET c RET
```

To see a complete list of built-in units, type *u v*. Press *M-# c* again to re-enter the Calculator when you're done looking at the units table.

(•) **Exercise 13.** How many seconds are there really in a year? See *XXX* [Types Answer 13], page *XXX*. (•)

(•) **Exercise 14.** Supercomputer designs are limited by the speed of light (and of electricity, which is nearly as fast). Suppose a computer has a 4.1 ns (nanosecond) clock cycle, and its cabinet is one meter across. Is speed of light going to be a significant factor in its design? See *XXX* [Types Answer 14], page *XXX*. (•)

(•) **Exercise 15.** Sam the Slug normally travels about five yards in an hour. He has obtained a supply of Power Pills; each Power Pill he eats doubles his speed. How many Power Pills can he swallow and still travel legally on most US highways? See *XXX* [Types Answer 15], page *XXX*. (•)

2.5 Algebra and Calculus Tutorial

This section shows how to use Calc's algebra facilities to solve equations, do simple calculus problems, and manipulate algebraic formulas.

2.5.1 Basic Algebra

If you enter a formula in algebraic mode that refers to variables, the formula itself is pushed onto the stack. You can manipulate formulas as regular data objects.

```
1:  2 x^2 - 6          1:  6 - 2 x^2          1:  (6 - 2 x^2) (3 x^2 + y)
    .                      .                      .

    ' 2x^2-6 RET           n                      ' 3x^2+y RET *
```

(•) **Exercise 1.** Do `' x RET Q 2 ^` and `' x RET 2 ^ Q` both wind up with the same result ('x')? Why or why not? See *XXX* [Algebra Answer 1], page *XXX*. (•)

There are also commands for doing common algebraic operations on formulas. Continuing with the formula from the last example,

```
1:  18 x^2 + 6 y - 6 x^4 - 2 x^2 y    1:  (18 - 2 y) x^2 - 6 x^4 + 6 y
    .                                     .

    a x                                   a c x RET
```

First we "expand" using the distributive law, then we "collect" terms involving like powers of x.

Let's find the value of this expression when x is 2 and y is one-half.

```
1:  17 x^2 - 6 x^4 + 3      1:  -25
    .                           .

    1:2 s l y RET               2 s l x RET
```

The `s l` command means "let"; it takes a number from the top of the stack and temporarily assigns it as the value of the variable you specify. It then evaluates (as if by the `=` key) the next expression on the stack. After this command, the variable goes back to its original value, if any.

(An earlier exercise in this tutorial involved storing a value in the variable `x`; if this value is still there, you will have to unstore it with `s u x RET` before the above example will work properly.)

Let's find the maximum value of our original expression when y is one-half and x ranges over all possible values. We can do this by taking the derivative with respect to x and examining values of x for which the derivative is zero. If the second derivative of the function at that value of x is negative, the function has a local maximum there.

```
1:  17 x^2 - 6 x^4 + 3      1:  34 x - 24 x^3
    .                           .

    U DEL  s 1                  a d x RET   s 2
```

Well, the derivative is clearly zero when x is zero. To find the other root(s), let's divide through by x and then solve:

```
1:  (34 x - 24 x^3) / x    1:  34 x / x - 24 x^3 / x    1:  34 - 24 x^2
    .                          .                            .

    ' x RET /                  a x                          a s


1:  34 - 24 x^2 = 0        1:  x = 1.19023
    .                          .

    0 a=  s 3                  a S x RET
```

Notice the use of `a s` to "simplify" the formula. When the default algebraic simplifications don't do enough, you can use `a s` to tell Calc to spend more time on the job.

Now we compute the second derivative and plug in our values of x:

```
1:  1.19023        2:  1.19023        2:  1.19023
    .              1:  34 x - 24 x^3  1:  34 - 72 x^2
                       .                  .

    a .                r 2                a d x RET s 4
```

(The `a .` command extracts just the righthand side of an equation. Another method would have been to use `v u` to unpack the equation '`x = 1.19`' to '`x`' and '`1.19`', then use `M-- M-2 DEL` to delete the '`x`'.)

```
2:  34 - 72 x^2   1:  -68.         2:  34 - 72 x^2     1:  34
1:  1.19023           .            1:  0                   .
    .                                  .

    TAB               s l x RET        U DEL 0             s l x RET
```

The first of these second derivatives is negative, so we know the function has a maximum value at $x = 1.19023$. (The function also has a local *minimum* at $x = 0$.)

When we solved for x, we got only one value even though $34 - 24x^2 = 0$ is a quadratic equation that ought to have two solutions. The reason is that `a S` normally returns a single "principal" solution. If it needs to come up with an arbitrary sign (as occurs in the quadratic formula) it picks $+$. If it needs an arbitrary integer, it picks zero. We can get a full solution by pressing `H` (the Hyperbolic flag) before `a S`.

```
1:  34 - 24 x^2 = 0    1:  x = 1.19023 s1       1:  x = -1.19023
    .                      .                        .

    r 3                    H a S x RET  s 5         1 n  s l s1 RET
```

Calc has invented the variable '`s1`' to represent an unknown sign; it is supposed to be either $+1$ or -1. Here we have used the "let" command to evaluate the expression when the sign is negative. If we plugged this into our second derivative we would get the same, negative, answer, so $x = -1.19023$ is also a maximum.

To find the actual maximum value, we must plug our two values of x into the original formula.

```
2:  17 x^2 - 6 x^4 + 3     1:  24.08333 s1^2 - 12.04166 s1^4 + 3
1:  x = 1.19023 s1             .
    .

    r 1 r 5                    s 1 RET
```

(Here we see another way to use *s 1*; if its input is an equation with a variable on the lefthand side, then *s 1* treats the equation like an assignment to that variable if you don't give a variable name.)

It's clear that this will have the same value for either sign of `s1`, but let's work it out anyway, just for the exercise:

```
2:  [-1, 1]              1:  [15.04166, 15.04166]
1:  24.08333 s1^2 ...        .
    .

   [ 1 n , 1 ] TAB          V M $ RET
```

Here we have used a vector mapping operation to evaluate the function at several values of '`s1`' at once. *V M $* is like *V M '* except that it takes the formula from the top of the stack. The formula is interpreted as a function to apply across the vector at the next-to-top stack level. Since a formula on the stack can't contain '`$`' signs, Calc assumes the variables in the formula stand for different arguments. It prompts you for an *argument list*, giving the list of all variables in the formula in alphabetical order as the default list. In this case the default is '`(s1)`', which is just what we want so we simply press *RET* at the prompt.

If there had been several different values, we could have used *V R X* to find the global maximum.

Calc has a built-in *a P* command that solves an equation using *H a S* and returns a vector of all the solutions. It simply automates the job we just did by hand. Applied to our original cubic polynomial, it would produce the vector of solutions $[1.19023, -1.19023, 0]$. (There is also an *a X* command which finds a local maximum of a function. It uses a numerical search method rather than examining the derivatives, and thus requires you to provide some kind of initial guess to show it where to look.)

(•) **Exercise 2.** Given a vector of the roots of a polynomial (such as the output of an *a P* command), what sequence of commands would you use to reconstruct the original polynomial? (The answer will be unique to within a constant multiple; choose the solution where the leading coefficient is one.) See *XXX* [Algebra Answer 2], page *XXX*. (•)

The *m s* command enables "symbolic mode," in which formulas like '`sqrt(5)`' that can't be evaluated exactly are left in symbolic form rather than giving a floating-point approximate answer. Fraction mode (*m f*) is also useful when doing algebra.

```
2:  34 x - 24 x^3        2:  34 x - 24 x^3
1:  34 x - 24 x^3        1:  [sqrt(51) / 6, sqrt(51) / -6, 0]
    .                        .

    r 2  RET     m s  m f    a P x RET
```

One more mode that makes reading formulas easier is "Big mode."

```
                  3
2:  34 x - 24 x


      ----    ----
     V 51    V 51
1:  [-----, -----, 0]
       6      -6


    .

    d B
```

Here things like powers, square roots, and quotients and fractions are displayed in a two-dimensional pictorial form. Calc has other language modes as well, such as C mode, FORTRAN mode, and TEX mode.

```
2:  34*x - 24*pow(x, 3)              2:  34*x - 24*x**3
1:  {sqrt(51) / 6, sqrt(51) / -6, 0}  1:  /sqrt(51) / 6, sqrt(51) / -6, 0/
    .                                     .

    d C                                   d F


3:  34 x - 24 x^3
2:  [{\sqrt{51} \over 6}, {\sqrt{51} \over -6}, 0]
1:  {2 \over 3} \sqrt{5}
    .

    d T   ' 2 \sqrt{5} \over 3 RET
```

As you can see, language modes affect both entry and display of formulas. They affect such things as the names used for built-in functions, the set of arithmetic operators and their precedences, and notations for vectors and matrices.

Notice that '`sqrt(51)`' may cause problems with older implementations of C and FORTRAN, which would require something more like '`sqrt(51.0)`'. It is always wise to check over the formulas produced by the various language modes to make sure they are fully correct.

Type *m s*, *m f*, and *d N* to reset these modes. (You may prefer to remain in Big mode, but all the examples in the tutorial are shown in normal mode.)

What is the area under the portion of this curve from $x = 1$ to 2? This is simply the integral of the function:

```
1:  17 x^2 - 6 x^4 + 3      1:  5.6666 x^3 - 1.2 x^5 + 3 x
    .                           .

    r 1                         a i x
```

We want to evaluate this at our two values for x and subtract. One way to do it is again with vector mapping and reduction:

```
2:  [2, 1]            1:  [12.93333, 7.46666]    1:  5.46666
1:  5.6666 x^3 ...        .                          .

   [ 2 , 1 ] TAB          V M $ RET                  V R -
```

(•) **Exercise 3.** Find the integral from 1 to y of $x \sin \pi x$ (where the sine is calculated in radians). Find the values of the integral for integers y from 1 to 5. See *XXX* [Algebra Answer 3], page *XXX*. (•)

Calc's integrator can do many simple integrals symbolically, but many others are beyond its capabilities. Suppose we wish to find the area under the curve $\sin x \ln x$ over the same range of x. If you entered this formula and typed `a i x RET` (don't bother to try this), Calc would work for a long time but would be unable to find a solution. In fact, there is no closed-form solution to this integral. Now what do we do?

One approach would be to do the integral numerically. It is not hard to do this by hand using vector mapping and reduction. It is rather slow, though, since the sine and logarithm functions take a long time. We can save some time by reducing the working precision.

```
     3:  10                  1:  [1, 1.1, 1.2,  ...  , 1.8, 1.9]
     2:  1                       .
     1:  0.1
         .

      10 RET 1 RET .1 RET        C-u v x
```

(Note that we have used the extended version of `v x`; we could also have used plain `v x` as follows: `v x 10 RET 9 + .1 *`.)

```
     2:  [1, 1.1, ... ]              1:  [0., 0.084941, 0.16993, ... ]
     1:  sin(x) ln(x)                    .
         .

         ' sin(x) ln(x) RET  s 1    m r  p 5 RET   V M $ RET


     1:  3.4195     0.34195
         .          .

         V R +      0.1 *
```

(If you got wildly different results, did you remember to switch to radians mode?)

Here we have divided the curve into ten segments of equal width; approximating these segments as rectangular boxes (i.e., assuming the curve is nearly flat at that resolution), we compute the areas of the boxes (height times width), then sum the areas. (It is faster to sum first, then multiply by the width, since the width is the same for every box.)

The true value of this integral turns out to be about 0.374, so we're not doing too well. Let's try another approach.

```
     1:  sin(x) ln(x)    1:  0.84147 x - 0.84147 + 0.11957 (x - 1)^2 - ...
         .                   .

         r 1                 a t x=1 RET 4 RET
```

Here we have computed the Taylor series expansion of the function about the point $x = 1$. We can now integrate this polynomial approximation, since polynomials are easy to integrate.

```
1:  0.42074 x^2 + ...     1:  [-0.0446, -0.42073]      1:  0.3761
    .                         .                             .

    a i x RET                 [ 2 , 1 ] TAB  V M $ RET      V R -
```

Better! By increasing the precision and/or asking for more terms in the Taylor series, we can get a result as accurate as we like. (Taylor series converge better away from singularities in the function such as the one at `ln(0)`, so it would also help to expand the series about the points $x = 2$ or $x = 1.5$ instead of $x = 1$.)

(•) **Exercise 4.** Our first method approximated the curve by stairsteps of width 0.1; the total area was then the sum of the areas of the rectangles under these stairsteps. Our second method approximated the function by a polynomial, which turned out to be a better approximation than stairsteps. A third method is *Simpson's rule*, which is like the stairstep method except that the steps are not required to be flat. Simpson's rule boils down to the formula,

$$\frac{h}{3}(f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + \cdots$$
$$+ 2f(a+(n-2)h) + 4f(a+(n-1)h) + f(a+nh))$$

where n (which must be even) is the number of slices and h is the width of each slice. These are 10 and 0.1 in our example. For reference, here is the corresponding formula for the stairstep method:

$$h(f(a) + f(a+h) + f(a+2h) + f(a+3h) + \cdots + f(a+(n-2)h) + f(a+(n-1)h))$$

Compute the integral from 1 to 2 of $\sin x \ln x$ using Simpson's rule with 10 slices. See *XXX* [Algebra Answer 4], page *XXX*. (•)

Calc has a built-in `a I` command for doing numerical integration. It uses *Romberg's method*, which is a more sophisticated cousin of Simpson's rule. In particular, it knows how to keep refining the result until the current precision is satisfied.

Aside from the commands we've seen so far, Calc also provides a large set of commands for operating on parts of formulas. You indicate the desired sub-formula by placing the cursor on any part of the formula before giving a *selection* command. Selections won't be covered in the tutorial; see *XXX* [Selecting Subformulas], page *XXX*, for details and examples.

2.5.2 Rewrite Rules

No matter how many built-in commands Calc provided for doing algebra, there would always be something you wanted to do that Calc didn't have in its repertoire. So Calc also provides a *rewrite rule* system that you can use to define your own algebraic manipulations.

Suppose we want to simplify this trigonometric formula:

```
1:  1 / cos(x) - sin(x) tan(x)
    .

    ' 1/cos(x) - sin(x) tan(x) RET   s 1
```

If we were simplifying this by hand, we'd probably replace the '`tan`' with a '`sin/cos`' first, then combine over a common denominator. There is no Calc command to do the former; the `a n` algebra command will do the latter but we'll do both with rewrite rules just for practice.

Rewrite rules are written with the ':=' symbol.

```
1:  1 / cos(x) - sin(x)^2 / cos(x)
    .

    a r tan(a) := sin(a)/cos(a) RET
```

(The "assignment operator" ':=' has several uses in Calc. All by itself the formula '`tan(a) := sin(a)/cos(a)`' doesn't do anything, but when it is given to the `a r` command, that command interprets it as a rewrite rule.)

The lefthand side, '`tan(a)`', is called the *pattern* of the rewrite rule. Calc searches the formula on the stack for parts that match the pattern. Variables in a rewrite pattern are called *meta-variables*, and when matching the pattern each meta-variable can match any sub-formula. Here, the meta-variable '`a`' matched the actual variable '`x`'.

When the pattern part of a rewrite rule matches a part of the formula, that part is replaced by the righthand side with all the meta-variables substituted with the things they matched. So the result is '`sin(x) / cos(x)`'. Calc's normal algebraic simplifications then mix this in with the rest of the original formula.

To merge over a common denominator, we can use another simple rule:

```
1:  (1 - sin(x)^2) / cos(x)
    .

    a r a/x + b/x := (a+b)/x RET
```

This rule points out several interesting features of rewrite patterns. First, if a meta-variable appears several times in a pattern, it must match the same thing everywhere. This rule detects common denominators because the same meta-variable '`x`' is used in both of the denominators.

Second, meta-variable names are independent from variables in the target formula. Notice that the meta-variable '`x`' here matches the subformula '`cos(x)`'; Calc never confuses the two meanings of '`x`'.

And third, rewrite patterns know a little bit about the algebraic properties of formulas. The pattern called for a sum of two quotients; Calc was able to match a difference of two quotients by matching '`a = 1`', '`b = -sin(x)^2`', and '`x = cos(x)`'.

We could just as easily have written '`a/x - b/x := (a-b)/x`' for the rule. It would have worked just the same in all cases. (If we really wanted the rule to apply only to '`+`' or only to '`-`', we could have used the `plain` symbol. See *XXX* [Algebraic Properties of Rewrite Rules], page *XXX*, for some examples of this.)

One more rewrite will complete the job. We want to use the identity '`sin(x)^2 + cos(x)^2 = 1`', but of course we must first rearrange the identity in a way that matches our formula. The obvious rule would be '`1 - sin(x)^2 := cos(x)^2`', but a little thought shows that the rule '`sin(x)^2 := 1 - cos(x)^2`' will also work. The latter rule has a more general pattern so it will work in many other situations, too.

```
1:  (1 + cos(x)^2 - 1) / cos(x)          1:  cos(x)
    .                                        .

    a r sin(x)^2 := 1 - cos(x)^2 RET         a s
```

You may ask, what's the point of using the most general rule if you have to type it in every time anyway? The answer is that Calc allows you to store a rewrite rule in a variable, then give the variable name in the *a r* command. In fact, this is the preferred way to use rewrites. For one, if you need a rule once you'll most likely need it again later. Also, if the rule doesn't work quite right you can simply Undo, edit the variable, and run the rule again without having to retype it.

```
' tan(x) := sin(x)/cos(x) RET      s t tsc RET
' a/x + b/x := (a+b)/x RET         s t merge RET
' sin(x)^2 := 1 - cos(x)^2 RET     s t sinsqr RET

1:  1 / cos(x) - sin(x) tan(x)     1:  cos(x)
    .                                  .

    r 1                  a r tsc RET  a r merge RET  a r sinsqr RET  a s
```

To edit a variable, type *s e* and the variable name, use regular Emacs editing commands as necessary, then type *M-# M-#* or *C-c C-c* to store the edited value back into the variable. You can also use *s e* to create a new variable if you wish.

Notice that the first time you use each rule, Calc puts up a "compiling" message briefly. The pattern matcher converts rules into a special optimized pattern-matching language rather than using them directly. This allows *a r* to apply even rather complicated rules very efficiently. If the rule is stored in a variable, Calc compiles it only once and stores the compiled form along with the variable. That's another good reason to store your rules in variables rather than entering them on the fly.

(•) **Exercise 1.** Type *m s* to get symbolic mode, then enter the formula '`(2 + sqrt(2)) / (1 + sqrt(2))`'. Using a rewrite rule, simplify this formula by multiplying both sides by the conjugate '`1 - sqrt(2)`'. The result will have to be expanded by the distributive law; do this with another rewrite. See *XXX* [Rewrites Answer 1], page *XXX*. (•)

The *a r* command can also accept a vector of rewrite rules, or a variable containing a vector of rules.

```
1:  [tsc, merge, sinsqr]          1:  [tan(x) := sin(x) / cos(x), ... ]
    .                                 .

    ' [tsc,merge,sinsqr] RET          =

1:  1 / cos(x) - sin(x) tan(x)    1:  cos(x)
    .                                 .

    s t trig RET  r 1                 a r trig RET  a s
```

Calc tries all the rules you give against all parts of the formula, repeating until no further change is possible. (The exact order in which things are tried is rather complex, but for simple rules like the ones we've used here the order doesn't really matter. See *XXX* [Nested Formulas with Rewrite Rules], page *XXX*.)

Calc actually repeats only up to 100 times, just in case your rule set has gotten into an infinite loop. You can give a numeric prefix argument to *a r* to specify any limit. In particular, *M-1 a r* does only one rewrite at a time.

```
1:  1 / cos(x) - sin(x)^2 / cos(x)    1:  (1 - sin(x)^2) / cos(x)
    .                                     .

    r 1  M-1 a r trig RET                 M-1 a r trig RET
```

You can type *M-0* `a r` if you want no limit at all on the number of rewrites that occur.

Rewrite rules can also be *conditional*. Simply follow the rule with a '`::`' symbol and the desired condition. For example,

```
1:  exp(2 pi i) + exp(3 pi i) + exp(4 pi i)
    .

    ' exp(2 pi i) + exp(3 pi i) + exp(4 pi i) RET


1:  1 + exp(3 pi i) + 1
    .

    a r exp(k pi i) := 1 :: k % 2 = 0 RET
```

(Recall, '`k % 2`' is the remainder from dividing '`k`' by 2, which will be zero only when '`k`' is an even integer.)

An interesting point is that the variables '`pi`' and '`i`' were matched literally rather than acting as meta-variables. This is because they are special-constant variables. The special constants '`e`', '`phi`', and so on also match literally. A common error with rewrite rules is to write, say, '`f(a,b,c,d,e) := g(a+b+c+d+e)`', expecting to match any '`f`' with five arguments but in fact matching only when the fifth argument is literally '`e`'!

fib Rewrite rules provide an interesting way to define your own functions. Suppose we want to define '`fib(n)`' to produce the *n*th Fibonacci number. The first two Fibonacci numbers are each 1; later numbers are formed by summing the two preceding numbers in the sequence. This is easy to express in a set of three rules:

```
' [fib(1) := 1, fib(2) := 1, fib(n) := fib(n-1) + fib(n-2)] RET  s t fib

1:  fib(7)               1:  13
    .                        .

    ' fib(7) RET             a r fib RET
```

One thing that is guaranteed about the order that rewrites are tried is that, for any given subformula, earlier rules in the rule set will be tried for that subformula before later ones. So even though the first and third rules both match '`fib(1)`', we know the first will be used preferentially.

This rule set has one dangerous bug: Suppose we apply it to the formula '`fib(x)`'? (Don't actually try this.) The third rule will match '`fib(x)`' and replace it with '`fib(x-1) + fib(x-2)`'. Each of these will then be replaced to get '`fib(x-2) + 2 fib(x-3) + fib(x-4)`', and so on, expanding forever. What we really want is to apply the third rule only when '`n`' is an integer greater than two. Type *s e fib RET*, then edit the third rule to:

```
fib(n) := fib(n-1) + fib(n-2) :: integer(n) :: n > 2
```

Now:

```
1:  fib(6) + fib(x) + fib(0)      1:  8 + fib(x) + fib(0)
    .                                 .

    ' fib(6)+fib(x)+fib(0) RET        a r fib RET
```

We've created a new function, `fib`, and a new command, *a r fib RET*, which means "evaluate all `fib` calls in this formula." To make things easier still, we can tell Calc to apply these rules automatically by storing them in the special variable `EvalRules`.

```
1:  [fib(1) := ...]    .                  1:  [8, 13]
    .                                         .

    s r fib RET         s t EvalRules RET    ' [fib(6), fib(7)] RET
```

It turns out that this rule set has the problem that it does far more work than it needs to when '`n`' is large. Consider the first few steps of the computation of '`fib(6)`':

```
fib(6) =
fib(5)              +               fib(4) =
fib(4)     +      fib(3)     +      fib(3)     +      fib(2) =
fib(3) + fib(2) + fib(2) + fib(1) + fib(2) + fib(1) + 1 = ...
```

Note that '`fib(3)`' appears three times here. Unless Calc's algebraic simplifier notices the multiple '`fib(3)`'s and combines them (and, as it happens, it doesn't), this rule set does lots of needless recomputation. To cure the problem, type `s e EvalRules` to edit the rules (or just *s E*, a shorthand command for editing `EvalRules`) and add another condition:

```
fib(n) := fib(n-1) + fib(n-2) :: integer(n) :: n > 2 :: remember
```

If a '`:: remember`' condition appears anywhere in a rule, then if that rule succeeds Calc will add another rule that describes that match to the front of the rule set. (Remembering works in any rule set, but for technical reasons it is most effective in `EvalRules`.) For example, if the rule rewrites '`fib(7)`' to something that evaluates to 13, then the rule '`fib(7) := 13`' will be added to the rule set.

Type *' fib(8) RET* to compute the eighth Fibonacci number, then type *s E* again to see what has happened to the rule set.

With the `remember` feature, our rule set can now compute '`fib(`*n*`)`' in just *n* steps. In the process it builds up a table of all Fibonacci numbers up to *n*. After we have computed the result for a particular *n*, we can get it back (and the results for all smaller *n*) later in just one step.

All Calc operations will run somewhat slower whenever `EvalRules` contains any rules. You should type *s u EvalRules RET* now to un-store the variable.

(•) **Exercise 2.** Sometimes it is possible to reformulate a problem to reduce the amount of recursion necessary to solve it. Create a rule that, in about *n* simple steps and without recourse to the `remember` option, replaces '`fib(`*n*`, 1, 1)`' with '`fib(1, `*x*`, `*y*`)`' where *x* and *y* are the *n*th and *n+1*st Fibonacci numbers, respectively. This rule is rather clunky to use, so add a couple more rules to make the "user interface" the same as for our first version: enter '`fib(`*n*`)`', get back a plain number. See *XXX* [Rewrites Answer 2], page *XXX*. (•)

There are many more things that rewrites can do. For example, there are '`&&&`' and '`|||`' pattern operators that create "and" and "or" combinations of rules. As one really simple example, we could combine our first two Fibonacci rules thusly:

```
[fib(1 ||| 2) := 1, fib(n) := ... ]
```

That means "`fib` of something matching either 1 or 2 rewrites to 1."

You can also make meta-variables optional by enclosing them in `opt`. For example, the pattern '`a + b x`' matches '`2 + 3 x`' but not '`2 + x`' or '`3 x`' or '`x`'. The pattern '`opt(a) + opt(b) x`' matches all of these forms, filling in a default of zero for '`a`' and one for '`b`'.

(●) **Exercise 3.** Your friend Joe had '`2 + 3 x`' on the stack and tried to use the rule '`opt(a) + opt(b) x := f(a, b, x)`'. What happened? See *XXX* [Rewrites Answer 3], page *XXX*. (●)

(●) **Exercise 4.** Starting with a positive integer a, divide a by two if it is even, otherwise compute $3a + 1$. Now repeat this step over and over. A famous unproved conjecture is that for any starting a, the sequence always eventually reaches 1. Given the formula '`seq(`*a*`, 0)`', write a set of rules that convert this into '`seq(1, `*n*`)`' where *n* is the number of steps it took the sequence to reach the value 1. Now enhance the rules to accept '`seq(`*a*`)`' as a starting configuration, and to stop with just the number *n* by itself. Now make the result be a vector of values in the sequence, from *a* to 1. (The formula '`x|`*y*' appends the vectors *x* and *y*.) For example, rewriting '`seq(6)`' should yield the vector $[6, 3, 10, 5, 16, 8, 4, 2, 1]$. See *XXX* [Rewrites Answer 4], page *XXX*. (●)

(●) **Exercise 5.** Define, using rewrite rules, a function '`nterms(`*x*`)`' that returns the number of terms in the sum *x*, or 1 if *x* is not a sum. (A *sum* for our purposes is one or more non-sum terms separated by '`+`' or '`-`' signs, so that $2 - 3(x + y) + xy$ is a sum of three terms.) See *XXX* [Rewrites Answer 5], page *XXX*. (●)

(●) **Exercise 6.** Calc considers the form 0^0 to be "indeterminate," and leaves it unevaluated (assuming infinite mode is not enabled). Some people prefer to define $0^0 = 1$, so that the identity $x^0 = 1$ can safely be used for all x. Find a way to make Calc follow this convention. What happens if you now type *m i* to turn on infinite mode? See *XXX* [Rewrites Answer 6], page *XXX*. (●)

(●) **Exercise 7.** A Taylor series for a function is an infinite series that exactly equals the value of that function at values of x near zero.

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

The `a t` command produces a *truncated Taylor series* which is obtained by dropping all the terms higher than, say, x^2. Calc represents the truncated Taylor series as a polynomial in x. Mathematicians often write a truncated series using a "big-O" notation that records what was the lowest term that was truncated.

$$\cos x = 1 - \frac{x^2}{2!} + O(x^3)$$

The meaning of $O(x^3)$ is "a quantity which is negligibly small if x^3 is considered negligibly small as x goes to zero."

The exercise is to create rewrite rules that simplify sums and products of power series represented as '*polynomial* `+ O(`*var*`^`*n*`)`'. For example, given '`1 - x^2 / 2 + O(x^3)`' and '`x - x^3 / 6 + O(x^4)`' on the stack, we want to be able to type `*` and get the result '`x - 2:3 x^3 + O(x^4)`'. Don't worry if the terms of the sum are rearranged or if *a s* needs to be typed after rewriting. (This one is rather tricky; the solution at the end of this chapter uses 6 rewrite rules. Hint: The '`constant(x)`' condition tests whether '`x`' is a number.) See *XXX* [Rewrites Answer 7], page *XXX*. (●)

See *XXX* [Rewrite Rules], page *XXX*, for the whole story on rewrite rules.

2.6 Programming Tutorial

The Calculator is written entirely in Emacs Lisp, a highly extensible language. If you know Lisp, you can program the Calculator to do anything you like. Rewrite rules also work as a powerful programming system. But Lisp and rewrite rules take a while to master, and often all you want to do is define a new function or repeat a command a few times. Calc has features that allow you to do these things easily.

(Note that the programming commands relating to user-defined keys are not yet supported under Lucid Emacs 19.)

One very limited form of programming is defining your own functions. Calc's *Z F* command allows you to define a function name and key sequence to correspond to any formula. Programming commands use the shift-*Z* prefix; the user commands they create use the lower case *z* prefix.

```
1:  1 + x + x^2 / 2 + x^3 / 6         1:  1 + x + x^2 / 2 + x^3 / 6
    .                                     .

    ' 1 + x + x^2/2! + x^3/3! RET         Z F e myexp RET RET RET y
```

This polynomial is a Taylor series approximation to '`exp(x)`'. The *Z F* command asks a number of questions. The above answers say that the key sequence for our function should be *z* `e`; the *M-x* equivalent should be `calc-myexp`; the name of the function in algebraic formulas should also be `myexp`; the default argument list '`(x)`' is acceptable; and finally *y* answers the question "leave it in symbolic form for non-constant arguments?"

```
1:  1.3495     2:  1.3495     3:  1.3495
    .          1:  1.34986    2:  1.34986
                   .          1:  myexp(a + 1)
                                  .

    .3 z e         .3 E           ' a+1 RET z e
```

First we call our new `exp` approximation with 0.3 as an argument, and compare it with the true `exp` function. Then we note that, as requested, if we try to give *z* `e` an argument that isn't a plain number, it leaves the `myexp` function call in symbolic form. If we had answered *n* to the final question, '`myexp(a + 1)`' would have evaluated by plugging in '`a + 1`' for '`x`' in the defining formula.

(•) **Exercise 1.** The "sine integral" function Si(x) is defined as the integral of '`sin(t)/t`' for $t = 0$ to x in radians. (It was invented because this integral has no solution in terms of basic functions; if you give it to Calc's `a` *i* command, it will ponder it for a long time and then give up.) We can use the numerical integration command, however, which in algebraic notation is written like '`ninteg(f(t), t, 0, x)`' with any integrand '`f(t)`'. Define a *z s* command and `Si` function that implement this. You will need to edit the default argument list a bit. As a test, '`Si(1)`' should return 0.946083. (Hint: `ninteg` will run a lot faster if you reduce the precision to, say, six digits beforehand.) See *XXX* [Programming Answer 1], page *XXX*. (•) `Si`

The simplest way to do real "programming" of Emacs is to define a *keyboard macro*. A keyboard macro is simply a sequence of keystrokes which Emacs has stored away and can play back on demand. For example, if you find yourself typing *H* `a` *S x RET* often, you may wish to program a keyboard macro to type this for you.

```
1:  y = sqrt(x)         1:  x = y^2
    .                       .

    ' y=sqrt(x) RET       C-x ( H a S x RET C-x )

1:  y = cos(x)          1:  x = s1 arccos(y) + 2 pi n1
    .                       .

    ' y=cos(x) RET            X
```

When you type *C-x (*, Emacs begins recording. But it is also still ready to execute your keystrokes, so you're really "training" Emacs by walking it through the procedure once. When you type *C-x)*, the macro is recorded. You can now type *X* to re-execute the same keystrokes.

You can give a name to your macro by typing *Z K*.

```
1:  .                 1:  y = x^4         1:  x = s2 sqrt(s1 sqrt(y))
                          .                   .


  Z K x RET               ' y=x^4 RET         z x
```

Notice that we use shift-*Z* to define the command, and lower-case *z* to call it up.

Keyboard macros can call other macros.

```
1:  abs(x)        1:  x = s1 y                1:  2 / x    1:  x = 2 / y
    .                 .                           .            .

  ' abs(x) RET   C-x ( ' y RET a = z x C-x )    ' 2/x RET       X
```

(•) **Exercise 2.** Define a keyboard macro to negate the item in level 3 of the stack, without disturbing the rest of the stack. See *XXX* [Programming Answer 2], page *XXX*. (•)

(•) **Exercise 3.** Define keyboard macros to compute the following functions:

1. Compute $\frac{\sin x}{x}$, where x is the number on the top of the stack.
2. Compute the base-b logarithm, just like the *B* key except the arguments are taken in the opposite order.
3. Produce a vector of integers from 1 to the integer on the top of the stack.

See *XXX* [Programming Answer 3], page *XXX*. (•)

(•) **Exercise 4.** Define a keyboard macro to compute the average (mean) value of a list of numbers. See *XXX* [Programming Answer 4], page *XXX*. (•)

In many programs, some of the steps must execute several times. Calc has *looping* commands that allow this. Loops are useful inside keyboard macros, but actually work at any time.

```
1:  x^6           2:  x^6          1: 360 x^2
    .             1:  4                .
                      .

  ' x^6 RET           4            Z < a d x RET Z >
```

Here we have computed the fourth derivative of x^6 by enclosing a derivative command in a "repeat loop" structure. This structure pops a repeat count from the stack, then executes the body of the loop that many times.

If you make a mistake while entering the body of the loop, type *Z C-g* to cancel the loop command.

Here's another example:

```
3:  1               2:  10946
2:  1               1:  17711
1:  20                  .
    .

1 RET RET 20        Z < TAB C-j + Z >
```

The numbers in levels 2 and 1 should be the 21st and 22nd Fibonacci numbers, respectively. (To see what's going on, try a few repetitions of the loop body by hand; *C-j*, also on the Line-Feed or *LFD* key if you have one, makes a copy of the number in level 2.)

A fascinating property of the Fibonacci numbers is that the nth Fibonacci number can be found directly by computing $\phi^n/\sqrt{5}$ and then rounding to the nearest integer, where ϕ ("phi"), the "golden ratio," is $(1+\sqrt{5})/2$. (For convenience, this constant is available from the `phi` variable, or the *I H P* command.)

```
1:  1.61803         1:  24476.0000409    1:  10945.9999817    1:  10946
    .                   .                    .                    .

    I H P               21 ^                 5 Q /                R
```

(•) **Exercise 5.** The *continued fraction* representation of ϕ is $1 + 1/(1 + 1/(1 + 1/(\ldots)))$. We can compute an approximate value by carrying this however far and then replacing the innermost $1/(\ldots)$ by 1. Approximate ϕ using a twenty-term continued fraction. See *XXX* [Programming Answer 5], page *XXX*. (•)

(•) **Exercise 6.** Linear recurrences like the one for Fibonacci numbers can be expressed in terms of matrices. Given a vector $[a, b]$ determine a matrix which, when multiplied by this vector, produces the vector $[b, c]$, where a, b and c are three successive Fibonacci numbers. Now write a program that, given an integer n, computes the nth Fibonacci number using matrix arithmetic. See *XXX* [Programming Answer 6], page *XXX*. (•)

A more sophisticated kind of loop is the *for* loop. Suppose we wish to compute the 20th "harmonic" number, which is equal to the sum of the reciprocals of the integers from 1 to 20.

```
3:  0               1:  3.597739
2:  1                   .
1:  20
    .

0 RET 1 RET 20        Z ( & + 1 Z )
```

The "for" loop pops two numbers, the lower and upper limits, then repeats the body of the loop as an internal counter increases from the lower limit to the upper one. Just before executing the loop body, it pushes the current loop counter. When the loop body finishes, it pops the "step," i.e., the amount by which to increment the loop counter. As you can see, our loop always uses a step of one.

This harmonic number function uses the stack to hold the running total as well as for the various loop housekeeping functions. If you find this disorienting, you can sum in a variable instead:

```
1:  0         2:  1                     .              1:  3.597739
    .         1:  20                                       .
                  .

    0 t 7        1 RET 20       Z ( & s + 7 1 Z )        r 7
```

The *s +* command adds the top-of-stack into the value in a variable (and removes that value from the stack).

It's worth noting that many jobs that call for a "for" loop can also be done more easily by Calc's high-level operations. Two other ways to compute harmonic numbers are to use vector mapping and reduction (*v x 20*, then *V M &*, then *V R +*), or to use the summation command *a +*. Both of these are probably easier than using loops. However, there are some situations where loops really are the way to go:

(•) **Exercise 7.** Use a "for" loop to find the first harmonic number which is greater than 4.0. See *XXX* [Programming Answer 7], page *XXX*. (•)

Of course, if we're going to be using variables in our programs, we have to worry about the programs clobbering values that the caller was keeping in those same variables. This is easy to fix, though:

```
    .         1:  0.6667       1:  0.6667     3:  0.6667
                  .                .          2:  3.597739
                                              1:  0.6667
                                                  .

  Z `     p 4 RET 2 RET 3 /    s 7 s s a RET    Z '  r 7 s r a RET
```

When we type *Z `* (that's a back-quote character), Calc saves its mode settings and the contents of the ten "quick variables" for later reference. When we type *Z '* (that's an apostrophe now), Calc restores those saved values. Thus the *p 4* and *s 7* commands have no effect outside this sequence. Wrapping this around the body of a keyboard macro ensures that it doesn't interfere with what the user of the macro was doing. Notice that the contents of the stack, and the values of named variables, survive past the *Z '* command.

The *Bernoulli numbers* are a sequence with the interesting property that all of the odd Bernoulli numbers are zero, and the even ones, while difficult to compute, can be roughly approximated by the formula $\frac{2n!}{(2\pi)^n}$. Let's write a keyboard macro to compute (approximate) Bernoulli numbers. (Calc has a command, *k b*, to compute exact Bernoulli numbers, but this command is very slow for large n since the higher Bernoulli numbers are very large fractions.)

```
1:  10                  1:  0.0756823
    .                       .

    10      C-x ( RET 2 % Z [ DEL 0 Z : ' 2 $! / (2 pi)^$ RET = Z ] C-x )
```

You can read *Z [* as "then," *Z :* as "else," and *Z]* as "end-if." There is no need for an explicit "if" command. For the purposes of *Z [*, the condition is "true" if the value it pops from the stack is a nonzero number, or "false" if it pops zero or something that is not a number (like a formula). Here we take our integer argument modulo 2; this will be nonzero if we're asking for an odd Bernoulli number.

The actual tenth Bernoulli number is 5/66.

```
3:  0.0756823    1:  0            1:  0.25305    1:  0            1:  1.16659
2:  5:66             .                .              .                .
1:  0.0757575
    .

10 k b RET c f   M-0 DEL 11 X   DEL 12 X       DEL 13 X       DEL 14 X
```

Just to exercise loops a bit more, let's compute a table of even Bernoulli numbers.

```
3:  []              1:  [0.10132, 0.03079, 0.02340, 0.033197, ...]
2:  2                   .
1:  30
    .

 [ ] 2 RET 30           Z ( X | 2 Z )
```

The vertical-bar `|` is the vector-concatenation command. When we execute it, the list we are building will be in stack level 2 (initially this is an empty list), and the next Bernoulli number will be in level 1. The effect is to append the Bernoulli number onto the end of the list. (To create a table of exact fractional Bernoulli numbers, just replace *X* with *k b* in the above sequence of keystrokes.)

With loops and conditionals, you can program essentially anything in Calc. One other command that makes looping easier is *Z /*, which takes a condition from the stack and breaks out of the enclosing loop if the condition is true (non-zero). You can use this to make "while" and "until" style loops.

If you make a mistake when entering a keyboard macro, you can edit it using *Z E*. First, you must attach it to a key with *Z K*. One technique is to enter a throwaway dummy definition for the macro, then enter the real one in the edit command.

```
1:  3                  1:  3           Keyboard Macro Editor.
    .                      .           Original keys: 1 RET 2 +

                                       type "1\r"
                                       type "2"
                                       calc-plus

C-x ( 1 RET 2 + C-x )    Z K h RET      Z E h
```

This shows the screen display assuming you have the '`macedit`' keyboard macro editing package installed, which is usually the case since a copy of '`macedit`' comes bundled with Calc.

A keyboard macro is stored as a pure keystroke sequence. The '`macedit`' package (invoked by *Z E*) scans along the macro and tries to decode it back into human-readable steps. If a key or keys are simply shorthand for some command with a *M-x* name, that name is shown. Anything that doesn't correspond to a *M-x* command is written as a '`type`' command.

Let's edit in a new definition, for computing harmonic numbers. First, erase the three lines of the old definition. Then, type in the new definition (or use Emacs *M-w* and *C-y* commands to copy it from this page of the Info file; you can skip typing the comments that begin with '`#`').

```
calc-kbd-push         # Save local values (Z ‘)
type "0"              # Push a zero
calc-store-into       # Store it in variable 1
type "1"
```

```
    type "1"               # Initial value for loop
    calc-roll-down         # This is the TAB key; swap initial & final
    calc-kbd-for           # Begin "for" loop...
    calc-inv               #   Take reciprocal
    calc-store-plus        #   Add to accumulator
    type "1"
    type "1"               #   Loop step is 1
    calc-kbd-end-for       # End "for" loop
    calc-recall            # Now recall final accumulated value
    type "1"
    calc-kbd-pop           # Restore values (Z ')
```

Press *M-# M-#* to finish editing and return to the Calculator.

```
    1:  20          1:  3.597739
        .               .

        20              z h
```

If you don't know how to write a particular command in '`macedit`' format, you can always write it as keystrokes in a `type` command. There is also a `keys` command which interprets the rest of the line as standard Emacs keystroke names. In fact, '`macedit`' defines a handy `read-kbd-macro` command which reads the current region of the current buffer as a sequence of keystroke names, and defines that sequence on the *X* (and *C-x* `e`) key. Because this is so useful, Calc puts this command on the *M-# m* key. Try reading in this macro in the following form: Press *C-@* (or *C-SPC*) at one end of the text below, then type *M-# m* at the other.

```
    Z ‘ 0 t 1
        1 TAB
        Z (  & s + 1  1 Z )
        r 1
    Z ’
```

(•) **Exercise 8.** A general algorithm for solving equations numerically is *Newton's Method*. Given the equation $f(x) = 0$ for any function f, and an initial guess x_0 which is reasonably close to the desired solution, apply this formula over and over:

$$x_{\text{new}} = x - \frac{f(x)}{f'(x)}$$

where $f'(x)$ is the derivative of f. The x values will quickly converge to a solution, i.e., eventually x_{new} and x will be equal to within the limits of the current precision. Write a program which takes a formula involving the variable x, and an initial guess x_0, on the stack, and produces a value of x for which the formula is zero. Use it to find a solution of $\sin(\cos x) = 0.5$ near $x = 4.5$. (Use angles measured in radians.) Note that the built-in `a` *R* (`calc-find-root`) command uses Newton's method when it is able. See *XXX* [Programming Answer 8], page *XXX*. (•)

(•) **Exercise 9.** The *digamma* function $\psi(z)$ ("psi") is defined as the derivative of $\ln\Gamma(z)$. For large values of z, it can be approximated by the infinite sum

$$\psi(z) \approx \ln z - \frac{1}{2z} - \sum_{n=1}^{\infty} \frac{\texttt{bern}(2n)}{2nz^{2n}}$$

where $\sum$ represents the sum over n from 1 to infinity (or to some limit high enough to give the desired accuracy), and the `bern` function produces (exact) Bernoulli numbers. While this sum is not guaranteed to converge, in practice it is safe. An interesting mathematical constant is Euler's gamma, which is equal to about 0.5772. One way to compute it is by the formula, $\gamma = -\psi(1)$. Unfortunately, 1 isn't a large enough argument for the above formula to work (5 is a much safer value for z). Fortunately, we can compute $\psi(1)$ from $\psi(5)$ using the recurrence $\psi(z+1) = \psi(z) + \frac{1}{z}$. Your task: Develop a program to compute $\psi(z)$; it should "pump up" z if necessary to be greater than 5, then use the above summation formula. Use looping commands to compute the sum. Use your function to compute γ to twelve decimal places. (Calc has a built-in command for Euler's constant, `I P`, which you can use to check your answer.) See *XXX* [Programming Answer 9], page *XXX*. (•)

(•) **Exercise 10.** Given a polynomial in x and a number m on the stack, where the polynomial is of degree m or less (i.e., does not have any terms higher than x^m), write a program to convert the polynomial into a list-of-coefficients notation. For example, $5x^4 + (x+1)^2$ with $m = 6$ should produce the list $[1, 2, 1, 0, 5, 0, 0]$. Also develop a way to convert from this form back to the standard algebraic form. See *XXX* [Programming Answer 10], page *XXX*. (•)

(•) **Exercise 11.** The *Stirling numbers of the first kind* are defined by the recurrences,

$$\begin{aligned} s(n,n) &= 1 \qquad \text{for } n \ge 0,\\ s(n,0) &= 0 \qquad \text{for } n > 0,\\ s(n+1,m) &= s(n,m-1) - n\,s(n,m) \qquad \text{for } n \ge m \ge 1. \end{aligned}$$

(These numbers are also sometimes written $\begin{bmatrix} n \\ m \end{bmatrix}$.)

This can be implemented using a *recursive* program in Calc; the program must invoke itself in order to calculate the two righthand terms in the general formula. Since it always invokes itself with "simpler" arguments, it's easy to see that it must eventually finish the computation. Recursion is a little difficult with Emacs keyboard macros since the macro is executed before its definition is complete. So here's the recommended strategy: Create a "dummy macro" and assign it to a key with, e.g., `Z K s`. Now enter the true definition, using the `z s` command to call itself recursively, then assign it to the same key with `Z K s`. Now the `z s` command will run the complete recursive program. (Another way is to use `Z E` or `M-# m` (`read-kbd-macro`) to read the whole macro at once, thus avoiding the "training" phase.) The task: Write a program that computes Stirling numbers of the first kind, given n and m on the stack. Test it with *small* inputs like $s(4,2)$. (There is a built-in command for Stirling numbers, `k s`, which you can use to check your answers.) See *XXX* [Programming Answer 11], page *XXX*. (•)

The programming commands we've seen in this part of the tutorial are low-level, general-purpose operations. Often you will find that a higher-level function, such as vector mapping or rewrite rules, will do the job much more easily than a detailed, step-by-step program can:

(•) **Exercise 12.** Write another program for computing Stirling numbers of the first kind, this time using rewrite rules. Once again, n and m should be taken from the stack. See *XXX* [Programming Answer 12], page *XXX*. (•)

This ends the tutorial section of the Calc manual. Now you know enough about Calc to use it effectively for many kinds of calculations. But Calc has many features that were not even touched upon in this tutorial. The rest of this manual tells the whole story.

2.7 Answers to Exercises

This section includes answers to all the exercises in the Calc tutorial.

2.7.1 RPN Tutorial Exercise 1

*1 RET 2 RET 3 RET 4 + * -*

The result is $1-(2\times(3+4))=-13$.

2.7.2 RPN Tutorial Exercise 2

$2\times4+7\times9.5+\frac{5}{4}=75.75$

After computing the intermediate term $2\times4=8$, you can leave that result on the stack while you compute the second term. With both of these results waiting on the stack you can then compute the final term, then press `+ +` to add everything up.

```
2:  2          1:  8          3:  8          2:  8
1:  4              .          2:  7          1:  66.5
    .                         1:  9.5            .
                                  .

  2 RET 4          *          7 RET 9.5          *


4:  8          3:  8          2:  8          1:  75.75
3:  66.5       2:  66.5       1:  67.75          .
2:  5          1:  1.25           .
1:  4              .
    .

  5 RET 4          /              +              +
```

Alternatively, you could add the first two terms before going on with the third term.

```
2:  8          1:  74.5       3:  74.5       2:  74.5       1:  75.75
1:  66.5           .          2:  5          1:  1.25           .
    .                         1:  4              .
                                  .

   ...             +            5 RET 4          /              +
```

On an old-style RPN calculator this second method would have the advantage of using only three stack levels. But since Calc's stack can grow arbitrarily large this isn't really an issue. Which method you choose is purely a matter of taste.

2.7.3 RPN Tutorial Exercise 3

The *TAB* key provides a way to operate on the number in level 2.

```
3:  10          3:  10          4:  10          3:  10          3:  10
2:  20          2:  30          3:  30          2:  30          2:  21
1:  30          1:  20          2:  20          1:  21          1:  30
    .               .           1:  1               .               .
                                    .

                    TAB             1               +               TAB
```

Similarly, *M-TAB* gives you access to the number in level 3.

```
3:  10          3:  21          3:  21          3:  30          3:  11
2:  21          2:  30          2:  30          2:  11          2:  21
1:  30          1:  10          1:  11          1:  21          1:  30
    .               .               .               .               .

                    M-TAB           1 +             M-TAB           M-TAB
```

2.7.4 RPN Tutorial Exercise 4

Either *(2 , 3)* or *(2 SPC 3)* would have worked, but using both the comma and the space at once yields:

```
1:  ( ...       2:  ( ...       1:  (2, ...     2:  (2, ...     2:  (2, ...
    .           1:  2               .           1:  (2, ...     1:  (2, 3)
                    .                               .               .

    (               2               ,               SPC             3 )
```

Joe probably tried to type *TAB DEL* to swap the extra incomplete object to the top of the stack and delete it. But a feature of Calc is that *DEL* on an incomplete object deletes just one component out of that object, so he had to press *DEL* twice to finish the job.

```
2:  (2, ...     2:  (2, 3)      2:  (2, 3)      1:  (2, 3)
1:  (2, 3)      1:  (2, ...     1:  ( ...           .
    .               .               .

                    TAB             DEL             DEL
```

(As it turns out, deleting the second-to-top stack entry happens often enough that Calc provides a special key, *M-DEL*, to do just that. *M-DEL* is just like *TAB DEL*, except that it doesn't exhibit the "feature" that tripped poor Joe.)

2.7.5 Algebraic Entry Tutorial Exercise 1

Type *' sqrt($) RET*.

If the *Q* key is broken, you could use *' $^0.5 RET*. Or, RPN style, *0.5 ^*.

(Actually, '`$^1:2`', using the fraction one-half as the power, is a closer equivalent, since '`9^0.5`' yields 3.0 whereas '`sqrt(9)`' and '`9^1:2`' yield the exact integer 3.)

2.7.6 Algebraic Entry Tutorial Exercise 2

In the formula '`2 x (1+y)`', '`x`' was interpreted as a function name with '`1+y`' as its argument. Assigning a value to a variable has no relation to a function by the same name. Joe needed to use an explicit '`*`' symbol here: '`2 x*(1+y)`'.

2.7.7 Algebraic Entry Tutorial Exercise 3

The result from *1 RET 0 /* will be the formula 1/0. The "function" '`/`' cannot be evaluated when its second argument is zero, so it is left in symbolic form. When you now type *0 **, the result will be zero because Calc uses the general rule that "zero times anything is zero."

The *m i* command enables an *infinite mode* in which 1/0 results in a special symbol that represents "infinity." If you multiply infinity by zero, Calc uses another special new symbol to show that the answer is "indeterminate." See *XXX* [Infinities], page *XXX*, for further discussion of infinite and indeterminate values.

2.7.8 Modes Tutorial Exercise 1

Calc always stores its numbers in decimal, so even though one-third has an exact base-3 representation ('`3#0.1`'), it is still stored as 0.3333333 (chopped off after 12 or however many decimal digits) inside the calculator's memory. When this inexact number is converted back to base 3 for display, it may still be slightly inexact. When we multiply this number by 3, we get 0.999999, also an inexact value.

When Calc displays a number in base 3, it has to decide how many digits to show. If the current precision is 12 (decimal) digits, that corresponds to '`12 / log10(3) = 25.15`' base-3 digits. Because 25.15 is not an exact integer, Calc shows only 25 digits, with the result that stored numbers carry a little bit of extra information that may not show up on the screen. When Joe entered '`3#0.2`', the stored number 0.666666 happened to round to a pleasing value when it lost that last 0.15 of a digit, but it was still inexact in Calc's memory. When he divided by 2, he still got the dreaded inexact value 0.333333. (Actually, he divided 0.666667 by 2 to get 0.333334, which is why he got something a little higher than `3#0.1` instead of a little lower.)

If Joe didn't want to be bothered with all this, he could have typed *M-24 d n* to display with one less digit than the default. (If you give *d n* a negative argument, it uses default-minus-that, so *M-- d n* would be an easier way to get the same effect.) Those inexact results would still be lurking there, but they would now be rounded to nice, natural-looking values for display purposes. (Remember, '0.022222' in base 3 is like '0.099999' in base 10; rounding off one digit will round the number up to '0.1'.) Depending on the nature of your work, this hiding of the inexactness may be a benefit or a danger. With the *d n* command, Calc gives you the choice.

Incidentally, another consequence of all this is that if you type *M-30 d n* to display more digits than are "really there," you'll see garbage digits at the end of the number. (In decimal display mode, with decimally-stored numbers, these garbage digits are always zero so they vanish and you don't notice them.) Because Calc rounds off that 0.15 digit, there is the danger that two numbers could be slightly different internally but still look the same. If you feel uneasy about this, set the *d n* precision to be a little higher than normal; you'll get ugly garbage digits, but you'll always be able to tell two distinct numbers apart.

An interesting side note is that most computers store their floating-point numbers in binary, and convert to decimal for display. Thus everyday programs have the same problem: Decimal 0.1 cannot be represented exactly in binary (try it: *0.1 d 2*), so '0.1 * 10' comes out as an inexact approximation to 1 on some machines (though they generally arrange to hide it from you by rounding off one digit as we did above). Because Calc works in decimal instead of binary, you can be sure that numbers that look exact *are* exact as long as you stay in decimal display mode.

It's not hard to show that any number that can be represented exactly in binary, octal, or hexadecimal is also exact in decimal, so the kinds of problems we saw in this exercise are likely to be severe only when you use a relatively unusual radix like 3.

2.7.9 Modes Tutorial Exercise 2

If the radix is 15 or higher, we can't use the letter 'e' to mark the exponent because 'e' is interpreted as a digit. When Calc needs to display scientific notation in a high radix, it writes '16#F.E8F*16.^15'. You can enter a number like this as an algebraic entry. Also, pressing *e* without any digits before it normally types *1e*, but in a high radix it types *16.^* and puts you in algebraic entry: *16#f.e8f RET e 15 RET ** is another way to enter this number.

The reason Calc puts a decimal point in the '16.^' is to prevent huge integers from being generated if the exponent is large (consider '16#1.23*16^1000', where we compute '16^1000' as a giant exact integer and then throw away most of the digits when we multiply it by the floating-point '16#1.23'). While this wouldn't normally matter for display purposes, it could give you a nasty surprise if you copied that number into a file and later moved it back into Calc.

2.7.10 Modes Tutorial Exercise 3

The answer he got was 0.5000000000006399.

The problem is not that the square operation is inexact, but that the sine of 45 that was already on the stack was accurate to only 12 places. Arbitrary-precision calculations still only give answers as good as their inputs.

The real problem is that there is no 12-digit number which, when squared, comes out to 0.5 exactly. The *f [* and *f]* commands decrease or increase a number by one unit in the last place (according to the current precision). They are useful for determining facts like this.

```
1:  0.707106781187      1:  0.500000000001
    .                       .

    45 S                    2 ^

1:  0.707106781187      1:  0.707106781186      1:  0.499999999999
    .                       .                       .

    U  DEL                  f [                     2 ^
```

A high-precision calculation must be carried out in high precision all the way. The only number in the original problem which was known exactly was the quantity 45 degrees, so the precision must be raised before anything is done after the number 45 has been entered in order for the higher precision to be meaningful.

2.7.11 Modes Tutorial Exercise 4

Many calculations involve real-world quantities, like the width and height of a piece of wood or the volume of a jar. Such quantities can't be measured exactly anyway, and if the data that is input to a calculation is inexact, doing exact arithmetic on it is a waste of time.

Fractions become unwieldy after too many calculations have been done with them. For example, the sum of the reciprocals of the integers from 1 to 10 is 7381:2520. The sum from 1 to 30 is 9304682830147:2329089562800. After a point it will take a long time to add even one more term to this sum, but a floating-point calculation of the sum will not have this problem.

Also, rational numbers cannot express the results of all calculations. There is no fractional form for the square root of two, so if you type `2 Q`, Calc has no choice but to give you a floating-point answer.

2.7.12 Arithmetic Tutorial Exercise 1

Dividing two integers that are larger than the current precision may give a floating-point result that is inaccurate even when rounded down to an integer. Consider 123456789/2 when the current precision is 6 digits. The true answer is 61728394.5, but with a precision of 6 this will be rounded to 12345700.0/2.0 = 61728500.0. The result, when converted to an integer, will be off by 106.

Here are two solutions: Raise the precision enough that the floating-point round-off error is strictly to the right of the decimal point. Or, convert to fraction mode so that 123456789/2 produces the exact fraction 123456789:2, which can be rounded down by the `F` command without ever switching to floating-point format.

2.7.13 Arithmetic Tutorial Exercise 2

27 RET 9 B could give the exact result 3:2, but it does a floating-point calculation instead and produces 1.5.

Calc will find an exact result for a logarithm if the result is an integer or the reciprocal of an integer. But there is no efficient way to search the space of all possible rational numbers for an exact answer, so Calc doesn't try.

2.7.14 Vector Tutorial Exercise 1

Duplicate the vector, compute its length, then divide the vector by its length: *RET A /*.

```
1:  [1, 2, 3]  2:  [1, 2, 3]      1:  [0.27, 0.53, 0.80]  1:  1.
    .          1:  3.74165738677      .                       .
                   .

    r 1            RET A              /                       A
```

The final *A* command shows that the normalized vector does indeed have unit length.

2.7.15 Vector Tutorial Exercise 2

The average position is equal to the sum of the products of the positions times their corresponding probabilities. This is the definition of the dot product operation. So all you need to do is to put the two vectors on the stack and press *.

2.7.16 Matrix Tutorial Exercise 1

The trick is to multiply by a vector of ones. Use *r 4 [1 1 1] ** to get the row sum. Similarly, use *[1 1] r 4 ** to get the column sum.

2.7.17 Matrix Tutorial Exercise 2

$$\begin{aligned} x + ay &= 6 \\ x + by &= 10 \end{aligned}$$

Just enter the righthand side vector, then divide by the lefthand side matrix as usual.

```
1:  [6, 10]     2:  [6, 10]          1:  [6 - 4 a / (b - a), 4 / (b - a) ]
    .           1:  [ [ 1, a ]           .
                      [ 1, b ] ]
                    .

 ' [6 10] RET      ' [1 a; 1 b] RET       /
```

This can be made more readable using `d B` to enable "big" display mode:

```
          4 a     4
1:  [6 - -----, -----]
         b - a  b - a
```

Type `d N` to return to "normal" display mode afterwards.

2.7.18 Matrix Tutorial Exercise 3

To solve $A^T A X = A^T B$, first we compute $A' = A^T A$ and $B' = A^T B$; now, we have a system $A'X = B'$ which we can solve using Calc's '`/`' command.

$$\begin{aligned} a + 2b + 3c &= 6 \\ 4a + 5b + 6c &= 2 \\ 7a + 6b &= 3 \\ 2a + 4b + 6c &= 11 \end{aligned}$$

The first step is to enter the coefficient matrix. We'll store it in quick variable number 7 for later reference. Next, we compute the B' vector.

```
1:  [ [ 1, 2, 3 ]              2:  [ [ 1, 4, 7, 2 ]     1:  [57, 84, 96]
      [ 4, 5, 6 ]                    [ 2, 5, 6, 4 ]         .
      [ 7, 6, 0 ]                    [ 3, 6, 0, 6 ] ]
      [ 2, 4, 6 ] ]            1:  [6, 2, 3, 11]
    .                              .

 ' [1 2 3; 4 5 6; 7 6 0; 2 4 6] RET  s 7  v t  [6 2 3 11]   *
```

Now we compute the matrix A' and divide.

```
2:  [57, 84, 96]          1:  [-11.64, 14.08, -3.64]
1:  [ [ 70, 72, 39 ]          .
      [ 72, 81, 60 ]
      [ 39, 60, 81 ] ]
    .

    r 7 v t r 7 *             /
```

(The actual computed answer will be slightly inexact due to round-off error.)

Notice that the answers are similar to those for the 3×3 system solved in the text. That's because the fourth equation that was added to the system is almost identical to the first one multiplied by two. (If it were identical, we would have gotten the exact same answer since the 4×3 system would be equivalent to the original 3×3 system.)

Since the first and fourth equations aren't quite equivalent, they can't both be satisfied at once. Let's plug our answers back into the original system of equations to see how well they match.

```
2:  [-11.64, 14.08, -3.64]     1:  [5.6, 2., 3., 11.2]
1:  [ [ 1, 2, 3 ]                  .
      [ 4, 5, 6 ]
      [ 7, 6, 0 ]
      [ 2, 4, 6 ] ]
    .

    r 7                            TAB *
```

This is reasonably close to our original B vector, $[6, 2, 3, 11]$.

2.7.19 List Tutorial Exercise 1

We can use *v x* to build a vector of integers. This needs to be adjusted to get the range of integers we desire. Mapping '-' across the vector will accomplish this, although it turns out the plain '-' key will work just as well.

```
2:  2                              2:  2
1:  [1, 2, 3, 4, 5, 6, 7, 8, 9]    1:  [-4, -3, -2, -1, 0, 1, 2, 3, 4]
    .                                  .

    2  v x 9 RET                       5 V M -   or   5 -
```

Now we use *V M ^* to map the exponentiation operator across the vector.

```
1:  [0.0625, 0.125, 0.25, 0.5, 1, 2, 4, 8, 16]
    .

    V M ^
```

2.7.20 List Tutorial Exercise 2

Given x and y vectors in quick variables 1 and 2 as before, the first job is to form the matrix that describes the problem.

$$m \times x + b \times 1 = y$$

Thus we want a 19×2 matrix with our x vector as one column and ones as the other column. So, first we build the column of ones, then we combine the two columns to form our A matrix.

```
2:  [1.34, 1.41, 1.49, ... ]    1:  [ [ 1.34, 1 ]
1:  [1, 1, 1, ...]                  [ 1.41, 1 ]
    .                               [ 1.49, 1 ]
                                    ...

    r 1 1 v b 19 RET               M-2 v p v t   s 3
```

Now we compute $A^T y$ and $A^T A$ and divide.

```
1:  [33.36554, 13.613]    2:  [33.36554, 13.613]
    .                     1:  [ [ 98.0003, 41.63 ]
                                [  41.63,   19   ] ]
                              .

 v t r 2 *                    r 3 v t r 3 *
```

(Hey, those numbers look familiar!)

```
1:  [0.52141679, -0.425978]
    .

    /
```

Since we were solving equations of the form $m \times x + b \times 1 = y$, these numbers should be m and b, respectively. Sure enough, they agree exactly with the result computed using `V M` and `V R`!

The moral of this story: `V M` and `V R` will probably solve your problem, but there is often an easier way using the higher-level arithmetic functions!

In fact, there is a built-in `a F` command that does least-squares fits. See *XXX* [Curve Fitting], page *XXX*.

2.7.21 List Tutorial Exercise 3

Move to one end of the list and press `C-@` (or `C-SPC` or whatever) to set the mark, then move to the other end of the list and type `M-# g`.

```
1:  [2.3, 6, 22, 15.1, 7, 15, 14, 7.5, 2.5]
    .
```

To make things interesting, let's assume we don't know at a glance how many numbers are in this list. Then we could type:

```
2:  [2.3, 6, 22, ... ]     2:  [2.3, 6, 22, ... ]
1:  [2.3, 6, 22, ... ]     1:  126356422.5
    .                          .

    RET                        V R *

2:  126356422.5            2:  126356422.5     1:  7.94652913734
1:  [2.3, 6, 22, ... ]     1:  9                   .
    .                          .

    TAB                        v l                 I ^
```

(The *I ^* command computes the nth root of a number. You could also type *& ^* to take the reciprocal of 9 and then raise the number to that power.)

2.7.22 List Tutorial Exercise 4

A number j is a divisor of n if $n \% j = 0$. The first step is to get a vector that identifies the divisors.

```
2:  30                   2:  [0, 0, 0, 2, ...]    1:  [1, 1, 1, 0, ...]
1:  [1, 2, 3, 4, ...]    1:  0                        .
    .                        .

 30 RET v x 30 RET   s 1    V M % 0                V M a =  s 2
```

This vector has 1's marking divisors of 30 and 0's marking non-divisors.

The zeroth divisor function is just the total number of divisors. The first divisor function is the sum of the divisors.

```
1:  8      3:  8                    2:  8                    2:  8
           2:  [1, 2, 3, 4, ...]    1:  [1, 2, 3, 0, ...]    1:  72
           1:  [1, 1, 1, 0, ...]        .                        .
               .

   V R +       r 1 r 2                  V M *                    V R +
```

Once again, the last two steps just compute a dot product for which a simple `*` would have worked equally well.

2.7.23 List Tutorial Exercise 5

The obvious first step is to obtain the list of factors with *k f*. This list will always be in sorted order, so if there are duplicates they will be right next to each other. A suitable method is to compare the list with a copy of itself shifted over by one.

```
1:  [3, 7, 7, 7, 19]   2:  [3, 7, 7, 7, 19]     2:  [3, 7, 7, 7, 19, 0]
    .                  1:  [3, 7, 7, 7, 19, 0]  1:  [0, 3, 7, 7, 7, 19]
                           .                        .

    19551 k f              RET 0 |                  TAB 0 TAB |


1:  [0, 0, 1, 1, 0, 0]   1:  2           1:  0
    .                        .               .

    V M a =                  V R +           0 a =
```

Note that we have to arrange for both vectors to have the same length so that the mapping operation works; no prime factor will ever be zero, so adding zeros on the left and right is safe. From then on the job is pretty straightforward.

Incidentally, Calc provides the *Möbius* μ function which is zero if and only if its argument is square-free. It would be a much more convenient way to do the above test in practice.

2.7.24 List Tutorial Exercise 6

First use *v x 6 RET* to get a list of integers, then *V M v x* to get a list of lists of integers!

2.7.25 List Tutorial Exercise 7

Here's one solution. First, compute the triangular list from the previous exercise and type *1 -* to subtract one from all the elements.

```
1:  [ [0],
      [0, 1],
      [0, 1, 2],
      ...

    1 -
```

The numbers down the lefthand edge of the list we desire are called the "triangular numbers" (now you know why!). The nth triangular number is the sum of the integers from 1 to n, and can be computed directly by the formula $\frac{n(n+1)}{2}$.

```
2:  [ [0], [0, 1], ... ]    2:  [ [0], [0, 1], ... ]
1:  [0, 1, 2, 3, 4, 5]      1:  [0, 1, 3, 6, 10, 15]
    .                           .

    v x 6 RET 1 -               V M ' $ ($+1)/2 RET
```

Adding this list to the above list of lists produces the desired result:

```
1:  [ [0],
      [1, 2],
      [3, 4, 5],
      [6, 7, 8, 9],
      [10, 11, 12, 13, 14],
      [15, 16, 17, 18, 19, 20] ]
      .

      V M +
```

If we did not know the formula for triangular numbers, we could have computed them using a *V U +* command. We could also have gotten them the hard way by mapping a reduction across the original triangular list.

```
2:  [ [0], [0, 1], ... ]    2:  [ [0], [0, 1], ... ]
1:  [ [0], [0, 1], ... ]    1:  [0, 1, 3, 6, 10, 15]
    .                           .

    RET                         V M V R +
```

(This means "map a *V R +* command across the vector," and since each element of the main vector is itself a small vector, *V R +* computes the sum of its elements.)

2.7.26 List Tutorial Exercise 8

The first step is to build a list of values of x.

```
1:  [1, 2, 3, ..., 21]  1:  [0, 1, 2, ..., 20]  1:  [0, 0.25, 0.5, ..., 5]
    .                       .                       .

    v x 21 RET              1 -                     4 /  s 1
```

Next, we compute the Bessel function values.

```
1:  [0., 0.124, 0.242, ..., -0.328]
    .

    V M ' besJ(1,$) RET
```

(Another way to do this would be *1 TAB V M f j*.)

A way to isolate the maximum value is to compute the maximum using *V R X*, then compare all the Bessel values with that maximum.

```
2:  [0., 0.124, 0.242, ... ]   1:  [0, 0, 0, ... ]    2:  [0, 0, 0, ... ]
1:  0.5801562                      .                  1:  1
    .                                                     .

    RET V R X                      V M a =                RET V R +    DEL
```

It's a good idea to verify, as in the last step above, that only one value is equal to the maximum. (After all, a plot of $\sin x$ might have many points all equal to the maximum value, 1.)

The vector we have now has a single 1 in the position that indicates the maximum value of x. Now it is a simple matter to convert this back into the corresponding value itself.

```
2:  [0, 0, 0, ... ]         1:  [0, 0., 0., ... ]    1:  1.75
1:  [0, 0.25, 0.5, ... ]        .                        .
    .

    r 1                         V M *                    V R +
```

If `a =` had produced more than one 1 value, this method would have given the sum of all maximum x values; not very useful! In this case we could have used *v m* (`calc-mask-vector`) instead. This command deletes all elements of a "data" vector that correspond to zeros in a "mask" vector, leaving us with, in this example, a vector of maximum x values.

The built-in `a` *X* command maximizes a function using more efficient methods. Just for illustration, let's use `a` *X* to maximize '`besJ(1,x)`' over this same interval.

```
2:  besJ(1, x)                 1:  [1.84115, 0.581865]
1:  [0 .. 5]                       .
    .

' besJ(1,x), [0..5] RET            a X x RET
```

The output from `a` *X* is a vector containing the value of x that maximizes the function, and the function's value at that maximum. As you can see, our simple search got quite close to the right answer.

2.7.27 List Tutorial Exercise 9

Step one is to convert our integer into vector notation.

```
1:  25129925999           3:  25129925999
    .                     2:  10
                          1:  [11, 10, 9, ..., 1, 0]
                              .

    25129925999 RET           10 RET 12 RET v x 12 RET -


1:  25129925999              1:  [0, 2, 25, 251, 2512, ... ]
2:  [100000000000, ... ]         .
    .

    V M ^  s 1                   V M \
```

(Recall, the \ command computes an integer quotient.)

```
1:  [0, 2, 5, 1, 2, 9, 9, 2, 5, 9, 9, 9]
    .

    10 V M %  s 2
```

Next we must increment this number. This involves adding one to the last digit, plus handling carries. There is a carry to the left out of a digit if that digit is a nine and all the digits to the right of it are nines.

```
1:  [0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1]   1:  [1, 1, 1, 0, 0, 1, ... ]
    .                                          .

    9 V M a =                                  v v


1:  [1, 1, 1, 0, 0, 0, ... ]   1:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
    .                              .

    V U *                          v v 1 |
```

Accumulating * across a vector of ones and zeros will preserve only the initial run of ones. These are the carries into all digits except the rightmost digit. Concatenating a one on the right takes care of aligning the carries properly, and also adding one to the rightmost digit.

```
2:  [0, 0, 0, 0, ... ]      1:  [0, 0, 2, 5, 1, 2, 9, 9, 2, 6, 0, 0, 0]
1:  [0, 0, 2, 5, ... ]          .
    .

    0 r 2 |                     V M +  10 V M %
```

Here we have concatenated 0 to the *left* of the original number; this takes care of shifting the carries by one with respect to the digits that generated them.

Finally, we must convert this list back into an integer.

```
3:  [0, 0, 2, 5, ... ]        2:  [0, 0, 2, 5, ... ]
2:  1000000000000             1:  [1000000000000, 100000000000, ... ]
1:  [100000000000, ... ]          .
    .

    10 RET 12 ^  r 1              |


1:  [0, 0, 20000000000, 5000000000, ... ]    1:  25129926000
    .                                            .

    V M *                                        V R +
```

Another way to do this final step would be to reduce the formula '`10 $$ + $`' across the vector of digits.

```
1:  [0, 0, 2, 5, ... ]        1:  25129926000
    .                             .

                                  V R ' 10 $$ + $ RET
```

2.7.28 List Tutorial Exercise 10

For the list $[a, b, c, d]$, the result is $((a = b) = c) = d$, which will compare a and b to produce a 1 or 0, which is then compared with c to produce another 1 or 0, which is then compared with d. This is not at all what Joe wanted.

Here's a more correct method:

```
1:  [7, 7, 7, 8, 7]       2:  [7, 7, 7, 8, 7]
    .                     1:  7
                              .

  ' [7,7,7,8,7] RET           RET v r 1 RET


1:  [1, 1, 1, 0, 1]       1:  0
    .                         .

    V M a =                   V R *
```

2.7.29 List Tutorial Exercise 11

The circle of unit radius consists of those points (x, y) for which $x^2 + y^2 < 1$. We start by generating a vector of x^2 and a vector of y^2.

We can make this go a bit faster by using the `v .` and `t .` commands.

```
2:  [2., 2., ..., 2.]         2:  [2., 2., ..., 2.]
1:  [2., 2., ..., 2.]         1:  [1.16, 1.98, ..., 0.81]
    .                             .

 v . t .  2. v b 100 RET RET      V M k r


2:  [2., 2., ..., 2.]         1:  [0.026, 0.96, ..., 0.036]
1:  [0.026, 0.96, ..., 0.036] 2:  [0.53, 0.81, ..., 0.094]
    .                             .

    1 -  2 V M ^                  TAB  V M k r  1 -  2 V M ^
```

Now we sum the x^2 and y^2 values, compare with 1 to get a vector of 1/0 truth values, then sum the truth values.

```
1:  [0.56, 1.78, ..., 0.13]   1:  [1, 0, ..., 1]    1:  84
    .                             .                     .

    +                             1 V M a <             V R +
```

The ratio 84/100 should approximate the ratio $\pi/4$.

```
1:  0.84         1:  3.36         2:  3.36         1:  1.0695
    .                .            1:  3.14159          .

    100 /            4 *              P                /
```

Our estimate, 3.36, is off by about 7%. We could get a better estimate by taking more points (say, 1000), but it's clear that this method is not very efficient!

(Naturally, since this example uses random numbers your own answer will be slightly different from the one shown here!)

If you typed `v .` and `t .` before, type them again to return to full-sized display of vectors.

2.7.30 List Tutorial Exercise 12

This problem can be made a lot easier by taking advantage of some symmetries. First of all, after some thought it's clear that the y axis can be ignored altogether. Just pick a random x component for one end of the match, pick a random direction θ, and see if x and $x + \cos\theta$ (which is the x coordinate of the other endpoint) cross a line. The lines are at integer coordinates, so this happens when the two numbers surround an integer.

Since the two endpoints are equivalent, we may as well choose the leftmost of the two endpoints as x. Then *theta* is an angle pointing to the right, in the range -90 to 90 degrees. (We could use radians, but it would feel like cheating to refer to $\pi/2$ radians while trying to estimate π!)

In fact, since the field of lines is infinite we can choose the coordinates 0 and 1 for the lines on either side of the leftmost endpoint. The rightmost endpoint will be between 0 and 1 if the match does not cross a line, or between 1 and 2 if it does. So: Pick random x and θ, compute $x + \cos\theta$, and count how many of the results are greater than one. Simple!

We can make this go a bit faster by using the `v .` and `t .` commands.

```
1:  [0.52, 0.71, ..., 0.72]    2:  [0.52, 0.71, ..., 0.72]
    .                          1:  [78.4, 64.5, ..., -42.9]
                                   .

v . t . 1. v b 100 RET  V M k r    180. v b 100 RET  V M k r  90 -
```

(The next step may be slow, depending on the speed of your computer.)

```
2:  [0.52, 0.71, ..., 0.72]    1:  [0.72, 1.14, ..., 1.45]
1:  [0.20, 0.43, ..., 0.73]        .
    .

    m d  V M C                     +


1:  [0, 1, ..., 1]       1:  0.64            1:  3.125
    .                        .                   .

    1 V M a >                V R + 100 /         2 TAB /
```

Let's try the third method, too. We'll use random integers up to one million. The *k r* command with an integer argument picks a random integer.

```
2:  [1000000, 1000000, ..., 1000000]   2:  [78489, 527587, ..., 814975]
1:  [1000000, 1000000, ..., 1000000]   1:  [324014, 358783, ..., 955450]
    .                                      .

    1000000 v b 100 RET RET                V M k r  TAB  V M k r


1:  [1, 1, ..., 25]      1:  [1, 1, ..., 0]     1:  0.56
    .                        .                      .

    V M k g                  1 V M a =              V R + 100 /


1:  10.714        1:  3.273
    .                 .

    6 TAB /           Q
```

For a proof of this property of the GCD function, see section 4.5.2, exercise 10, of Knuth's *Art of Computer Programming*, volume II.

If you typed `v .` and `t .` before, type them again to return to full-sized display of vectors.

2.7.31 List Tutorial Exercise 13

First, we put the string on the stack as a vector of ASCII codes.

```
1:  [84, 101, 115, ..., 51]
    .

    "Testing, 1, 2, 3 RET
```

Note that the " key, like *$*, initiates algebraic entry so there was no need to type an apostrophe. Also, Calc didn't mind that we omitted the closing ". (The same goes for all closing delimiters like *)* and *]* at the end of a formula.

We'll show two different approaches here. In the first, we note that if the input vector is $[a, b, c, d]$, then the hash code is $3(3(3a + b) + c) + d = 27a + 9b + 3c + d$. In other words, it's a sum of descending powers of three times the ASCII codes.

```
2:  [84, 101, 115, ..., 51]    2:  [84, 101, 115, ..., 51]
1:  16                         1:  [15, 14, 13, ..., 0]
    .                              .

    RET v l                        v x 16 RET -


2:  [84, 101, 115, ..., 51]    1:  1960915098    1:  121
1:  [14348907, ..., 1]             .                 .
    .

    3 TAB V M ^                    *                 511 %
```

Once again, `*` elegantly summarizes most of the computation. But there's an even more elegant approach: Reduce the formula *3 $$ + $* across the vector. Recall that this represents a function of two arguments that computes its first argument times three plus its second argument.

```
1:  [84, 101, 115, ..., 51]    1:  1960915098
    .                              .

    "Testing, 1, 2, 3 RET          V R ' 3$$+$ RET
```

If you did the decimal arithmetic exercise, this will be familiar. Basically, we're turning a base-3 vector of digits into an integer, except that our "digits" are much larger than real digits.

Instead of typing *511 %* again to reduce the result, we can be cleverer still and notice that rather than computing a huge integer and taking the modulo at the end, we can take the modulo at each step without affecting the result. While this means there are more arithmetic operations, the numbers we operate on remain small so the operations are faster.

```
1:  [84, 101, 115, ..., 51]    1:  121
    .                              .

    "Testing, 1, 2, 3 RET          V R ' (3$$+$)%511 RET
```

Why does this work? Think about a two-step computation: $3(3a + b) + c$. Taking a result modulo 511 basically means subtracting off enough 511's to put the result in the desired range. So the result when we take the modulo after every step is,

$$3(3a + b - 511m) + c - 511n$$

for some suitable integers m and n. Expanding out by the distributive law yields

$$9a + 3b + c - 511 \times 3m - 511n$$

The m term in the latter formula is redundant because any contribution it makes could just as easily be made by the n term. So we can take it out to get an equivalent formula with $n' = 3m + n$,

$$9a + 3b + c - 511n'$$

which is just the formula for taking the modulo only at the end of the calculation. Therefore the two methods are essentially the same.

Later in the tutorial we will encounter *modulo forms*, which basically automate the idea of reducing every intermediate result modulo some value M.

2.7.32 List Tutorial Exercise 14

We want to use *H V U* to nest a function which adds a random step to an (x, y) coordinate. The function is a bit long, but otherwise the problem is quite straightforward.

```
2:  [0, 0]     1:  [ [    0,       0    ]
1:  50               [  0.4288, -0.1695 ]
    .                [ -0.4787, -0.9027 ]
                     ...

    [0,0] 50       H V U ' <# + [random(2.0)-1, random(2.0)-1]> RET
```

Just as the text recommended, we used '`< >`' nameless function notation to keep the two `random` calls from being evaluated before nesting even begins.

We now have a vector of $[x, y]$ sub-vectors, which by Calc's rules acts like a matrix. We can transpose this matrix and unpack to get a pair of vectors, x and y, suitable for graphing.

```
2:  [ 0, 0.4288, -0.4787, ... ]
1:  [ 0, -0.1696, -0.9027, ... ]
    .

    v t  v u  g f
```

Incidentally, because the x and y are completely independent in this case, we could have done two separate commands to create our x and y vectors of numbers directly.

To make a random walk of unit steps, we note that `sincos` of a random direction exactly gives us an $[x, y]$ step of unit length; in fact, the new nesting function is even briefer, though we might want to lower the precision a bit for it.

```
2:  [0, 0]     1:  [ [    0,      0    ]
1:  50               [  0.1318, 0.9912 ]
    .                [ -0.5965, 0.3061 ]
                     ...

    [0,0] 50   m d  p 6 RET   H V U ' <# + sincos(random(360.0))> RET
```

Another *v t v u g f* sequence will graph this new random walk.

An interesting twist on these random walk functions would be to use complex numbers instead of 2-vectors to represent points on the plane. In the first example, we'd use something like '`random + random*(0,1)`', and in the second we could use polar complex numbers with random phase angles. (This exercise was first suggested in this form by Randal Schwartz.)

2.7.33 Types Tutorial Exercise 1

If the number is the square root of π times a rational number, then its square, divided by π, should be a rational number.

```
1:  1.26508260337     1:  0.509433962268    1:  2486645810:4881193627
    .                     .                     .

                          2 ^ P /               c F
```

Technically speaking this is a rational number, but not one that is likely to have arisen in the original problem. More likely, it just happens to be the fraction which most closely represents some irrational number to within 12 digits.

But perhaps our result was not quite exact. Let's reduce the precision slightly and try again:

```
1:  0.509433962268      1:  27:53
    .                       .

    U p 10 RET              c F
```

Aha! It's unlikely that an irrational number would equal a fraction this simple to within ten digits, so our original number was probably $\sqrt{27\pi/53}$.

Notice that we didn't need to re-round the number when we reduced the precision. Remember, arithmetic operations always round their inputs to the current precision before they begin.

2.7.34 Types Tutorial Exercise 2

'`inf / inf = nan`'. Perhaps '`1`' is the "obvious" answer. But if '`17 inf = inf`', then '`17 inf / inf = inf / inf = 17`', too.

'`exp(inf) = inf`'. It's tempting to say that the exponential of infinity must be "bigger" than "regular" infinity, but as far as Calc is concerned all infinities are as just as big. In other words, as x goes to infinity, e^x also goes to infinity, but the fact the e^x grows much faster than x is not relevant here.

'`exp(-inf) = 0`'. Here we have a finite answer even though the input is infinite.

'`sqrt(-inf) = (0, 1) inf`'. Remember that $(0, 1)$ represents the imaginary number i. Here's a derivation: '`sqrt(-inf) = sqrt((-1) * inf) = sqrt(-1) * sqrt(inf)`'. The first part is, by definition, i; the second is `inf` because, once again, all infinities are the same size.

'`sqrt(uinf) = uinf`'. In fact, we do know something about the direction because `sqrt` is defined to return a value in the right half of the complex plane. But Calc has no notation for this, so it settles for the conservative answer `uinf`.

'`abs(uinf) = inf`'. No matter which direction x points, '`abs(x)`' always points along the positive real axis.

'`ln(0) = -inf`'. Here we have an infinite answer to a finite input. As in the $1/0$ case, Calc will only use infinities here if you have turned on "infinite" mode. Otherwise, it will treat '`ln(0)`' as an error.

2.7.35 Types Tutorial Exercise 3

We can make '`inf - inf`' be any real number we like, say, *a*, just by claiming that we added *a* to the first infinity but not to the second. This is just as true for complex values of *a*, so `nan` can stand for a complex number. (And, similarly, `uinf` can stand for an infinity that points in any direction in the complex plane, such as '`(0, 1) inf`').

In fact, we can multiply the first `inf` by two. Surely '`2 inf - inf = inf`', but also '`2 inf - inf = inf - inf = nan`'. So `nan` can even stand for infinity. Obviously it's just as easy to make it stand for minus infinity as for plus infinity.

The moral of this story is that "infinity" is a slippery fish indeed, and Calc tries to handle it by having a very simple model for infinities (only the direction counts, not the "size"); but Calc is careful to write `nan` any time this simple model is unable to tell what the true answer is.

2.7.36 Types Tutorial Exercise 4

```
2:  0@ 47' 26"              1:  0@ 2' 47.411765"
1:  17                          .
    .

    0@ 47' 26" RET 17           /
```
The average song length is two minutes and 47.4 seconds.

```
2:  0@ 2' 47.411765"     1:  0@ 3' 7.411765"     1:  0@ 53' 6.000005"
1:  0@ 0' 20"                .                       .
    .

    20"                      +                       17 *
```
The album would be 53 minutes and 6 seconds long.

2.7.37 Types Tutorial Exercise 5

Let's suppose it's January 14, 1991. The easiest thing to do is to keep trying 13ths of months until Calc reports a Friday. We can do this by manually entering dates, or by using `t I`:

```
1:  <Wed Feb 13, 1991>    1:  <Wed Mar 13, 1991>    1:  <Sat Apr 13, 1991>
    .                         .                         .

    ' <2/13> RET        DEL    ' <3/13> RET             t I
```
(Calc assumes the current year if you don't say otherwise.)

This is getting tedious—we can keep advancing the date by typing `t I` over and over again, but let's automate the job by using vector mapping. The `t I` command actually takes a second "how-many-months" argument, which defaults to one. This argument is exactly what we want to map over:

```
2:  <Sat Apr 13, 1991>     1:  [<Mon May 13, 1991>, <Thu Jun 13, 1991>,
1:  [1, 2, 3, 4, 5, 6]          <Sat Jul 13, 1991>, <Tue Aug 13, 1991>,
    .                           <Fri Sep 13, 1991>, <Sun Oct 13, 1991>]
                                .

    v x 6 RET                  V M t I
```

Et voilà, September 13, 1991 is a Friday.

```
1:  242
    .

  ' <sep 13> - <jan 14> RET
```

And the answer to our original question: 242 days to go.

2.7.38 Types Tutorial Exercise 6

The full rule for leap years is that they occur in every year divisible by four, except that they don't occur in years divisible by 100, except that they *do* in years divisible by 400. We could work out the answer by carefully counting the years divisible by four and the exceptions, but there is a much simpler way that works even if we don't know the leap year rule.

Let's assume the present year is 1991. Years have 365 days, except that leap years (whenever they occur) have 366 days. So let's count the number of days between now and then, and compare that to the number of years times 365. The number of extra days we find must be equal to the number of leap years there were.

```
1:  <Mon Jan 1, 10001>     2:  <Mon Jan 1, 10001>     1:  2925593
    .                      1:  <Tue Jan 1, 1991>          .
                               .

  ' <jan 1 10001> RET        ' <jan 1 1991> RET           -


3:  2925593       2:  2925593     2:  2925593     1:  1943
2:  10001         1:  8010        1:  2923650         .
1:  1991              .               .
    .

  10001 RET 1991      -               365 *           -
```

There will be 1943 leap years before the year 10001. (Assuming, of course, that the algorithm for computing leap years remains unchanged for that long. See *XXX* [Date Forms], page *XXX*, for some interesting background information in that regard.)

2.7.39 Types Tutorial Exercise 7

The relative errors must be converted to absolute errors so that '+/-' notation may be used.

```
1:  1.              2:  1.
    .               1:  0.2
                        .

    20 RET .05 *        4 RET .05 *
```

Now we simply chug through the formula.

```
1:  19.7392088022    1:  394.78 +/- 19.739    1:  6316.5 +/- 706.21
    .                    .                        .

    2 P 2 ^ *            20 p 1 *                 4 p .2 RET 2 ^ *
```

It turns out the *v u* command will unpack an error form as well as a vector. This saves us some retyping of numbers.

```
3:  6316.5 +/- 706.21     2:  6316.5 +/- 706.21
2:  6316.5                1:  0.1118
1:  706.21                    .
    .

    RET v u                   TAB /
```

Thus the volume is 6316 cubic centimeters, within about 11 percent.

2.7.40 Types Tutorial Exercise 8

The first answer is pretty simple: '`1 / (0 .. 10) = (0.1 .. inf)`'. Since a number in the interval '`(0 .. 10)`' can get arbitrarily close to zero, its reciprocal can get arbitrarily large, so the answer is an interval that effectively means, "any number greater than 0.1" but with no upper bound.

The second answer, similarly, is '`1 / (-10 .. 0) = (-inf .. -0.1)`'.

Calc normally treats division by zero as an error, so that the formula '`1 / 0`' is left unsimplified. Our third problem, '`1 / [0 .. 10]`', also (potentially) divides by zero because zero is now a member of the interval. So Calc leaves this one unevaluated, too.

If you turn on "infinite" mode by pressing *m i*, you will instead get the answer '`[0.1 .. inf]`', which includes infinity as a possible value.

The fourth calculation, '`1 / (-10 .. 10)`', has the same problem. Zero is buried inside the interval, but it's still a possible value. It's not hard to see that the actual result of '`1 / (-10 .. 10)`' will be either greater than 0.1, or less than −0.1. Thus the interval goes from minus infinity to plus infinity, with a "hole" in it from −0.1 to 0.1. Calc doesn't have any way to represent this, so it just reports '`[-inf .. inf]`' as the answer. It may be disappointing to hear "the answer lies somewhere between minus infinity and plus infinity, inclusive," but that's the best that interval arithmetic can do in this case.

2.7.41 Types Tutorial Exercise 9

```
1:  [-3 .. 3]       2:  [-3 .. 3]      2:  [0 .. 9]
    .               1:  [0 .. 9]       1:  [-9 .. 9]
                        .                  .

    [ 3 n .. 3 ]        RET 2 ^            TAB RET *
```

In the first case the result says, "if a number is between −3 and 3, its square is between 0 and 9." The second case says, "the product of two numbers each between −3 and 3 is between −9 and 9."

An interval form is not a number; it is a symbol that can stand for many different numbers. Two identical-looking interval forms can stand for different numbers.

The same issue arises when you try to square an error form.

2.7.42 Types Tutorial Exercise 10

Testing the first number, we might arbitrarily choose 17 for x.

```
1:  17 mod 811749613   2:  17 mod 811749613   1:  533694123 mod 811749613
    .                      811749612              .
                           .

    17 M 811749613 RET     811749612              ^
```

Since 533694123 is (considerably) different from 1, the number 811749613 must not be prime.

It's awkward to type the number in twice as we did above. There are various ways to avoid this, and algebraic entry is one. In fact, using a vector mapping operation we can perform several tests at once. Let's use this method to test the second number.

```
2:  [17, 42, 100000]             1:  [1 mod 15485863, 1 mod ... ]
1:  15485863                         .
    .

   [17 42 100000] 15485863 RET           V M ' ($$ mod $)^($-1) RET
```

The result is three ones (modulo n), so it's very probable that 15485863 is prime. (In fact, this number is the millionth prime.)

Note that the functions '`($$^($-1)) mod $`' or '`$$^($-1) % $`' would have been hopelessly inefficient, since they would have calculated the power using full integer arithmetic.

Calc has a ***k p*** command that does primality testing. For small numbers it does an exact test; for large numbers it uses a variant of the Fermat test we used here. You can use ***k p*** repeatedly to prove that a large integer is prime with any desired probability.

2.7.43 Types Tutorial Exercise 11

There are several ways to insert a calculated number into an HMS form. One way to convert a number of seconds to an HMS form is simply to multiply the number by an HMS form representing one second:

```
1:  31415926.5359     2:  31415926.5359     1:  8726@ 38' 46.5359"
    .                 1:  0@ 0' 1"              .
                          .

    P 1e7 *               0@ 0' 1"              *


2:  8726@ 38' 46.5359"             1:  6@ 6' 2.5359" mod 24@ 0' 0"
1:  15@ 27' 16" mod 24@ 0' 0"          .
    .

    x time RET                         +
```

It will be just after six in the morning.

The algebraic hms function can also be used to build an HMS form:

```
1:  hms(0, 0, 10000000. pi)       1:  8726@ 38' 46.5359"
    .                                 .

  ' hms(0, 0, 1e7 pi) RET             =
```

The = key is necessary to evaluate the symbol 'pi' to the actual number 3.14159...

2.7.44 Types Tutorial Exercise 12

As we recall, there are 17 songs of about 2 minutes and 47 seconds each.

```
2:  0@ 2' 47"                     1:  [0@ 3' 7" .. 0@ 3' 47"]
1:  [0@ 0' 20" .. 0@ 1' 0"]           .
    .

    [ 0@ 20" .. 0@ 1' ]               +


1:  [0@ 52' 59." .. 1@ 4' 19."]
    .

    17 *
```

No matter how long it is, the album will fit nicely on one CD.

2.7.45 Types Tutorial Exercise 13

Type *' 1 yr RET u c s RET*. The answer is 31557600 seconds.

2.7.46 Types Tutorial Exercise 14

How long will it take for a signal to get from one end of the computer to the other?

```
1:  m / c          1:  3.3356 ns
    .                  .

  ' 1 m / c RET        u c ns RET
```

(Recall, 'c' is a "unit" corresponding to the speed of light.)

```
1:  3.3356 ns      1:  0.81356 ns / ns     1:  0.81356
2:  4.1 ns             .                       .
    .

   ' 4.1 ns RET        /                       u s
```

Thus a signal could take up to 81 percent of a clock cycle just to go from one place to another inside the computer, assuming the signal could actually attain the full speed of light. Pretty tight!

2.7.47 Types Tutorial Exercise 15

The speed limit is 55 miles per hour on most highways. We want to find the ratio of Sam's speed to the US speed limit.

```
1:  55 mph         2:  55 mph           3:  11 hr mph / yd
    .              1:  5 yd / hr            .
                       .

   ' 55 mph RET       ' 5 yd/hr RET          /
```

The *u s* command cancels out these units to get a plain number. Now we take the logarithm base two to find the final answer, assuming that each successive pill doubles his speed.

```
1:  19360.       2:  19360.       1:  14.24
    .            1:  2                .
                     .

    u s              2                B
```

Thus Sam can take up to 14 pills without a worry.

2.7.48 Algebra Tutorial Exercise 1

The result '`sqrt(x)^2`' is simplified back to x by the Calculator, but '`sqrt(x^2)`' is not. (Consider what happens if $x = -4$.) If x is real, this formula could be simplified to '`abs(x)`', but for general complex arguments even that is not safe. (See *XXX* [Declarations], page *XXX*, for a way to tell Calc that x is known to be real.)

2.7.49 Algebra Tutorial Exercise 2

Suppose our roots are $[a, b, c]$. We want a polynomial which is zero when x is any of these values. The trivial polynomial $x - a$ is zero when $x = a$, so the product $(x - a)(x - b)(x - c)$ will do the job. We can use `a c x` to write this in a more familiar form.

```
1:  34 x - 24 x^3          1:  [1.19023, -1.19023, 0]
    .                          .

    r 2                        a P x RET


1:  [x - 1.19023, x + 1.19023, x]     1:  (x - 1.19023) (x + 1.19023) x
    .                                     .

    V M ' x-$ RET                         V R *


1:  x^3 - 1.41666 x        1:  34 x - 24 x^3
    .                          .

    a c x RET                  24 n *  a x
```

Sure enough, our answer (multiplied by a suitable constant) is the same as the original polynomial.

2.7.50 Algebra Tutorial Exercise 3

```
1:  x sin(pi x)         1:  (sin(pi x) - pi x cos(pi x)) / pi^2
    .                       .

  ' x sin(pi x) RET   m r   a i x RET


1:  [y, 1]
2:  (sin(pi x) - pi x cos(pi x)) / pi^2
    .

  ' [y,1] RET TAB
```

```
1:  [(sin(pi y) - pi y cos(pi y)) / pi^2, (sin(pi) - pi cos(pi)) / pi^2]
    .

    V M $ RET

1:  (sin(pi y) - pi y cos(pi y)) / pi^2 + (pi cos(pi) - sin(pi)) / pi^2
    .

    V R -

1:  (sin(3.14159 y) - 3.14159 y cos(3.14159 y)) / 9.8696 - 0.3183
    .

    =

1:  [0., -0.95493, 0.63662, -1.5915, 1.2732]
    .

    v x 5 RET  TAB  V M $ RET
```

2.7.51 Algebra Tutorial Exercise 4

The hard part is that `V R +` is no longer sufficient to add up all the contributions from the slices, since the slices have varying coefficients. So first we must come up with a vector of these coefficients. Here's one way:

```
2:  -1                 2:  3                    1:  [4, 2, ..., 4]
1:  [1, 2, ..., 9]     1:  [-1, 1, ..., -1]         .
    .                      .

    1 n v x 9 RET          V M ^  3 TAB             -

1:  [4, 2, ..., 4, 1]      1:  [1, 4, 2, ..., 4, 1]
    .                          .

    1 |                        1 TAB |
```

Now we compute the function values. Note that for this method we need eleven values, including both endpoints of the desired interval.

```
2:  [1, 4, 2, ..., 4, 1]
1:  [1, 1.1, 1.2,  ...  , 1.8, 1.9, 2.]
    .

 11 RET 1 RET .1 RET  C-u v x
```

```
2:  [1, 4, 2, ..., 4, 1]
1:  [0., 0.084941, 0.16993, ... ]
    .

    ' sin(x) ln(x) RET   m r  p 5 RET   V M $ RET
```

Once again this calls for *V M * V R +*; a simple `*` does the same thing.

```
1:  11.22      1:  1.122      1:  0.374
    .              .              .

    *              .1 *           3 /
```

Wow! That's even better than the result from the Taylor series method.

2.7.52 Rewrites Tutorial Exercise 1

We'll use Big mode to make the formulas more readable.

```
                                               ___
                                          2 + V 2
1:  (2 + sqrt(2)) / (1 + sqrt(2))     1:  --------
    .                                      ___
                                          1 + V 2

                                          .

   ' (2+sqrt(2)) / (1+sqrt(2)) RET        d B
```

Multiplying by the conjugate helps because $(a+b)(a-b) = a^2 - b^2$.

```
          ___    ___
1:  (2 + V 2 ) (V 2  - 1)
    .

   a r a/(b+c) := a*(b-c) / (b^2-c^2) RET
```

```
         ___                         ___
1:  2 + V 2  - 2                1:  V 2
    .                               .

   a r a*(b+c) := a*b + a*c         a s
```

(We could have used `a x` instead of a rewrite rule for the second step.)

The multiply-by-conjugate rule turns out to be useful in many different circumstances, such as when the denominator involves sines and cosines or the imaginary constant `i`.

2.7.53 Rewrites Tutorial Exercise 2

Here is the rule set:

```
[ fib(n) := fib(n, 1, 1) :: integer(n) :: n >= 1,
  fib(1, x, y) := x,
  fib(n, x, y) := fib(n-1, y, x+y) ]
```

The first rule turns a one-argument `fib` that people like to write into a three-argument `fib` that makes computation easier. The second rule converts back from three-argument form once the computation is done. The third rule does the computation itself. It basically says that if x and y are two consecutive Fibonacci numbers, then y and $x + y$ are the next (overlapping) pair of Fibonacci numbers.

Notice that because the number n was "validated" by the conditions on the first rule, there is no need to put conditions on the other rules because the rule set would never get that far unless the input were valid. That further speeds computation, since no extra conditions need to be checked at every step.

Actually, a user with a nasty sense of humor could enter a bad three-argument `fib` call directly, say, '`fib(0, 1, 1)`', which would get the rules into an infinite loop. One thing that would help keep this from happening by accident would be to use something like '`ZzFib`' instead of `fib` as the name of the three-argument function.

2.7.54 Rewrites Tutorial Exercise 3

He got an infinite loop. First, Calc did as expected and rewrote '`2 + 3 x`' to '`f(2, 3, x)`'. Then it looked for ways to apply the rule again, and found that '`f(2, 3, x)`' looks like '`a + b x`' with '`a = 0`' and '`b = 1`', so it rewrote to '`f(0, 1, f(2, 3, x))`'. It then wrapped another '`f(0, 1, ...)`' around that, and so on, ad infinitum. Joe should have used *`M-1 a r`* to make sure the rule applied only once.

(Actually, even the first step didn't work as he expected. What Calc really gives for *`M-1 a r`* in this situation is '`f(3 x, 1, 2)`', treating 2 as the "variable," and '`3 x`' as a constant being added to it. While this may seem odd, it's just as valid a solution as the "obvious" one. One way to fix this would be to add the condition '`:: variable(x)`' to the rule, to make sure the thing that matches '`x`' is indeed a variable, or to change '`x`' to '`quote(x)`' on the lefthand side, so that the rule matches the actual variable '`x`' rather than letting '`x`' stand for something else.)

2.7.55 Rewrites Tutorial Exercise 4

`seq` Here is a suitable set of rules to solve the first part of the problem:

```
[ seq(n, c) := seq(n/2,  c+1) :: n%2 = 0,
  seq(n, c) := seq(3n+1, c+1) :: n%2 = 1 :: n > 1 ]
```

Given the initial formula '`seq(6, 0)`', application of these rules produces the following sequence of formulas:

```
    seq( 3, 1)
    seq(10, 2)
    seq( 5, 3)
    seq(16, 4)
    seq( 8, 5)
    seq( 4, 6)
    seq( 2, 7)
    seq( 1, 8)
```

whereupon neither of the rules match, and rewriting stops.

We can pretty this up a bit with a couple more rules:

```
    [ seq(n) := seq(n, 0),
      seq(1, c) := c,
      ... ]
```

Now, given '`seq(6)`' as the starting configuration, we get 8 as the result.

The change to return a vector is quite simple:

```
    [ seq(n) := seq(n, []) :: integer(n) :: n > 0,
      seq(1, v) := v | 1,
      seq(n, v) := seq(n/2,  v | n) :: n%2 = 0,
      seq(n, v) := seq(3n+1, v | n) :: n%2 = 1 ]
```

Given '`seq(6)`', the result is '`[6, 3, 10, 5, 16, 8, 4, 2, 1]`'.

Notice that the $n > 1$ guard is no longer necessary on the last rule since the $n = 1$ case is now detected by another rule. But a guard has been added to the initial rule to make sure the initial value is suitable before the computation begins.

While still a good idea, this guard is not as vitally important as it was for the `fib` function, since calling, say, '`seq(x, [])`' will not get into an infinite loop. Calc will not be able to prove the symbol '`x`' is either even or odd, so none of the rules will apply and the rewrites will stop right away.

2.7.56 Rewrites Tutorial Exercise 5

If x is the sum $a + b$, then '`nterms(`x`)`' must be '`nterms(`a`)`' plus '`nterms(`b`)`'. If x is not a sum, then '`nterms(`x`)`' $= 1$. `nterms`

```
    [ nterms(a + b) := nterms(a) + nterms(b),
      nterms(x)     := 1 ]
```

Here we have taken advantage of the fact that earlier rules always match before later rules; '`nterms(x)`' will only be tried if we already know that '`x`' is not a sum.

2.7.57 Rewrites Tutorial Exercise 6

Just put the rule '`0^0 := 1`' into `EvalRules`. For example, before making this definition we have:

```
2:  [-2, -1, 0, 1, 2]              1:  [1, 1, 0^0, 1, 1]
1:  0                                  .
    .

    v x 5 RET  3 -  0                  V M ^
```

But then:

```
2:  [-2, -1, 0, 1, 2]              1:  [1, 1, 1, 1, 1]
1:  0                                  .
    .

    U  ' 0^0:=1 RET s t EvalRules RET    V M ^
```

Perhaps more surprisingly, this rule still works with infinite mode turned on. Calc tries `EvalRules` before any built-in rules for a function. This allows you to override the default behavior of any Calc feature: Even though Calc now wants to evaluate 0^0 to `nan`, your rule gets there first and evaluates it to 1 instead.

Just for kicks, try adding the rule `2+3 := 6` to `EvalRules`. What happens? (Be sure to remove this rule afterward, or you might get a nasty surprise when you use Calc to balance your checkbook!)

2.7.58 Rewrites Tutorial Exercise 7

Here is a rule set that will do the job:

```
[ a*(b + c) := a*b + a*c,
  opt(a) O(x^n) + opt(b) O(x^m) := O(x^n) :: n <= m
     :: constant(a) :: constant(b),
  opt(a) O(x^n) + opt(b) x^m := O(x^n) :: n <= m
     :: constant(a) :: constant(b),
  a O(x^n) := O(x^n) :: constant(a),
  x^opt(m) O(x^n) := O(x^(n+m)),
  O(x^n) O(x^m) := O(x^(n+m)) ]
```

If we really want the `+` and `*` keys to operate naturally on power series, we should put these rules in `EvalRules`. For testing purposes, it is better to put them in a different variable, say, `O`, first.

The first rule just expands products of sums so that the rest of the rules can assume they have an expanded-out polynomial to work with. Note that this rule does not mention '`O`' at all, so it will apply to any product-of-sum it encounters—this rule may surprise you if you put it into `EvalRules`!

In the second rule, the sum of two O's is changed to the smaller O. The optional constant coefficients are there mostly so that '`O(x^2) - O(x^3)`' and '`O(x^3) - O(x^2)`' are handled as well as '`O(x^2) + O(x^3)`'.

The third rule absorbs higher powers of '`x`' into O's.

The fourth rule says that a constant times a negligible quantity is still negligible. (This rule will also match '`O(x^3) / 4`', with '`a = 1/4`'.)

The fifth rule rewrites, for example, '`x^2 O(x^3)`' to '`O(x^5)`'. (It is easy to see that if one of these forms is negligible, the other is, too.) Notice the '`x^opt(m)`' to pick up terms like '`x O(x^3)`'. Optional powers will match '`x`' as '`x^1`' but not 1 as '`x^0`'. This turns out to be exactly what we want here.

The sixth rule is the corresponding rule for products of two O's.

Another way to solve this problem would be to create a new "data type" that represents truncated power series. We might represent these as function calls '`series(`*coefs*`, `*x*`)`' where *coefs* is a vector of coefficients for x^0, x^1, x^2, and so on. Rules would exist for sums and products of such `series` objects, and as an optional convenience could also know how to combine a `series` object with a normal polynomial. (With this, and with a rule that rewrites '`O(x^n)`' to the equivalent `series` form, you could still enter power series in exactly the same notation as before.) Operations on such objects would probably be more efficient, although the objects would be a bit harder to read.

Some other symbolic math programs provide a power series data type similar to this. Mathematica, for example, has an object that looks like '`PowerSeries[`*x*`, `*x0*`, `*coefs*`, `*nmin*`, `*nmax*`, `*den*`]`', where *x0* is the point about which the power series is taken (we've been assuming this was always zero), and *nmin*, *nmax*, and *den* allow pseudo-power-series with fractional or negative powers. Also, the `PowerSeries` objects have a special display format that makes them look like '`2 x^2 + O(x^4)`' when they are printed out. (See *XXX* [Compositions], page *XXX*, for a way to do this in Calc, although for something as involved as this it would probably be better to write the formatting routine in Lisp.)

2.7.59 Programming Tutorial Exercise 1

Just enter the formula '`ninteg(sin(t)/t, t, 0, x)`', type *Z F*, and answer the questions. Since this formula contains two variables, the default argument list will be '`(t x)`'. We want to change this to '`(x)`' since *t* is really a dummy variable to be used within `ninteg`.

The exact keystrokes are *`Z F s Si RET RET C-b C-b DEL DEL RET y`*. (The *`C-b C-b DEL DEL`* are what fix the argument list.)

2.7.60 Programming Tutorial Exercise 2

One way is to move the number to the top of the stack, operate on it, then move it back: *`C-x ( M-TAB n M-TAB M-TAB C-x )`*.

Another way is to negate the top three stack entries, then negate again the top two stack entries: *`C-x ( M-3 n M-2 n C-x )`*.

Finally, it turns out that a negative prefix argument causes a command like *`n`* to operate on the specified stack entry only, which is just what we want: *`C-x ( M-- 3 n C-x )`*.

Just for kicks, let's also do it algebraically: *`C-x ( ' -$$$, $$, $ RET C-x )`*.

2.7.61 Programming Tutorial Exercise 3

Each of these functions can be computed using the stack, or using algebraic entry, whichever way you prefer:

Computing $\frac{\sin x}{x}$:

Using the stack: `C-x ( RET S TAB / C-x )`.

Using algebraic entry: `C-x ( ' sin($)/$ RET C-x )`.

Computing the logarithm:

Using the stack: `C-x ( TAB B C-x )`

Using algebraic entry: `C-x ( ' log($,$$) RET C-x )`.

Computing the vector of integers:

Using the stack: `C-x ( 1 RET 1 C-u v x C-x )`. (Recall that `C-u v x` takes the vector size, starting value, and increment from the stack.)

Alternatively: `C-x ( ~ v x C-x )`. (The `~` key pops a number from the stack and uses it as the prefix argument for the next command.)

Using algebraic entry: `C-x ( ' index($) RET C-x )`.

2.7.62 Programming Tutorial Exercise 4

Here's one way: `C-x ( RET V R + TAB v l / C-x )`.

2.7.63 Programming Tutorial Exercise 5

```
2:  1                1:  1.61803398502          2:  1.61803398502
1:  20                   .                      1:  1.61803398875
    .                                               .

   1 RET 20          Z < & 1 + Z >                  I H P
```

This answer is quite accurate.

2.7.64 Programming Tutorial Exercise 6

Here is the matrix:

```
[ [ 0, 1 ]   * [a, b] = [b, a + b]
  [ 1, 1 ] ]
```

Thus '`[0, 1; 1, 1]^n * [1, 1]`' computes Fibonacci numbers $n+1$ and $n+2$. Here's one program that does the job:

```
C-x ( ' [0, 1; 1, 1] ^ ($-1) * [1, 1] RET v u DEL C-x )
```

This program is quite efficient because Calc knows how to raise a matrix (or other value) to the power n in only $\log_2 n$ steps. For example, this program can compute the 1000th Fibonacci number (a 209-digit integer!) in about 10 steps; even though the `Z < ... Z >` solution had much simpler steps, it would have required so many steps that it would not have been practical.

2.7.65 Programming Tutorial Exercise 7

The trick here is to compute the harmonic numbers differently, so that the loop counter itself accumulates the sum of reciprocals. We use a separate variable to hold the integer counter.

```
1:  1           2:  1        1:  .
    .           1:  4
                    .

    1 t 1        1 RET 4      Z ( t 2 r 1 1 + s 1 & Z )
```

The body of the loop goes as follows: First save the harmonic sum so far in variable 2. Then delete it from the stack; the for loop itself will take care of remembering it for us. Next, recall the count from variable 1, add one to it, and feed its reciprocal to the for loop to use as the step value. The for loop will increase the "loop counter" by that amount and keep going until the loop counter exceeds 4.

```
2:  31                   3:  31
1:  3.99498713092        2:  3.99498713092
    .                    1:  4.02724519544
                             .

    r 1 r 2                  RET 31 & +
```

Thus we find that the 30th harmonic number is 3.99, and the 31st harmonic number is 4.02.

2.7.66 Programming Tutorial Exercise 8

The first step is to compute the derivative $f'(x)$ and thus the formula $x - \frac{f(x)}{f'(x)}$.

(Because this definition is long, it will be repeated in concise form below. You can use *M-# m* to load it from there. While you are entering a *Z ‘ Z ’* body in a macro, Calc simply collects keystrokes without executing them. In the following diagrams we'll pretend Calc actually executed the keystrokes as you typed them, just for purposes of illustration.)

```
2:  sin(cos(x)) - 0.5            3:  4.5
1:  4.5                          2:  sin(cos(x)) - 0.5
    .                            1:  -(sin(x) cos(cos(x)))
                                     .

' sin(cos(x))-0.5 RET 4.5  m r  C-x ( Z `  TAB RET a d x RET


2:  4.5
1:  x + (sin(cos(x)) - 0.5) / sin(x) cos(cos(x))
    .

    /  ' x RET TAB -   t 1
```

Now, we enter the loop. We'll use a repeat loop with a 20-repetition limit just in case the method fails to converge for some reason. (Normally, the `Z /` command will stop the loop before all 20 repetitions are done.)

```
1:  4.5         3:  4.5                   2:  4.5
    .           2:  x + (sin(cos(x)) ...  1:  5.24196456928
                1:  4.5                       .
                    .

  20 Z <          RET r 1 TAB                s 1 x RET
```

This is the new guess for x. Now we compare it with the old one to see if we've converged.

```
3:  5.24196     2:  5.24196     1:  5.24196     1:  5.26345856348
2:  5.24196     1:  0               .               .
1:  4.5             .
    .

  RET M-TAB         a =             Z /             Z > Z ' C-x )
```

The loop converges in just a few steps to this value. To check the result, we can simply substitute it back into the equation.

```
2:  5.26345856348
1:  0.499999999997
    .

 RET ' sin(cos($)) RET
```

Let's test the new definition again:

```
2:  x^2 - 9           1:  3.
1:  1                     .
    .

  ' x^2-9 RET 1           X
```

Once again, here's the full Newton's Method definition:

```
C-x ( Z `  TAB RET a d x RET  /  ' x RET TAB -  t 1
           20 Z <  RET r 1 TAB  s 1 x RET
                   RET M-TAB  a =  Z /
              Z >
      Z '
C-x )
```

It turns out that Calc has a built-in command for applying a formula repeatedly until it converges to a number. See *XXX* [Nesting and Fixed Points], page *XXX*, to see how to use it.

Also, of course, `a R` is a built-in command that uses Newton's method (among others) to look for numerical solutions to any equation. See *XXX* [Root Finding], page *XXX*.

2.7.67 Programming Tutorial Exercise 9

The first step is to adjust z to be greater than 5. A simple "for" loop will do the job here. If z is less than 5, we reduce the problem using $\psi(z) = \psi(z+1) - 1/z$. We go on to compute $\psi(z+1)$, and remember to add back a factor of $-1/z$ when we're done. This step is repeated until $z > 5$.

(Because this definition is long, it will be repeated in concise form below. You can use `M-# m` to load it from there. While you are entering a `Z ' Z '` body in a macro, Calc simply collects keystrokes without executing them. In the following diagrams we'll pretend Calc actually executed the keystrokes as you typed them, just for purposes of illustration.)

```
1:  1.                  1:  1.
    .                       .

  1.0 RET            C-x ( Z '  s 1  0 t 2
```

Here, variable 1 holds z and variable 2 holds the adjustment factor. If $z < 5$, we use a loop to increase it.

(By the way, we started with '`1.0`' instead of the integer 1 because otherwise the calculation below will try to do exact fractional arithmetic, and will never converge because fractions compare equal only if they are exactly equal, not just equal to within the current precision.)

```
3:  1.          2:  1.          1:  6.
2:  1.          1:  1               .
1:  5               .
    .

   RET 5          a <    Z [  5 Z (  & s + 2  1 s + 1  1 Z ) r 1  Z ]
```

Now we compute the initial part of the sum: $\ln z - \frac{1}{2z}$ minus the adjustment factor.

```
2:  1.79175946923       2:  1.7084261359        1:  -0.57490719743
1:  0.0833333333333     1:  2.28333333333           .
    .                       .

    L  r 1 2 * &            -  r 2                  -
```

Now we evaluate the series. We'll use another "for" loop counting up the value of $2n$. (Calc does have a summation command, `a +`, but we'll use loops just to get more practice with them.)

```
3:  -0.5749         3:  -0.5749         4:  -0.5749         2:  -0.5749
2:  2               2:  1:6             3:  1:6             1:  2.3148e-3
1:  40              1:  2               2:  2                   .
    .                   .               1:  36.
                                            .

  2 RET 40          Z ( RET k b TAB     RET r 1 TAB ^       * /
3:  -0.5749       3:  -0.5772       2:  -0.5772      1:  -0.577215664892
2:  -0.5749       2:  -0.5772       1:  0                .
1:  2.3148e-3     1:  -0.5749           .
    .                 .

   TAB RET M-TAB        - RET M-TAB       a =      Z /     2  Z )  Z ' C-x )
```

This is the value of $-\gamma$, with a slight bit of roundoff error. To get a full 12 digits, let's use a higher precision:

```
2:  -0.577215664892       2:  -0.577215664892
1:  1.                    1:  -0.577215664901532

    1. RET                    p 16 RET X
```

Here's the complete sequence of keystrokes:

```
C-x ( Z `  s 1  0 t 2
           RET 5 a <  Z [  5 Z (  & s + 2  1 s + 1  1 Z ) r 1  Z ]
           L r 1 2 * & - r 2 -
           2 RET 40  Z (  RET k b TAB RET r 1 TAB ^ * /
                          TAB RET M-TAB - RET M-TAB a = Z /
                     2  Z )
      Z '
C-x )
```

2.7.68 Programming Tutorial Exercise 10

Taking the derivative of a term of the form x^n will produce a term like nx^{n-1}. Taking the derivative of a constant produces zero. From this it is easy to see that the nth derivative of a polynomial, evaluated at $x = 0$, will equal the coefficient on the x^n term times $n!$.

(Because this definition is long, it will be repeated in concise form below. You can use *M-# m* to load it from there. While you are entering a *Z ` Z '* body in a macro, Calc simply collects keystrokes without executing them. In the following diagrams we'll pretend Calc actually executed the keystrokes as you typed them, just for purposes of illustration.)

```
2:  5 x^4 + (x + 1)^2          3:  5 x^4 + (x + 1)^2
1:  6                          2:  0
    .                          1:  6
                                   .

   ' 5 x^4 + (x+1)^2 RET 6         C-x ( Z `  [ ] t 1  0 TAB
```

Variable 1 will accumulate the vector of coefficients.

```
2:  0                3:  0                    2:  5 x^4 + ...
1:  5 x^4 + ...      2:  5 x^4 + ...          1:  1
    .                1:  1                        .
                         .

    Z ( TAB          RET 0 s l x RET              M-TAB ! /  s | 1
```

Note that *s | 1* appends the top-of-stack value to the vector in a variable; it is completely analogous to *s + 1*. We could have written instead, *r 1 TAB | t 1*.

```
1:  20 x^3 + 2 x + 2      1:  0          1:  [1, 2, 1, 0, 5, 0, 0]
    .                         .              .

    a d x RET                 1 Z )          DEL r 1  Z ' C-x )
```

To convert back, a simple method is just to map the coefficients against a table of powers of x.

```
2:  [1, 2, 1, 0, 5, 0, 0]    2:  [1, 2, 1, 0, 5, 0, 0]
1:  6                        1:  [0, 1, 2, 3, 4, 5, 6]
    .                            .

    6 RET                        1 + 0 RET 1 C-u v x


2:  [1, 2, 1, 0, 5, 0, 0]    2:  1 + 2 x + x^2 + 5 x^4
1:  [1, x, x^2, x^3, ... ]       .
    .

    ' x RET TAB V M ^            *
```

Once again, here are the whole polynomial to/from vector programs:

```
C-x ( Z `  [ ] t 1  0 TAB
           Z (  TAB RET 0 s 1 x RET M-TAB ! /  s | 1
                a d x RET
         1 Z ) r 1
      Z '
C-x )

C-x (  1 + 0 RET 1 C-u v x ' x RET TAB V M ^ *  C-x )
```

2.7.69 Programming Tutorial Exercise 11

First we define a dummy program to go on the *z s* key. The true *z s* key is supposed to take two numbers from the stack and return one number, so *DEL* as a dummy definition will make sure the stack comes out right.

```
2:  4          1:  4                            2:  4
1:  2              .                            1:  2
    .                                               .

  4 RET 2        C-x ( DEL C-x )  Z K s RET       2
```

The last step replaces the 2 that was eaten during the creation of the dummy *z s* command. Now we move on to the real definition. The recurrence needs to be rewritten slightly, to the form $s(n,m) = s(n-1,m-1) - (n-1)s(n-1,m)$.

(Because this definition is long, it will be repeated in concise form below. You can use *M-# m* to load it from there.)

```
2:  4       4:  4      3:  4      2:  4
1:  2       3:  2      2:  2      1:  2
    .       2:  4      1:  0          .
            1:  2          .
                .

  C-x (       M-2 RET       a =        Z [  DEL DEL 1  Z :

4:  4     2:  4                  2:  3     4:  3    4:  3    3:  3
3:  2     1:  2                  1:  2     3:  2    3:  2    2:  2
2:  2         .                      .     2:  3    2:  3    1:  3
1:  0                                      1:  2    1:  1        .
    .                                          .        .

  RET 0   a = Z [  DEL DEL 0  Z :  TAB 1 - TAB  M-2 RET    1 -     z s
```

(Note that the value 3 that our dummy *z s* produces is not correct; it is merely a placeholder that will do just as well for now.)

```
3:  3             4:  3        3:  3      2:  3     1:  -6
2:  3             3:  3        2:  3      1:  9         .
1:  2             2:  3        1:  3          .
    .             1:  2            .
                      .

 M-TAB M-TAB     TAB RET M-TAB       z s         *         -

1:  -6                      2:  4       1:  11     2:  11
    .                       1:  2           .      1:  11
                                .                      .

  Z ] Z ] C-x )   Z K s RET     DEL 4 RET 2      z s     M-RET k s
```

Even though the result that we got during the definition was highly bogus, once the definition is complete the *z s* command gets the right answers.

Here's the full program once again:

```
C-x (  M-2 RET a =
       Z [  DEL DEL 1
       Z :  RET 0 a =
            Z [  DEL DEL 0
            Z :  TAB 1 - TAB M-2 RET 1 - z s
                 M-TAB M-TAB TAB RET M-TAB z s * -
            Z ]
       Z ]
C-x )
```

You can read this definition using *M-# m* (`read-kbd-macro`) followed by *Z K s*, without having to make a dummy definition first, because `read-kbd-macro` doesn't need to execute the definition as it reads it in. For this reason, `M-# m` is often the easiest way to create recursive programs in Calc.

2.7.70 Programming Tutorial Exercise 12

This turns out to be a much easier way to solve the problem. Let's denote Stirling numbers as calls of the function '`s`'.

First, we store the rewrite rules corresponding to the definition of Stirling numbers in a convenient variable:

```
s e StirlingRules RET
[ s(n,n) := 1  :: n >= 0,
  s(n,0) := 0  :: n > 0,
  s(n,m) := s(n-1,m-1) - (n-1) s(n-1,m) :: n >= m :: m >= 1 ]
C-c C-c
```

Now, it's just a matter of applying the rules:

```
2:  4          1:  s(4, 2)              1:  11
1:  2              .                        .
    .

  4 RET 2        C-x (  ' s($$,$) RET     a r StirlingRules RET  C-x )
```

As in the case of the `fib` rules, it would be useful to put these rules in `EvalRules` and to add a '`:: remember`' condition to the last rule.

3 Introduction

This chapter is the beginning of the Calc reference manual. It covers basic concepts such as the stack, algebraic and numeric entry, undo, numeric prefix arguments, etc.

3.1 Basic Commands

To start the Calculator in its standard interface, type *`M-x calc`*. By default this creates a pair of small windows, '`*Calculator*`' and '`*Calc Trail*`'. The former displays the contents of the Calculator stack and is manipulated exclusively through Calc commands. It is possible (though not usually necessary) to create several Calc Mode buffers each of which has an independent stack, undo list, and mode settings. There is exactly one Calc Trail buffer; it records a list of the results of all calculations that have been done. The Calc Trail buffer uses a variant of Calc Mode, so Calculator commands still work when the trail buffer's window is selected. It is possible to turn the trail window off, but the '`*Calc Trail*`' buffer itself still exists and is updated silently. See *XXX* [Trail Commands], page *XXX*.

`M-# c`
`M-# M-#`
In most installations, the *`M-# c`* key sequence is a more convenient way to start the Calculator. Also, *`M-# M-#`* and *`M-# #`* are synonyms for *`M-# c`* unless you last used Calc in its "keypad" mode.

`x`
`M-x`
Most Calc commands use one or two keystrokes. Lower- and upper-case letters are distinct. Commands may also be entered in full *`M-x`* form; for some commands this is the only form. As a convenience, the *`x`* key (`calc-execute-extended-command`) is like *`M-x`* except that it enters the initial string '`calc-`' for you. For example, the following key sequences are equivalent: *`S`*, *`M-x calc-sin RET`*, *`x sin RET`*.

The Calculator exists in many parts. When you type *`M-# c`*, the Emacs "auto-load" mechanism will bring in only the first part, which contains the basic arithmetic functions. The other parts will be auto-loaded the first time you use the more advanced commands like trig functions or matrix operations. This is done to improve the response time of the Calculator in the common case when all you need to do is a little arithmetic. If for some reason the Calculator fails to load an extension module automatically, you can force it to load all the extensions by using the *`M-# L`* (`calc-load-everything`) command. See *XXX* [Mode Settings], page *XXX*.

If you type *`M-x calc`* or *`M-# c`* with any numeric prefix argument, the Calculator is loaded if necessary, but it is not actually started. If the argument is positive, the '`calc-ext`' extensions are also loaded if necessary. User-written Lisp code that wishes to make use of Calc's arithmetic routines can use '`(calc 0)`' or '`(calc 1)`' to auto-load the Calculator.

`M-# b`
If you type *`M-# b`*, then next time you use *`M-# c`* you will get a Calculator that uses the full height of the Emacs screen. When full-screen mode is on, *`M-# c`* runs the `full-calc` command instead of `calc`. From the Unix shell you can type '`emacs -f full-calc`' to start a new Emacs specifically for use as a calculator. When Calc is started from the Emacs command line like this, Calc's normal "quit" commands actually quit Emacs itself.

`M-# o`
The *`M-# o`* command is like *`M-# c`* except that the Calc window is not actually selected. If you are already in the Calc window, *`M-# o`* switches you out of it. (The regular Emacs *`C-x o`* command would also work for this, but it has a tendency to drop you into the Calc Trail window instead, which *`M-# o`* takes care not to do.)

For one quick calculation, you can type *M-# q* (`quick-calc`) which prompts you for a formula (like '`2+3/4`'). The result is displayed at the bottom of the Emacs screen without ever creating any special Calculator windows. See *XXX* [Quick Calculator], page *XXX*. `M-# q`

Finally, if you are using the X window system you may want to try *M-# k* (`calc-keypad`) which runs Calc with a "calculator keypad" picture as well as a stack display. Click on the keys with the mouse to operate the calculator. See *XXX* [Keypad Mode], page *XXX*. `M-# k`

The *q* key (`calc-quit`) exits Calc Mode and closes the Calculator's window(s). It does not delete the Calculator buffers. If you type *M-x calc* again, the Calculator will reappear with the contents of the stack intact. Typing *M-# c* or *M-# M-#* again from inside the Calculator buffer is equivalent to executing `calc-quit`; you can think of *M-# M-#* as toggling the Calculator on and off. `q`

The *M-# x* command also turns the Calculator off, no matter which user interface (standard, Keypad, or Embedded) is currently active. It also cancels `calc-edit` mode if used from there. `M-# x`

The *d SPC* key sequence (`calc-refresh`) redraws the contents of the Calculator buffer from memory. Use this if the contents of the buffer have been damaged somehow. `d SPC`

The *o* key (`calc-realign`) moves the cursor back to its "home" position at the bottom of the Calculator buffer. `o`

The `<` and `>` keys are bound to `calc-scroll-left` and `calc-scroll-right`. These are just like the normal horizontal scrolling commands except that they scroll one half-screen at a time by default. (Calc formats its output to fit within the bounds of the window whenever it can.) `<` `>`

The `{` and `}` keys are bound to `calc-scroll-down` and `calc-scroll-up`. They scroll up or down by one-half the height of the Calc window. `{` `}`

The *M-# 0* command (`calc-reset`; that's *M-#* followed by a zero) resets the Calculator to its default state. This clears the stack, resets all the modes, clears the caches (see *XXX* [Caches], page *XXX*), and so on. (It does *not* erase the values of any variables.) With a numeric prefix argument, *M-# 0* preserves the contents of the stack but resets everything else. `M-# 0`

The *M-x calc-version* command displays the current version number of Calc and the name of the person who installed it on your system. (This information is also present in the '`*Calc Trail*`' buffer, and in the output of the *h h* command.)

3.2 Help Commands

The *?* key (`calc-help`) displays a series of brief help messages. Some keys (such as *b* and *d*) are prefix keys, like Emacs' *ESC* and *C-x* prefixes. You can type *?* after a prefix to see a list of commands beginning with that prefix. (If the message includes '`[MORE]`', press *?* again to see additional commands for that prefix.) `?`

The *h h* (`calc-full-help`) command displays all the *?* responses at once. When printed, this makes a nice, compact (three pages) summary of Calc keystrokes. `h h`

In general, the *h* key prefix introduces various commands that provide help within Calc. Many of the *h* key functions are Calc-specific analogues to the *C-h* functions for Emacs help.

The *h i* (`calc-info`) command runs the Emacs Info system to read this manual on-line. This is basically the same as typing *C-h i* (the regular way to run the Info system), then, if Info is not already in the Calc manual, selecting the beginning of the manual. The *M-# i* command is another way to read the Calc manual; it is different from *h i* in that it works any time, not just inside Calc. The plain *i* key is also equivalent to *h i*, though this key is obsolete and may be replaced with a different command in a future version of Calc. `h i` `M-# i` `i`

`h t`
`M-# t`
The *h t* (`calc-tutorial`) command runs the Info system on the Tutorial section of the Calc manual. It is like *h i*, except that it selects the starting node of the tutorial rather than the beginning of the whole manual. (It actually selects the node "Interactive Tutorial" which tells a few things about using the Info system before going on to the actual tutorial.) The *M-# t* key is equivalent to *h t* (but it works at all times).

`h s`
`M-# s`
The *h s* (`calc-info-summary`) command runs the Info system on the Summary node of the Calc manual. See *XXX* [Summary], page *XXX*. The *M-# s* key is equivalent to *h s*.

`h k`
The *h k* (`calc-describe-key`) command looks up a key sequence in the Calc manual. For example, *h k H a S* looks up the documentation on the *H a S* (`calc-solve-for`) command. This works by looking up the textual description of the key(s) in the Key Index of the manual, then jumping to the node indicated by the index.

Most Calc commands do not have traditional Emacs documentation strings, since the *h k* command is both more convenient and more instructive. This means the regular Emacs *C-h k* (`describe-key`) command will not be useful for Calc keystrokes.

`h c`
The *h c* (`calc-describe-key-briefly`) command reads a key sequence and displays a brief one-line description of it at the bottom of the screen. It looks for the key sequence in the Summary node of the Calc manual; if it doesn't find the sequence there, it acts just like its regular Emacs counterpart *C-h c* (`describe-key-briefly`). For example, *h c H a S* gives the description:

```
H a S runs calc-solve-for:  a `H a S' v  => fsolve(a,v)  (?=notes)
```

which means the command *H a S* or *H M-x calc-solve-for* takes a value a from the stack, prompts for a value v, then applies the algebraic function `fsolve` to these values. The '`?=notes`' message means you can now type *?* to see additional notes from the summary that apply to this command.

`h f`
The *h f* (`calc-describe-function`) command looks up an algebraic function or a command name in the Calc manual. The prompt initially contains '`calcFunc-`'; follow this with an algebraic function name to look up that function in the Function Index. Or, backspace and enter a command name beginning with '`calc-`' to look it up in the Command Index. This command will also look up operator symbols that can appear in algebraic formulas, like '`%`' and '`=>`'.

`h v`
The *h v* (`calc-describe-variable`) command looks up a variable in the Calc manual. The prompt initially contains the '`var-`' prefix; just add a variable name like `pi` or `PlotRejects`.

`h b`
The *h b* (`calc-describe-bindings`) command is just like *C-h b*, except that only local (Calc-related) key bindings are listed.

`h n`
The *h n* or *h C-n* (`calc-view-news`) command displays the "news" or change history of Calc. This is kept in the file '`README`', which Calc looks for in the same directory as the Calc source files.

`h C-c`
`h C-d`
`h C-w`
The *h C-c*, *h C-d*, and *h C-w* keys display copying, distribution, and warranty information about Calc. These work by pulling up the appropriate parts of the "Copying" or "Reporting Bugs" sections of the manual.

3.3 Stack Basics

Calc uses RPN notation. If you are not familar with RPN, see *XXX* [RPN Tutorial], page *XXX*.

To add the numbers 1 and 2 in Calc you would type the keys: *1 RET 2 +*. (*RET* corresponds to the *ENTER* key on most calculators.) The first three keystrokes "push" the numbers 1 and 2 onto the stack. The `+` key always "pops" the top two numbers from the stack, adds them, and pushes the result (3) back onto the stack. This number is ready for further calculations: *5 -* pushes 5 onto the stack, then pops the 3 and 5, subtracts them, and pushes the result (−2).

Note that the "top" of the stack actually appears at the *bottom* of the buffer. A line containing a single '`.`' character signifies the end of the buffer; Calculator commands operate on the number(s) directly above this line. The *d t* (`calc-truncate-stack`) command allows you to move the '`.`' marker up and down in the stack; see *XXX* [Truncating the Stack], page *XXX*.

`d l` Stack elements are numbered consecutively, with number 1 being the top of the stack. These line numbers are ordinarily displayed on the lefthand side of the window. The *d l* (`calc-line-numbering`) command controls whether these numbers appear. (Line numbers may be turned off since they slow the Calculator down a bit and also clutter the display.)

`o` The unshifted letter *o* (`calc-realign`) command repositions the cursor to its top-of-stack "home" position. It also undoes any horizontal scrolling in the window. If you give it a numeric prefix argument, it instead moves the cursor to the specified stack element.

The *RET* (or equivalent *SPC*) key is only required to separate two consecutive numbers. (After all, if you typed *1 2* by themselves the Calculator would enter the number 12.) If you press *RET* or *SPC* *not* right after typing a number, the key duplicates the number on the top of the stack. *RET ** is thus a handy way to square a number.

The *DEL* key pops and throws away the top number on the stack. The *TAB* key swaps the top two objects on the stack. See *XXX* [Stack and Trail], page *XXX*, for descriptions of these and other stack-related commands.

3.4 Numeric Entry

`0-9` `.` `e` Pressing a digit or other numeric key begins numeric entry using the minibuffer. The number is pushed on the stack when you press the *RET* or *SPC* keys. If you press any other non-numeric key, the number is pushed onto the stack and the appropriate operation is performed. If you press a numeric key which is not valid, the key is ignored.

`_` There are three different concepts corresponding to the word "minus," typified by $a - b$ (subtraction), $-x$ (change-sign), and −5 (negative number). Calc uses three different keys for these operations, respectively: *-*, *n*, and *_* (the underscore). The *-* key subtracts the two numbers on the top of the stack. The *n* key changes the sign of the number on the top of the stack or the number currently being entered. The *_* key begins entry of a negative number or changes the sign of the number currently being entered. The following sequences all enter the number −5 onto the stack: *0 RET 5 -*, *5 n RET*, *5 RET n*, *_ 5 RET*, *5 _ RET*.

Some other keys are active during numeric entry, such as *#* for non-decimal numbers, *:* for fractions, and *@* for HMS forms. These notations are described later in this manual with the corresponding data types. See *XXX* [Data Types], page *XXX*.

During numeric entry, the only editing key available is *DEL*.

3.5 Algebraic Entry

'

Calculations can also be entered in algebraic form. This is accomplished by typing the apostrophe key, *'*, followed by the expression in standard format: *' 2+3*4 RET* computes $2 + (3 \times 4) = 14$ and pushes that on the stack. If you wish you can ignore the RPN aspect of Calc altogether and simply enter algebraic expressions in this way. You may want to use *DEL* every so often to clear previous results off the stack.

You can press the apostrophe key during normal numeric entry to switch the half-entered number into algebraic entry mode. One reason to do this would be to use the full Emacs cursor motion and editing keys, which are available during algebraic entry but not during numeric entry.

In the same vein, during either numeric or algebraic entry you can press *`* (backquote) to switch to `calc-edit` mode, where you complete your half-finished entry in a separate buffer. See *XXX* [Editing Stack Entries], page *XXX*.

m a

If you prefer algebraic entry, you can use the command *m a* (`calc-algebraic-mode`) to set Algebraic mode. In this mode, digits and other keys that would normally start numeric entry instead start full algebraic entry; as long as your formula begins with a digit you can omit the apostrophe. Open parentheses and square brackets also begin algebraic entry. You can still do RPN calculations in this mode, but you will have to press *RET* to terminate every number: *2 RET 3 RET * 4 RET +* would accomplish the same thing as *2*3+4 RET*.

If you give a numeric prefix argument like *C-u* to the *m a* command, it enables Incomplete Algebraic mode; this is like regular Algebraic mode except that it applies to the *(* and *[* keys only. Numeric keys still begin a numeric entry in this mode.

m t

The *m t* (`calc-total-algebraic-mode`) gives you an even stronger algebraic-entry mode, in which *all* regular letter and punctuation keys begin algebraic entry. Use this if you prefer typing *sqrt( )* instead of *Q*, *factor( )* instead of *a f*, and so on. To type regular Calc commands when you are in "total" algebraic mode, hold down the *META* key. Thus *M-q* is the command to quit Calc, *M-p* sets the precision, and *M-m t* (or *M-m M-t*, if you prefer) turns total algebraic mode back off again. Meta keys also terminate algebraic entry, so that *2+3 M-S* is equivalent to *2+3 RET M-S*. The symbol '`Alg*`' will appear in the mode line whenever you are in this mode.

Pressing *'* (the apostrophe) a second time re-enters the previous algebraic formula. You can then use the normal Emacs editing keys to modify this formula to your liking before pressing *RET*.

$

Within a formula entered from the keyboard, the symbol *$* represents the number on the top of the stack. If an entered formula contains any *$* characters, the Calculator replaces the top of stack with that formula rather than simply pushing the formula onto the stack. Thus, *' 1+2 RET* pushes 3 on the stack, and *$*2 RET* replaces it with 6. Note that the *$* key always initiates algebraic entry; the *'* is unnecessary if *$* is the first character in the new formula.

Higher stack elements can be accessed from an entered formula with the symbols *$$*, *$$$*, and so on. The number of stack elements removed (to be replaced by the entered values) equals the number of dollar signs in the longest such symbol in the formula. For example, '`$$+$$$`' adds the second and third stack elements, replacing the top three elements with the answer. (All information about the top stack element is thus lost since no single '`$`' appears in this formula.)

A slightly different way to refer to stack elements is with a dollar sign followed by a number: '`$1`', '`$2`', and so on are much like '`$`', '`$$`', etc., except that stack entries referred to numerically are not replaced by the algebraic entry. That is, while '`$+1`' replaces 5 on the stack with 6, '`$1+1`' leaves the 5 on the stack and pushes an additional 6.

If a sequence of formulas are entered separated by commas, each formula is pushed onto the stack in turn. For example, '`1,2,3`' pushes those three numbers onto the stack (leaving the 3 at the top), and '`$+1,$-1`' replaces a 5 on the stack with 4 followed by 6. Also, '`$,$$`' exchanges the top two elements of the stack, just like the *TAB* key.

You can finish an algebraic entry with *M-=* or *M-RET* instead of *RET*. This uses `=` to evaluate the variables in each formula that goes onto the stack. (Thus *' pi RET* pushes the variable '`pi`', but *' pi M-RET* pushes 3.1415.)

If you finish your algebraic entry by pressing *LFD* (or *C-j*) instead of *RET*, Calc disables the default simplifications (as if by *m O*; see *XXX* [Simplification Modes], page *XXX*) while the entry is being pushed on the stack. Thus *' 1+2 RET* pushes 3 on the stack, but *' 1+2 LFD* pushes the formula $1 + 2$; you might then press `=` when it is time to evaluate this formula.

3.6 "Quick Calculator" Mode

`M-# q`

There is another way to invoke the Calculator if all you need to do is make one or two quick calculations. Type *M-# q* (or *M-x quick-calc*), then type any formula as an algebraic entry. The Calculator will compute the result and display it in the echo area, without ever actually putting up a Calc window.

You can use the *$* character in a Quick Calculator formula to refer to the previous Quick Calculator result. Older results are not retained; the Quick Calculator has no effect on the full Calculator's stack or trail. If you compute a result and then forget what it was, just run `M-# q` again and enter '`$`' as the formula.

If this is the first time you have used the Calculator in this Emacs session, the *M-# q* command will create the `*Calculator*` buffer and perform all the usual initializations; it simply will refrain from putting that buffer up in a new window. The Quick Calculator refers to the `*Calculator*` buffer for all mode settings. Thus, for example, to set the precision that the Quick Calculator uses, simply run the full Calculator momentarily and use the regular *p* command.

If you use `M-# q` from inside the Calculator buffer, the effect is the same as pressing the apostrophe key (algebraic entry).

The result of a Quick calculation is placed in the Emacs "kill ring" as well as being displayed. A subsequent *C-y* command will yank the result into the editing buffer. You can also use this to yank the result into the next *M-# q* input line as a more explicit alternative to *$* notation, or to yank the result into the Calculator stack after typing *M-# c*.

If you finish your formula by typing *LFD* (or *C-j*) instead of *RET*, the result is inserted immediately into the current buffer rather than going into the kill ring.

Quick Calculator results are actually evaluated as if by the `=` key (which replaces variable names by their stored values, if any). If the formula you enter is an assignment to a variable using the '`:=`' operator, say, '`foo := 2 + 3`' or '`foo := foo + 1`', then the result of the evaluation is stored in that Calc variable. See *XXX* [Store and Recall], page *XXX*.

If the result is an integer and the current display radix is decimal, the number will also be displayed in hex and octal formats. If the integer is in the range from 1 to 126, it will also be displayed as an ASCII character.

For example, the quoted character '`"x"`' produces the vector result '`[120]`' (because 120 is the ASCII code of the lower-case 'x'; see *XXX* [Strings], page *XXX*). Since this is a vector, not an integer, it is displayed only according to the current mode settings. But running Quick Calc again and entering '`120`' will produce the result '`120 (16#78, 8#170, x)`' which shows the number in its decimal, hexadecimal, octal, and ASCII forms.

Please note that the Quick Calculator is not any faster at loading or computing the answer than the full Calculator; the name "quick" merely refers to the fact that it's much less hassle to use for small calculations.

3.7 Numeric Prefix Arguments

Many Calculator commands use numeric prefix arguments. Some, such as *d s* (`calc-sci-notation`), set a parameter to the value of the prefix argument or use a default if you don't use a prefix. Others (like *d f* (`calc-fix-notation`)) require an argument and prompt for a number if you don't give one as a prefix.

As a rule, stack-manipulation commands accept a numeric prefix argument which is interpreted as an index into the stack. A positive argument operates on the top *n* stack entries; a negative argument operates on the *n*th stack entry in isolation; and a zero argument operates on the entire stack.

Most commands that perform computations (such as the arithmetic and scientific functions) accept a numeric prefix argument that allows the operation to be applied across many stack elements. For unary operations (that is, functions of one argument like absolute value or complex conjugate), a positive prefix argument applies that function to the top *n* stack entries simultaneously, and a negative argument applies it to the *n*th stack entry only. For binary operations (functions of two arguments like addition, GCD, and vector concatenation), a positive prefix argument "reduces" the function across the top *n* stack elements (for example, *C-u 5 +* sums the top 5 stack entries; see *XXX* [Reducing and Mapping], page *XXX*), and a negative argument maps the next-to-top *n* stack elements with the top stack element as a second argument (for example, *7 c-u -5 +* adds 7 to the top 5 stack elements). This feature is not available for operations which use the numeric prefix argument for some other purpose.

Numeric prefixes are specified the same way as always in Emacs: Press a sequence of *META*-digits, or press *ESC* followed by digits, or press *C-u* followed by digits. Some commands treat plain *C-u* (without any actual digits) specially.

~ You can type ~ (`calc-num-prefix`) to pop an integer from the top of the stack and enter it as the numeric prefix for the next command. For example, *C-u 16 p* sets the precision to 16 digits; an alternate (silly) way to do this would be *2 RET 4 ^ ~ p*, i.e., compute 2 to the fourth power and set the precision to that value.

Conversely, if you have typed a numeric prefix argument the ~ key pushes it onto the stack in the form of an integer.

3.8 Undoing Mistakes

The shift-*U* key (`calc-undo`) undoes the most recent operation. If that operation added or dropped objects from the stack, those objects are removed or restored. If it was a "store" operation, you are queried whether or not to restore the variable to its original value. The *U* key may be pressed any number of times to undo successively farther back in time; with a numeric prefix argument it undoes a specified number of operations. The undo history is cleared only by the *q* (`calc-quit`) command. (Recall that *M-# c* is synonymous with `calc-quit` while inside the Calculator; this also clears the undo history.) `U` `C-_`

Currently the mode-setting commands (like `calc-precision`) are not undoable. You can undo past a point where you changed a mode, but you will need to reset the mode yourself.

The shift-*D* key (`calc-redo`) redoes an operation that was mistakenly undone. Pressing *U* with a negative prefix argument is equivalent to executing `calc-redo`. You can redo any number of times, up to the number of recent consecutive undo commands. Redo information is cleared whenever you give any command that adds new undo information, i.e., if you undo, then enter a number on the stack or make any other change, then it will be too late to redo. `D`

The *M-RET* key (`calc-last-args`) is like undo in that it restores the arguments of the most recent command onto the stack; however, it does not remove the result of that command. Given a numeric prefix argument, this command applies to the nth most recent command which removed items from the stack; it pushes those items back onto the stack. `M-RET`

The *K* (`calc-keep-args`) command provides a related function to *M-RET*. See *XXX* [Stack and Trail], page *XXX*.

It is also possible to recall previous results or inputs using the trail. See *XXX* [Trail Commands], page *XXX*.

The standard Emacs *C-_* undo key is recognized as a synonym for *U*.

3.9 Error Messages

Many situations that would produce an error message in other calculators simply create unsimplified formulas in the Emacs Calculator. For example, *1 RET 0 /* pushes the formula 1/0; *0 L* pushes the formula '`ln(0)`'. Floating-point overflow and underflow are also reasons for this to happen. `w`

When a function call must be left in symbolic form, Calc usually produces a message explaining why. Messages that are probably surprising or indicative of user errors are displayed automatically. Other messages are simply kept in Calc's memory and are displayed only if you type *w* (`calc-why`). You can also press *w* if the same computation results in several messages. (The first message will end with '`[w=more]`' in this case.)

The *d w* (`calc-auto-why`) command controls when error messages are displayed automatically. (Calc effectively presses *w* for you after your computation finishes.) By default, this occurs only for "important" messages. The other possible modes are to report *all* messages automatically, or to report none automatically (so that you must always press *w* yourself to see the messages). `d w`

3.10 Multiple Calculators

It is possible to have any number of Calc Mode buffers at once. Usually this is done by executing *M-x another-calc*, which is similar to *M-# c* except that if a '`*Calculator*`' buffer already exists, a new, independent one with a name of the form '`*Calculator*<`*n*`>`' is created. You can also use the command `calc-mode` to put any buffer into Calculator mode, but this would ordinarily never be done.

The *q* (`calc-quit`) command does not destroy a Calculator buffer; it only closes its window. Use *M-x kill-buffer* to destroy a Calculator buffer.

Each Calculator buffer keeps its own stack, undo list, and mode settings such as precision, angular mode, and display formats. In Emacs terms, variables such as `calc-stack` are buffer-local variables. The global default values of these variables are used only when a new Calculator buffer is created. The `calc-quit` command saves the stack and mode settings of the buffer being quit as the new defaults.

There is only one trail buffer, '`*Calc Trail*`', used by all Calculator buffers.

3.11 Troubleshooting Commands

This section describes commands you can use in case a computation incorrectly fails or gives the wrong answer.

See *XXX* [Reporting Bugs], page *XXX*, if you find a problem that appears to be due to a bug or deficiency in Calc.

3.11.1 Autoloading Problems

The Calc program is split into many component files; components are loaded automatically as you use various commands that require them. Occasionally Calc may lose track of when a certain component is necessary; typically this means you will type a command and it won't work because some function you've never heard of was undefined.

`M-# L` If this happens, the easiest workaround is to type *M-# L* (`calc-load-everything`) to force all the parts of Calc to be loaded right away. This will cause Emacs to take up a lot more memory than it would otherwise, but it's guaranteed to fix the problem.

If you seem to run into this problem no matter what you do, or if even the *M-# L* command crashes, Calc may have been improperly installed. See *XXX* [Installation], page *XXX*, for details of the installation process.

3.11.2 Recursion Depth

Calc uses recursion in many of its calculations. Emacs Lisp keeps a variable `max-lisp-eval-depth` which limits the amount of recursion possible in an attempt to recover from program bugs. If a calculation ever halts incorrectly with the message "Computation got stuck or ran too long," use the *M* command (`calc-more-recursion-depth`) to increase this limit. (Of course, this will not help if the calculation really did get stuck due to some problem inside Calc.) `M` `I M`

The limit is always increased (multiplied) by a factor of two. There is also an *I M* (`calc-less-recursion-depth`) command which decreases this limit by a factor of two, down to a minimum value of 200. The default value is 1000.

These commands also double or halve `max-specpdl-size`, another internal Lisp recursion limit. The minimum value for this limit is 600.

3.11.3 Caches

Calc saves certain values after they have been computed once. For example, the *P* (`calc-pi`) command initially "knows" the constant π to about 20 decimal places; if the current precision is greater than this, it will recompute π using a series approximation. This value will not need to be recomputed ever again unless you raise the precision still further. Many operations such as logarithms and sines make use of similarly cached values such as $\frac{\pi}{4}$ and $\ln 2$. The visible effect of caching is that high-precision computations may seem to do extra work the first time. Other things cached include powers of two (for the binary arithmetic functions), matrix inverses and determinants, symbolic integrals, and data points computed by the graphing commands.

If you suspect a Calculator cache has become corrupt, you can use the `calc-flush-caches` command to reset all caches to the empty state. (This should only be necessary in the event of bugs in the Calculator.) The *M-# 0* (with the zero key) command also resets caches along with all other aspects of the Calculator's state.

3.11.4 Debugging Calc

A few commands exist to help in the debugging of Calc commands. See *XXX* [Programming], page *XXX*, to see the various ways that you can write your own Calc commands.

The *Z T* (`calc-timing`) command turns on and off a mode in which the timing of slow commands is reported in the Trail. Any Calc command that takes two seconds or longer writes a line to the Trail showing how many seconds it took. This value is accurate only to within one second. `Z T`

All steps of executing a command are included; in particular, time taken to format the result for display in the stack and trail is counted. Some prompts also count time taken waiting for them to be answered, while others do not; this depends on the exact implementation of the command. For best results, if you are timing a sequence that includes prompts or multiple commands, define a keyboard macro to run the whole sequence at once. Calc's *X* command (see *XXX* [Keyboard Macros], page *XXX*) will then report the time taken to execute the whole macro.

Another advantage of the *X* command is that while it is executing, the stack and trail are not updated from step to step. So if you expect the output of your test sequence to leave a result that may take a long time to format and you don't wish to count this formatting time, end your sequence with a *DEL* keystroke to clear the result from the stack. When you run the sequence with *X*, Calc will never bother to format the large result.

Another thing *Z T* does is to increase the Emacs variable `gc-cons-threshold` to a much higher value (two million; the usual default in Calc is 250,000) for the duration of each command. This generally prevents garbage collection during the timing of the command, though it may cause your Emacs process to grow abnormally large. (Garbage collection time is a major unpredictable factor in the timing of Emacs operations.)

Another command that is useful when debugging your own Lisp extensions to Calc is *M-x calc-pass-errors*, which disables the error handler that changes the "`max-lisp-eval-depth` exceeded" message to the much more friendly "Computation got stuck or ran too long." This handler interferes with the Emacs Lisp debugger's `debug-on-error` mode. Errors are reported in the handler itself rather than at the true location of the error. After you have executed `calc-pass-errors`, Lisp errors will be reported correctly but the user-friendly message will be lost.

4 Data Types

This chapter discusses the various types of objects that can be placed on the Calculator stack, how they are displayed, and how they are entered. (See *XXX* [Data Type Formats], page *XXX*, for information on how these data types are represented as underlying Lisp objects.)

Integers, fractions, and floats are various ways of describing real numbers. HMS forms also for many purposes act as real numbers. These types can be combined to form complex numbers, modulo forms, error forms, or interval forms. (But these last four types cannot be combined arbitrarily: error forms may not contain modulo forms, for example.) Finally, all these types of numbers may be combined into vectors, matrices, or algebraic formulas.

4.1 Integers

The Calculator stores integers to arbitrary precision. Addition, subtraction, and multiplication of integers always yields an exact integer result. (If the result of a division or exponentiation of integers is not an integer, it is expressed in fractional or floating-point form according to the current Fraction Mode. See *XXX* [Fraction Mode], page *XXX*.)

A decimal integer is represented as an optional sign followed by a sequence of digits. Grouping (see *XXX* [Grouping Digits], page *XXX*) can be used to insert a comma at every third digit for display purposes, but you must not type commas during the entry of numbers.

A non-decimal integer is represented as an optional sign, a radix between 2 and 36, a '`#`' symbol, and one or more digits. For radix 11 and above, the letters A through Z (upper- or lower-case) count as digits and do not terminate numeric entry mode. See *XXX* [Radix Modes], page *XXX*, for how to set the default radix for display of integers. Numbers of any radix may be entered at any time. If you press `#` at the beginning of a number, the current display radix is used. #

4.2 Fractions

A *fraction* is a ratio of two integers. Fractions are traditionally written "2/3" but Calc uses the notation '`2:3`'. (The `/` key performs RPN division; the following two sequences push the number '`2:3`' on the stack: *`2 : 3 RET`*, or *`2 RET 3 /`* assuming Fraction Mode has been enabled.) When the Calculator produces a fractional result it always reduces it to simplest form, which may in fact be an integer.

Fractions may also be entered in a three-part form, where '`2:3:4`' represents two-and-three-quarters. See *XXX* [Fraction Formats], page *XXX*, for fraction display formats.

Non-decimal fractions are entered and displayed as '*radix*`#`*num*`:`*denom*' (or in the analogous three-part form). The numerator and denominator always use the same radix.

4.3 Floats

A floating-point number or *float* is a number stored in scientific notation. The number of significant digits in the fractional part is governed by the current floating precision (see *XXX* [Precision], page *XXX*). The range of acceptable values is from $10^{-3999999}$ (inclusive) to $10^{4000000}$ (exclusive), plus the corresponding negative values and zero.

Calculations that would exceed the allowable range of values (such as '`exp(exp(20))`') are left in symbolic form by Calc. The messages "floating-point overflow" or "floating-point underflow" indicate that during the calculation a number would have been produced that was too large or too close to zero, respectively, to be represented by Calc. This does not necessarily mean the final result would have overflowed, just that an overflow occurred while computing the result. (In fact, it could report an underflow even though the final result would have overflowed!)

If a rational number and a float are mixed in a calculation, the result will in general be expressed as a float. Commands that require an integer value (such as ***k g*** [`gcd`]) will also accept integer-valued floats, i.e., floating-point numbers with nothing after the decimal point.

Floats are identified by the presence of a decimal point and/or an exponent. In general a float consists of an optional sign, digits including an optional decimal point, and an optional exponent consisting of an '`e`', an optional sign, and up to seven exponent digits. For example, '`23.5e-2`' is 23.5 times ten to the minus-second power, or 0.235.

Floating-point numbers are normally displayed in decimal notation with all significant figures shown. Exceedingly large or small numbers are displayed in scientific notation. Various other display options are available. See *XXX* [Float Formats], page *XXX*.

Floating-point numbers are stored in decimal, not binary. The result of each operation is rounded to the nearest value representable in the number of significant digits specified by the current precision, rounding away from zero in the case of a tie. Thus (in the default display mode) what you see is exactly what you get. Some operations such as square roots and transcendental functions are performed with several digits of extra precision and then rounded down, in an effort to make the final result accurate to the full requested precision. However, accuracy is not rigorously guaranteed. If you suspect the validity of a result, try doing the same calculation in a higher precision. The Calculator's arithmetic is not intended to be IEEE-conformant in any way.

While floats are always *stored* in decimal, they can be entered and displayed in any radix just like integers and fractions. The notation '*radix*`#`*ddd*`.`*ddd*' is a floating-point number whose digits are in the specified radix. Note that the '`.`' is more aptly referred to as a "radix point" than as a decimal point in this case. The number '`8#123.4567`' is defined as '`8#1234567 * 8^-4`'. If the radix is 14 or less, you can use '`e`' notation to write a non-decimal number in scientific notation. The exponent is written in decimal, and is considered to be a power of the radix: '`8#1234567e-4`'. If the radix is 15 or above, the letter '`e`' is a digit, so scientific notation must be written out, e.g., '`16#123.4567*16^2`'. The first two exercises of the Modes Tutorial explore some of the properties of non-decimal floats.

4.4 Complex Numbers

There are two supported formats for complex numbers: rectangular and polar. The default format is rectangular, displayed in the form '`(`*real*`,`*imag*`)`' where *real* is the real part and *imag* is the imaginary part, each of which may be any real number. Rectangular complex numbers can also be displayed in '*a*`+`*b*`i`' notation; see *XXX* [Complex Formats], page *XXX*.

Polar complex numbers are displayed in the form '`(`r`;`θ`)`' where r is the nonnegative magnitude and θ is the argument or phase angle. The range of θ depends on the current angular mode (see *XXX* [Angular Modes], page *XXX*); it is generally between -180 and $+180$ degrees or the equivalent range in radians.

Complex numbers are entered in stages using incomplete objects. See *XXX* [Incomplete Objects], page *XXX*.

Operations on rectangular complex numbers yield rectangular complex results, and similarly for polar complex numbers. Where the two types are mixed, or where new complex numbers arise (as for the square root of a negative real), the current *Polar Mode* is used to determine the type. See *XXX* [Polar Mode], page *XXX*.

A complex result in which the imaginary part is zero (or the phase angle is 0 or 180 degrees or π radians) is automatically converted to a real number.

4.5 Infinities

The word `inf` represents the mathematical concept of *infinity*. Calc actually has three slightly different infinity-like values: `inf`, `uinf`, and `nan`. These are just regular variable names (see *XXX* [Variables], page *XXX*); you should avoid using these names for your own variables because Calc gives them special treatment. Infinities, like all variable names, are normally entered using algebraic entry.

Mathematically speaking, it is not rigorously correct to treat "infinity" as if it were a number, but mathematicians often do so informally. When they say that '`1 / inf = 0`', what they really mean is that $1/x$, as x becomes larger and larger, becomes arbitrarily close to zero. So you can imagine that if x got "all the way to infinity," then $1/x$ would go all the way to zero. Similarly, when they say that '`exp(inf) = inf`', they mean that e^x grows without bound as x grows. The symbol '`-inf`' likewise stands for an infinitely negative real value; for example, we say that '`exp(-inf) = 0`'. You can have an infinity pointing in any direction on the complex plane: '`sqrt(-inf) = i inf`'.

The same concept of limits can be used to define $1/0$. We really want the value that $1/x$ approaches as x approaches zero. But if all we have is $1/0$, we can't tell which direction x was coming from. If x was positive and decreasing toward zero, then we should say that '`1 / 0 = inf`'. But if x was negative and increasing toward zero, the answer is '`1 / 0 = -inf`'. In fact, x could be an imaginary number, giving the answer '`i inf`' or '`-i inf`'. Calc uses the special symbol '`uinf`' to mean *undirected infinity*, i.e., a value which is infinitely large but with an unknown sign (or direction on the complex plane).

Calc actually has three modes that say how infinities are handled. Normally, infinities never arise from calculations that didn't already have them. Thus, $1/0$ is treated simply as an error and left unevaluated. The *`m i`* (`calc-infinite-mode`) command (see *XXX* [Infinite Mode], page *XXX*) enables a mode in which 1/0 evaluates to `uinf` instead. There is also an alternative type of infinite mode which says to treat zeros as if they were positive, so that '`1 / 0 = inf`'. While this is less mathematically correct, it may be the answer you want in some cases.

Since all infinities are "as large" as all others, Calc simplifies, e.g., '`5 inf`' to '`inf`'. Another example is '`5 - inf = -inf`', where the '`-inf`' is so large that adding a finite number like five to it does not affect it. Note that '`a - inf`' also results in '`-inf`'; Calc assumes that variables like `a` always stand for finite quantities. Just to show that infinities really are all the same size, note that '`sqrt(inf) = inf^2 = exp(inf) = inf`' in Calc's notation.

It's not so easy to define certain formulas like '`0 * inf`' and '`inf / inf`'. Depending on where these zeros and infinities came from, the answer could be literally anything. The latter formula could be the limit of x/x (giving a result of one), or $2x/x$ (giving two), or x^2/x (giving `inf`), or x/x^2 (giving zero). Calc uses the symbol `nan` to represent such an *indeterminate* value. (The name "nan" comes from analogy with the "NAN" concept of IEEE standard arithmetic; it stands for "Not A Number." This is somewhat of a misnomer, since `nan` *does* stand for some number or infinity, it's just that *which* number it stands for cannot be determined.) In Calc's notation, '`0 * inf = nan`' and '`inf / inf = nan`'. A few other common indeterminate expressions are '`inf - inf`' and '`inf ^ 0`'. Also, '`0 / 0 = nan`' if you have turned on "infinite mode" (as described above).

Infinities are especially useful as parts of *intervals*. See *XXX* [Interval Forms], page *XXX*.

4.6 Vectors and Matrices

The *vector* data type is flexible and general. A vector is simply a list of zero or more data objects. When these objects are numbers, the whole is a vector in the mathematical sense. When these objects are themselves vectors of equal (nonzero) length, the whole is a *matrix*. A vector which is not a matrix is referred to here as a *plain vector*.

A vector is displayed as a list of values separated by commas and enclosed in square brackets: '`[1, 2, 3]`'. Thus the following is a 2 row by 3 column matrix: '`[[1, 2, 3], [4, 5, 6]]`'. Vectors, like complex numbers, are entered as incomplete objects. See *XXX* [Incomplete Objects], page *XXX*. During algebraic entry, vectors are entered all at once in the usual brackets-and-commas form. Matrices may be entered algebraically as nested vectors, or using the shortcut notation '`[1, 2, 3; 4, 5, 6]`', with rows separated by semicolons. The commas may usually be omitted when entering vectors: '`[1 2 3]`'. Curly braces may be used in place of brackets: '`{1, 2, 3}`', but the commas are required in this case.

Traditional vector and matrix arithmetic is also supported; see *XXX* [Basic Arithmetic], page *XXX* and see *XXX* [Matrix Functions], page *XXX*. Many other operations are applied to vectors element-wise. For example, the complex conjugate of a vector is a vector of the complex conjugates of its elements.

`vec` Algebraic functions for building vectors include '`vec(a, b, c)`' to build '`[a, b, c]`', '`cvec(a, n, m)`' to build an $n \times m$ matrix of '`a`'s, and '`index(n)`' to build a vector of integers from 1 to '`n`'.

4.7 Strings

Character strings are not a special data type in the Calculator. Rather, a string is represented simply as a vector all of whose elements are integers in the range 0 to 255 (ASCII codes). You can enter a string at any time by pressing the `"` key. Quotation marks and backslashes are written '`\"`' and '`\\`', respectively, inside strings. Other notations introduced by backslashes are: `"`

`\a`	7	`\^@`	0
`\b`	8	`\^a-z`	1-26
`\e`	27	`\^[`	27
`\f`	12	`\^\\`	28
`\n`	10	`\^]`	29
`\r`	13	`\^^`	30
`\t`	9	`\^_`	31
		`\^?`	127

Finally, a backslash followed by three octal digits produces any character from its ASCII code.

Strings are normally displayed in vector-of-integers form. The *d* `"` (`calc-display-strings`) command toggles a mode in which any vectors of small integers are displayed as quoted strings instead. `d "`

The backslash notations shown above are also used for displaying strings. Characters 128 and above are not translated by Calc; unless you have an Emacs modified for 8-bit fonts, these will show up in backslash-octal-digits notation. For characters below 32, and for character 127, Calc uses the backslash-letter combination if there is one, or otherwise uses a '`\^`' sequence.

The only Calc feature that uses strings is *compositions*; see *XXX* [Compositions], page *XXX*. Strings also provide a convenient way to do conversions between ASCII characters and integers.

There is a **`string`** function which provides a different display format for strings. Basically, '`string(`*s*`)`', where *s* is a vector of integers in the proper range, is displayed as the corresponding string of characters with no surrounding quotation marks or other modifications. Thus '`string("ABC")`' (or '`string([65 66 67])`') will look like '`ABC`' on the stack. This happens regardless of whether *d* `"` has been used. The only way to turn it off is to use *d U* (unformatted language mode) which will display '`string("ABC")`' instead. `string`

Control characters are displayed somewhat differently by `string`. Characters below 32, and character 127, are shown using '`^`' notation (same as shown above, but without the backslash). The quote and backslash characters are left alone, as are characters 128 and above.

The `bstring` function is just like `string` except that the resulting string is breakable across multiple lines if it doesn't fit all on one line. Potential break points occur at every space character in the string. `bstring`

4.8 HMS Forms

HMS stands for Hours-Minutes-Seconds; when used as an angular argument, the interpretation is Degrees-Minutes-Seconds. All functions that operate on angles accept HMS forms. These are interpreted as degrees regardless of the current angular mode. It is also possible to use HMS as the angular mode so that calculated angles are expressed in degrees, minutes, and seconds.

@ The default format for HMS values is '*hours*`@` *mins*`'` *secs*`"`'. During entry, the letters '`h`' (for "hours") or '`o`' (approximating the "degrees" symbol) are accepted as well as '`@`', '`m`' is accepted in place of '`'`', and '`s`' is accepted in place of '`"`'. The *hours* value is an integer (or integer-valued float). The *mins* value is an integer or integer-valued float between 0 and 59. The *secs* value is a real number between 0 (inclusive) and 60 (exclusive). A positive HMS form is interpreted as *hours* + *mins*/60 + *secs*/3600. A negative HMS form is interpreted as $-hours - mins/60 - secs/3600$. Display format for HMS forms is quite flexible. See *XXX* [HMS Formats], page *XXX*.

HMS forms can be added and subtracted. When they are added to numbers, the numbers are interpreted according to the current angular mode. HMS forms can also be multiplied and divided by real numbers. Dividing two HMS forms produces a real-valued ratio of the two angles.

Just for kicks, *`M-x calc-time`* pushes the current time of day on the stack as an HMS form.

4.9 Date Forms

A *date form* represents a date and possibly an associated time. Simple date arithmetic is supported: Adding a number to a date produces a new date shifted by that many days; adding an HMS form to a date shifts it by that many hours. Subtracting two date forms computes the number of days between them (represented as a simple number). Many other operations, such as multiplying two date forms, are nonsensical and are not allowed by Calc.

Date forms are entered and displayed enclosed in '`< >`' brackets. The default format is, e.g., '`<Wed Jan 9, 1991>`' for dates, or '`<3:32:20pm Wed Jan 9, 1991>`' for dates with times. Input is flexible; date forms can be entered in any of the usual notations for dates and times. See *XXX* [Date Formats], page *XXX*.

Date forms are stored internally as numbers, specifically the number of days since midnight on the morning of January 1 of the year 1 AD. If the internal number is an integer, the form represents a date only; if the internal number is a fraction or float, the form represents a date and time. For example, '`<6:00am Wed Jan 9, 1991>`' is represented by the number 726842.25. The standard precision of 12 decimal digits is enough to ensure that a (reasonable) date and time can be stored without roundoff error.

If the current precision is greater than 12, date forms will keep additional digits in the seconds position. For example, if the precision is 15, the seconds will keep three digits after the decimal point. Decreasing the precision below 12 may cause the time part of a date form to become inaccurate. This can also happen if astronomically high years are used, though this will not be an issue in everyday (or even everymillenium) use. Note that date forms without times are stored as exact integers, so roundoff is never an issue for them.

You can use the *`v p`* (`calc-pack`) and *`v u`* (`calc-unpack`) commands to get at the numerical representation of a date form. See *XXX* [Packing and Unpacking], page *XXX*.

Date forms can go arbitrarily far into the future or past. Negative year numbers represent years BC. Calc uses a combination of the Gregorian and Julian calendars, following the history of Great Britain and the British colonies. This is the same calendar that is used by the `cal` program in most Unix implementations.

Some historical background: The Julian calendar was created by Julius Caesar in the year 46 BC as an attempt to fix the gradual drift caused by the lack of leap years in the calendar used until that time. The Julian calendar introduced an extra day in all years divisible by four. After some initial confusion, the calendar was adopted around the year we call 8 AD. Some centuries later it became apparent that the Julian year of 365.25 days was itself not quite right. In 1582 Pope Gregory XIII introduced the Gregorian calendar, which added the new rule that years divisible by 100, but not by 400, were not to be considered leap years despite being divisible by four. Many countries delayed adoption of the Gregorian calendar because of religious differences; in Britain it was put off until the year 1752, by which time the Julian calendar had fallen eleven days behind the true seasons. So the switch to the Gregorian calendar in early September 1752 introduced a discontinuity: The day after Sep 2, 1752 is Sep 14, 1752. Calc follows this convention. To take another example, Russia waited until 1918 before adopting the new calendar, and thus needed to remove thirteen days (between Feb 1, 1918 and Feb 14, 1918). This means that Calc's reckoning will be inconsistent with Russian history between 1752 and 1918, and similarly for various other countries.

Today's timekeepers introduce an occasional "leap second" as well, but Calc does not take these minor effects into account. (If it did, it would have to report a non-integer number of days between, say, '`<12:00am Mon Jan 1, 1900>`' and '`<12:00am Sat Jan 1, 2000>`'.)

Calc uses the Julian calendar for all dates before the year 1752, including dates BC when the Julian calendar technically had not yet been invented. Thus the claim that day number -10000 is called "August 16, 28 BC" should be taken with a grain of salt.

Please note that there is no "year 0"; the day before '`<Sat Jan 1, +1>`' is '`<Fri Dec 31, -1>`'. These are days 0 and -1 respectively in Calc's internal numbering scheme.

Another day counting system in common use is, confusingly, also called "Julian." It was invented in 1583 by Joseph Justus Scaliger, who named it in honor of his father Julius Caesar Scaliger. For obscure reasons he chose to start his day numbering on Jan 1, 4713 BC at noon, which in Calc's scheme is -1721423.5 (recall that Calc starts at midnight instead of noon). Thus to convert a Calc date code obtained by unpacking a date form into a Julian day number, simply add 1721423.5. The Julian code for '`6:00am Jan 9, 1991`' is 2448265.75. The built-in *`t J`* command performs this conversion for you.

The Unix operating system measures time as an integer number of seconds since midnight, Jan 1, 1970. To convert a Calc date value into a Unix time stamp, first subtract 719164 (the code for '`<Jan 1, 1970>`'), then multiply by 86400 (the number of seconds in a day) and press *`R`* to round to the nearest integer. If you have a date form, you can simply subtract the day '`<Jan 1, 1970>`' instead of unpacking and subtracting 719164. Likewise, divide by 86400 and add '`<Jan 1, 1970>`' to convert from Unix time to a Calc date form. (Note that Unix normally maintains the time in the GMT time zone; you may need to subtract five hours to get New York time, or eight hours for California time. The same is usually true of Julian day counts.) The built-in *`t U`* command performs these conversions.

4.10 Modulo Forms

A *modulo form* is a real number which is taken modulo (i.e., within an integer multiple of) some value M. Arithmetic modulo M often arises in number theory. Modulo forms are written 'a `mod` M', where a and M are real numbers or HMS forms, and $0 \le a < M$. In many applications a and M will be integers but this is not required.

Modulo forms are not to be confused with the modulo operator '`%`'. The expression '`27 % 10`' means to compute 27 modulo 10 to produce the result 7. Further computations treat this 7 as just a regular integer. The expression '`27 mod 10`' produces the result '`7 mod 10`'; further computations with this value are again reduced modulo 10 so that the result always lies in the desired range.

When two modulo forms with identical M's are added or multiplied, the Calculator simply adds or multiplies the values, then reduces modulo M. If one argument is a modulo form and the other a plain number, the plain number is treated like a compatible modulo form. It is also possible to raise modulo forms to powers; the result is the value raised to the power, then reduced modulo M. (When all values involved are integers, this calculation is done much more efficiently than actually computing the power and then reducing.)

Two modulo forms 'a `mod` M' and 'b `mod` M' can be divided if a, b, and M are all integers. The result is the modulo form which, when multiplied by 'b `mod` M', produces 'a `mod` M'. If there is no solution to this equation (which can happen only when M is non-prime), or if any of the arguments are non-integers, the division is left in symbolic form. Other operations, such as square roots, are not yet supported for modulo forms. (Note that, although '`(`a `mod` M`)^.5`' will compute a "modulo square root" in the sense of reducing $\sqrt{a}$ modulo M, this is not a useful definition from the number-theoretical point of view.)

`M`
`mod`
To create a modulo form during numeric entry, press the shift-*`M`* key to enter the word '`mod`'. As a special convenience, pressing shift-*`M`* a second time automatically enters the value of M that was most recently used before. During algebraic entry, either type '`mod`' by hand or press *`M-m`* (that's *`META-m`*). Once again, pressing this a second time enters the current modulo.

You can also use *`v p`* and *`%`* to modify modulo forms. See *XXX* [Building Vectors], page *XXX*. See *XXX* [Basic Arithmetic], page *XXX*.

It is possible to mix HMS forms and modulo forms. For example, an HMS form modulo 24 could be used to manipulate clock times; an HMS form modulo 360 would be suitable for angles. Making the modulo M also be an HMS form eliminates troubles that would arise if the angular mode were inadvertently set to Radians, in which case '`2@ 0' 0" mod 24`' would be interpreted as two degrees modulo 24 radians!

Modulo forms cannot have variables or formulas for components. If you enter the formula '`(x + 2) mod 5`', Calc propagates the modulus to each of the coefficients: '`(1 mod 5) x + (2 mod 5)`'.

`makemod`
The algebraic function '`makemod(a, m)`' builds the modulo form '`a mod m`'.

4.11 Error Forms

An *error form* is a number with an associated standard deviation, as in '`2.3 +/- 0.12`'. The notation 'x `+/-` σ' stands for an uncertain value which follows a normal or Gaussian distribution of mean x and standard deviation or "error" σ. Both the mean and the error can be either numbers or formulas. Generally these are real numbers but the mean may also be complex. If the error is negative or complex, it is changed to its absolute value. An error form with zero error is converted to a regular number by the Calculator.

All arithmetic and transcendental functions accept error forms as input. Operations on the mean-value part work just like operations on regular numbers. The error part for any function $f(x)$ (such as $\sin x$) is defined by the error of x times the derivative of f evaluated at the mean value of x. For a two-argument function $f(x,y)$ (such as addition) the error is the square root of the sum of the squares of the errors due to x and y.

$$f(x \texttt{ +/- } \sigma) = f(x) \texttt{ +/- } \sigma \left| \frac{df(x)}{dx} \right|$$

$$f(x \texttt{ +/- } \sigma_x, y \texttt{ +/- } \sigma_y) = f(x,y) \texttt{ +/- } \sqrt{\left(\sigma_x \left| \frac{\partial f(x,y)}{\partial x} \right| \right)^2 + \left(\sigma_y \left| \frac{\partial f(x,y)}{\partial y} \right| \right)^2}$$

Note that this definition assumes the errors in x and y are uncorrelated. A side effect of this definition is that '`(2 +/- 1) * (2 +/- 1)`' is not the same as '`(2 +/- 1)^2`'; the former represents the product of two independent values which happen to have the same probability distributions, and the latter is the product of one random value with itself. The former will produce an answer with less error, since on the average the two independent errors can be expected to cancel out.

Consult a good text on error analysis for a discussion of the proper use of standard deviations. Actual errors often are neither Gaussian-distributed nor uncorrelated, and the above formulas are valid only when errors are small. As an example, the error arising from '`sin(`x `+/-` σ`)`' is 'σ `abs(cos(`x`))`'. When x is close to zero, $\cos x$ is close to one so the error in the sine is close to σ; this makes sense, since $\sin x$ is approximately x near zero, so a given error in x will produce about the same error in the sine. Likewise, near 90 degrees $\cos x$ is nearly zero and so the computed error is small: The sine curve is nearly flat in that region, so an error in x has relatively little effect on the value of $\sin x$. However, consider '`sin(90 +/- 1000)`'. The cosine of 90 is zero, so Calc will report zero error! We get an obviously wrong result because we have violated the small-error approximation underlying the error analysis. If the error in x had been small, the error in $\sin x$ would indeed have been negligible.

To enter an error form during regular numeric entry, use the *p* ("plus-or-minus") key to type the '`+/-`' symbol. (If you try actually typing '`+/-`' the `+` key will be interpreted as the Calculator's `+` command!) Within an algebraic formula, you can press *M-p* to type the '`+/-`' symbol, or type it out by hand. `p` `+/-`

Error forms and complex numbers can be mixed; the formulas shown above are used for complex numbers, too; note that if the error part evaluates to a complex number its absolute value (or the square root of the sum of the squares of the absolute values of the two error contributions) is used. Mathematically, this corresponds to a radially symmetric Gaussian distribution of numbers on the complex plane. However, note that Calc considers an error form with real components to represent a real number, not a complex distribution around a real mean.

Error forms may also be composed of HMS forms. For best results, both the mean and the error should be HMS forms if either one is.

The algebraic function '`sdev(a, b)`' builds the error form '`a +/- b`'. `sdev`

4.12 Interval Forms

An *interval* is a subset of consecutive real numbers. For example, the interval '`[2 .. 4]`' represents all the numbers from 2 to 4, inclusive. If you multiply it by the interval '`[0.5 .. 2]`' you obtain '`[1 .. 8]`'. This calculation represents the fact that if you multiply some number in the range '`[2 .. 4]`' by some other number in the range '`[0.5 .. 2]`', your result will lie in the range from 1 to 8. Interval arithmetic is used to get a worst-case estimate of the possible range of values a computation will produce, given the set of possible values of the input.

Calc supports several varieties of intervals, including *closed* intervals of the type shown above, *open* intervals such as '`(2 .. 4)`', which represents the range of numbers from 2 to 4 *exclusive*, and *semi-open* intervals in which one end uses a round parenthesis and the other a square bracket. In mathematical terms,

$$\begin{array}{rcl} [2..4] & \text{means} & 2 \le x \le 4 \\ [2..4) & \text{means} & 2 \le x < 4 \\ (2..4] & \text{means} & 2 < x \le 4 \\ (2..4) & \text{means} & 2 < x < 4 \end{array}$$

The lower and upper limits of an interval must be either real numbers (or HMS or date forms), or symbolic expressions which are assumed to be real-valued, or '`-inf`' and '`inf`'. In general the lower limit must be less than the upper limit. A closed interval containing only one value, '`[3 .. 3]`', is converted to a plain number (3) automatically. An interval containing no values at all (such as '`[3 .. 2]`' or '`[2 .. 2)`') can be represented but is not guaranteed to behave well when used in arithmetic. Note that the interval '`[3 .. inf)`' represents all real numbers greater than or equal to 3, and '`(-inf .. inf)`' represents all real numbers. In fact, '`[-inf .. inf]`' represents all real numbers including the real infinities.

Intervals are entered in the notation shown here, either as algebraic formulas, or using incomplete forms. (See *XXX* [Incomplete Objects], page *XXX*.) In algebraic formulas, multiple periods in a row are collected from left to right, so that '`1...1e2`' is interpreted as '`1.0 .. 1e2`' rather than '`1 .. 0.1e2`'. Add spaces or zeros if you want to get the other interpretation. If you omit the lower or upper limit, a default of '`-inf`' or '`inf`' (respectively) is furnished.

"Infinite mode" also affects operations on intervals (see *XXX* [Infinities], page *XXX*). Calc will always introduce an open infinity, as in '`1 / (0 .. 2] = [0.5 .. inf)`'. But closed infinities, '`1 / [0 .. 2] = [0.5 .. inf]`', arise only in infinite mode; otherwise they are left unevaluated. Note that the "direction" of a zero is not an issue in this case since the zero is always assumed to be continuous with the rest of the interval. For intervals that contain zero inside them Calc is forced to give the result, '`1 / (-2 .. 2) = [-inf .. inf]`'.

While it may seem that intervals and error forms are similar, they are based on entirely different concepts of inexact quantities. An error form 'x `+/-` σ' means a variable is random, and its value could be anything but is "probably" within one σ of the mean value x. An interval '`[`a `..` b`]`' means a variable's value is unknown, but guaranteed to lie in the specified range. Error forms are statistical or "average case" approximations; interval arithmetic tends to produce "worst case" bounds on an answer.

Intervals may not contain complex numbers, but they may contain HMS forms or date forms.

See *XXX* [Set Operations], page *XXX*, for commands that interpret interval forms as subsets of the set of real numbers.

The algebraic function '`intv(n, a, b)`' builds an interval form from '`a`' to '`b`'; '`n`' is an integer code which must be 0 for '`(..)`', 1 for '`(..]`', 2 for '`[..)`', or 3 for '`[..]`'. `intv`

Please note that in fully rigorous interval arithmetic, care would be taken to make sure that the computation of the lower bound rounds toward minus infinity, while upper bound computations round toward plus infinity. Calc's arithmetic always uses a round-to-nearest mode, which means that roundoff errors could creep into an interval calculation to produce intervals slightly smaller than they ought to be. For example, entering '`[1..2]`' and pressing *Q 2 ^* should yield the interval '`[1..2]`' again, but in fact it yields the (slightly too small) interval '`[1..1.9999999]`' due to roundoff error.

4.13 Incomplete Objects

When *(* or *[* is typed to begin entering a complex number or vector, respectively, the effect is to push an *incomplete* complex number or vector onto the stack. The *,* key adds the value(s) at the top of the stack onto the current incomplete object. The *)* and *]* keys "close" the incomplete object after adding any values on the top of the stack in front of the incomplete object. `[ ]` `( )` `,`

As a result, the sequence of keystrokes *[2 , 3 RET 2 * , 9]* pushes the vector '`[2, 6, 9]`' onto the stack. Likewise, *(1 , 2 Q)* pushes the complex number '`(1, 1.414)`' (approximately).

If several values lie on the stack in front of the incomplete object, all are collected and appended to the object. Thus the *,* key is redundant: *[2 RET 3 RET 2 * 9]*. Some people prefer the equivalent *SPC* key to *RET*.

As a special case, typing *,* immediately after *(*, *[*, or *,* adds a zero or duplicates the preceding value in the list being formed. Typing *DEL* during incomplete entry removes the last item from the list.

The *;* key is used in the same way as *,* to create polar complex numbers: *(1 ; 2)*. When entering a vector, *;* is useful for creating a matrix. In particular, *[[1 , 2 ; 3 , 4 ; 5 , 6]]* is equivalent to *[[1 , 2] , [3 , 4] , [5 , 6]]*. `;`

Incomplete entry is also used to enter intervals. For example, *[2 .. 4)* enters a semi-open interval. Note that when you type the first period, it will be interpreted as a decimal point, but when you type a second period immediately afterward, it is re-interpreted as part of the interval symbol. Typing *..* corresponds to executing the `calc-dots` command. `..`

If you find incomplete entry distracting, you may wish to enter vectors and complex numbers as algebraic formulas by pressing the apostrophe key.

4.14 Variables

A *variable* is somewhere between a storage register on a conventional calculator, and a variable in a programming language. (In fact, a Calc variable is really just an Emacs Lisp variable that contains a Calc number or formula.) A variable's name is normally composed of letters and digits. Calc also allows apostrophes and # signs in variable names. The Calc variable `foo` corresponds to the Emacs Lisp variable `var-foo`. Commands like *s s* (`calc-store`) that operate on variables can be made to use any arbitrary Lisp variable simply by backspacing over the '`var-`' prefix in the minibuffer.

In a command that takes a variable name, you can either type the full name of a variable, or type a single digit to use one of the special convenience variables `var-q0` through `var-q9`. For example, *3 s s 2* stores the number 3 in variable `var-q2`, and *3 s s foo RET* stores that number in variable `var-foo`.

To push a variable itself (as opposed to the variable's value) on the stack, enter its name as an algebraic expression using the apostrophe (*'*) key. Variable names in algebraic formulas implicitly have '`var-`' prefixed to their names. The '`#`' character in variable names used in algebraic formulas corresponds to a dash '`-`' in the Lisp variable name. If the name contains any dashes, the prefix '`var-`' is *not* automatically added. Thus the two formulas '`foo + 1`' and '`var#foo + 1`' both refer to the same variable.

= The = (`calc-evaluate`) key "evaluates" a formula by replacing all variables in the formula which have been given values by a `calc-store` or `calc-let` command by their stored values. Other variables are left alone. Thus a variable that has not been stored acts like an abstract variable in algebra; a variable that has been stored acts more like a register in a traditional calculator. With a positive numeric prefix argument, = evaluates the top *n* stack entries; with a negative argument, = evaluates the *n*th stack entry.

A few variables are called *special constants*. Their names are '`e`', '`pi`', '`i`', '`phi`', and '`gamma`'. (See *XXX* [Scientific Functions], page *XXX*.) When they are evaluated with =, their values are calculated if necessary according to the current precision or complex polar mode. If you wish to use these symbols for other purposes, simply undefine or redefine them using `calc-store`.

The variables '`inf`', '`uinf`', and '`nan`' stand for infinite or indeterminate values. It's best not to use them as regular variables, since Calc uses special algebraic rules when it manipulates them. Calc displays a warning message if you store a value into any of these special variables.

See *XXX* [Store and Recall], page *XXX*, for a discussion of commands dealing with variables.

4.15 Formulas

When you press the apostrophe key you may enter any expression or formula in algebraic form. (Calc uses the terms "expression" and "formula" interchangeably.) An expression is built up of numbers, variable names, and function calls, combined with various arithmetic operators. Parentheses may be used to indicate grouping. Spaces are ignored within formulas, except that spaces are not permitted within variable names or numbers. Arithmetic operators, in order from highest to lowest precedence, and with their equivalent function names, are:

'`_`' [`subscr`] (subscripts);

postfix '`%`' [`percent`] (as in '`25% = 0.25`');

prefix '`+`' and '`-`' [`neg`] (as in '`-x`') and prefix '`!`' [`lnot`] (logical "not," as in '`!x`');

'`+/-`' [`sdev`] (the standard deviation symbol) and '`mod`' [`makemod`] (the symbol for modulo forms);

postfix '`!`' [`fact`] (factorial, as in '`n!`') and postfix '`!!`' [`dfact`] (double factorial);

'`^`' [`pow`] (raised-to-the-power-of);

'`*`' [`mul`];

'`/`' [`div`], '`%`' [`mod`] (modulo), and '`\`' [`idiv`] (integer division);

infix '`+`' [`add`] and '`-`' [`sub`] (as in '`x-y`');

'|' [vconcat] (vector concatenation);

relations '=' [eq], '!=' [neq], '<' [lt], '>' [gt], '<=' [leq], and '>=' [geq];

'&&' [land] (logical "and");

'||' [lor] (logical "or");

the C-style "if" operator 'a?b:c' [if];

'!!!' [pnot] (rewrite pattern "not");

'&&&' [pand] (rewrite pattern "and");

'|||' [por] (rewrite pattern "or");

':=' [assign] (for assignments and rewrite rules);

'::' [condition] (rewrite pattern condition);

'=>' [evalto].

Note that, unlike in usual computer notation, multiplication binds more strongly than division: 'a*b/c*d' is equivalent to $\frac{ab}{cd}$.

The multiplication sign '*' may be omitted in many cases. In particular, if the righthand side is a number, variable name, or parenthesized expression, the '*' may be omitted. Implicit multiplication has the same precedence as the explicit '*' operator. The one exception to the rule is that a variable name followed by a parenthesized expression, as in 'f(x)', is interpreted as a function call, not an implicit '*'. In many cases you must use a space if you omit the '*': '2a' is the same as '2*a', and 'a b' is the same as 'a*b', but 'ab' is a variable called ab, *not* the product of 'a' and 'b'! Also note that 'f (x)' is still a function call.

The rules are slightly different for vectors written with square brackets. In vectors, the space character is interpreted (like the comma) as a separator of elements of the vector. Thus '[2a b+c d]' is equivalent to '[2*a, b+c, d]', whereas '2a b+c d' is equivalent to '2*a*b + c*d'. Note that spaces around the brackets, and around explicit commas, are ignored. To force spaces to be interpreted as multiplication you can enclose a formula in parentheses as in '[(a b) 2(c d)]', which is interpreted as '[a*b, 2*c*d]'. An implicit comma is also inserted between '][', as in the matrix '[[1 2][3 4]]'.

Vectors that contain commas (not embedded within nested parentheses or brackets) do not treat spaces specially: '[a b, 2 c d]' is a vector of two elements. Also, if it would be an error to treat spaces as separators, but not otherwise, then Calc will ignore spaces: '[a - b]' is a vector of one element, but '[a -b]' is a vector of two elements. Finally, vectors entered with curly braces instead of square brackets do not give spaces any special treatment. When Calc displays a vector that does not contain any commas, it will insert parentheses if necessary to make the meaning clear: '[(a b)]'.

The expression '5%-2' is ambiguous; is this five-percent minus two, or five modulo minus-two? Calc always interprets the leftmost symbol as an infix operator preferentially (modulo, in this case), so you would need to write '(5%)-2' to get the former interpretation.

A function call is, e.g., 'sin(1+x)'. Function names follow the same rules as variable names except that the default prefix 'calcFunc-' is used (instead of 'var-') for the internal Lisp form. Most mathematical Calculator commands like calc-sin have function equivalents like sin. If no Lisp function is defined for a function called by a formula, the call is left as it is during algebraic manipulation: 'f(x+y)' is left alone. Beware that many innocent-looking short names like in and re have predefined meanings which could surprise you; however, single letters or single letters followed by digits are always safe to use for your own function names. See *XXX* [Function Index], page *XXX*.

In the documentation for particular commands, the notation ***H S*** (`calc-sinh`) [`sinh`] means that the key sequence ***H S***, the command ***M-x calc-sinh***, and the algebraic function `sinh(x)` all represent the same operation.

Commands that interpret ("parse") text as algebraic formulas include algebraic entry (***'***), editing commands like ***`*** which parse the contents of the editing buffer when you finish, the ***M-# g*** and ***M-# r*** commands, the ***C-y*** command, the X window system "paste" mouse operation, and Embedded Mode. All of these operations use the same rules for parsing formulas; in particular, language modes (see *XXX* [Language Modes], page *XXX*) affect them all in the same way.

When you read a large amount of text into the Calculator (say a vector which represents a big set of rewrite rules; see *XXX* [Rewrite Rules], page *XXX*), you may wish to include comments in the text. Calc's formula parser ignores the symbol '`%%`' and anything following it on a line:

```
[ a + b,   %% the sum of "a" and "b"
  c + d,
  %% last line is coming up:
  e + f ]
```

This is parsed exactly the same as '`[ a + b, c + d, e + f ]`'.

See *XXX* [Syntax Tables], page *XXX*, for a way to create your own operators and other input notations. See *XXX* [Compositions], page *XXX*, for a way to create new display formats.

See *XXX* [Algebra], page *XXX*, for commands for manipulating formulas symbolically.

5 Stack and Trail Commands

This chapter describes the Calc commands for manipulating objects on the stack and in the trail buffer. (These commands operate on objects of any type, such as numbers, vectors, formulas, and incomplete objects.)

5.1 Stack Manipulation Commands

To duplicate the top object on the stack, press *RET* or *SPC* (two equivalent keys for the `calc-enter` command). Given a positive numeric prefix argument, these commands duplicate several elements at the top of the stack. Given a negative argument, these commands duplicate the specified element of the stack. Given an argument of zero, they duplicate the entire stack. For example, with '`10 20 30`' on the stack, *RET* creates '`10 20 30 30`', *C-u 2 RET* creates '`10 20 30 20 30`', *C-u - 2 RET* creates '`10 20 30 20`', and *C-u 0 RET* creates '`10 20 30 10 20 30`'. `RET` `SPC`

The *LFD* (`calc-over`) command (on a key marked Line-Feed if you have it, else on *C-j*) is like `calc-enter` except that the sign of the numeric prefix argument is interpreted oppositely. Also, with no prefix argument the default argument is 2. Thus with '`10 20 30`' on the stack, *LFD* and *C-u 2 LFD* are both equivalent to *C-u - 2 RET*, producing '`10 20 30 20`'. `LFD`

To remove the top element from the stack, press *DEL* (`calc-pop`). The *C-d* key is a synonym for *DEL*. (If the top element is an incomplete object with at least one element, the last element is removed from it.) Given a positive numeric prefix argument, several elements are removed. Given a negative argument, the specified element of the stack is deleted. Given an argument of zero, the entire stack is emptied. For example, with '`10 20 30`' on the stack, *DEL* leaves '`10 20`', *C-u 2 DEL* leaves '`10`', *C-u - 2 DEL* leaves '`10 30`', and *C-u 0 DEL* leaves an empty stack. `DEL` `C-d`

The *M-DEL* (`calc-pop-above`) command is to *DEL* what *LFD* is to *RET*: It interprets the sign of the numeric prefix argument in the opposite way, and the default argument is 2. Thus *M-DEL* by itself removes the second-from-top stack element, leaving the first, third, fourth, and so on; *M-3 M-DEL* deletes the third stack element. `M-DEL`

To exchange the top two elements of the stack, press *TAB* (`calc-roll-down`). Given a positive numeric prefix argument, the specified number of elements at the top of the stack are rotated downward. Given a negative argument, the entire stack is rotated downward the specified number of times. Given an argument of zero, the entire stack is reversed top-for-bottom. For example, with '`10 20 30 40 50`' on the stack, *TAB* creates '`10 20 30 50 40`', *C-u 3 TAB* creates '`10 20 50 30 40`', *C-u - 2 TAB* creates '`40 50 10 20 30`', and *C-u 0 TAB* creates '`50 40 30 20 10`'. `TAB`

The command *M-TAB* (`calc-roll-up`) is analogous to *TAB* except that it rotates upward instead of downward. Also, the default with no prefix argument is to rotate the top 3 elements. For example, with '`10 20 30 40 50`' on the stack, *M-TAB* creates '`10 20 40 50 30`', *C-u 4 M-TAB* creates '`10 30 40 50 20`', *C-u - 2 M-TAB* creates '`30 40 50 10 20`', and *C-u 0 M-TAB* creates '`50 40 30 20 10`'. `M-TAB`

A good way to view the operation of *TAB* and *M-TAB* is in terms of moving a particular element to a new position in the stack. With a positive argument n, *TAB* moves the top stack element down to level n, making room for it by pulling all the intervening stack elements toward the top. *M-TAB* moves the element at level n up to the top. (Compare with *LFD*, which copies instead of moving the element in level n.)

With a negative argument $-n$, *TAB* rotates the stack to move the object in level n to the deepest place in the stack, and the object in level $n+1$ to the top. *M-TAB* rotates the deepest stack element to be in level n, also putting the top stack element in level $n+1$.

See *XXX* [Selecting Subformulas], page *XXX*, for a way to apply these commands to any portion of a vector or formula on the stack.

5.2 Editing Stack Entries

`

The backquote, *`* (`calc-edit`) command creates a temporary buffer (‘`*Calc Edit*`’) for editing the top-of-stack value using regular Emacs commands. With a numeric prefix argument, it edits the specified number of stack entries at once. (An argument of zero edits the entire stack; a negative argument edits one specific stack entry.)

When you are done editing, press *M-# M-#* to finish and return to Calc. The *RET* and *LFD* keys also work to finish most sorts of editing, though in some cases Calc leaves *RET* with its usual meaning (“insert a newline”) if it’s a situation where you might want to insert new lines into the editing buffer. The traditional Emacs “finish” key sequence, *C-c C-c*, also works to finish editing and may be easier to type, depending on your keyboard.

When you finish editing, the Calculator parses the lines of text in the ‘`*Calc Edit*`’ buffer as numbers or formulas, replaces the original stack elements in the original buffer with these new values, then kills the ‘`*Calc Edit*`’ buffer. The original Calculator buffer continues to exist during editing, but for best results you should be careful not to change it until you have finished the edit. You can also cancel the edit by pressing *M-#* `x`.

The formula is normally reevaluated as it is put onto the stack. For example, editing ‘`a + 2`’ to ‘`3 + 2`’ and pressing *M-# M-#* will push 5 on the stack. If you use *LFD* to finish, Calc will put the result on the stack without evaluating it.

If you give a prefix argument to *M-# M-#* (or *C-c C-c*), Calc will not kill the ‘`*Calc Edit*`’ buffer. You can switch back to that buffer and continue editing if you wish. However, you should understand that if you initiated the edit with *`*, the *M-# M-#* operation will be programmed to replace the top of the stack with the new edited value, and it will do this even if you have rearranged the stack in the meanwhile. This is not so much of a problem with other editing commands, though, such as *s* `e` (`calc-edit-variable`; see *XXX* [Operations on Variables], page *XXX*).

If the `calc-edit` command involves more than one stack entry, each line of the ‘`*Calc Edit*`’ buffer is interpreted as a separate formula. Otherwise, the entire buffer is interpreted as one formula, with line breaks ignored. (You can use *C-o* or *C-q C-j* to insert a newline in the buffer without pressing *RET*.)

The *`* key also works during numeric or algebraic entry. The text entered so far is moved to the `*Calc Edit*` buffer for more extensive editing than is convenient in the minibuffer.

5.3 Trail Commands

The commands for manipulating the Calc Trail buffer are two-key sequences beginning with the *t* prefix.

The *t d* (`calc-trail-display`) command turns display of the trail on and off. Normally the trail display is toggled on if it was off, off if it was on. With a numeric prefix of zero, this command always turns the trail off; with a prefix of one, it always turns the trail on. The other trail-manipulation commands described here automatically turn the trail on. Note that when the trail is off values are still recorded there; they are simply not displayed. To set Emacs to turn the trail off by default, type *t d* and then save the mode settings with *m m* (`calc-save-modes`). `t d`

The *t i* (`calc-trail-in`) and *t o* (`calc-trail-out`) commands switch the cursor into and out of the Calc Trail window. In practice they are rarely used, since the commands shown below are a more convenient way to move around in the trail, and they work "by remote control" when the cursor is still in the Calculator window. `t i` `t o`

There is a *trail pointer* which selects some entry of the trail at any given time. The trail pointer looks like a '`>`' symbol right before the selected number. The following commands operate on the trail pointer in various ways.

The *t y* (`calc-trail-yank`) command reads the selected value in the trail and pushes it onto the Calculator stack. It allows you to re-use any previously computed value without retyping. With a numeric prefix argument *n*, it yanks the value *n* lines above the current trail pointer. `t y`

The *t <* (`calc-trail-scroll-left`) and *t >* (`calc-trail-scroll-right`) commands horizontally scroll the trail window left or right by one half of its width. `t <` `t >`

The *t n* (`calc-trail-next`) and *t p* (`calc-trail-previous`) commands move the trail pointer down or up one line. The *t f* (`calc-trail-forward`) and *t b* (`calc-trail-backward`) commands move the trail pointer down or up one screenful at a time. All of these commands accept numeric prefix arguments to move several lines or screenfuls at a time. `t n` `t p` `t f` `t b`

The *t [* (`calc-trail-first`) and *t]* (`calc-trail-last`) commands move the trail pointer to the first or last line of the trail. The *t h* (`calc-trail-here`) command moves the trail pointer to the cursor position; unlike the other trail commands, *t h* works only when Calc Trail is the selected window. `t [` `t ]` `t h`

The *t s* (`calc-trail-isearch-forward`) and *t r* (`calc-trail-isearch-backward`) commands perform an incremental search forward or backward through the trail. You can press *RET* to terminate the search; the trail pointer moves to the current line. If you cancel the search with *C-g*, the trail pointer stays where it was when the search began. `t s` `t r`

The *t m* (`calc-trail-marker`) command allows you to enter a line of text of your own choosing into the trail. The text is inserted after the line containing the trail pointer; this usually means it is added to the end of the trail. Trail markers are useful mainly as the targets for later incremental searches in the trail. `t m`

The *t k* (`calc-trail-kill`) command removes the selected line from the trail. The line is saved in the Emacs kill ring suitable for yanking into another buffer, but it is not easy to yank the text back into the trail buffer. With a numeric prefix argument, this command kills the *n* lines below or above the selected one. `t k`

The *t .* (`calc-full-trail-vectors`) command is described elsewhere; see *XXX* [Vector and Matrix Formats], page *XXX*.

5.4 Keep Arguments

`K`

The *K* (`calc-keep-args`) command acts like a prefix for the following command. It prevents that command from removing its arguments from the stack. For example, after *2 RET 3 +*, the stack contains the sole number 5, but after *2 RET 3 K +*, the stack contains the arguments and the result: '`2 3 5`'.

This works for all commands that take arguments off the stack. As another example, *K* `a` *s* simplifies a formula, pushing the simplified version of the formula onto the stack after the original formula (rather than replacing the original formula).

Note that you could get the same effect by typing *RET* `a` *s*, copying the formula and then simplifying the copy. One difference is that for a very large formula the time taken to format the intermediate copy in *RET* `a` *s* could be noticeable; *K* `a` *s* would avoid this extra work.

Even stack manipulation commands are affected. *TAB* works by popping two values and pushing them back in the opposite order, so *2 RET 3 K TAB* produces '`2 3 3 2`'.

A few Calc commands provide other ways of doing the same thing. For example, *' sin($)* replaces the number on the stack with its sine using algebraic entry; to push the sine and keep the original argument you could use either *' sin($1)* or *K ' sin($)*. See *XXX* [Algebraic Entry], page *XXX*. Also, the *s s* command is effectively the same as *K s t*. See *XXX* [Storing Variables], page *XXX*.

Keyboard macros may interact surprisingly with the *K* prefix. If you have defined a keyboard macro to be, say, '`Q +`' to add one number to the square root of another, then typing *K X* will execute *K Q +*, probably not what you expected. The *K* prefix will apply to just the first command in the macro rather than the whole macro.

If you execute a command and then decide you really wanted to keep the argument, you can press *M-RET* (`calc-last-args`). This command pushes the last arguments that were popped by any command onto the stack. Note that the order of things on the stack will be different than with *K*: *2 RET 3 + M-RET* leaves '`5 2 3`' on the stack instead of '`2 3 5`'. See *XXX* [Undo], page *XXX*.

6 Mode Settings

This chapter describes commands that set modes in the Calculator. They do not affect the contents of the stack, although they may change the *appearance* or *interpretation* of the stack's contents.

6.1 General Mode Commands

You can save all of the current mode settings in your '`.emacs`' file with the *m m* (`calc-save-modes`) command. This will cause Emacs to reestablish these modes each time it starts up. The modes saved in the file include everything controlled by the *m* and *d* prefix keys, the current precision and binary word size, whether or not the trail is displayed, the current height of the Calc window, and more. The current interface (used when you type *M-# M-#*) is also saved. If there were already saved mode settings in the file, they are replaced. Otherwise, the new mode information is appended to the end of the file. `m m`

The *m R* (`calc-mode-record-mode`) command tells Calc to record the new mode settings (as if by pressing *m m*) every time a mode setting changes. If Embedded Mode is enabled, other options are available; see *XXX* [Mode Settings in Embedded Mode], page *XXX*. `m R`

The *m F* (`calc-settings-file-name`) command allows you to choose a different place than your '`.emacs`' file for *m m*, *Z P*, and similar commands to save permanent information. You are prompted for a file name. All Calc modes are then reset to their default values, then settings from the file you named are loaded if this file exists, and this file becomes the one that Calc will use in the future for commands like *m m*. The default settings file name is '`~/.emacs`'. You can see the current file name by giving a blank response to the *m F* prompt. See also the discussion of the `calc-settings-file` variable; see *XXX* [Installation], page *XXX*. `m F`

If the file name you give contains the string '`.emacs`' anywhere inside it, *m F* will not automatically load the new file. This is because you are presumably switching to your '`~/.emacs`' file, which may contain other things you don't want to reread. You can give a numeric prefix argument of 1 to *m F* to force it to read the file no matter what its name. Conversely, an argument of −1 tells *m F* *not* to read the new file. An argument of 2 or −2 tells *m F* not to reset the modes to their defaults beforehand, which is useful if you intend your new file to have a variant of the modes present in the file you were using before.

The *m x* (`calc-always-load-extensions`) command enables a mode in which the first use of Calc loads the entire program, including all extensions modules. Otherwise, the extensions modules will not be loaded until the various advanced Calc features are used. Since this mode only has effect when Calc is first loaded, *m x* is usually followed by *m m* to make the mode-setting permanent. To load all of Calc just once, rather than always in the future, you can press *M-# L*. `m x`

The *m S* (`calc-shift-prefix`) command enables a mode in which all of Calc's letter prefix keys may be typed shifted as well as unshifted. If you are typing, say, *a S* (`calc-solve-for`) quite often you might find it easier to turn this mode on so that you can type *A S* instead. When this mode is enabled, the commands that used to be on those single shifted letters (e.g., *A* (`calc-abs`)) can now be invoked by pressing the shifted letter twice: *A A*. Note that the *v* prefix key always works both shifted and unshifted, and the *z* and *Z* prefix keys are always distinct. Also, the *h* prefix is not affected by this mode. Press *m S* again to disable shifted-prefix mode. `m S`

6.2 Precision

p The *p* (`calc-precision`) command controls the precision to which floating-point calculations are carried. The precision must be at least 3 digits and may be arbitrarily high, within the limits of memory and time. This affects only floats: Integer and rational calculations are always carried out with as many digits as necessary.

The *p* key prompts for the current precision. If you wish you can instead give the precision as a numeric prefix argument.

Many internal calculations are carried to one or two digits higher precision than normal. Results are rounded down afterward to the current precision. Unless a special display mode has been selected, floats are always displayed with their full stored precision, i.e., what you see is what you get. Reducing the current precision does not round values already on the stack, but those values will be rounded down before being used in any calculation. The *c 0* through *c 9* commands (see *XXX* [Conversions], page *XXX*) can be used to round an existing value to a new precision.

It is important to distinguish the concepts of *precision* and *accuracy*. In the normal usage of these words, the number 123.4567 has a precision of 7 digits but an accuracy of 4 digits. The precision is the total number of digits not counting leading or trailing zeros (regardless of the position of the decimal point). The accuracy is simply the number of digits after the decimal point (again not counting trailing zeros). In Calc you control the precision, not the accuracy of computations. If you were to set the accuracy instead, then calculations like '`exp(100)`' would generate many more digits than you would typically need, while '`exp(-100)`' would probably round to zero! In Calc, both these computations give you exactly 12 (or the requested number of) significant digits.

The only Calc features that deal with accuracy instead of precision are fixed-point display mode for floats (*d f*; see *XXX* [Float Formats], page *XXX*), and the rounding functions like `floor` and `round` (see *XXX* [Integer Truncation], page *XXX*). Also, *c 0* through *c 9* deal with both precision and accuracy depending on the magnitudes of the numbers involved.

If you need to work with a particular fixed accuracy (say, dollars and cents with two digits after the decimal point), one solution is to work with integers and an "implied" decimal point. For example, $8.99 divided by 6 would be entered *899 RET 6 /*, yielding 149.833 (actually $1.49833 with our implied decimal point); pressing *R* would round this to 150 cents, i.e., $1.50.

See *XXX* [Floats], page *XXX*, for still more on floating-point precision and related issues.

6.3 Inverse and Hyperbolic Flags

I There is no single-key equivalent to the `calc-arcsin` function. Instead, you must first press *I* (`calc-inverse`) to set the *Inverse Flag*, then press *S* (`calc-sin`). The *I* key actually toggles the Inverse Flag. When this flag is set, the word '`Inv`' appears in the mode line.

H Likewise, the *H* key (`calc-hyperbolic`) sets or clears the Hyperbolic Flag, which transforms `calc-sin` into `calc-sinh`. If both of these flags are set at once, the effect will be `calc-arcsinh`. (The Hyperbolic flag is also used by some non-trigonometric commands; for example *H L* computes a base-10, instead of base-e, logarithm.)

Command names like `calc-arcsin` are provided for completeness, and may be executed with *x* or *M-x*. Their effect is simply to toggle the Inverse and/or Hyperbolic flags and then execute the corresponding base command (`calc-sin` in this case).

The Inverse and Hyperbolic flags apply only to the next Calculator command, after which they are automatically cleared. (They are also cleared if the next keystroke is not a Calc command.) Digits you type after *I* or *H* (or *K*) are treated as prefix arguments for the next command, not as numeric entries. The same is true of *C-u*, but not of the minus sign (*K -* means to subtract and keep arguments).

The third Calc prefix flag, *K* (keep-arguments), is discussed elsewhere. See *XXX* [Keep Arguments], page *XXX*.

6.4 Calculation Modes

The commands in this section are two-key sequences beginning with the *m* prefix. (That's the letter *m*, not the *META* key.) The '`m a`' (`calc-algebraic-mode`) command is described elsewhere (see *XXX* [Algebraic Entry], page *XXX*).

6.4.1 Angular Modes

The Calculator supports three notations for angles: radians, degrees, and degrees-minutes-seconds. When a number is presented to a function like `sin` that requires an angle, the current angular mode is used to interpret the number as either radians or degrees. If an HMS form is presented to `sin`, it is always interpreted as degrees-minutes-seconds.

Functions that compute angles produce a number in radians, a number in degrees, or an HMS form depending on the current angular mode. If the result is a complex number and the current mode is HMS, the number is instead expressed in degrees. (Complex-number calculations would normally be done in radians mode, though. Complex numbers are converted to degrees by calculating the complex result in radians and then multiplying by 180 over π.)

The *m r* (`calc-radians-mode`), *m d* (`calc-degrees-mode`), and *m h* (`calc-hms-mode`) commands control the angular mode. The current angular mode is displayed on the Emacs mode line. The default angular mode is degrees. `m r` `m d` `m h`

6.4.2 Polar Mode

The Calculator normally "prefers" rectangular complex numbers in the sense that rectangular form is used when the proper form can not be decided from the input. This might happen by multiplying a rectangular number by a polar one, by taking the square root of a negative real number, or by entering *(2 SPC 3)*.

The *m p* (`calc-polar-mode`) command toggles complex-number preference between rectangular and polar forms. In polar mode, all of the above example situations would produce polar complex numbers. `m p`

6.4.3 Fraction Mode

Division of two integers normally yields a floating-point number if the result cannot be expressed as an integer. In some cases you would rather get an exact fractional answer. One way to accomplish this is to multiply fractions instead: *6 RET 1:4 ** produces 3:2 even though *6 RET 4 /* produces 1.5.

m f

To set the Calculator to produce fractional results for normal integer divisions, use the *m f* (`calc-frac-mode`) command. For example, 8/4 produces 2 in either mode, but 6/4 produces 3:2 in Fraction Mode, 1.5 in Float Mode.

At any time you can use *c f* (`calc-float`) to convert a fraction to a float, or *c F* (`calc-fraction`) to convert a float to a fraction. See *XXX* [Conversions], page *XXX*.

6.4.4 Infinite Mode

The Calculator normally treats results like 1/0 as errors; formulas like this are left in unsimplified form. But Calc can be put into a mode where such calculations instead produce "infinite" results.

m i

The *m i* (`calc-infinite-mode`) command turns this mode on and off. When the mode is off, infinities do not arise except in calculations that already had infinities as inputs. (One exception is that infinite open intervals like '`[0 .. inf)`' can be generated; however, intervals closed at infinity ('`[0 .. inf]`') will not be generated when infinite mode is off.)

With infinite mode turned on, '`1 / 0`' will generate `uinf`, an undirected infinity. See *XXX* [Infinities], page *XXX*, for a discussion of the difference between `inf` and `uinf`. Also, 0/0 evaluates to `nan`, the "indeterminate" symbol. Various other functions can also return infinities in this mode; for example, '`ln(0) = -inf`', and '`gamma(-7) = uinf`'. Once again, note that '`exp(inf) = inf`' regardless of infinite mode because this calculation has infinity as an input.

The *m i* command with a numeric prefix argument of zero, i.e., *C-u 0 m i*, turns on a "positive infinite mode" in which zero is treated as positive instead of being directionless. Thus, '`1 / 0 = inf`' and '`-1 / 0 = -inf`' in this mode. Note that zero never actually has a sign in Calc; there are no separate representations for $+0$ and -0. Positive infinite mode merely changes the interpretation given to the single symbol, '`0`'. One consequence of this is that, while you might expect '`1 / -0 = -inf`', actually '`1 / -0`' is equivalent to '`1 / 0`', which is equal to positive `inf`.

6.4.5 Symbolic Mode

Calculations are normally performed numerically wherever possible. For example, the `calc-sqrt` command, or `sqrt` function in an algebraic expression, produces a numeric answer if the argument is a number or a symbolic expression if the argument is an expression: *2 Q* pushes 1.4142 but *' x+1 RET Q* pushes '`sqrt(x+1)`'.

m s

In *symbolic mode*, controlled by the *m s* (`calc-symbolic-mode`) command, functions which would produce inexact, irrational results are left in symbolic form. Thus *16 Q* pushes 4, but *2 Q* pushes '`sqrt(2)`'.

The shift-*N* (`calc-eval-num`) command evaluates numerically the expression at the top of the stack, by temporarily disabling `calc-symbolic-mode` and executing `=` (`calc-evaluate`). Given a numeric prefix argument, it also sets the floating-point precision to the specified value for the duration of the command. `N`

To evaluate a formula numerically without expanding the variables it contains, you can use the key sequence *m s* `a` *v m s* (this uses `calc-alg-evaluate`, which resimplifies but doesn't evaluate variables.)

6.4.6 Matrix and Scalar Modes

Calc sometimes makes assumptions during algebraic manipulation that are awkward or incorrect when vectors and matrices are involved. Calc has two modes, *matrix mode* and *scalar mode*, which modify its behavior around vectors in useful ways.

Press *m v* (`calc-matrix-mode`) once to enter matrix mode. In this mode, all objects are assumed to be matrices unless provably otherwise. One major effect is that Calc will no longer consider multiplication to be commutative. (Recall that in matrix arithmetic, '`A*B`' is not the same as '`B*A`'.) This assumption affects rewrite rules and algebraic simplification. Another effect of this mode is that calculations that would normally produce constants like 0 and 1 (e.g., $a - a$ and a/a, respectively) will now produce function calls that represent "generic" zero or identity matrices: '`idn(0)`', '`idn(1)`'. The `idn` function '`idn(`*a*`,`*n*`)`' returns *a* times an *n*x*n* identity matrix; if *n* is omitted, it doesn't know what dimension to use and so the `idn` call remains in symbolic form. However, if this generic identity matrix is later combined with a matrix whose size is known, it will be converted into a true identity matrix of the appropriate size. On the other hand, if it is combined with a scalar (as in '`idn(1) + 2`'), Calc will assume it really was a scalar after all and produce, e.g., 3. `m v`

Press *m v* a second time to get scalar mode. Here, objects are assumed *not* to be vectors or matrices unless provably so. For example, normally adding a variable to a vector, as in '`[x, y, z] + a`', will leave the sum in symbolic form because as far as Calc knows, '`a`' could represent either a number or another 3-vector. In scalar mode, '`a`' is assumed to be a non-vector, and the addition is evaluated to '`[x+a, y+a, z+a]`'.

Press *m v* a third time to return to the normal mode of operation.

If you press *m v* with a numeric prefix argument *n*, you get a special "dimensioned matrix mode" in which matrices of unknown size are assumed to be *n*x*n* square matrices. Then, the function call '`idn(1)`' will expand into an actual matrix rather than representing a "generic" matrix.

Of course these modes are approximations to the true state of affairs, which is probably that some quantities will be matrices and others will be scalars. One solution is to "declare" certain variables or functions to be scalar-valued. See *XXX* [Declarations], page *XXX*, to see how to make declarations in Calc.

There is nothing stopping you from declaring a variable to be scalar and then storing a matrix in it; however, if you do, the results you get from Calc may not be valid. Suppose you let Calc get the result '`[x+a, y+a, z+a]`' shown above, and then stored '`[1, 2, 3]`' in '`a`'. The result would not be the same as for '`[x, y, z] + [1, 2, 3]`', but that's because you have broken your earlier promise to Calc that '`a`' would be scalar.

Another way to mix scalars and matrices is to use selections (see *XXX* [Selecting Subformulas], page *XXX*). Use matrix mode when operating on your formula normally; then, to apply scalar mode to a certain part of the formula without affecting the rest just select that part, change into scalar mode and press `=` to resimplify the part under this mode, then change back to matrix mode before deselecting.

6.4.7 Automatic Recomputation

The *evaluates-to* operator, '`=>`', has the special property that any '`=>`' formulas on the stack are recomputed whenever variable values or mode settings that might affect them are changed. See *XXX* [Evaluates-To Operator], page *XXX*.

`m C` The *`m C`* (`calc-auto-recompute`) command turns this automatic recomputation on and off. If you turn it off, Calc will not update '`=>`' operators on the stack (nor those in the attached Embedded Mode buffer, if there is one). They will not be updated unless you explicitly do so by pressing `=` or until you press *`m C`* to turn recomputation back on. (While automatic recomputation is off, you can think of *`m C m C`* as a command to update all '`=>`' operators while leaving recomputation off.)

To update '`=>`' operators in an Embedded buffer while automatic recomputation is off, use *`M-# u`*. See *XXX* [Embedded Mode], page *XXX*.

6.4.8 Working Messages

Since the Calculator is written entirely in Emacs Lisp, which is not designed for heavy numerical work, many operations are quite slow. The Calculator normally displays the message '`Working...`' in the echo area during any command that may be slow. In addition, iterative operations such as square roots and trigonometric functions display the intermediate result at each step. Both of these types of messages can be disabled if you find them distracting.

`m w` Type *`m w`* (`calc-working`) with a numeric prefix of 0 to disable all "working" messages. Use a numeric prefix of 1 to enable only the plain '`Working...`' message. Use a numeric prefix of 2 to see intermediate results as well. With no numeric prefix this displays the current mode.

While it may seem that the "working" messages will slow Calc down considerably, experiments have shown that their impact is actually quite small. But if your terminal is slow you may find that it helps to turn the messages off.

6.5 Simplification Modes

The current *simplification mode* controls how numbers and formulas are "normalized" when being taken from or pushed onto the stack. Some normalizations are unavoidable, such as rounding floating-point results to the current precision, and reducing fractions to simplest form. Others, such as simplifying a formula like $a + a$ (or $2 + 3$), are done by default but can be turned off when necessary.

When you press a key like `+` when 2 and 3 are on the stack, Calc pops these numbers, normalizes them, creates the formula $2 + 3$, normalizes it, and pushes the result. Of course the standard rules for normalizing $2 + 3$ will produce the result 5.

Simplification mode commands consist of the lower-case `m` prefix key followed by a shifted letter.

The *m O* (`calc-no-simplify-mode`) command turns off all optional simplifications. These would leave a formula like $2 + 3$ alone. In fact, nothing except simple numbers are ever affected by normalization in this mode. `m O`

`m N` The *m N* (`calc-num-simplify-mode`) command turns off simplification of any formulas except those for which all arguments are constants. For example, $1+2$ is simplified to 3, and $a+(2-2)$ is simplified to $a+0$ but no further, since one argument of the sum is not a constant. Unfortunately, $(a+2)-2$ is *not* simplified because the top-level '`-`' operator's arguments are not both constant numbers (one of them is the formula $a+2$). A constant is a number or other numeric object (such as a constant error form or modulo form), or a vector all of whose elements are constant.

`m D` The *m D* (`calc-default-simplify-mode`) command restores the default simplifications for all formulas. This includes many easy and fast algebraic simplifications such as $a+0$ to a, and $a+2a$ to $3a$, as well as evaluating functions like $\mathtt{deriv}(x^2, x)$ to $2x$.

`m B` The *m B* (`calc-bin-simplify-mode`) mode applies the default simplifications to a result and then, if the result is an integer, uses the *b c* (`calc-clip`) command to clip the integer according to the current binary word size. See *XXX* [Binary Functions], page *XXX*. Real numbers are rounded to the nearest integer and then clipped; other kinds of results (after the default simplifications) are left alone.

`m A` The *m A* (`calc-alg-simplify-mode`) mode does algebraic simplification; it applies all the default simplifications, and also the more powerful (and slower) simplifications made by `a s` (`calc-simplify`). See *XXX* [Algebraic Simplifications], page *XXX*.

`m E` The *m E* (`calc-ext-simplify-mode`) mode does "extended" algebraic simplification, as by the `a e` (`calc-simplify-extended`) command. See *XXX* [Unsafe Simplifications], page *XXX*.

`m U` The *m U* (`calc-units-simplify-mode`) mode does units simplification; it applies the command *u s* (`calc-simplify-units`), which in turn is a superset of `a s`. In this mode, variable names which are identifiable as unit names (like '`mm`' for "millimeters") are simplified with their unit definitions in mind.

A common technique is to set the simplification mode down to the lowest amount of simplification you will allow to be applied automatically, then use manual commands like `a s` and *c c* (`calc-clean`) to perform higher types of simplifications on demand. See *XXX* [Algebraic Definitions], page *XXX*, for another sample use of no-simplification mode.

6.6 Declarations

A *declaration* is a statement you make that promises you will use a certain variable or function in a restricted way. This may give Calc the freedom to do things that it couldn't do if it had to take the fully general situation into account.

6.6.1 Declaration Basics

The *s d* (`calc-declare-variable`) command is the easiest way to make a declaration for a variable. This command prompts for the variable name, then prompts for the declaration. The default at the declaration prompt is the previous declaration, if any. You can edit this declaration, or press *C-k* to erase it and type a new declaration. (Or, erase it and press *RET* to clear the declaration, effectively "undeclaring" the variable.) `s d`

A declaration is in general a vector of *type symbols* and *range* values. If there is only one type symbol or range value, you can write it directly rather than enclosing it in a vector. For example, *s d foo RET real RET* declares `foo` to be a real number, and *s d bar RET [int, const, [1..6]] RET* declares `bar` to be a constant integer between 1 and 6. (Actually, you can omit the outermost brackets and Calc will provide them for you: *s d bar RET int, const, [1..6] RET*.)

Declarations in Calc are kept in a special variable called `Decls`. This variable encodes the set of all outstanding declarations in the form of a matrix. Each row has two elements: A variable or vector of variables declared by that row, and the declaration specifier as described above. You can use the *s D* command to edit this variable if you wish to see all the declarations at once. See *XXX* [Operations on Variables], page *XXX*, for a description of this command and the *s p* command that allows you to save your declarations permanently if you wish.

Items being declared can also be function calls. The arguments in the call are ignored; the effect is to say that this function returns values of the declared type for any valid arguments. The *s d* command declares only variables, so if you wish to make a function declaration you will have to edit the `Decls` matrix yourself.

For example, the declaration matrix

```
[ [ foo,       real       ]
  [ [j, k, n], int        ]
  [ f(1,2,3),  [0 .. inf) ] ]
```

declares that `foo` represents a real number, `j`, `k` and `n` represent integers, and the function `f` always returns a real number in the interval shown.

If there is a declaration for the variable `All`, then that declaration applies to all variables that are not otherwise declared. It does not apply to function names. For example, using the row '`[All, real]`' says that all your variables are real unless they are explicitly declared without `real` in some other row. The *s d* command declares `All` if you give a blank response to the variable-name prompt.

6.6.2 Kinds of Declarations

The type-specifier part of a declaration (that is, the second prompt in the *s d* command) can be a type symbol, an interval, or a vector consisting of zero or more type symbols followed by zero or more intervals or numbers that represent the set of possible values for the variable.

```
[ [ a, [1, 2, 3, 4, 5] ]
  [ b, [1 .. 5]        ]
  [ c, [int, 1 .. 5]   ] ]
```

Here `a` is declared to contain one of the five integers shown; `b` is any number in the interval from 1 to 5 (any real number since we haven't specified), and `c` is any integer in that interval. Thus the declarations for `a` and `c` are nearly equivalent (see below).

The type-specifier can be the empty vector '`[]`' to say that nothing is known about a given variable's value. This is the same as not declaring the variable at all except that it overrides any `All` declaration which would otherwise apply.

The initial value of `Decls` is the empty vector '`[]`'. If `Decls` has no stored value or if the value stored in it is not valid, it is ignored and there are no declarations as far as Calc is concerned. (The *`s d`* command will replace such a malformed value with a fresh empty matrix, '`[]`', before recording the new declaration.) Unrecognized type symbols are ignored.

The following type symbols describe what sorts of numbers will be stored in a variable:

`int`	Integers.
`numint`	Numerical integers. (Integers or integer-valued floats.)
`frac`	Fractions. (Rational numbers which are not integers.)
`rat`	Rational numbers. (Either integers or fractions.)
`float`	Floating-point numbers.
`real`	Real numbers. (Integers, fractions, or floats. Actually, intervals and error forms with real components also count as reals here.)
`pos`	Positive real numbers. (Strictly greater than zero.)
`nonneg`	Nonnegative real numbers. (Greater than or equal to zero.)
`number`	Numbers. (Real or complex.)

Calc uses this information to determine when certain simplifications of formulas are safe. For example, '`(x^y)^z`' cannot be simplified to '`x^(y z)`' in general; for example, '`((-3)^2)^1:2`' is 3, but '`(-3)^(2*1:2) = (-3)^1`' is -3. However, this simplification *is* safe if `z` is known to be an integer, or if `x` is known to be a nonnegative real number. If you have given declarations that allow Calc to deduce either of these facts, Calc will perform this simplification of the formula.

Calc can apply a certain amount of logic when using declarations. For example, '`(x^y)^(2n+1)`' will be simplified if `n` has been declared `int`; Calc knows that an integer times an integer, plus an integer, must always be an integer. (In fact, Calc would simplify '`(-x)^(2n+1)`' to '`-(x^(2n+1))`' since it is able to determine that '`2n+1`' must be an odd integer.)

Similarly, '`(abs(x)^y)^z`' will be simplified to '`abs(x)^(y z)`' because Calc knows that the `abs` function always returns a nonnegative real. If you had a `myabs` function that also had this property, you could get Calc to recognize it by adding the row '`[myabs(), nonneg]`' to the `Decls` matrix.

One instance of this simplification is '`sqrt(x^2)`' (since the `sqrt` function is effectively a one-half power). Normally Calc leaves this formula alone. After the command *`s d x RET real RET`*, however, it can simplify the formula to '`abs(x)`'. And after *`s d x RET nonneg RET`*, Calc can simplify this formula all the way to '`x`'.

If there are any intervals or real numbers in the type specifier, they comprise the set of possible values that the variable or function being declared can have. In particular, the type symbol `real` is effectively the same as the range '`[-inf .. inf]`' (note that infinity is included in the range of possible values); `pos` is the same as '`(0 .. inf]`', and `nonneg` is the same as '`[0 .. inf]`'. Saying '`[real, [-5 .. 5]]`' is redundant because the fact that the variable is real can be deduced just from the interval, but '`[int, [-5 .. 5]]`' and '`[rat, [-5 .. 5]]`' are useful combinations.

Note that the vector of intervals or numbers is in the same format used by Calc's set-manipulation commands. See *XXX* [Set Operations], page *XXX*.

The type specifier '`[1, 2, 3]`' is equivalent to '`[numint, 1, 2, 3]`', *not* to '`[int, 1, 2, 3]`'. In other words, the range of possible values means only that the variable's value must be numerically equal to a number in that range, but not that it must be equal in type as well. Calc's set operations act the same way; '`in(2, [1., 2., 3.])`' and '`in(1.5, [1:2, 3:2, 5:2])`' both report "true."

If you use a conflicting combination of type specifiers, the results are unpredictable. An example is '`[pos, [0 .. 5]]`', where the interval does not lie in the range described by the type symbol.

"Real" declarations mostly affect simplifications involving powers like the one described above. Another case where they are used is in the `a P` command which returns a list of all roots of a polynomial; if the variable has been declared real, only the real roots (if any) will be included in the list.

"Integer" declarations are used for simplifications which are valid only when certain values are integers (such as '`(x^y)^z`' shown above).

Another command that makes use of declarations is `a s`, when simplifying equations and inequalities. It will cancel `x` from both sides of '`a x = b x`' only if it is sure `x` is non-zero, say, because it has a `pos` declaration. To declare specifically that `x` is real and non-zero, use '`[[-inf .. 0), (0 .. inf]]`'. (There is no way in the current notation to say that `x` is nonzero but not necessarily real.) The `a e` command does "unsafe" simplifications, including cancelling '`x`' from the equation when '`x`' is not known to be nonzero.

Another set of type symbols distinguish between scalars and vectors.

`scalar`	The value is not a vector.
`vector`	The value is a vector.
`matrix`	The value is a matrix (a rectangular vector of vectors).

These type symbols can be combined with the other type symbols described above; '`[int, matrix]`' describes an object which is a matrix of integers.

Scalar/vector declarations are used to determine whether certain algebraic operations are safe. For example, '`[a, b, c] + x`' is normally not simplified to '`[a + x, b + x, c + x]`', but it will be if `x` has been declared `scalar`. On the other hand, multiplication is usually assumed to be commutative, but the terms in '`x y`' will never be exchanged if both `x` and `y` are known to be vectors or matrices. (Calc currently never distinguishes between `vector` and `matrix` declarations.)

See *XXX* [Matrix Mode], page *XXX*, for a discussion of "matrix mode" and "scalar mode," which are similar to declaring '`[All, matrix]`' or '`[All, scalar]`' but much more convenient.

One more type symbol that is recognized is used with the `H a d` command for taking total derivatives of a formula. See *XXX* [Calculus], page *XXX*.

`const` The value is a constant with respect to other variables.

Calc does not check the declarations for a variable when you store a value in it. However, storing -3.5 in a variable that has been declared `pos`, `int`, or `matrix` may have unexpected effects; Calc may evaluate '`sqrt(x^2)`' to 3.5 if it substitutes the value first, or to -3.5 if `x` was declared `pos` and the formula '`sqrt(x^2)`' is simplified to '`x`' before the value is substituted. Before using a variable for a new purpose, it is best to use *s d* or *s D* to check to make sure you don't still have an old declaration for the variable that will conflict with its new meaning.

6.6.3 Functions for Declarations

Calc has a set of functions for accessing the current declarations in a convenient manner. These functions return 1 if the argument can be shown to have the specified property, or 0 if the argument can be shown *not* to have that property; otherwise they are left unevaluated. These functions are suitable for use with rewrite rules (see *XXX* [Conditional Rewrite Rules], page *XXX*) or programming constructs (see *XXX* [Conditionals in Macros], page *XXX*). They can be entered only using algebraic notation. See *XXX* [Logical Operations], page *XXX*, for functions that perform other tests not related to declarations.

For example, '`dint(17)`' returns 1 because 17 is an integer, as do '`dint(n)`' and '`dint(2 n - 3)`' if `n` has been declared `int`, but '`dint(2.5)`' and '`dint(n + 0.5)`' return 0. Calc consults knowledge of its own built-in functions as well as your own declarations: '`dint(floor(x))`' returns 1.

`dint`
`dnumint`
`dnatnum`
The `dint` function checks if its argument is an integer. The `dnatnum` function checks if its argument is a natural number, i.e., a nonnegative integer. The `dnumint` function checks if its argument is numerically an integer, i.e., either an integer or an integer-valued float. Note that these and the other data type functions also accept vectors or matrices composed of suitable elements, and that real infinities '`inf`' and '`-inf`' are considered to be integers for the purposes of these functions.

`drat` The `drat` function checks if its argument is rational, i.e., an integer or fraction. Infinities count as rational, but intervals and error forms do not.

`dreal` The `dreal` function checks if its argument is real. This includes integers, fractions, floats, real error forms, and intervals.

`dimag` The `dimag` function checks if its argument is imaginary, i.e., is mathematically equal to a real number times i.

`dpos`
`dneg`
`dnonneg`
The `dpos` function checks for positive (but nonzero) reals. The `dneg` function checks for negative reals. The `dnonneg` function checks for nonnegative reals, i.e., reals greater than or equal to zero. Note that the `a s` command can simplify an expression like $x > 0$ to 1 or 0 using `dpos`, and that `a s` is effectively applied to all conditions in rewrite rules, so the actual functions `dpos`, `dneg`, and `dnonneg` are rarely necessary.

`dnonzero` The `dnonzero` function checks that its argument is nonzero. This includes all nonzero real or complex numbers, all intervals that do not include zero, all nonzero modulo forms, vectors all of whose elements are nonzero, and variables or formulas whose values can be deduced to be nonzero. It does not include error forms, since they represent values which could be anything including zero. (This is also the set of objects considered "true" in conditional contexts.)

The `deven` function returns 1 if its argument is known to be an even integer (or integer-valued float); it returns 0 if its argument is known not to be even (because it is known to be odd or a non-integer). The *a s* command uses this to simplify a test of the form '`x % 2 = 0`'. There is also an analogous `dodd` function. `deven` `dodd`

The `drange` function returns a set (an interval or a vector of intervals and/or numbers; see *XXX* [Set Operations], page *XXX*) that describes the set of possible values of its argument. If the argument is a variable or a function with a declaration, the range is copied from the declaration. Otherwise, the possible signs of the expression are determined using a method similar to `dpos`, etc., and a suitable set like '`[0 .. inf]`' is returned. If the expression is not provably real, the `drange` function remains unevaluated. `drange`

The `dscalar` function returns 1 if its argument is provably scalar, or 0 if its argument is provably non-scalar. It is left unevaluated if this cannot be determined. (If matrix mode or scalar mode are in effect, this function returns 1 or 0, respectively, if it has no other information.) When Calc interprets a condition (say, in a rewrite rule) it considers an unevaluated formula to be "false." Thus, '`dscalar(a)`' is "true" only if `a` is provably scalar, and '`!dscalar(a)`' is "true" only if `a` is provably non-scalar; both are "false" if there is insufficient information to tell. `dscalar`

6.7 Display Modes

The commands in this section are two-key sequences beginning with the *d* prefix. The *d l* (`calc-line-numbering`) and *d b* (`calc-line-breaking`) commands are described elsewhere; see *XXX* [Stack Basics], page *XXX* and see *XXX* [Normal Language Modes], page *XXX*, respectively. Display formats for vectors and matrices are also covered elsewhere; see *XXX* [Vector and Matrix Formats], page *XXX*.

One thing all display modes have in common is their treatment of the *H* prefix. This prefix causes any mode command that would normally refresh the stack to leave the stack display alone. The word "Dirty" will appear in the mode line when Calc thinks the stack display may not reflect the latest mode settings.

The *d RET* (`calc-refresh-top`) command reformats the top stack entry according to all the current modes. Positive prefix arguments reformat the top *n* entries; negative prefix arguments reformat the specified entry, and a prefix of zero is equivalent to *d SPC* (`calc-refresh`), which reformats the entire stack. For example, *H d s M-2 d RET* changes to scientific notation but reformats only the top two stack entries in the new mode. `d RET`

The *I* prefix has another effect on the display modes. The mode is set only temporarily; the top stack entry is reformatted according to that mode, then the original mode setting is restored. In other words, *I d s* is equivalent to *H d s d RET H d (old mode)*.

6.7.1 Radix Modes

Calc normally displays numbers in decimal (*base-10* or *radix-10*) notation. Calc can actually display in any radix from two (binary) to 36. When the radix is above 10, the letters `A` to `Z` are used as digits. When entering such a number, letter keys are interpreted as potential digits rather than terminating numeric entry mode.

`d 2`
`d 8`
`d 6`
`d 0`

The key sequences *d 2*, *d 8*, *d 6*, and *d 0* select binary, octal, hexadecimal, and decimal as the current display radix, respectively. Numbers can always be entered in any radix, though the current radix is used as a default if you press **#** without any initial digits. A number entered without a **#** is *always* interpreted as decimal.

`d r`

To set the radix generally, use *d r* (`calc-radix`) and enter an integer from 2 to 36. You can specify the radix as a numeric prefix argument; otherwise you will be prompted for it.

`d z`

Integers normally are displayed with however many digits are necessary to represent the integer and no more. The *d z* (`calc-leading-zeros`) command causes integers to be padded out with leading zeros according to the current binary word size. (See *XXX* [Binary Functions], page *XXX*, for a discussion of word size.) If the absolute value of the word size is w, all integers are displayed with at least enough digits to represent $2^w - 1$ in the current radix. (Larger integers will still be displayed in their entirety.)

6.7.2 Grouping Digits

`d g`

Long numbers can be hard to read if they have too many digits. For example, the factorial of 30 is 33 digits long! Press *d g* (`calc-group-digits`) to enable *grouping* mode, in which digits are displayed in clumps of 3 or 4 (depending on the current radix) separated by commas.

The *d g* command toggles grouping on and off. With a numerix prefix of 0, this command displays the current state of the grouping flag; with an argument of minus one it disables grouping; with a positive argument N it enables grouping on every N digits. For floating-point numbers, grouping normally occurs only before the decimal point. A negative prefix argument $-N$ enables grouping every N digits both before and after the decimal point.

`d ,`

The *d ,* (`calc-group-char`) command allows you to choose any character as the grouping separator. The default is the comma character. If you find it difficult to read vectors of large integers grouped with commas, you may wish to use spaces or some other character instead. This command takes the next character you type, whatever it is, and uses it as the digit separator. As a special case, *d , * selects '`\,`' (TeX's thin-space symbol) as the digit separator.

Please note that grouped numbers will not generally be parsed correctly if re-read in textual form, say by the use of *M-# y* and *M-# g*. (See *XXX* [Kill and Yank], page *XXX*, for details on these commands.) One exception is the '`\,`' separator, which doesn't interfere with parsing because it is ignored by TeX language mode.

6.7.3 Float Formats

Floating-point quantities are normally displayed in standard decimal form, with scientific notation used if the exponent is especially high or low. All significant digits are normally displayed. The commands in this section allow you to choose among several alternative display formats for floats.

The *d n* (`calc-normal-notation`) command selects the normal display format. All significant figures in a number are displayed. With a positive numeric prefix, numbers are rounded if necessary to that number of significant digits. With a negative numerix prefix, the specified number of significant digits less than the current precision is used. (Thus *C-u -2 d n* displays 10 digits if the current precision is 12.) `d n`

The *d f* (`calc-fix-notation`) command selects fixed-point notation. The numeric argument is the number of digits after the decimal point, zero or more. This format will relax into scientific notation if a nonzero number would otherwise have been rounded all the way to zero. Specifying a negative number of digits is the same as for a positive number, except that small nonzero numbers will be rounded to zero rather than switching to scientific notation. `d f`

The *d s* (`calc-sci-notation`) command selects scientific notation. A positive argument sets the number of significant figures displayed, of which one will be before and the rest after the decimal point. A negative argument works the same as for *d n* format. The default is to display all significant digits. `d s`

The *d e* (`calc-eng-notation`) command selects engineering notation. This is similar to scientific notation except that the exponent is rounded down to a multiple of three, with from one to three digits before the decimal point. An optional numeric prefix sets the number of significant digits to display, as for *d s*. `d e`

It is important to distinguish between the current *precision* and the current *display format*. After the commands *C-u 10 p* and *C-u 6 d n* the Calculator computes all results to ten significant figures but displays only six. (In fact, intermediate calculations are often carried to one or two more significant figures, but values placed on the stack will be rounded down to ten figures.) Numbers are never actually rounded to the display precision for storage, except by commands like *C-k* and *M-# y* which operate on the actual displayed text in the Calculator buffer.

The *d .* (`calc-point-char`) command selects the character used as a decimal point. Normally this is a period; users in some countries may wish to change this to a comma. Note that this is only a display style; on entry, periods must always be used to denote floating-point numbers, and commas to separate elements in a list. `d .`

6.7.4 Complex Formats

There are three supported notations for complex numbers in rectangular form. The default is as a pair of real numbers enclosed in parentheses and separated by a comma: '`(a,b)`'. The *d c* (`calc-complex-notation`) command selects this style. `d c`

The other notations are *d i* (`calc-i-notation`), in which numbers are displayed in '`a+bi`' form, and *d j* (`calc-j-notation`) which displays the form '`a+bj`' preferred in some disciplines. `d i` `d j`

Complex numbers are normally entered in '`(a,b)`' format. If you enter '`2+3i`' as an algebraic formula, it will be stored as the formula '`2 + 3 * i`'. However, if you use `=` to evaluate this formula and you have not changed the variable '`i`', the '`i`' will be interpreted as '`(0,1)`' and the formula will be simplified to '`(2,3)`'. Other commands (like `calc-sin`) will *not* interpret the formula '`2 + 3 * i`' as a complex number. See *XXX* [Variables], page *XXX*, under "special constants."

6.7.5 Fraction Formats

`d o` Display of fractional numbers is controlled by the *`d o`* (`calc-over-notation`) command. By default, a number like eight thirds is displayed in the form '`8:3`'. The *`d o`* command prompts for a one- or two-character format. If you give one character, that character is used as the fraction separator. Common separators are '`:`' and '`/`'. (During input of numbers, the *`:`* key must be used regardless of the display format; in particular, the *`/`* is used for RPN-style division, *not* for entering fractions.)

If you give two characters, fractions use "integer-plus-fractional-part" notation. For example, the format '`+/`' would display eight thirds as '`2+2/3`'. If two colons are present in a number being entered, the number is interpreted in this form (so that the entries *`2:2:3`* and *`8:3`* are equivalent).

It is also possible to follow the one- or two-character format with a number. For example: '`:10`' or '`+/3`'. In this case, Calc adjusts all fractions that are displayed to have the specified denominator, if possible. Otherwise it adjusts the denominator to be a multiple of the specified value. For example, in '`:6`' mode the fraction 1:6 will be unaffected, but 2:3 will be displayed as 4:6, 1:2 will be displayed as 3:6, and 1:8 will be displayed as 3:24. Integers are also affected by this mode: 3 is displayed as 18:6. Note that the format '`:1`' writes fractions the same as '`:`', but it writes integers as n:1.

The fraction format does not affect the way fractions or integers are stored, only the way they appear on the screen. The fraction format never affects floats.

6.7.6 HMS Formats

`d h` The *`d h`* (`calc-hms-notation`) command controls the display of HMS (hours-minutes-seconds) forms. It prompts for a string which consists basically of an "hours" marker, optional punctuation, a "minutes" marker, more optional punctuation, and a "seconds" marker. Punctuation is zero or more spaces, commas, or semicolons. The hours marker is one or more non-punctuation characters. The minutes and seconds markers must be single non-punctuation characters.

The default HMS format is '`@ ' "`', producing HMS values of the form '`23@ 30' 15.75"`'. The format '`deg, ms`' would display this same value as '`23deg, 30m15.75s`'. During numeric entry, the *`h`* or *`o`* keys are recognized as synonyms for `@` regardless of display format. The *`m`* and *`s`* keys are recognized as synonyms for *`'`* and `"`, respectively, but only if an `@` (or *`h`* or *`o`*) has already been typed; otherwise, they have their usual meanings (*`m-`* prefix and *`s-`* prefix). Thus, *`5 "`*, *`0 @ 5 "`*, and *`0 h 5 s`* are some of the ways to enter the quantity "five seconds." The *`'`* key is recognized as "minutes" only if `@` (or *`h`* or *`o`*) has already been pressed; otherwise it means to switch to algebraic entry.

6.7.7 Date Formats

The *d d* (`calc-date-notation`) command controls the display of date forms (see *XXX* [Date Forms], page *XXX*). It prompts for a string which contains letters that represent the various parts of a date and time. To show which parts should be omitted when the form represents a pure date with no time, parts of the string can be enclosed in '`< >`' marks. If you don't include '`< >`' markers in the format, Calc guesses at which parts, if any, should be omitted when formatting pure dates. `d d`

The default format is: '`<H:mm:SSpp >Www Mmm D, YYYY`'. An example string in this format is '`3:32pm Wed Jan 9, 1991`'. If you enter a blank format string, this default format is reestablished.

Calc uses '`< >`' notation for nameless functions as well as for dates. See *XXX* [Specifying Operators], page *XXX*. To avoid confusion with nameless functions, your date formats should avoid using the '`#`' character.

6.7.7.1 Date Formatting Codes

When displaying a date, the current date format is used. All characters except for letters and '`<`' and '`>`' are copied literally when dates are formatted. The portion between '`< >`' markers is omitted for pure dates, or included for date/time forms. Letters are interpreted according to the table below.

When dates are read in during algebraic entry, Calc first tries to match the input string to the current format either with or without the time part. The punctuation characters (including spaces) must match exactly; letter fields must correspond to suitable text in the input. If this doesn't work, Calc checks if the input is a simple number; if so, the number is interpreted as a number of days since Jan 1, 1 AD. Otherwise, Calc tries a much more relaxed and flexible algorithm which is described in the next section.

Weekday names are ignored during reading.

Two-digit year numbers are interpreted as lying in the range from 1941 to 2039. Years outside that range are always entered and displayed in full. Year numbers with a leading '`+`' sign are always interpreted exactly, allowing the entry and display of the years 1 through 99 AD.

Here is a complete list of the formatting codes for dates:

Y	Year: "91" for 1991, "7" for 2007, "+23" for 23 AD.
YY	Year: "91" for 1991, "07" for 2007, "+23" for 23 AD.
BY	Year: "91" for 1991, " 7" for 2007, "+23" for 23 AD.
YYY	Year: "1991" for 1991, "23" for 23 AD.
YYYY	Year: "1991" for 1991, "+23" for 23 AD.
aa	Year: "ad" or blank.
AA	Year: "AD" or blank.

aaa	Year: “ad ” or blank. (Note trailing space.)
AAA	Year: “AD ” or blank.
aaaa	Year: “a.d.” or blank.
AAAA	Year: “A.D.” or blank.
bb	Year: “bc” or blank.
BB	Year: “BC” or blank.
bbb	Year: “ bc” or blank. (Note leading space.)
BBB	Year: “ BC” or blank.
bbbb	Year: “b.c.” or blank.
BBBB	Year: “B.C.” or blank.
M	Month: “8” for August.
MM	Month: “08” for August.
BM	Month: “ 8” for August.
MMM	Month: “AUG” for August.
Mmm	Month: “Aug” for August.
mmm	Month: “aug” for August.
MMMM	Month: “AUGUST” for August.
Mmmm	Month: “August” for August.
D	Day: “7” for 7th day of month.
DD	Day: “07” for 7th day of month.
BD	Day: “ 7” for 7th day of month.
W	Weekday: “0” for Sunday, “6” for Saturday.
WWW	Weekday: “SUN” for Sunday.
Www	Weekday: “Sun” for Sunday.
www	Weekday: “sun” for Sunday.
WWWW	Weekday: “SUNDAY” for Sunday.
Wwww	Weekday: “Sunday” for Sunday.

d	Day of year: “34” for Feb. 3.
ddd	Day of year: “034” for Feb. 3.
bdd	Day of year: “ 34” for Feb. 3.
h	Hour: “5” for 5 AM; “17” for 5 PM.
hh	Hour: “05” for 5 AM; “17” for 5 PM.
bh	Hour: “ 5” for 5 AM; “17” for 5 PM.
H	Hour: “5” for 5 AM and 5 PM.
HH	Hour: “05” for 5 AM and 5 PM.
BH	Hour: “ 5” for 5 AM and 5 PM.
p	AM/PM: “a” or “p”.
P	AM/PM: “A” or “P”.
pp	AM/PM: “am” or “pm”.
PP	AM/PM: “AM” or “PM”.
pppp	AM/PM: “a.m.” or “p.m.”.
PPPP	AM/PM: “A.M.” or “P.M.”.
m	Minutes: “7” for 7.
mm	Minutes: “07” for 7.
bm	Minutes: “ 7” for 7.
s	Seconds: “7” for 7; “7.23” for 7.23.
ss	Seconds: “07” for 7; “07.23” for 7.23.
bs	Seconds: “ 7” for 7; “ 7.23” for 7.23.
SS	Optional seconds: “07” for 7; blank for 0.
BS	Optional seconds: “ 7” for 7; blank for 0.
N	Numeric date/time: “726842.25” for 6:00am Wed Jan 9, 1991.
n	Numeric date: “726842” for any time on Wed Jan 9, 1991.
J	Julian date/time: “2448265.75” for 6:00am Wed Jan 9, 1991.
j	Julian date: “2448266” for any time on Wed Jan 9, 1991.

U	Unix time: "663400800" for 6:00am Wed Jan 9, 1991.
X	Brackets suppression. An "X" at the front of the format causes the surrounding '`< >`' delimiters to be omitted when formatting dates. Note that the brackets are still required for algebraic entry.

If "SS" or "BS" (optional seconds) is preceded by a colon, the colon is also omitted if the seconds part is zero.

If "bb," "bbb" or "bbbb" or their upper-case equivalents appear in the format, then negative year numbers are displayed without a minus sign. Note that "aa" and "bb" are mutually exclusive. Some typical usages would be '`YYYY AABB`'; '`AAAYYYYBBB`'; '`YYYYBBB`'.

The formats "YY," "YYYY," "MM," "DD," "ddd," "hh," "HH," "mm," "ss," and "SS" actually match any number of digits during reading unless several of these codes are strung together with no punctuation in between, in which case the input must have exactly as many digits as there are letters in the format.

The "j," "J," and "U" formats do not make any time zone adjustment. They effectively use '`julian(x,0)`' and '`unixtime(x,0)`' to make the conversion; see *XXX* [Date Arithmetic], page *XXX*.

6.7.7.2 Free-Form Dates

When reading a date form during algebraic entry, Calc falls back on the algorithm described here if the input does not exactly match the current date format. This algorithm generally "does the right thing" and you don't have to worry about it, but it is described here in full detail for the curious.

Calc does not distinguish between upper- and lower-case letters while interpreting dates.

First, the time portion, if present, is located somewhere in the text and then removed. The remaining text is then interpreted as the date.

A time is of the form '`hh:mm:ss`', possibly with the seconds part omitted and possibly with an AM/PM indicator added to indicate 12-hour time. If the AM/PM is present, the minutes may also be omitted. The AM/PM part may be any of the words '`am`', '`pm`', '`noon`', or '`midnight`'; each of these may be abbreviated to one letter, and the alternate forms '`a.m.`', '`p.m.`', and '`mid`' are also understood. Obviously '`noon`' and '`midnight`' are allowed only on 12:00:00. The words '`noon`', '`mid`', and '`midnight`' are also recognized with no number attached.

If there is no AM/PM indicator, the time is interpreted in 24-hour format.

To read the date portion, all words and numbers are isolated from the string; other characters are ignored. All words must be either month names or day-of-week names (the latter of which are ignored). Names can be written in full or as three-letter abbreviations.

Large numbers, or numbers with '`+`' or '`-`' signs, are interpreted as years. If one of the other numbers is greater than 12, then that must be the day and the remaining number in the input is therefore the month. Otherwise, Calc assumes the month, day and year are in the same order that they appear in the current date format. If the year is omitted, the current year is taken from the system clock.

If there are too many or too few numbers, or any unrecognizable words, then the input is rejected.

If there are any large numbers (of five digits or more) other than the year, they are ignored on the assumption that they are something like Julian dates that were included along with the traditional date components when the date was formatted.

One of the words '`ad`', '`a.d.`', '`bc`', or '`b.c.`' may optionally be used; the latter two are equivalent to a minus sign on the year value.

If you always enter a four-digit year, and use a name instead of a number for the month, there is no danger of ambiguity.

6.7.7.3 Standard Date Formats

There are actually ten standard date formats, numbered 0 through 9. Entering a blank line at the *`d d`* command's prompt gives you format number 1, Calc's usual format. You can enter any digit to select the other formats.

To create your own standard date formats, give a numeric prefix argument from 0 to 9 to the *`d d`* command. The format you enter will be recorded as the new standard format of that number, as well as becoming the new current date format. You can save your formats permanently with the *`m m`* command (see *XXX* [Mode Settings], page *XXX*).

0	'`N`' (Numerical format)
1	'`<H:mm:SSpp >Www Mmm D, YYYY`' (American format)
2	'`D Mmm YYYY<, h:mm:SS>`' (European format)
3	'`Www Mmm BD< hh:mm:ss> YYYY`' (Unix written date format)
4	'`M/D/Y< H:mm:SSpp>`' (American slashed format)
5	'`D.M.Y< h:mm:SS>`' (European dotted format)
6	'`M-D-Y< H:mm:SSpp>`' (American dashed format)
7	'`D-M-Y< h:mm:SS>`' (European dashed format)
8	'`j<, h:mm:ss>`' (Julian day plus time)
9	'`YYddd< hh:mm:ss>`' (Year-day format)

6.7.8 Truncating the Stack

`d t`

The *d t* (`calc-truncate-stack`) command moves the '`.`' line that marks the top-of-stack up or down in the Calculator buffer. The number right above that line is considered to the be at the top of the stack. Any numbers below that line are "hidden" from all stack operations. This is similar to the Emacs "narrowing" feature, except that the values below the '`.`' are *visible*, just temporarily frozen. This feature allows you to keep several independent calculations running at once in different parts of the stack, or to apply a certain command to an element buried deep in the stack.

Pressing *d t* by itself moves the '`.`' to the line the cursor is on. Thus, this line and all those below it become hidden. To un-hide these lines, move down to the end of the buffer and press *d t*. With a positive numeric prefix argument n, *d t* hides the bottom n values in the buffer. With a negative argument, it hides all but the top n values. With an argument of zero, it hides zero values, i.e., moves the '`.`' all the way down to the bottom.

`d [`
`d ]`

The *d [* (`calc-truncate-up`) and *d]* (`calc-truncate-down`) commands move the '`.`' up or down one line at a time (or several lines with a prefix argument).

6.7.9 Justification

`d <`
`d =`
`d >`

Values on the stack are normally left-justified in the window. You can control this arrangement by typing *d <* (`calc-left-justify`), *d >* (`calc-right-justify`), or *d =* (`calc-center-justify`). For example, in right-justification mode, stack entries are displayed flush-right against the right edge of the window.

If you change the width of the Calculator window you may have to type *d SPC* (`calc-refresh`) to re-align right-justified or centered text.

Right-justification is especially useful together with fixed-point notation (see `d f`; `calc-fix-notation`). With these modes together, the decimal points on numbers will always line up.

With a numeric prefix argument, the justification commands give you a little extra control over the display. The argument specifies the horizontal "origin" of a display line. It is also possible to specify a maximum line width using the *d b* command (see *XXX* [Normal Language Modes], page *XXX*). For reference, the precise rules for formatting and breaking lines are given below. Notice that the interaction between origin and line width is slightly different in each justification mode.

In left-justified mode, the line is indented by a number of spaces given by the origin (default zero). If the result is longer than the maximum line width, if given, or too wide to fit in the Calc window otherwise, then it is broken into lines which will fit; each broken line is indented to the origin.

In right-justified mode, lines are shifted right so that the rightmost character is just before the origin, or just before the current window width if no origin was specified. If the line is too long for this, then it is broken; the current line width is used, if specified, or else the origin is used as a width if that is specified, or else the line is broken to fit in the window.

In centering mode, the origin is the column number of the center of each stack entry. If a line width is specified, lines will not be allowed to go past that width; Calc will either indent less or break the lines if necessary. If no origin is specified, half the line width or Calc window width is used.

Note that, in each case, if line numbering is enabled the display is indented an additional four spaces to make room for the line number. The width of the line number is taken into account when positioning according to the current Calc window width, but not when positioning by explicit origins and widths. In the latter case, the display is formatted as specified, and then uniformly shifted over four spaces to fit the line numbers.

6.7.10 Labels

The *d {* (`calc-left-label`) command prompts for a string, then displays that string to the left of every stack entry. If the entries are left-justified (see *XXX* [Justification], page *XXX*), then they will appear immediately after the label (unless you specified an origin greater than the length of the label). If the entries are centered or right-justified, the label appears on the far left and does not affect the horizontal position of the stack entry. `d {`

Give a blank string (with *d { RET*) to turn the label off.

The *d }* (`calc-right-label`) command similarly adds a label on the righthand side. It does not affect positioning of the stack entries unless they are right-justified. Also, if both a line width and an origin are given in right-justified mode, the stack entry is justified to the origin and the righthand label is justified to the line width. `d }`

One application of labels would be to add equation numbers to formulas you are manipulating in Calc and then copying into a document (possibly using Embedded Mode). The equations would typically be centered, and the equation numbers would be on the left or right as you prefer.

6.8 Language Modes

The commands in this section change Calc to use a different notation for entry and display of formulas, corresponding to the conventions of some other common language such as Pascal or TeX. Objects displayed on the stack or yanked from the Calculator to an editing buffer will be formatted in the current language; objects entered in algebraic entry or yanked from another buffer will be interpreted according to the current language.

The current language has no effect on things written to or read from the trail buffer, nor does it affect numeric entry. Only algebraic entry is affected. You can make even algebraic entry ignore the current language and use the standard notation by giving a numeric prefix, e.g., *C-u '*.

For example, suppose the formula '`2*a[1] + atan(a[2])`' occurs in a C program; elsewhere in the program you need the derivatives of this formula with respect to '`a[1]`' and '`a[2]`'. First, type *d C* to switch to C notation. Now use `C-u M-# g` to grab the formula into the Calculator, *a d a[1] RET* to differentiate with respect to the first variable, and *M-# y* to yank the formula for the derivative back into your C program. Press *U* to undo the differentiation and repeat with *a d a[2] RET* for the other derivative.

Without being switched into C mode first, Calc would have misinterpreted the brackets in '`a[1]`' and '`a[2]`', would not have known that `atan` was equivalent to Calc's built-in `arctan` function, and would have written the formula back with notations (like implicit multiplication) which would not have been legal for a C program.

As another example, suppose you are maintaining a C program and a TeX document, each of which needs a copy of the same formula. You can grab the formula from the program in C mode, switch to TeX mode, and yank the formula into the document in TeX math-mode format.

Language modes are selected by typing the letter *d* followed by a shifted letter key.

6.8.1 Normal Language Modes

`d N` The *d N* (`calc-normal-language`) command selects the usual notation for Calc formulas, as described in the rest of this manual. Matrices are displayed in a multi-line tabular format, but all other objects are written in linear form, as they would be typed from the keyboard.

`d O` The *d O* (`calc-flat-language`) command selects a language identical with the normal one, except that matrices are written in one-line form along with everything else. In some applications this form may be more suitable for yanking data into other buffers.

`d b` Even in one-line mode, long formulas or vectors will still be split across multiple lines if they exceed the width of the Calculator window. The *d b* (`calc-line-breaking`) command turns this line-breaking feature on and off. (It works independently of the current language.) If you give a numeric prefix argument of five or greater to the *d b* command, that argument will specify the line width used when breaking long lines.

`d B` The *d B* (`calc-big-language`) command selects a language which uses textual approximations to various mathematical notations, such as powers, quotients, and square roots:

```
    ____________
   | a + 1    2
   | ----- + c
  \|   b
```

in place of '`sqrt((a+1)/b + c^2)`'.

Subscripts like '`a_i`' are displayed as actual subscripts in "big" mode. Double subscripts, '`a_i_j`' ('`subscr(subscr(a, i), j)`') are displayed as '`a`' with subscripts separated by commas: '`i, j`'. They must still be entered in the usual underscore notation.

One slight ambiguity of Big notation is that

```
    3
  - -
    4
```

can represent either the negative rational number $-3{:}4$, or the actual expression '`-(3/4)`'; but the latter formula would normally never be displayed because it would immediately be evaluated to $-3{:}4$ or -0.75, so this ambiguity is not a problem in typical use.

Non-decimal numbers are displayed with subscripts. Thus there is no way to tell the difference between '`16#C2`' and '`C2_16`', though generally you will know which interpretation is correct. Logarithms '`log(x,b)`' and '`log10(x)`' also use subscripts in Big mode.

In Big mode, stack entries often take up several lines. To aid readability, stack entries are separated by a blank line in this mode. You may find it useful to expand the Calc window's height using *C-x ^* (`enlarge-window`) or to make the Calc window the only one on the screen with *C-x 1* (`delete-other-windows`).

Long lines are currently not rearranged to fit the window width in Big mode, so you may need to use the `<` and `>` keys to scroll across a wide formula. For really big formulas, you may even need to use `{` and `}` to scroll up and down.

The *d U* (`calc-unformatted-language`) command altogether disables the use of operator notation in formulas. In this mode, the formula shown above would be displayed: `d U`

```
sqrt(add(div(add(a, 1), b), pow(c, 2)))
```

These four modes differ only in display format, not in the format expected for algebraic entry. The standard Calc operators work in all four modes, and unformatted notation works in any language mode (except that Mathematica mode expects square brackets instead of parentheses).

6.8.2 C, FORTRAN, and Pascal Modes

The *d C* (`calc-c-language`) command selects the conventions of the C language for display and entry of formulas. This differs from the normal language mode in a variety of (mostly minor) ways. In particular, C language operators and operator precedences are used in place of Calc's usual ones. For example, '`a^b`' means '`xor(a,b)`' in C mode; a value raised to a power is written as a function call, '`pow(a,b)`'. `d C`

In C mode, vectors and matrices use curly braces instead of brackets. Octal and hexadecimal values are written with leading '`0`' or '`0x`' rather than using the '`#`' symbol. Array subscripting is translated into `subscr` calls, so that '`a[i]`' in C mode is the same as '`a_i`' in normal mode. Assignments turn into the `assign` function, which Calc normally displays using the '`:=`' symbol.

The variables `var-pi` and `var-e` would be displayed '`pi`' and '`e`' in normal mode, but in C mode they are displayed as '`M_PI`' and '`M_E`', corresponding to the names of constants typically provided in the '`<math.h>`' header. Functions whose names are different in C are translated automatically for entry and display purposes. For example, entering '`asin(x)`' will push the formula '`arcsin(x)`' onto the stack; this formula will be displayed as '`asin(x)`' as long as C mode is in effect.

The *d P* (`calc-pascal-language`) command selects Pascal conventions. Like C mode, Pascal mode interprets array brackets and uses a different table of operators. Hexadecimal numbers are entered and displayed with a preceding dollar sign. (Thus the regular meaning of *$2* during algebraic entry does not work in Pascal mode, though *$* (and *$$*, etc.) not followed by digits works the same as always.) No special provisions are made for other non-decimal numbers, vectors, and so on, since there is no universally accepted standard way of handling these in Pascal. `d P`

The *d F* (`calc-fortran-language`) command selects FORTRAN conventions. Various function names are transformed into FORTRAN equivalents. Vectors are written as '`/1, 2, 3/`', and may be entered this way or using square brackets. Since FORTRAN uses round parentheses for both function calls and array subscripts, Calc displays both in the same way; '`a(i)`' is interpreted as a function call upon reading, and subscripts must be entered as '`subscr(a, i)`'. Also, if the variable `a` has been declared to have type `vector` or `matrix` then '`a(i)`' will be parsed as a subscript. (See *XXX* [Declarations], page *XXX*.) Usually it doesn't matter, though; if you enter the subscript expression '`a(i)`' and Calc interprets it as a function call, you'll never know the difference unless you switch to another language mode or replace `a` with an actual vector (or unless `a` happens to be the name of a built-in function!). `d F`

Underscores are allowed in variable and function names in all of these language modes. The underscore here is equivalent to the '`#`' in normal mode, or to hyphens in the underlying Emacs Lisp variable names.

FORTRAN and Pascal modes normally do not adjust the case of letters in formulas. Most built-in Calc names use lower-case letters. If you use a positive numeric prefix argument with *d P* or *d F*, these modes will use upper-case letters exclusively for display, and will convert to lower-case on input. With a negative prefix, these modes convert to lower-case for display and input.

6.8.3 TEX Language Mode

`d T` The *d T* (`calc-tex-language`) command selects the conventions of "math mode" in the TEX typesetting language, by Donald Knuth. Formulas are entered and displayed in TEX notation, as in '`\sin\left( a \over b \right)`'. Math formulas are usually enclosed by '`$ $`' signs in TEX; these should be omitted when interfacing with Calc. To Calc, the '`$`' sign has the same meaning it always does in algebraic formulas (a reference to an existing entry on the stack).

Complex numbers are displayed as in '`3 + 4i`'. Fractions and quotients are written using `\over`; binomial coefficients are written with `\choose`. Interval forms are written with `\ldots`, and error forms are written with `\pm`. Absolute values are written as in '`|x + 1|`', and the floor and ceiling functions are written with `\lfloor`, `\rfloor`, etc. The words `\left` and `\right` are ignored when reading formulas in TEX mode. Both `inf` and `uinf` are written as `\infty`; when read, `\infty` always translates to `inf`.

Function calls are written the usual way, with the function name followed by the arguments in parentheses. However, functions for which TEX has special names (like `\sin`) will use curly braces instead of parentheses for very simple arguments. During input, curly braces and parentheses work equally well for grouping, but when the document is formatted the curly braces will be invisible. Thus the printed result is $\sin 2x$ but $\sin(2 + x)$.

Function and variable names not treated specially by TEX are simply written out as-is, which will cause them to come out in italic letters in the printed document. If you invoke *d T* with a positive numeric prefix argument, names of more than one character will instead be written '`\hbox{`*name*`}`'. The '`\hbox{ }`' notation is ignored during reading. If you use a negative prefix argument, such function names are written '`\`*name*', and function names that begin with `\` during reading have the `\` removed. (Note that in this mode, long variable names are still written with `\hbox`. However, you can always make an actual variable name like `\bar` in any TEX mode.)

During reading, text of the form '`\matrix{ ... }`' is replaced by '`[ ... ]`'. The same also applies to `\pmatrix` and `\bmatrix`. The symbol '`&`' is interpreted as a comma, and the symbols '`\cr`' and '`\\`' are interpreted as semicolons. During output, matrices are displayed in '`\matrix{ a & b \\ c & d}`' format; you may need to edit this afterwards to change `\matrix` to `\pmatrix` or `\\` to `\cr`.

Accents like `\tilde` and `\bar` translate into function calls internally ('`tilde(x)`', '`bar(x)`'). The `\underline` sequence is treated as an accent. The `\vec` accent corresponds to the function name `Vec`, because `vec` is the name of a built-in Calc function. The following table shows the accents in Calc, TEX, and *eqn* (described in the next section):

Calc	TeX	eqn
acute	\acute	
bar	\bar	bar
breve	\breve	
check	\check	
dot	\dot	dot
dotdot	\ddot	dotdot
dyad		dyad
grave	\grave	
hat	\hat	hat
Prime		prime
tilde	\tilde	tilde
under	\underline	under
Vec	\vec	vec

The ‘`=>`’ (evaluates-to) operator appears as a `\to` symbol: ‘`{`*a* `\to` *b*`}`’. TeX defines `\to` as an alias for `\rightarrow`. However, if the ‘`=>`’ is the top-level expression being formatted, a slightly different notation is used: ‘`\evalto` *a* `\to` *b*’. The `\evalto` word is ignored by Calc's input routines, and is undefined in TeX. You will typically want to include one of the following definitions at the top of a TeX file that uses `\evalto`:

```
\def\evalto{}
\def\evalto#1\to{}
```

The first definition formats evaluates-to operators in the usual way. The second causes only the *b* part to appear in the printed document; the *a* part and the arrow are hidden. Another definition you may wish to use is ‘`\let\to=\Rightarrow`’ which causes `\to` to appear more like Calc's ‘`=>`’ symbol. See *XXX* [Evaluates-To Operator], page *XXX*, for a discussion of `evalto`.

The complete set of TeX control sequences that are ignored during reading is:

```
\hbox  \mbox  \text  \left  \right
\,  \>  \:  \;  \!  \quad  \qquad  \hfil  \hfill
\displaystyle  \textstyle  \dsize  \tsize
\scriptstyle  \scriptscriptstyle  \ssize  \ssize
\rm  \bf  \it  \sl  \roman  \bold  \italic  \slanted
\cal  \mit  \Cal  \Bbb  \frak  \goth
\evalto
```

Note that, because these symbols are ignored, reading a TeX formula into Calc and writing it back out may lose spacing and font information.

Also, the “discretionary multiplication sign” ‘`\*`’ is read the same as ‘`*`’.

Here are some examples of how various Calc formulas are formatted in TeX:

```
sin(a^2 / b_i)
\sin\left( a^2 \over b_i \right)
```

$$\sin\left(a^2 \over b_i \right)$$

```
[(3, 4), 3:4, 3 +/- 4, [3 .. inf)]
[3 + 4i, {3 \over 4}, 3 \pm 4, [3 \ldots \infty)]
```

$$[3+4i, {3 \over 4}, 3 \pm 4, [3 \ldots \infty)]$$

```
[abs(a), abs(a / b), floor(a), ceil(a / b)]
[|a|, \left| a \over b \right|,
 \lfloor a \rfloor, \left\lceil a \over b \right\rceil]
```

$$[|a|, \left| a \over b \right|, \lfloor a \rfloor, \left\lceil a \over b \right\rceil]$$

```
[sin(a), sin(2 a), sin(2 + a), sin(a / b)]
[\sin{a}, \sin{2 a}, \sin(2 + a),
 \sin\left( {a \over b} \right)]
```

$$[\sin{a}, \sin{2 a}, \sin(2 + a), \sin\left({a \over b} \right)]$$

First with plain *d T*, then with *C-u d T*, then finally with *C-u - d T* (using the example definition '`\def\foo#1{\tilde F(#1)}`':

```
[f(a), foo(bar), sin(pi)]
[f(a), foo(bar), \sin\pi]
[f(a), \hbox{foo}(\hbox{bar}), \sin{\pi}]
[f(a), \foo{\hbox{bar}}, \sin{\pi}]
```

$$[f(a), foo(bar), \sin\pi]$$

$$[f(a), \hbox{foo}(\hbox{bar}), \sin{\pi}]$$

$$[f(a), \tilde F(\hbox{bar}), \sin{\pi}]$$

First with '`\def\evalto{}`', then with '`\def\evalto#1\to{}`':

```
2 + 3 => 5
\evalto 2 + 3 \to 5
```

$$2 + 3 \to 5$$

$$5$$

First with standard `\to`, then with '`\let\to\Rightarrow`':

```
[2 + 3 => 5, a / 2 => (b + c) / 2]
[{2 + 3 \to 5}, {{a \over 2} \to {b + c \over 2}}]
```

$$[2 + 3 \to 5, \frac{a}{2} \to \frac{b + c}{2}]$$

$$[2 + 3 \Rightarrow 5, \frac{a}{2} \Rightarrow \frac{b + c}{2}]$$

Matrices normally, then changing `\matrix` to `\pmatrix`:

```
[ [ a / b, 0 ], [ 0, 2^(x + 1) ] ]
\matrix{ {a \over b} & 0 \\ 0 & 2^{(x + 1)} }
\pmatrix{ {a \over b} & 0 \\ 0 & 2^{(x + 1)} }
```

$$\begin{matrix} \frac{a}{b} & 0 \\ 0 & 2^{(x+1)} \end{matrix}$$

$$\begin{pmatrix} \frac{a}{b} & 0 \\ 0 & 2^{(x+1)} \end{pmatrix}$$

6.8.4 Eqn Language Mode

Eqn is another popular formatter for math formulas. It is designed for use with the TROFF text formatter, and comes standard with many versions of Unix. The *d E* (`calc-eqn-language`) command selects *eqn* notation.

The *eqn* language's main idiosyncrasy is that whitespace plays a significant part in the parsing of the language. For example, '`sqrt x+1 + y`' treats '`x+1`' as the argument of the `sqrt` operator. *Eqn* also understands more conventional grouping using curly braces: '`sqrt{x+1} + y`'. Braces are required only when the argument contains spaces.

In Calc's *eqn* mode, however, curly braces are required to delimit arguments of operators like `sqrt`. The first of the above examples would treat only the '`x`' as the argument of `sqrt`, and in fact '`sin x+1`' would be interpreted as '`sin * x + 1`', because `sin` is not a special operator in the *eqn* language. If you always surround the argument with curly braces, Calc will never misunderstand.

Calc also understands parentheses as grouping characters. Another peculiarity of *eqn*'s syntax makes it advisable to separate words with spaces from any surrounding characters that aren't curly braces, so Calc writes '`sin ( x + y )`' in *eqn* mode. (The spaces around `sin` are important to make *eqn* recognize that `sin` should be typeset in a roman font, and the spaces around `x` and `y` are a good idea just in case the *eqn* document has defined special meanings for these names, too.)

Powers and subscripts are written with the `sub` and `sup` operators, respectively. Note that the caret symbol '`^`' is treated the same as a space in *eqn* mode, as is the '`~`' symbol (these are used to introduce spaces of various widths into the typeset output of *eqn*).

As in TeX mode, Calc's formatter omits parentheses around the arguments of functions like `ln` and `sin` if they are "simple-looking"; in this case Calc surrounds the argument with braces, separated by a '`~`' from the function name: '`sin~{x}`'.

Font change codes (like '`roman` *x*') and positioning codes (like '`~`' and '`down` *n x*') are ignored by the *eqn* reader. Also ignored are the words `left`, `right`, `mark`, and `lineup`. Quotation marks in *eqn* mode input are treated the same as curly braces: '`sqrt "1+x"`' is equivalent to '`sqrt {1+x}`'; this is only an approximation to the true meaning of quotes in *eqn*, but it is good enough for most uses.

Accent codes ('*x* `dot`') are handled by treating them as function calls ('`dot(`*x*`)`') internally. See *XXX* [TeX Language Mode], page *XXX* for a table of these accent functions. The `prime` accent is treated specially if it occurs on a variable or function name: '`f prime prime ( x prime )`' is stored internally as '`f''(x')`'. For example, taking the derivative of '`f(2 x)`' with `a d x` will produce '`2 f'(2 x)`', which *eqn* mode will display as '`2 f prime ( 2 x )`'.

Assignments are written with the '`<-`' (left-arrow) symbol, and `evalto` operators are written with '`->`' or '`evalto ... ->`' (see *XXX* [TeX Language Mode], page *XXX*, for a discussion of this). The regular Calc symbols '`:=`' and '`=>`' are also recognized for these operators during reading.

Vectors in *eqn* mode use regular Calc square brackets, but matrices are formatted as '`matrix { ccol { a above b } ... }`'. The words `lcol` and `rcol` are recognized as synonyms for `ccol` during input, and are generated instead of `ccol` if the matrix justification mode so specifies.

6.8.5 Mathematica Language Mode

`d M` The *d M* (`calc-mathematica-language`) command selects the conventions of Mathematica, a powerful and popular mathematical tool from Wolfram Research, Inc. Notable differences in Mathematica mode are that the names of built-in functions are capitalized, and function calls use square brackets instead of parentheses. Thus the Calc formula '`sin(2 x)`' is entered and displayed '`Sin[2 x]`' in Mathematica mode.

Vectors and matrices use curly braces in Mathematica. Complex numbers are written '`3 + 4 I`'. The standard special constants in Calc are written `Pi`, `E`, `I`, `GoldenRatio`, `EulerGamma`, `Infinity`, `ComplexInfinity`, and `Indeterminate` in Mathematica mode. Non-decimal numbers are written, e.g., '`16^^7fff`'. Floating-point numbers in scientific notation are written '`1.23*10.^3`'. Subscripts use double square brackets: '`a[[i]]`'.

6.8.6 Maple Language Mode

The *d W* (`calc-maple-language`) command selects the conventions of Maple, another mathematical tool from the University of Waterloo. `d W`

Maple's language is much like C. Underscores are allowed in symbol names; square brackets are used for subscripts; explicit '`*`'s for multiplications are required. Use either '`^`' or '`**`' to denote powers.

Maple uses square brackets for lists and curly braces for sets. Calc interprets both notations as vectors, and displays vectors with square brackets. This means Maple sets will be converted to lists when they pass through Calc. As a special case, matrices are written as calls to the function `matrix`, given a list of lists as the argument, and can be read in this form or with all-capitals `MATRIX`.

The Maple interval notation '`2 .. 3`' has no surrounding brackets; Calc reads '`2 .. 3`' as the closed interval '`[2 .. 3]`', and writes any kind of interval as '`2 .. 3`'. This means you cannot see the difference between an open and a closed interval while in Maple display mode.

Maple writes complex numbers as '`3 + 4*I`'. Its special constants are `Pi`, `E`, `I`, and `infinity` (all three of `inf`, `uinf`, and `nan` display as `infinity`). Floating-point numbers are written '`1.23*10.^3`'.

Among things not currently handled by Calc's Maple mode are the various quote symbols, procedures and functional operators, and inert ('`&`') operators.

6.8.7 Compositions

There are several *composition functions* which allow you to get displays in a variety of formats similar to those in Big language mode. Most of these functions do not evaluate to anything; they are placeholders which are left in symbolic form by Calc's evaluator but are recognized by Calc's display formatting routines.

Two of these, `string` and `bstring`, are described elsewhere. See *XXX* [Strings], page *XXX*. For example, '`string("ABC")`' is displayed as '`ABC`'. When viewed on the stack it will be indistinguishable from the variable `ABC`, but internally it will be stored as '`string([65, 66, 67])`' and can still be manipulated this way; for example, the selection and vector commands *j 1 v v j u* would select the vector portion of this object and reverse the elements, then deselect to reveal a string whose characters had been reversed.

The composition functions do the same thing in all language modes (although their components will of course be formatted in the current language mode). The one exception is Unformatted mode (*d U*), which does not give the composition functions any special treatment. The functions are discussed here because of their relationship to the language modes.

6.8.7.1 Composition Basics

Compositions are generally formed by stacking formulas together horizontally or vertically in various ways. Those formulas are themselves compositions. TeX users will find this analogous to TeX's "boxes." Each multi-line composition has a *baseline*; horizontal compositions use the baselines to decide how formulas should be positioned relative to one another. For example, in the Big mode formula

```
           2
      a + b
17 + ------
        c
```

the second term of the sum is four lines tall and has line three as its baseline. Thus when the term is combined with 17, line three is placed on the same level as the baseline of 17.

Another important composition concept is *precedence*. This is an integer that represents the binding strength of various operators. For example, '`*`' has higher precedence (195) than '`+`' (180), which means that '`(a * b) + c`' will be formatted without the parentheses, but '`a * (b + c)`' will keep the parentheses.

The operator table used by normal and Big language modes has the following precedences:

`_`	1200	(subscripts)
`%`	1100	(as in n%)
`-`	1000	(as in -n)
`!`	1000	(as in !n)
`mod`	400	
`+/-`	300	
`!!`	210	(as in n!!)
`!`	210	(as in n!)
`^`	200	
`*`	195	(or implicit multiplication)
`/ % \`	190	
`+ -`	180	(as in a+b)
`|`	170	
`< =`	160	(and other relations)
`&&`	110	
`||`	100	
`? :`	90	
`!!!`	85	
`&&&`	80	
`|||`	75	
`:=`	50	
`::`	45	
`=>`	40	

The general rule is that if an operator with precedence n occurs as an argument to an operator with precedence m, then the argument is enclosed in parentheses if $n < m$. Top-level expressions and expressions which are function arguments, vector components, etc., are formatted with precedence zero (so that they normally never get additional parentheses).

For binary left-associative operators like '`+`', the righthand argument is actually formatted with one-higher precedence than shown in the table. This makes sure '`(a + b) + c`' omits the parentheses, but the unnatural form '`a + (b + c)`' keeps its parentheses. Right-associative operators like '`^`' format the lefthand argument with one-higher precedence.

`cprec` The `cprec` function formats an expression with an arbitrary precedence. For example, '`cprec(abc, 185)`' will combine into sums and products as follows: '`7 + abc`', '`7 (abc)`' (because this `cprec` form has higher precedence than addition, but lower precedence than multiplication).

A final composition issue is *line breaking*. Calc uses two different strategies for "flat" and "non-flat" compositions. A non-flat composition is anything that appears on multiple lines (not counting line breaking). Examples would be matrices and Big mode powers and quotients. Non-flat compositions are displayed exactly as specified. If they come out wider than the current window, you must use horizontal scrolling (`<` and `>`) to view them.

Flat compositions, on the other hand, will be broken across several lines if they are too wide to fit the window. Certain points in a composition are noted internally as *break points*. Calc's general strategy is to fill each line as much as possible, then to move down to the next line starting at the first break point that didn't fit. However, the line breaker understands the hierarchical structure of formulas. It will not break an "inner" formula if it can use an earlier break point from an "outer" formula instead. For example, a vector of sums might be formatted as:

```
[ a + b + c, d + e + f,
  g + h + i, j + k + l, m ]
```

If the '`m`' can fit, then so, it seems, could the '`g`'. But Calc prefers to break at the comma since the comma is part of a "more outer" formula. Calc would break at a plus sign only if it had to, say, if the very first sum in the vector had itself been too large to fit.

Of the composition functions described below, only `choriz` generates break points. The `bstring` function (see *XXX* [Strings], page *XXX*) also generates breakable items: A break point is added after every space (or group of spaces) except for spaces at the very beginning or end of the string.

Composition functions themselves count as levels in the formula hierarchy, so a `choriz` that is a component of a larger `choriz` will be less likely to be broken. As a special case, if a `bstring` occurs as a component of a `choriz` or `choriz`-like object (such as a vector or a list of arguments in a function call), then the break points in that `bstring` will be on the same level as the break points of the surrounding object.

6.8.7.2 Horizontal Compositions

`choriz` The `choriz` function takes a vector of objects and composes them horizontally. For example, '`choriz([17, a b/c, d])`' formats as '`17a b / cd`' in normal language mode, or as

```
  a b
17---d
   c
```

in Big language mode. This is actually one case of the general function '`choriz(`*vec*`, `*sep*`, `*prec*`)`', where either or both of *sep* and *prec* may be omitted. *Prec* gives the *precedence* to use when formatting each of the components of *vec*. The default precedence is the precedence from the surrounding environment.

Sep is a string (i.e., a vector of character codes as might be entered with `" "` notation) which should separate components of the composition. Also, if *sep* is given, the line breaker will allow lines to be broken after each occurrence of *sep*. If *sep* is omitted, the composition will not be breakable (unless any of its component compositions are breakable).

For example, '`2 choriz([a, b c, d = e], " + ", 180)`' is formatted as '`2 a + b c + (d = e)`'. To get the `choriz` to have precedence 180 "outwards" as well as "inwards," enclose it in a `cprec` form: '`2 cprec(choriz(...), 180)`' formats as '`2 (a + b c + (d = e))`'.

The baseline of a horizontal composition is the same as the baselines of the component compositions, which are all aligned.

6.8.7.3 Vertical Compositions

`cvert` The `cvert` function makes a vertical composition. Each component of the vector is centered in a column. The baseline of the result is by default the top line of the resulting composition. For example, '`f(cvert([a, bb, ccc]), cvert([a^2 + 1, b^2]))`' formats in Big mode as

```
f( a ,  2    )
  bb   a  + 1
  ccc     2
         b
```

`cbase` There are several special composition functions that work only as components of a vertical composition. The `cbase` function controls the baseline of the vertical composition; the baseline will be the same as the baseline of whatever component is enclosed in `cbase`. Thus '`f(cvert([a, cbase(bb), ccc]), cvert([a^2 + 1, cbase(b^2)]))`' displays as

```
        2
       a  + 1
   a      2
f(bb ,   b   )
  ccc
```

`ctbase`
`cbbase` There are also `ctbase` and `cbbase` functions which make the baseline of the vertical composition equal to the top or bottom line (rather than the baseline) of that component. Thus '`cvert([cbase(a / b)]) + cvert([ctbase(a / b)]) + cvert([cbbase(a / b)])`' gives

```
        a
a       -
- + a + b
b   -
    b
```

There should be only one `cbase`, `ctbase`, or `cbbase` function in a given vertical composition. These functions can also be written with no arguments: '`ctbase()`' is a zero-height object which means the baseline is the top line of the following item, and '`cbbase()`' means the baseline is the bottom line of the preceding item.

`crule` The `crule` function builds a "rule," or horizontal line, across a vertical composition. By itself '`crule()`' uses '`-`' characters to build the rule. You can specify any other character, e.g., '`crule("=")`'. The argument must be a character code or vector of exactly one character code. It is repeated to match the width of the widest item in the stack. For example, a quotient with a thick line is '`cvert([a + 1, cbase(crule("=")), b^2])`':

```
a + 1
=====
  2
 b
```

Finally, the functions `clvert` and `crvert` act exactly like `cvert` except that the items are left- or right-justified in the stack. Thus '`clvert([a, bb, ccc]) + crvert([a, bb, ccc])`' gives: clvert crvert

```
a   +   a
bb     bb
ccc   ccc
```

Like `choriz`, the vertical compositions accept a second argument which gives the precedence to use when formatting the components. Vertical compositions do not support separator strings.

6.8.7.4 Other Compositions

The `csup` function builds a superscripted expression. For example, '`csup(a, b)`' looks the same as '`a^b`' does in Big language mode. This is essentially a horizontal composition of '`a`' and '`b`', where '`b`' is shifted up so that its bottom line is one above the baseline. csup

Likewise, the `csub` function builds a subscripted expression. This shifts '`b`' down so that its top line is one below the bottom line of '`a`' (note that this is not quite analogous to `csup`). Other arrangements can be obtained by using `choriz` and `cvert` directly. csub

The `cflat` function formats its argument in "flat" mode, as obtained by '`d 0`', if the current language mode is normal or Big. It has no effect in other language modes. For example, '`a^(b/c)`' is formatted by Big mode like '`csup(a, cflat(b/c))`' to improve its readability. cflat

The `cspace` function creates horizontal space. For example, '`cspace(4)`' is effectively the same as '`string("    ")`'. A second string (i.e., vector of characters) argument is repeated instead of the space character. For example, '`cspace(4, "ab")`' looks like '`abababab`'. If the second argument is not a string, it is formatted in the normal way and then several copies of that are composed together: '`cspace(4, a^2)`' yields cspace

```
 2 2 2 2
a a a a
```

If the number argument is zero, this is a zero-width object.

The `cvspace` function creates vertical space, or a vertical stack of copies of a certain string or formatted object. The baseline is the center line of the resulting stack. A numerical argument of zero will produce an object which contributes zero height if used in a vertical composition. cvspace

There are also `ctspace` and `cbspace` functions which create vertical space with the baseline the same as the baseline of the top or bottom copy, respectively, of the second argument. Thus '`cvspace(2, a/b) + ctspace(2, a/b) + cbspace(2, a/b)`' displays as: ctspace cbspace

```
            a
            -
a           b
-     a     a
b  +  -  +  -
a     b     b
-     a
b     -
      b
```

6.8.7.5 Information about Compositions

The functions in this section are actual functions; they compose their arguments according to the current language and other display modes, then return a certain measurement of the composition as an integer.

`cwidth` The `cwidth` function measures the width, in characters, of a composition. For example, '`cwidth(a + b)`' is 5, and '`cwidth(a / b)`' is 5 in normal mode, 1 in Big mode, and 11 in TeX mode (for '`{a \over b}`'). The argument may involve the composition functions described in this section.

`cheight` The `cheight` function measures the height of a composition. This is the total number of lines in the argument's printed form.

`cascent`
`cdescent` The functions `cascent` and `cdescent` measure the amount of the height that is above (and including) the baseline, or below the baseline, respectively. Thus '`cascent(`*x*`) + cdescent(`*x*`)`' always equals '`cheight(`*x*`)`'. For a one-line formula like '`a + b`', `cascent` returns 1 and `cdescent` returns 0. For '`a / b`' in Big mode, `cascent` returns 2 and `cdescent` returns 1. The only formula for which `cascent` will return zero is '`cvspace(0)`' or equivalents.

6.8.7.6 User-Defined Compositions

`Z C` The *Z C* (`calc-user-define-composition`) command lets you define the display format for any algebraic function. You provide a formula containing a certain number of argument variables on the stack. Any time Calc formats a call to the specified function in the current language mode and with that number of arguments, Calc effectively replaces the function call with that formula with the arguments replaced.

Calc builds the default argument list by sorting all the variable names that appear in the formula into alphabetical order. You can edit this argument list before pressing *RET* if you wish. Any variables in the formula that do not appear in the argument list will be displayed literally; any arguments that do not appear in the formula will not affect the display at all.

You can define formats for built-in functions, for functions you have defined with *Z F* (see *XXX* [Algebraic Definitions], page *XXX*), or for functions which have no definitions but are being used as purely syntactic objects. You can define different formats for each language mode, and for each number of arguments, using a succession of *Z C* commands. When Calc formats a function call, it first searches for a format defined for the current language mode (and number of arguments); if there is none, it uses the format defined for the Normal language mode. If neither format exists, Calc uses its built-in standard format for that function (usually just '*func*(*args*)').

If you execute *Z C* with the number 0 on the stack instead of a formula, any defined formats for the function in the current language mode will be removed. The function will revert to its standard format.

For example, the default format for the binomial coefficient function 'choose(n, m)' in the Big language mode is

```
 n
( )
 m
```

You might prefer the notation,

```
 C
n m
```

To define this notation, first make sure you are in Big mode, then put the formula

```
choriz([cvert([cvspace(1), n]), C, cvert([cvspace(1), m])])
```

on the stack and type *Z C*. Answer the first prompt with `choose`. The second prompt will be the default argument list of '(C m n)'. Edit this list to be '(n m)' and press *RET*. Now, try it out: For example, turn simplification off with *m O* and enter 'choose(a,b) + choose(7,3)' as an algebraic entry.

```
 C  +  C
a b   7 3
```

As another example, let's define the usual notation for Stirling numbers of the first kind, 'stir1(n, m)'. This is just like the regular format for binomial coefficients but with square brackets instead of parentheses.

```
choriz([string("["), cvert([n, cbase(cvspace(1)), m]), string("]")])
```

Now type *Z C stir1 RET*, edit the argument list to '(n m)', and type *RET*.

The formula provided to *Z C* usually will involve composition functions, but it doesn't have to. Putting the formula 'a + b + c' onto the stack and typing *Z C foo RET RET* would define the function 'foo(x,y,z)' to display like 'x + y + z'. This "sum" will act exactly like a real sum for all formatting purposes (it will be parenthesized the same, and so on). However it will be computationally unrelated to a sum. For example, the formula '2 * foo(1, 2, 3)' will display as '2 (1 + 2 + 3)'. Operator precedences have caused the "sum" to be written in parentheses, but the arguments have not actually been summed. (Generally a display format like this would be undesirable, since it can easily be confused with a real sum.)

The special function `eval` can be used inside a *Z C* composition formula to cause all or part of the formula to be evaluated at display time. For example, if the formula is 'a + eval(b + c)', then 'foo(1, 2, 3)' will be displayed as '1 + 5'. Evaluation will use the default simplifications, regardless of the current simplification mode. There are also `evalsimp` and `evalextsimp` which simplify as if by *a s* and *a e* (respectively). Note that these "functions" operate only in the context of composition formulas (and also in rewrite rules, where they serve a similar purpose; see *XXX* [Rewrite Rules], page *XXX*). On the stack, a call to `eval` will be left in symbolic form.

It is not a good idea to use `eval` except as a last resort. It can cause the display of formulas to be extremely slow. For example, while 'eval(a + b)' might seem quite fast and simple, there are several situations where it could be slow. For example, '`a`' and/or '`b`' could be polar complex numbers, in which case doing the sum requires trigonometry. Or, '`a`' could be the factorial '`fact(100)`' which is unevaluated because you have typed *m O*; `eval` will evaluate it anyway to produce a large, unwieldy integer.

You can save your display formats permanently using the *Z P* command (see *XXX* [Creating User Keys], page *XXX*).

6.8.8 Syntax Tables

Syntax tables do for input what compositions do for output: They allow you to teach custom notations to Calc's formula parser. Calc keeps a separate syntax table for each language mode.

(Note that the Calc "syntax tables" discussed here are completely unrelated to the syntax tables described in the Emacs manual.)

`Z S` The *Z S* (`calc-edit-user-syntax`) command edits the syntax table for the current language mode. If you want your syntax to work in any language, define it in the normal language mode. Type *M-# M-#* to finish editing the syntax table, or *M-# x* to cancel the edit. The *m m* command saves all the syntax tables along with the other mode settings; see *XXX* [General Mode Commands], page *XXX*.

6.8.8.1 Syntax Table Basics

Parsing is the process of converting a raw string of characters, such as you would type in during algebraic entry, into a Calc formula. Calc's parser works in two stages. First, the input is broken down into *tokens*, such as words, numbers, and punctuation symbols like '`+`', '`:=`', and '`+/-`'. Space between tokens is ignored (except when it serves to separate adjacent words). Next, the parser matches this string of tokens against various built-in syntactic patterns, such as "an expression followed by '`+`' followed by another expression" or "a name followed by '`(`', zero or more expressions separated by commas, and '`)`'."

A *syntax table* is a list of user-defined *syntax rules*, which allow you to specify new patterns to define your own favorite input notations. Calc's parser always checks the syntax table for the current language mode, then the table for the normal language mode, before it uses its built-in rules to parse an algebraic formula you have entered. Each syntax rule should go on its own line; it consists of a *pattern*, a '`:=`' symbol, and a Calc formula with an optional *condition*. (Syntax rules resemble algebraic rewrite rules, but the notation for patterns is completely different.)

A syntax pattern is a list of tokens, separated by spaces. Except for a few special symbols, tokens in syntax patterns are matched literally, from left to right. For example, the rule,

```
foo ( ) := 2+3
```

would cause Calc to parse the formula '`4+foo()*5`' as if it were '`4+(2+3)*5`'. Notice that the parentheses were written as two separate tokens in the rule. As a result, the rule works for both '`foo()`' and '`foo ( )`'. If we had written the rule as '`foo () := 2+3`', then Calc would treat '`()`' as a single, indivisible token, so that '`foo( )`' would not be recognized by the rule. (It would be parsed as a regular zero-argument function call instead.) In fact, this rule would also make trouble for the rest of Calc's parser: An unrelated formula like '`bar()`' would now be tokenized into '`bar ()`' instead of '`bar ( )`', so that the standard parser for function calls would no longer recognize it!

While it is possible to make a token with a mixture of letters and punctuation symbols, this is not recommended. It is better to break it into several tokens, as we did with '`foo()`' above.

The symbol '`#`' in a syntax pattern matches any Calc expression. On the righthand side, the things that matched the '`#`'s can be referred to as '`#1`', '`#2`', and so on (where '`#1`' matches the leftmost '`#`' in the pattern). For example, these rules match a user-defined function, prefix operator, infix operator, and postfix operator, respectively:

```
foo ( # ) := myfunc(#1)
foo # := myprefix(#1)
# foo # := myinfix(#1,#2)
# foo := mypostfix(#1)
```

Thus '`foo(3)`' will parse as '`myfunc(3)`', and '`2+3 foo`' will parse as '`mypostfix(2+3)`'.

It is important to write the first two rules in the order shown, because Calc tries rules in order from first to last. If the pattern '`foo #`' came first, it would match anything that could match the '`foo ( # )`' rule, since an expression in parentheses is itself a valid expression. Thus the '`foo ( # )`' rule would never get to match anything. Likewise, the last two rules must be written in the order shown or else '`3 foo 4`' will be parsed as '`mypostfix(3) * 4`'. (Of course, the best way to avoid these ambiguities is not to use the same symbol in more than one way at the same time! In case you're not convinced, try the following exercise: How will the above rules parse the input '`foo(3,4)`', if at all? Work it out for yourself, then try it in Calc and see.)

Calc is quite flexible about what sorts of patterns are allowed. The only rule is that every pattern must begin with a literal token (like '`foo`' in the first two patterns above), or with a '`#`' followed by a literal token (as in the last two patterns). After that, any mixture is allowed, although putting two '`#`'s in a row will not be very useful since two expressions with nothing between them will be parsed as one expression that uses implicit multiplication.

As a more practical example, Maple uses the notation '`sum(a(i), i=1..10)`' for sums, which Calc's Maple mode doesn't recognize at present. To handle this syntax, we simply add the rule,

```
sum ( # , # = # .. # ) := sum(#1,#2,#3,#4)
```

to the Maple mode syntax table. As another example, C mode can't read assignment operators like '`++`' and '`*=`'. We can define these operators quite easily:

```
# *= # := muleq(#1,#2)
# ++ := postinc(#1)
++ # := preinc(#1)
```

To complete the job, we would use corresponding composition functions and *Z C* to cause these functions to display in their respective Maple and C notations. (Note that the C example ignores issues of operator precedence, which are discussed in the next section.)

You can enclose any token in quotes to prevent its usual interpretation in syntax patterns:

```
# ":=" # := becomes(#1,#2)
```

Quotes also allow you to include spaces in a token, although once again it is generally better to use two tokens than one token with an embedded space. To include an actual quotation mark in a quoted token, precede it with a backslash. (This also works to include backslashes in tokens.)

```
# "bad token" # "/\"\\" # := silly(#1,#2,#3)
```

This will parse '`3 bad token 4 /"\ 5`' to '`silly(3,4,5)`'.

The token `#` has a predefined meaning in Calc's formula parser; it is not legal to use '`"#"`' in a syntax rule. However, longer tokens that include the '`#`' character are allowed. Also, while '`"$"`' and '`"\""`' are allowed as tokens, their presence in the syntax table will prevent those characters from working in their usual ways (referring to stack entries and quoting strings, respectively).

Finally, the notation '`%%`' anywhere in a syntax table causes the rest of the line to be ignored as a comment.

6.8.8.2 Precedence

Different operators are generally assigned different *precedences*. By default, an operator defined by a rule like

```
# foo # := foo(#1,#2)
```

will have an extremely low precedence, so that '`2*3+4 foo 5 == 6`' will be parsed as '`(2*3+4) foo (5 == 6)`'. To change the precedence of an operator, use the notation '`#/`*p*' in place of '`#`', where *p* is an integer precedence level. For example, 185 lies between the precedences for '`+`' and '`*`', so if we change this rule to

```
#/185 foo #/186 := foo(#1,#2)
```

then '`2+3 foo 4*5`' will be parsed as '`2+(3 foo (4*5))`'. Also, because we've given the righthand expression slightly higher precedence, our new operator will be left-associative: '`1 foo 2 foo 3`' will be parsed as '`(1 foo 2) foo 3`'. By raising the precedence of the lefthand expression instead, we can create a right-associative operator.

See *XXX* [Composition Basics], page *XXX*, for a table of precedences of the standard Calc operators. For the precedences of operators in other language modes, look in the Calc source file '`calc-lang.el`'.

6.8.8.3 Advanced Syntax Patterns

To match a function with a variable number of arguments, you could write

```
foo ( # ) := myfunc(#1)
foo ( # , # ) := myfunc(#1,#2)
foo ( # , # , # ) := myfunc(#1,#2,#3)
```

but this isn't very elegant. To match variable numbers of items, Calc uses some notations inspired regular expressions and the "extended BNF" style used by some language designers.

```
foo ( { # }*, ) := apply(myfunc,#1)
```

The token '`{`' introduces a repeated or optional portion. One of the three tokens '`}*`', '`}+`', or '`}?`' ends the portion. These will match zero or more, one or more, or zero or one copies of the enclosed pattern, respectively. In addition, '`}*`' and '`}+`' can be followed by a separator token (with no space in between, as shown above). Thus '`{ # }*,`' matches nothing, or one expression, or several expressions separated by commas.

A complete '`{ ... }`' item matches as a vector of the items that matched inside it. For example, the above rule will match '`foo(1,2,3)`' to get '`apply(myfunc,[1,2,3])`'. The Calc `apply` function takes a function name and a vector of arguments and builds a call to the function with those arguments, so the net result is the formula '`myfunc(1,2,3)`'.

If the body of a '`{ ... }`' contains several '`#`'s (or nested '`{ ... }`' constructs), then the items will be strung together into the resulting vector. If the body does not contain anything but literal tokens, the result will always be an empty vector.

```
foo ( { # , # }+, ) := bar(#1)
foo ( { { # }*, }*; ) := matrix(#1)
```

will parse '`foo(1,2,3,4)`' as '`bar([1,2,3,4])`', and '`foo(1,2;3,4)`' as '`matrix([[1,2],[3,4]])`'. Also, after some thought it's easy to see how this pair of rules will parse '`foo(1,2,3)`' as '`matrix([[1,2,3]])`', since the first rule will only match an even number of arguments. The rule

```
foo ( # { , # , # }? ) := bar(#1,#2)
```

will parse '`foo(2,3,4)`' as '`bar(2,[3,4])`', and '`foo(2)`' as '`bar(2,[])`'.

The notation '`{ ... }?.`' (note the trailing period) works just the same as regular '`{ ... }?`', except that it does not count as an argument; the following two rules are equivalent:

```
foo ( # , { also }? # ) := bar(#1,#3)
foo ( # , { also }?. # ) := bar(#1,#2)
```

Note that in the first case the optional text counts as '`#2`', which will always be an empty vector, but in the second case no empty vector is produced.

Another variant is '`{ ... }?$`', which means the body is optional only at the end of the input formula. All built-in syntax rules in Calc use this for closing delimiters, so that during algebraic entry you can type *`[sqrt(2), sqrt(3` RET*, omitting the closing parenthesis and bracket. Calc does this automatically for trailing '`)`', '`]`', and '`>`' tokens in syntax rules, but you can use '`{ ... }?$`' explicitly to get this effect with any token (such as '`"}"`' or '`end`'). Like '`{ ... }?.`', this notation does not count as an argument. Conversely, you can use quotes, as in '`")"`', to prevent a closing-delimiter token from being automatically treated as optional.

Calc's parser does not have full backtracking, which means some patterns will not work as you might expect:

```
foo ( { # , }? # , # ) := bar(#1,#2,#3)
```

Here we are trying to make the first argument optional, so that '`foo(2,3)`' parses as '`bar([],2,3)`'. Unfortunately, Calc first tries to match '`2,`' against the optional part of the pattern, finds a match, and so goes ahead to match the rest of the pattern. Later on it will fail to match the second comma, but it doesn't know how to go back and try the other alternative at that point. One way to get around this would be to use two rules:

```
foo ( # , # , # ) := bar([#1],#2,#3)
foo ( # , # ) := bar([],#1,#2)
```

More precisely, when Calc wants to match an optional or repeated part of a pattern, it scans forward attempting to match that part. If it reaches the end of the optional part without failing, it "finalizes" its choice and proceeds. If it fails, though, it backs up and tries the other alternative. Thus Calc has "partial" backtracking. A fully backtracking parser would go on to make sure the rest of the pattern matched before finalizing the choice.

6.8.8.4 Conditional Syntax Rules

It is possible to attach a *condition* to a syntax rule. For example, the rules

```
foo ( # ) := ifoo(#1) :: integer(#1)
foo ( # ) := gfoo(#1)
```

will parse '`foo(3)`' as '`ifoo(3)`', but will parse '`foo(3.5)`' and '`foo(x)`' as calls to `gfoo`. Any number of conditions may be attached; all must be true for the rule to succeed. A condition is "true" if it evaluates to a nonzero number. See *XXX* [Logical Operations], page *XXX*, for a list of Calc functions like `integer` that perform logical tests.

The exact sequence of events is as follows: When Calc tries a rule, it first matches the pattern as usual. It then substitutes '`#1`', '`#2`', etc., in the conditions, if any. Next, the conditions are simplified and evaluated in order from left to right, as if by the `a s` algebra command (see *XXX* [Simplifying Formulas], page *XXX*). Each result is true if it is a nonzero number, or an expression that can be proven to be nonzero (see *XXX* [Declarations], page *XXX*). If the results of all conditions are true, the expression (such as '`ifoo(#1)`') has its '`#`'s substituted, and that is the result of the parse. If the result of any condition is false, Calc goes on to try the next rule in the syntax table.

Syntax rules also support `let` conditions, which operate in exactly the same way as they do in algebraic rewrite rules. See *XXX* [Other Features of Rewrite Rules], page *XXX*, for details. A `let` condition is always true, but as a side effect it defines a variable which can be used in later conditions, and also in the expression after the '`:=`' sign:

```
foo ( # ) := hifoo(x) :: let(x := #1 + 0.5) :: dnumint(x)
```

The `dnumint` function tests if a value is numerically an integer, i.e., either a true integer or an integer-valued float. This rule will parse `foo` with a half-integer argument, like '`foo(3.5)`', to a call like '`hifoo(4.)`'.

The lefthand side of a syntax rule `let` must be a simple variable, not the arbitrary pattern that is allowed in rewrite rules.

The `matches` function is also treated specially in syntax rule conditions (again, in the same way as in rewrite rules). See *XXX* [Matching Commands], page *XXX*. If the matching pattern contains meta-variables, then those meta-variables may be used in later conditions and in the result expression. The arguments to `matches` are not evaluated in this situation.

```
sum ( # , # ) := sum(#1,a,b,c) :: matches(#2, a=[b..c])
```

This is another way to implement the Maple mode `sum` notation. In this approach, we allow '`#2`' to equal the whole expression '`i=1..10`'. Then, we use `matches` to break it apart into its components. If the expression turns out not to match the pattern, the syntax rule will fail. Note that `Z S` always uses Calc's normal language mode for editing expressions in syntax rules, so we must use regular Calc notation for the interval '`[b..c]`' that will correspond to the Maple mode interval '`1..10`'.

6.9 The `Modes` Variable

The *m g* (`calc-get-modes`) command pushes onto the stack a vector of numbers that describes the various mode settings that are in effect. With a numeric prefix argument, it pushes only the *n*th mode, i.e., the *n*th element of this vector. Keyboard macros can use the *m g* command to modify their behavior based on the current mode settings. `m g`

The modes vector is also available in the special variable `Modes`. In other words, *m g* is like *s r Modes RET*. It will not work to store into this variable; in fact, if you do, `Modes` will cease to track the current modes. (The *m g* command will continue to work, however.)

In general, each number in this vector is suitable as a numeric prefix argument to the associated mode-setting command. (Recall that the `~` key takes a number from the stack and gives it as a numeric prefix to the next command.)

The elements of the modes vector are as follows:

1. Current precision. Default is 12; associated command is *p*.
2. Binary word size. Default is 32; associated command is *b w*.
3. Stack size (not counting the value about to be pushed by *m g*). This is zero if *m g* is executed with an empty stack.
4. Number radix. Default is 10; command is *d r*.
5. Floating-point format. This is the number of digits, plus the constant 0 for normal notation, 10000 for scientific notation, 20000 for engineering notation, or 30000 for fixed-point notation. These codes are acceptable as prefix arguments to the *d n* command, but note that this may lose information: For example, *d s* and *C-u 12 d s* have similar (but not quite identical) effects if the current precision is 12, but they both produce a code of 10012, which will be treated by *d n* as *C-u 12 d s*. If the precision then changes, the float format will still be frozen at 12 significant figures.
6. Angular mode. Default is 1 (degrees). Other values are 2 (radians) and 3 (HMS). The *m d* command accepts these prefixes.
7. Symbolic mode. Value is 0 or 1; default is 0. Command is *m s*.
8. Fraction mode. Value is 0 or 1; default is 0. Command is *m f*.
9. Polar mode. Value is 0 (rectangular) or 1 (polar); default is 0. Command is *m p*.
10. Matrix/scalar mode. Default value is -1. Value is 0 for scalar mode, -2 for matrix mode, or N for $N \times N$ matrix mode. Command is *m v*.
11. Simplification mode. Default is 1. Value is -1 for off (*m O*), 0 for *m N*, 2 for *m B*, 3 for *m A*, 4 for *m E*, or 5 for *m U*. The *m D* command accepts these prefixes.
12. Infinite mode. Default is -1 (off). Value is 1 if the mode is on, or 0 if the mode is on with positive zeros. Command is *m i*.

For example, the sequence *M-1 m g RET 2 + ~ p* increases the precision by two, leaving a copy of the old precision on the stack. Later, *~ p* will restore the original precision using that stack value. (This sequence might be especially useful inside a keyboard macro.)

As another example, *M-3 m g 1 - ~ DEL* deletes all but the oldest (bottommost) stack entry.

Yet another example: The HP-48 "round" command rounds a number to the current displayed precision. You could roughly emulate this in Calc with the sequence *M-5 m g 10000 % ~ c c*. (This would not work for fixed-point mode, but it wouldn't be hard to do a full emulation with the help of the *Z [* and *Z]* programming commands. See *XXX* [Conditionals in Macros], page *XXX*.)

6.10 The Calc Mode Line

This section is a summary of all symbols that can appear on the Calc mode line, the highlighted bar that appears under the Calc stack window (or under an editing window in Embedded Mode).

The basic mode line format is:

```
--%%-Calc: 12 Deg other modes         (Calculator)
```

The '`%%`' is the Emacs symbol for "read-only"; it shows that regular Emacs commands are not allowed to edit the stack buffer as if it were text.

The word '`Calc:`' changes to '`CalcEmbed:`' if Embedded Mode is enabled. The words after this describe the various Calc modes that are in effect.

The first mode is always the current precision, an integer. The second mode is always the angular mode, either `Deg`, `Rad`, or `Hms`.

Here is a complete list of the remaining symbols that can appear on the mode line:

`Alg`	Algebraic mode (`m a`; see *XXX* [Algebraic Entry], page *XXX*).
`Alg[(`	Incomplete algebraic mode (`C-u m a`).
`Alg*`	Total algebraic mode (`m t`).
`Symb`	Symbolic mode (`m s`; see *XXX* [Symbolic Mode], page *XXX*).
`Matrix`	Matrix mode (`m v`; see *XXX* [Matrix Mode], page *XXX*).
`Matrix`*n*	Dimensioned matrix mode (`C-u` *n* `m v`).
`Scalar`	Scalar mode (`m v`; see *XXX* [Matrix Mode], page *XXX*).
`Polar`	Polar complex mode (`m p`; see *XXX* [Polar Mode], page *XXX*).
`Frac`	Fraction mode (`m f`; see *XXX* [Fraction Mode], page *XXX*).
`Inf`	Infinite mode (`m i`; see *XXX* [Infinite Mode], page *XXX*).
`+Inf`	Positive infinite mode (`C-u 0 m i`).
`NoSimp`	Default simplifications off (`m O`; see *XXX* [Simplification Modes], page *XXX*).
`NumSimp`	Default simplifications for numeric arguments only (`m N`).
`BinSimp`*w*	Binary-integer simplification mode; word size *w* (`m B`, `b w`).
`AlgSimp`	Algebraic simplification mode (`m A`).
`ExtSimp`	Extended algebraic simplification mode (`m E`).
`UnitSimp`	Units simplification mode (`m U`).
`Bin`	Current radix is 2 (`d 2`; see *XXX* [Radix Modes], page *XXX*).

`Oct`	Current radix is 8 (`d 8`).
`Hex`	Current radix is 16 (`d 6`).
`Radix`*n*	Current radix is *n* (`d r`).
`Zero`	Leading zeros (`d z`; see *XXX* [Radix Modes], page *XXX*).
`Big`	Big language mode (`d B`; see *XXX* [Normal Language Modes], page *XXX*).
`Flat`	One-line normal language mode (`d O`).
`Unform`	Unformatted language mode (`d U`).
`C`	C language mode (`d C`; see *XXX* [C FORTRAN Pascal], page *XXX*).
`Pascal`	Pascal language mode (`d P`).
`Fortran`	FORTRAN language mode (`d F`).
`TeX`	TeX language mode (`d T`; see *XXX* [TeX Language Mode], page *XXX*).
`Eqn`	*Eqn* language mode (`d E`; see *XXX* [Eqn Language Mode], page *XXX*).
`Math`	Mathematica language mode (`d M`; see *XXX* [Mathematica Language Mode], page *XXX*).
`Maple`	Maple language mode (`d W`; see *XXX* [Maple Language Mode], page *XXX*).
`Norm`*n*	Normal float mode with *n* digits (`d n`; see *XXX* [Float Formats], page *XXX*).
`Fix`*n*	Fixed point mode with *n* digits after the point (`d f`).
`Sci`	Scientific notation mode (`d s`).
`Sci`*n*	Scientific notation with *n* digits (`d s`).
`Eng`	Engineering notation mode (`d e`).
`Eng`*n*	Engineering notation with *n* digits (`d e`).
`Left`*n*	Left-justified display indented by *n* (`d <`; see *XXX* [Justification], page *XXX*).
`Right`	Right-justified display (`d >`).
`Right`*n*	Right-justified display with width *n* (`d >`).
`Center`	Centered display (`d =`).
`Center`*n*	Centered display with center column *n* (`d =`).
`Wid`*n*	Line breaking with width *n* (`d b`; see *XXX* [Normal Language Modes], page *XXX*).
`Wide`	No line breaking (`d b`).

`Break`	Selections show deep structure (*j b*; see *XXX* [Making Selections], page *XXX*).
`Save`	Record modes in '`~/.emacs`' (*m R*; see *XXX* [General Mode Commands], page *XXX*).
`Local`	Record modes in Embedded buffer (*m R*).
`LocEdit`	Record modes as editing-only in Embedded buffer (*m R*).
`LocPerm`	Record modes as permanent-only in Embedded buffer (*m R*).
`Global`	Record modes as global in Embedded buffer (*m R*).
`Manual`	Automatic recomputation turned off (*m C*; see *XXX* [Automatic Recomputation], page *XXX*).
`Graph`	GNUPLOT process is alive in background (see *XXX* [Graphics], page *XXX*).
`Sel`	Top-of-stack has a selection (Embedded only; see *XXX* [Making Selections], page *XXX*).
`Dirty`	The stack display may not be up-to-date (see *XXX* [Display Modes], page *XXX*).
`Inv`	"Inverse" prefix was pressed (*I*; see *XXX* [Inverse and Hyperbolic], page *XXX*).
`Hyp`	"Hyperbolic" prefix was pressed (*H*).
`Keep`	"Keep-arguments" prefix was pressed (*K*).
`Narrow`	Stack is truncated (*d t*; see *XXX* [Truncating the Stack], page *XXX*).

In addition, the symbols `Active` and `~Active` can appear as minor modes on an Embedded buffer's mode line. See *XXX* [Embedded Mode], page *XXX*.

7 Arithmetic Functions

This chapter describes the Calc commands for doing simple calculations on numbers, such as addition, absolute value, and square roots. These commands work by removing the top one or two values from the stack, performing the desired operation, and pushing the result back onto the stack. If the operation cannot be performed, the result pushed is a formula instead of a number, such as '`2/0`' (because division by zero is illegal) or '`sqrt(x)`' (because the argument '`x`' is a formula).

Most of the commands described here can be invoked by a single keystroke. Some of the more obscure ones are two-letter sequences beginning with the `f` ("functions") prefix key.

See *XXX* [Prefix Arguments], page *XXX*, for a discussion of the effect of numeric prefix arguments on commands in this chapter which do not otherwise interpret a prefix argument.

7.1 Basic Arithmetic

The `+` (`calc-plus`) command adds two numbers. The numbers may be any of the standard Calc data types. The resulting sum is pushed back onto the stack. `+`

If both arguments of `+` are vectors or matrices (of matching dimensions), the result is a vector or matrix sum. If one argument is a vector and the other a scalar (i.e., a non-vector), the scalar is added to each of the elements of the vector to form a new vector. If the scalar is not a number, the operation is left in symbolic form: Suppose you added '`x`' to the vector '`[1,2]`'. You may want the result '`[1+x,2+x]`', or you may plan to substitute a 2-vector for '`x`' in the future. Since the Calculator can't tell which interpretation you want, it makes the safest assumption. See *XXX* [Reducing and Mapping], page *XXX*, for a way to add '`x`' to every element of a vector.

If either argument of `+` is a complex number, the result will in general be complex. If one argument is in rectangular form and the other polar, the current Polar Mode determines the form of the result. If Symbolic Mode is enabled, the sum may be left as a formula if the necessary conversions for polar addition are non-trivial.

If both arguments of `+` are HMS forms, the forms are added according to the usual conventions of hours-minutes-seconds notation. If one argument is an HMS form and the other is a number, that number is converted from degrees or radians (depending on the current Angular Mode) to HMS format and then the two HMS forms are added.

If one argument of `+` is a date form, the other can be either a real number, which advances the date by a certain number of days, or an HMS form, which advances the date by a certain amount of time. Subtracting two date forms yields the number of days between them. Adding two date forms is meaningless, but Calc interprets it as the subtraction of one date form and the negative of the other. (The negative of a date form can be understood by remembering that dates are stored as the number of days before or after Jan 1, 1 AD.)

If both arguments of `+` are error forms, the result is an error form with an appropriately computed standard deviation. If one argument is an error form and the other is a number, the number is taken to have zero error. Error forms may have symbolic formulas as their mean and/or error parts; adding these will produce a symbolic error form result. However, adding an error form to a plain symbolic formula (as in '`(a +/- b) + c`') will not work, for the same reasons just mentioned for vectors. Instead you must write '`(a +/- b) + (c +/- 0)`'.

If both arguments of `+` are modulo forms with equal values of M, or if one argument is a modulo form and the other a plain number, the result is a modulo form which represents the sum, modulo M, of the two values.

If both arguments of `+` are intervals, the result is an interval which describes all possible sums of the possible input values. If one argument is a plain number, it is treated as the interval '`[x .. x]`'.

If one argument of `+` is an infinity and the other is not, the result is that same infinity. If both arguments are infinite and in the same direction, the result is the same infinity, but if they are infinite in different directions the result is `nan`.

`-` The `-` (`calc-minus`) command subtracts two values. The top number on the stack is subtracted from the one behind it, so that the computation *5 RET 2 -* produces 3, not −3. All options available for `+` are available for `-` as well.

`*` The `*` (`calc-times`) command multiplies two numbers. If one argument is a vector and the other a scalar, the scalar is multiplied by the elements of the vector to produce a new vector. If both arguments are vectors, the interpretation depends on the dimensions of the vectors: If both arguments are matrices, a matrix multiplication is done. If one argument is a matrix and the other a plain vector, the vector is interpreted as a row vector or column vector, whichever is dimensionally correct. If both arguments are plain vectors, the result is a single scalar number which is the dot product of the two vectors.

If one argument of `*` is an HMS form and the other a number, the HMS form is multiplied by that amount. It is an error to multiply two HMS forms together, or to attempt any multiplication involving date forms. Error forms, modulo forms, and intervals can be multiplied; see the comments for addition of those forms. When two error forms or intervals are multiplied they are considered to be statistically independent; thus, '`[-2 .. 3] * [-2 .. 3]`' is '`[-6 .. 9]`', whereas '`[-2 .. 3] ^ 2`' is '`[0 .. 9]`'.

`/` The `/` (`calc-divide`) command divides two numbers. When dividing a scalar B by a square matrix A, the computation performed is B times the inverse of A. This also occurs if B is itself a vector or matrix, in which case the effect is to solve the set of linear equations represented by B. If B is a matrix with the same number of rows as A, or a plain vector (which is interpreted here as a column vector), then the equation $AX = B$ is solved for the vector or matrix X. Otherwise, if B is a non-square matrix with the same number of *columns* as A, the equation $XA = B$ is solved. If you wish a vector B to be interpreted as a row vector to be solved as $XA = B$, make it into a one-row matrix with *C-u 1 v p* first. To force a left-handed solution with a square matrix B, transpose A and B before dividing, then transpose the result.

HMS forms can be divided by real numbers or by other HMS forms. Error forms can be divided in any combination of ways. Modulo forms where both values and the modulo are integers can be divided to get an integer modulo form result. Intervals can be divided; dividing by an interval that encompasses zero or has zero as a limit will result in an infinite interval.

`^` The `^` (`calc-power`) command raises a number to a power. If the power is an integer, an exact result is computed using repeated multiplications. For non-integer powers, Calc uses Newton's method or logarithms and exponentials. Square matrices can be raised to integer powers. If either argument is an error (or interval or modulo) form, the result is also an error (or interval or modulo) form.

`I ^`
`nroot` If you press the *I* (inverse) key first, the *I ^* command computes an Nth root: *125 RET 3 I ^* computes the number 5. (This is entirely equivalent to *125 RET 1:3 ^*.)

The `\` (`calc-idiv`) command divides two numbers on the stack to produce an integer result. It is equivalent to dividing with `/`, then rounding down with `F` (`calc-floor`), only a bit more convenient and efficient. Also, since it is an all-integer operation when the arguments are integers, it avoids problems that `/ F` would have with floating-point roundoff. `\` `idiv`

The `%` (`calc-mod`) command performs a "modulo" (or "remainder") operation. Mathematically, '`a%b = a - (a\b)*b`', and is defined for all real numbers a and b (except $b = 0$). For positive b, the result will always be between 0 (inclusive) and b (exclusive). Modulo does not work for HMS forms and error forms. If a is a modulo form, its modulo is changed to b, which must be positive real number. `%`

The `:` (`calc-fdiv`) command [`fdiv` function in a formula] divides the two integers on the top of the stack to produce a fractional result. This is a convenient shorthand for enabling Fraction Mode (with `m f`) temporarily and using '`/`'. Note that during numeric entry the `:` key is interpreted as a fraction separator, so to divide 8 by 6 you would have to type `8 RET 6 RET :`. (Of course, in this case, it would be much easier simply to enter the fraction directly as `8:6 RET`!) `:` `fdiv`

The `n` (`calc-change-sign`) command negates the number on the top of the stack. It works on numbers, vectors and matrices, HMS forms, date forms, error forms, intervals, and modulo forms. `n`

The `A` (`calc-abs`) [`abs`] command computes the absolute value of a number. The result of `abs` is always a nonnegative real number: With a complex argument, it computes the complex magnitude. With a vector or matrix argument, it computes the Frobenius norm, i.e., the square root of the sum of the squares of the absolute values of the elements. The absolute value of an error form is defined by replacing the mean part with its absolute value and leaving the error part the same. The absolute value of a modulo form is undefined. The absolute value of an interval is defined in the obvious way. `A` `abs`

The `f A` (`calc-abssqr`) [`abssqr`] command computes the absolute value squared of a number, vector or matrix, or error form. `f A` `abssqr`

The `f s` (`calc-sign`) [`sign`] command returns 1 if its argument is positive, −1 if its argument is negative, or 0 if its argument is zero. In algebraic form, you can also write '`sign(a,x)`' which evaluates to '`x * sign(a)`', i.e., either '`x`', '`-x`', or zero depending on the sign of '`a`'. `f s` `sign`

The `&` (`calc-inv`) [`inv`] command computes the reciprocal of a number, i.e., $1/x$. Operating on a square matrix, it computes the inverse of that matrix. `&` `inv`

The `Q` (`calc-sqrt`) [`sqrt`] command computes the square root of a number. For a negative real argument, the result will be a complex number whose form is determined by the current Polar Mode. `Q` `sqrt`

The `f h` (`calc-hypot`) [`hypot`] command computes the square root of the sum of the squares of two numbers. That is, '`hypot(a,b)`' is the length of the hypotenuse of a right triangle with sides a and b. If the arguments are complex numbers, their squared magnitudes are used. `f h` `hypot`

The `f Q` (`calc-isqrt`) [`isqrt`] command computes the integer square root of an integer. This is the true square root of the number, rounded down to an integer. For example, '`isqrt(10)`' produces 3. Note that, like `\` [`idiv`], this uses exact integer arithmetic throughout to avoid roundoff problems. If the input is a floating-point number or other non-integer value, this is exactly the same as '`floor(sqrt(x))`'. `f Q` `isqrt`

The `f n` (`calc-min`) [`min`] and `f x` (`calc-max`) [`max`] commands take the minimum or maximum of two real numbers, respectively. These commands also work on HMS forms, date forms, intervals, and infinities. (In algebraic expressions, these functions take any number of arguments and return the maximum or minimum among all the arguments.) `f n` `f x` `min` `max`

f M
f X
mant
xpon

The *f M* (`calc-mant-part`) [`mant`] function extracts the "mantissa" part m of its floating-point argument; *f X* (`calc-xpon-part`) [`xpon`] extracts the "exponent" part e. The original number is equal to $m \times 10^e$, where m is in the interval '`[1.0 .. 10.0)`' except that $m = e = 0$ if the original number is zero. For integers and fractions, `mant` returns the number unchanged and `xpon` returns zero. The *v u* (`calc-unpack`) command can also be used to "unpack" a floating-point number; this produces an integer mantissa and exponent, with the constraint that the mantissa is not a multiple of ten (again except for the $m = e = 0$ case).

f S
scf

The *f S* (`calc-scale-float`) [`scf`] function scales a number by a given power of ten. Thus, '`scf(mant(x), xpon(x)) = x`' for any real '`x`'. The second argument must be an integer, but the first may actually be any numeric value. For example, '`scf(5,-2) = 0.05`' or '`1:20`' depending on the current Fraction Mode.

f [
f]
decr
incr

The *f [* (`calc-decrement`) [`decr`] and *f]* (`calc-increment`) [`incr`] functions decrease or increase a number by one unit. For integers, the effect is obvious. For floating-point numbers, the change is by one unit in the last place. For example, incrementing '`12.3456`' when the current precision is 6 digits yields '`12.3457`'. If the current precision had been 8 digits, the result would have been '`12.345601`'. Incrementing '`0.0`' produces 10^{-p}, where p is the current precision. These operations are defined only on integers and floats. With numeric prefix arguments, they change the number by n units.

Note that incrementing followed by decrementing, or vice-versa, will almost but not quite always cancel out. Suppose the precision is 6 digits and the number '`9.99999`' is on the stack. Incrementing will produce '`10.0000`'; decrementing will produce '`9.9999`'. One digit has been dropped. This is an unavoidable consequence of the way floating-point numbers work.

Incrementing a date/time form adjusts it by a certain number of seconds. Incrementing a pure date form adjusts it by a certain number of days.

7.2 Integer Truncation

There are four commands for truncating a real number to an integer, differing mainly in their treatment of negative numbers. All of these commands have the property that if the argument is an integer, the result is the same integer. An integer-valued floating-point argument is converted to integer form.

If you press *H* (`calc-hyperbolic`) first, the result will be expressed as an integer-valued floating-point number.

F
floor
ffloor

The *F* (`calc-floor`) [`floor` or `ffloor`] command truncates a real number to the next lower integer, i.e., toward minus infinity. Thus *3.6 F* produces 3, but *_3.6 F* produces −4.

I F
ceil
fceil

The *I F* (`calc-ceiling`) [`ceil` or `fceil`] command truncates toward positive infinity. Thus *3.6 I F* produces 4, and *_3.6 I F* produces −3.

R
round
fround

The *R* (`calc-round`) [`round` or `fround`] command rounds to the nearest integer. When the fractional part is .5 exactly, this command rounds away from zero. (All other rounding in the Calculator uses this convention as well.) Thus *3.5 R* produces 4 but *3.4 R* produces 3; *_3.5 R* produces −4.

The *I R* (`calc-trunc`) [`trunc` or `ftrunc`] command truncates toward zero. In other words, it "chops off" everything after the decimal point. Thus *3.6 I R* produces 3 and *_3.6 I R* produces −3.

`I R`
`trunc`
`ftrunc`

These functions may not be applied meaningfully to error forms, but they do work for intervals. As a convenience, applying `floor` to a modulo form floors the value part of the form. Applied to a vector, these functions operate on all elements of the vector one by one. Applied to a date form, they operate on the internal numerical representation of dates, converting a date/time form into a pure date.

There are two more rounding functions which can only be entered in algebraic notation. The `roundu` function is like `round` except that it rounds up, toward plus infinity, when the fractional part is .5. This distinction matters only for negative arguments. Also, `rounde` rounds to an even number in the case of a tie, rounding up or down as necessary. For example, '`rounde(3.5)`' and '`rounde(4.5)`' both return 4, but '`rounde(5.5)`' returns 6. The advantage of round-to-even is that the net error due to rounding after a long calculation tends to cancel out to zero. An important subtle point here is that the number being fed to `rounde` will already have been rounded to the current precision before `rounde` begins. For example, '`rounde(2.500001)`' with a current precision of 6 will incorrectly, or at least surprisingly, yield 2 because the argument will first have been rounded down to 2.5 (which `rounde` sees as an exact tie between 2 and 3).

`rounde`
`roundu`
`frounde`
`froundu`

Each of these functions, when written in algebraic formulas, allows a second argument which specifies the number of digits after the decimal point to keep. For example, '`round(123.4567, 2)`' will produce the answer 123.46, and '`round(123.4567, -1)`' will produce 120 (i.e., the cutoff is one digit to the *left* of the decimal point). A second argument of zero is equivalent to no second argument at all.

To compute the fractional part of a number (i.e., the amount which, when added to '`floor(`N`)`', will produce N) just take N modulo 1 using the `%` command.

Note also the `\` (integer quotient), *f I* (integer logarithm), and *f Q* (integer square root) commands, which are analogous to `/`, *B*, and *Q*, respectively, except that they take integer arguments and return the result rounded down to an integer.

7.3 Complex Number Functions

The *J* (`calc-conj`) [`conj`] command computes the complex conjugate of a number. For complex number $a + bi$, the complex conjugate is $a - bi$. If the argument is a real number, this command leaves it the same. If the argument is a vector or matrix, this command replaces each element by its complex conjugate.

`J`
`conj`

The *G* (`calc-argument`) [`arg`] command computes the "argument" or polar angle of a complex number. For a number in polar notation, this is simply the second component of the pair '`(`r`;`θ`)`'. The result is expressed according to the current angular mode and will be in the range −180 degrees (exclusive) to +180 degrees (inclusive), or the equivalent range in radians.

`G`
`arg`

The `calc-imaginary` command multiplies the number on the top of the stack by the imaginary number $i = (0, 1)$. This command is not normally bound to a key in Calc, but it is available on the *IMAG* button in Keypad Mode.

`f r`
`re`
The *f r* (`calc-re`) [`re`] command replaces a complex number by its real part. This command has no effect on real numbers. (As an added convenience, `re` applied to a modulo form extracts the value part.)

`f i`
`im`
The *f i* (`calc-im`) [`im`] command replaces a complex number by its imaginary part; real numbers are converted to zero. With a vector or matrix argument, these functions operate element-wise.

`v p`
The *v p* (`calc-pack`) command can pack the top two numbers on the the stack into a composite object such as a complex number. With a prefix argument of −1, it produces a rectangular complex number; with an argument of −2, it produces a polar complex number. (Also, see *XXX* [Building Vectors], page *XXX*.)

`v u`
The *v u* (`calc-unpack`) command takes the complex number (or other composite object) on the top of the stack and unpacks it into its separate components.

7.4 Conversions

The commands described in this section convert numbers from one form to another; they are two-key sequences beginning with the letter *c*.

`c f`
`pfloat`
The *c f* (`calc-float`) [`pfloat`] command converts the number on the top of the stack to floating-point form. For example, 23 is converted to 23.0, 3:2 is converted to 1.5, and 2.3 is left the same. If the value is a composite object such as a complex number or vector, each of the components is converted to floating-point. If the value is a formula, all numbers in the formula are converted to floating-point. Note that depending on the current floating-point precision, conversion to floating-point format may lose information.

As a special exception, integers which appear as powers or subscripts are not floated by *c f*. If you really want to float a power, you can use a *j s* command to select the power followed by *c f*. Because *c f* cannot examine the formula outside of the selection, it does not notice that the thing being floated is a power. See *XXX* [Selecting Subformulas], page *XXX*.

The normal *c f* command is "pervasive" in the sense that it applies to all numbers throughout the formula. The `pfloat` algebraic function never stays around in a formula; '`pfloat(a + 1)`' changes to '`a + 1.0`' as soon as it is evaluated.

`H c f`
`float`
With the Hyperbolic flag, *H c f* [`float`] operates only on the number or vector of numbers at the top level of its argument. Thus, '`float(1)`' is 1.0, but '`float(a + 1)`' is left unevaluated because its argument is not a number.

You should use *H c f* if you wish to guarantee that the final value, once all the variables have been assigned, is a float; you would use *c f* if you wish to do the conversion on the numbers that appear right now.

`c F`
`pfrac`
The *c F* (`calc-fraction`) [`pfrac`] command converts a floating-point number into a fractional approximation. By default, it produces a fraction whose decimal representation is the same as the input number, to within the current precision. You can also give a numeric prefix argument to specify a tolerance, either directly, or, if the prefix argument is zero, by using the number on top of the stack as the tolerance. If the tolerance is a positive integer, the fraction is correct to within that many significant figures. If the tolerance is a non-positive integer, it specifies how many digits fewer than the current precision to use. If the tolerance is a floating-point number, the fraction is correct to within that absolute amount.

The `pfrac` function is pervasive, like `pfloat`. There is also a non-pervasive version, *`H c F`* [`frac`], which is analogous to *`H c f`* discussed above. `H c F` `frac`

The *`c d`* (`calc-to-degrees`) [`deg`] command converts a number into degrees form. The value on the top of the stack may be an HMS form (interpreted as degrees-minutes-seconds), or a real number which will be interpreted in radians regardless of the current angular mode. `c d` `deg`

The *`c r`* (`calc-to-radians`) [`rad`] command converts an HMS form or angle in degrees into an angle in radians. `c r` `rad`

The *`c h`* (`calc-to-hms`) [`hms`] command converts a real number, interpreted according to the current angular mode, to an HMS form describing the same angle. In algebraic notation, the `hms` function also accepts three arguments: '`hms(`*h*`,` *m*`,` *s*`)`'. (The three-argument version is independent of the current angular mode.) `c h` `hms`

The `calc-from-hms` command converts the HMS form on the top of the stack into a real number according to the current angular mode.

The *`c p`* (`calc-polar`) command converts the complex number on the top of the stack from polar to rectangular form, or from rectangular to polar form, whichever is appropriate. Real numbers are left the same. This command is equivalent to the `rect` or `polar` functions in algebraic formulas, depending on the direction of conversion. (It uses `polar`, except that if the argument is already a polar complex number, it uses `rect` instead. The *`I c p`* command always uses `rect`.) `c p` `I c p` `polar` `rect`

The *`c c`* (`calc-clean`) [`pclean`] command "cleans" the number on the top of the stack. Floating point numbers are re-rounded according to the current precision. Polar numbers whose angular components have strayed from the −180 to +180 degree range are normalized. (Note that results will be undesirable if the current angular mode is different from the one under which the number was produced!) Integers and fractions are generally unaffected by this operation. Vectors and formulas are cleaned by cleaning each component number (i.e., pervasively). `c c` `pclean`

If the simplification mode is set below the default level, it is raised to the default level for the purposes of this command. Thus, *`c c`* applies the default simplifications even if their automatic application is disabled. See *XXX* [Simplification Modes], page *XXX*.

A numeric prefix argument to *`c c`* sets the floating-point precision to that value for the duration of the command. A positive prefix (of at least 3) sets the precision to the specified value; a negative or zero prefix decreases the precision by the specified amount.

The keystroke sequences *`c 0`* through *`c 9`* are equivalent to *`c c`* with the corresponding negative prefix argument. If roundoff errors have changed 2.0 into 1.999999, typing *`c 1`* to clip off one decimal place often conveniently does the trick. `c 0-9`

The *`c c`* command with a numeric prefix argument, and the *`c 0`* through *`c 9`* commands, also "clip" very small floating-point numbers to zero. If the exponent is less than or equal to the negative of the specified precision, the number is changed to 0.0. For example, if the current precision is 12, then *`c 2`* changes the vector '`[1e-8, 1e-9, 1e-10, 1e-11]`' to '`[1e-8, 1e-9, 0, 0]`'. Numbers this small generally arise from roundoff noise.

If the numbers you are using really are legitimately this small, you should avoid using the *`c 0`* through *`c 9`* commands. (The plain *`c c`* command rounds to the current precision but does not clip small numbers.)

One more property of *`c 0`* through *`c 9`*, and of *`c c`* with a prefix argument, is that integer-valued floats are converted to plain integers, so that *`c 1`* on '`[1., 1.5, 2., 2.5, 3.]`' produces '`[1, 1.5, 2, 2.5, 3]`'. This is not done for huge numbers ('`1e100`' is technically an integer-valued float, but you wouldn't want it automatically converted to a 100-digit integer).

`H c 0-9` `H c c` `clean`

With the Hyperbolic flag, *H c c* and *H c 0* through *H c 9* operate non-pervasively [`clean`].

7.5 Date Arithmetic

The commands described in this section perform various conversions and calculations involving date forms (see *XXX* [Date Forms], page *XXX*). They use the `t` (for time/date) prefix key followed by shifted letters.

The simplest date arithmetic is done using the regular `+` and `-` commands. In particular, adding a number to a date form advances the date form by a certain number of days; adding an HMS form to a date form advances the date by a certain amount of time; and subtracting two date forms produces a difference measured in days. The commands described here provide additional, more specialized operations on dates.

Many of these commands accept a numeric prefix argument; if you give plain *C-u* as the prefix, these commands will instead take the additional argument from the top of the stack.

7.5.1 Date Conversions

`t D` `date`

The *t D* (`calc-date`) [`date`] command converts a date form into a number, measured in days since Jan 1, 1 AD. The result will be an integer if *date* is a pure date form, or a fraction or float if *date* is a date/time form. Or, if its argument is a number, it converts this number into a date form.

With a numeric prefix argument, *t D* takes that many objects (up to six) from the top of the stack and interprets them in one of the following ways:

The '`date(`*year*`,` *month*`,` *day*`)`' function builds a pure date form out of the specified year, month, and day, which must all be integers. *Year* is a year number, such as 1991 (*not* the same as 91!). *Month* must be an integer in the range 1 to 12; *day* must be in the range 1 to 31. If the specified month has fewer than 31 days and *day* is too large, the equivalent day in the following month will be used.

The '`date(`*month*`,` *day*`)`' function builds a pure date form using the current year, as determined by the real-time clock.

The '`date(`*year*`,` *month*`,` *day*`,` *hms*`)`' function builds a date/time form using an *hms* form.

The '`date(`*year*`,` *month*`,` *day*`,` *hour*`,` *minute*`,` *second*`)`' function builds a date/time form. *hour* should be an integer in the range 0 to 23; *minute* should be an integer in the range 0 to 59; *second* should be any real number in the range '`[0 .. 60)`'. The last two arguments default to zero if omitted.

`t J` `julian`

The *t J* (`calc-julian`) [`julian`] command converts a date form into a Julian day count, which is the number of days since noon on Jan 1, 4713 BC. A pure date is converted to an integer Julian count representing noon of that day. A date/time form is converted to an exact floating-point Julian count, adjusted to interpret the date form in the current time zone but the Julian day count in Greenwich Mean Time. A numeric prefix argument allows you to specify the time zone; see *XXX* [Time Zones], page *XXX*. Use a prefix of zero to suppress the time zone adjustment. Note that pure date forms are never time-zone adjusted.

This command can also do the opposite conversion, from a Julian day count (either an integer day, or a floating-point day and time in the GMT zone), into a pure date form or a date/time form in the current or specified time zone.

The *t U* (`calc-unix-time`) [`unixtime`] command converts a date form into a Unix time value, which is the number of seconds since midnight on Jan 1, 1970, or vice-versa. The numeric result will be an integer if the current precision is 12 or less; for higher precisions, the result may be a float with (*precision*−12) digits after the decimal. Just as for *t J*, the numeric time is interpreted in the GMT time zone and the date form is interpreted in the current or specified zone. Some systems use Unix-like numbering but with the local time zone; give a prefix of zero to suppress the adjustment if so. `t U` `unixtime`

The *t C* (`calc-convert-time-zones`) [`tzconv`] command converts a date form from one time zone to another. You are prompted for each time zone name in turn; you can answer with any suitable Calc time zone expression (see *XXX* [Time Zones], page *XXX*). If you answer either prompt with a blank line, the local time zone is used for that prompt. You can also answer the first prompt with *$* to take the two time zone names from the stack (and the date to be converted from the third stack level). `t C` `tzconv`

7.5.2 Date Functions

The *t N* (`calc-now`) [`now`] command pushes the current date and time on the stack as a date form. The time is reported in terms of the specified time zone; with no numeric prefix argument, *t N* reports for the current time zone. `t N` `now`

The *t P* (`calc-date-part`) command extracts one part of a date form. The prefix argument specifies the part; with no argument, this command prompts for a part code from 1 to 9. The various part codes are described in the following paragraphs. `t P`

The *M-1 t P* [`year`] function extracts the year number from a date form as an integer, e.g., 1991. This and the following functions will also accept a real number for an argument, which is interpreted as a standard Calc day number. Note that this function will never return zero, since the year 1 BC immediately precedes the year 1 AD. `year`

The *M-2 t P* [`month`] function extracts the month number from a date form as an integer in the range 1 to 12. `month`

The *M-3 t P* [`day`] function extracts the day number from a date form as an integer in the range 1 to 31. `day`

The *M-4 t P* [`hour`] function extracts the hour from a date form as an integer in the range 0 (midnight) to 23. Note that 24-hour time is always used. This returns zero for a pure date form. This function (and the following two) also accept HMS forms as input. `hour`

The *M-5 t P* [`minute`] function extracts the minute from a date form as an integer in the range 0 to 59. `minute`

The *M-6 t P* [`second`] function extracts the second from a date form. If the current precision is 12 or less, the result is an integer in the range 0 to 59. For higher precisions, the result may instead be a floating-point number. `second`

The *M-7 t P* [`weekday`] function extracts the weekday number from a date form as an integer in the range 0 (Sunday) to 6 (Saturday). `weekday`

The *M-8 t P* [`yearday`] function extracts the day-of-year number from a date form as an integer in the range 1 (January 1) to 366 (December 31 of a leap year). `yearday` `time` The *M-9 t P* [`time`] function extracts the time portion of a date form as an HMS form. This returns '`0@ 0' 0"`' for a pure date form.

`t M`
`newmonth` The *t M* (`calc-new-month`) [`newmonth`] command computes a new date form that represents the first day of the month specified by the input date. The result is always a pure date form; only the year and month numbers of the input are retained. With a numeric prefix argument *n* in the range from 1 to 31, *t M* computes the *n*th day of the month. (If *n* is greater than the actual number of days in the month, or if *n* is zero, the last day of the month is used.)

`t Y`
`newyear` The *t Y* (`calc-new-year`) [`newyear`] command computes a new pure date form that represents the first day of the year specified by the input. The month, day, and time of the input date form are lost. With a numeric prefix argument *n* in the range from 1 to 366, *t Y* computes the *n*th day of the year (366 is treated as 365 in non-leap years). A prefix argument of 0 computes the last day of the year (December 31). A negative prefix argument from −1 to −12 computes the first day of the *n*th month of the year.

`t W`
`newweek` The *t W* (`calc-new-week`) [`newweek`] command computes a new pure date form that represents the Sunday on or before the input date. With a numeric prefix argument, it can be made to use any day of the week as the starting day; the argument must be in the range from 0 (Sunday) to 6 (Saturday). This function always subtracts between 0 and 6 days from the input date.

Here's an example use of `newweek`: Find the date of the next Wednesday after a given date. Using *M-3 t W* or '`newweek(d, 3)`' will give you the *preceding* Wednesday, so '`newweek(d+7, 3)`' will give you the following Wednesday. A further look at the definition of `newweek` shows that if the input date is itself a Wednesday, this formula will return the Wednesday one week in the future. An exercise for the reader is to modify this formula to yield the same day if the input is already a Wednesday. Another interesting exercise is to preserve the time-of-day portion of the input (`newweek` resets the time to midnight; hint: how can `newweek` be defined in terms of the `weekday` function?).

`pwday` The '`pwday(`*date*`)`' function (not on any key) computes the day-of-month number of the Sunday on or before *date*. With two arguments, '`pwday(`*date*`, `*day*`)`' computes the day number of the Sunday on or before day number *day* of the month specified by *date*. The *day* must be in the range from 7 to 31; if the day number is greater than the actual number of days in the month, the true number of days is used instead. Thus '`pwday(`*date*`, 7)`' finds the first Sunday of the month, and '`pwday(`*date*`, 31)`' finds the last Sunday of the month. With a third *weekday* argument, `pwday` can be made to look for any day of the week instead of Sunday.

`t I`
`incmonth` The *t I* (`calc-inc-month`) [`incmonth`] command increases a date form by one month, or by an arbitrary number of months specified by a numeric prefix argument. The time portion, if any, of the date form stays the same. The day also stays the same, except that if the new month has fewer days the day number may be reduced to lie in the valid range. For example, '`incmonth(<Jan 31, 1991>)`' produces '`<Feb 28, 1991>`'. Because of this, *t I t I* and *M-2 t I* do not always give the same results ('`<Mar 28, 1991>`' versus '`<Mar 31, 1991>`' in this case).

`incyear` The '`incyear(`*date*`, `*step*`)`' function increases a date form by the specified number of years, which may be any positive or negative integer. Note that '`incyear(d, n)`' is equivalent to '`incmonth(d, 12*n)`', but these do not have simple equivalents in terms of day arithmetic because months and years have varying lengths. If the *step* argument is omitted, 1 year is assumed. There is no keyboard command for this function; use *C-u 12 t I* instead.

There is no `newday` function at all because *F* [`floor`] serves this purpose. Similarly, instead of `incday` and `incweek` simply use $d + n$ or $d + 7n$.

See *XXX* [Basic Arithmetic], page *XXX*, for the *f]* [incr] command which can adjust a date/time form by a certain number of seconds.

7.5.3 Business Days

Often time is measured in "business days" or "working days," where weekends and holidays are skipped. Calc's normal date arithmetic functions use calendar days, so that subtracting two consecutive Mondays will yield a difference of 7 days. By contrast, subtracting two consecutive Mondays would yield 5 business days (assuming two-day weekends and the absence of holidays).

The *t +* (`calc-business-days-plus`) [`badd`] and *t -* (`calc-business-days-minus`) [`bsub`] commands perform arithmetic using business days. For *t +*, one argument must be a date form and the other must be a real number (positive or negative). If the number is not an integer, then a certain amount of time is added as well as a number of days; for example, adding 0.5 business days to a time in Friday evening will produce a time in Monday morning. It is also possible to add an HMS form; adding '`12@ 0' 0"`' also adds half a business day. For *t -*, the arguments are either a date form and a number or HMS form, or two date forms, in which case the result is the number of business days between the two dates.

`t +`
`t -`
`badd`
`bsub`

By default, Calc considers any day that is not a Saturday or Sunday to be a business day. You can define any number of additional holidays by editing the variable `Holidays`. (There is an *s H* convenience command for editing this variable.) Initially, `Holidays` contains the vector '`[sat, sun]`'. Entries in the `Holidays` vector may be any of the following kinds of objects:

- Date forms (pure dates, not date/time forms). These specify particular days which are to be treated as holidays.
- Intervals of date forms. These specify a range of days, all of which are holidays (e.g., Christmas week). See *XXX* [Interval Forms], page *XXX*.
- Nested vectors of date forms. Each date form in the vector is considered to be a holiday.
- Any Calc formula which evaluates to one of the above three things. If the formula involves the variable y, it stands for a yearly repeating holiday; y will take on various year numbers like 1992. For example, '`date(y, 12, 25)`' specifies Christmas day, and '`newweek(date(y, 11, 7), 4) + 21`' specifies Thanksgiving (which is held on the fourth Thursday of November). If the formula involves the variable m, that variable takes on month numbers from 1 to 12: '`date(y, m, 15)`' is a holiday that takes place on the 15th of every month.
- A weekday name, such as `sat` or `sun`. This is really a variable whose name is a three-letter, lower-case day name.
- An interval of year numbers (integers). This specifies the span of years over which this holiday list is to be considered valid. Any business-day arithmetic that goes outside this range will result in an error message. Use this if you are including an explicit list of holidays, rather than a formula to generate them, and you want to make sure you don't accidentally go beyond the last point where the holidays you entered are complete. If there is no limiting interval in the `Holidays` vector, the default '`[1 .. 2737]`' is used. (This is the absolute range of years for which Calc's business-day algorithms will operate.)
- An interval of HMS forms. This specifies the span of hours that are to be considered one business day. For example, if this range is '`[9@ 0' 0" .. 17@ 0' 0"]`' (i.e., 9am to 5pm), then

the business day is only eight hours long, so that *1.5 t +* on '<4:00pm Fri Dec 13, 1991>' will add one business day and four business hours to produce '<12:00pm Tue Dec 17, 1991>'. Likewise, *t -* will now express differences in time as fractions of an eight-hour day. Times before 9am will be treated as 9am by business date arithmetic, and times at or after 5pm will be treated as 4:59:59pm. If there is no HMS interval in Holidays, the full 24-hour day '[0 0' 0" .. 24 0' 0"]' is assumed. (Regardless of the type of bounds you specify, the interval is treated as inclusive on the low end and exclusive on the high end, so that the work day goes from 9am up to, but not including, 5pm.)

If the Holidays vector is empty, then *t +* and *t -* will act just like + and - because there will then be no difference between business days and calendar days.

Calc expands the intervals and formulas you give into a complete list of holidays for internal use. This is done mainly to make sure it can detect multiple holidays. (For example, '<Jan 1, 1989>' is both New Year's Day and a Sunday, but Calc's algorithms take care to count it only once when figuring the number of holidays between two dates.)

Since the complete list of holidays for all the years from 1 to 2737 would be huge, Calc actually computes only the part of the list between the smallest and largest years that have been involved in business-day calculations so far. Normally, you won't have to worry about this. Keep in mind, however, that if you do one calculation for 1992, and another for 1792, even if both involve only a small range of years, Calc will still work out all the holidays that fall in that 200-year span.

If you add a (positive) number of days to a date form that falls on a weekend or holiday, the date form is treated as if it were the most recent business day. (Thus adding one business day to a Friday, Saturday, or Sunday will all yield the following Monday.) If you subtract a number of days from a weekend or holiday, the date is effectively on the following business day. (So subtracting one business day from Saturday, Sunday, or Monday yields the preceding Friday.) The difference between two dates one or both of which fall on holidays equals the number of actual business days between them. These conventions are consistent in the sense that, if you add *n* business days to any date, the difference between the result and the original date will come out to *n* business days. (It can't be completely consistent though; a subtraction followed by an addition might come out a bit differently, since *t +* is incapable of producing a date that falls on a weekend or holiday.)

holiday There is a holiday function, not on any keys, that takes any date form and returns 1 if that date falls on a weekend or holiday, as defined in Holidays, or 0 if the date is a business day.

7.5.4 Time Zones

Time zones and daylight savings time are a complicated business. The conversions to and from Julian and Unix-style dates automatically compute the correct time zone and daylight savings adjustment to use, provided they can figure out this information. This section describes Calc's time zone adjustment algorithm in detail, in case you want to do conversions in different time zones or in case Calc's algorithms can't determine the right correction to use.

Adjustments for time zones and daylight savings time are done by *t U*, *t J*, *t N*, and *t C*, but not by any other commands. In particular, '<may 1 1991> - <apr 1 1991>' evaluates to exactly 30 days even though there is a daylight-savings transition in between. This is also true for Julian pure dates: 'julian(<may 1 1991>) - julian(<apr 1 1991>)'. But Julian and Unix date/times will adjust for daylight savings time: 'julian(<12am may 1 1991>) - julian(<12am apr 1 1991>)'

evaluates to '`29.95834`' (that's 29 days and 23 hours) because one hour was lost when daylight savings commenced on April 7, 1991.

In brief, the idiom '`julian(`*date1*`) - julian(`*date2*`)`' computes the actual number of 24-hour periods between two dates, whereas '*date1* `-` *date2*' computes the number of calendar days between two dates without taking daylight savings into account.

The `calc-time-zone` [`tzone`] command converts the time zone specified by its numeric prefix argument into a number of seconds difference from Greenwich mean time (GMT). If the argument is a number, the result is simply that value multiplied by 3600. Typical arguments for North America are 5 (Eastern) or 8 (Pacific). If Daylight Savings time is in effect, one hour should be subtracted from the normal difference. `tzone`

If you give a prefix of plain *`C-u`*, `calc-time-zone` (like other date arithmetic commands that include a time zone argument) takes the zone argument from the top of the stack. (In the case of *`t J`* and *`t U`*, the normal argument is then taken from the second-to-top stack position.) This allows you to give a non-integer time zone adjustment. The time-zone argument can also be an HMS form, or it can be a variable which is a time zone name in upper- or lower-case. For example '`tzone(PST) = tzone(8)`' and '`tzone(pdt) = tzone(7)`' (for Pacific standard and daylight savings times, respectively).

North American and European time zone names are defined as follows; note that for each time zone there is one name for standard time, another for daylight savings time, and a third for "generalized" time in which the daylight savings adjustment is computed from context.

```
YST  PST  MST  CST  EST  AST    NST    GMT   WET     MET    MEZ
 9    8    7    6    5    4     3.5     0     -1      -2     -2

YDT  PDT  MDT  CDT  EDT  ADT    NDT    BST  WETDST  METDST  MESZ
 8    7    6    5    4    3     2.5     -1    -2      -3     -3

YGT  PGT  MGT  CGT  EGT  AGT    NGT    BGT   WEGT    MEGT   MEGZ
9/8  8/7  7/6  6/5  5/4  4/3  3.5/2.5  0/-1 -1/-2   -2/-3  -2/-3
```

To define time zone names that do not appear in the above table, you must modify the Lisp variable `math-tzone-names`. This is a list of lists describing the different time zone names; its structure is best explained by an example. The three entries for Pacific Time look like this:

```
( ( "PST" 8 0 )    ; Name as an upper-case string, then standard
  ( "PDT" 8 -1 )   ; adjustment, then daylight savings adjustment.
  ( "PGT" 8 "PST" "PDT" ) )   ; Generalized time zone.
```

With no arguments, `calc-time-zone` or '`tzone()`' obtains an argument from the Calc variable `TimeZone` if a value has been stored for that variable. If not, Calc runs the Unix '`date`' command and looks for one of the above time zone names in the output; if this does not succeed, '`tzone()`' leaves itself unevaluated. The time zone name in the '`date`' output may be followed by a signed adjustment, e.g., '`GMT+5`' or '`GMT+0500`' which specifies a number of hours and minutes to be added to the base time zone. Calc stores the time zone it finds into `TimeZone` to speed later calls to '`tzone()`'.

The special time zone name `local` is equivalent to no argument, i.e., it uses the local time zone as obtained from the `date` command.

If the time zone name found is one of the standard or daylight savings zone names from the above table, and Calc's internal daylight savings algorithm says that time and zone are consistent (e.g., PDT accompanies a date that Calc's algorithm would also consider to be daylight savings, or PST accompanies a date that Calc would consider to be standard time), then Calc substitutes the corresponding generalized time zone (like PGT).

If your system does not have a suitable '`date`' command, you may wish to put a '`(setq var-TimeZone ...)`' in your Emacs initialization file to set the time zone. The easiest way to do this is to edit the `TimeZone` variable using Calc's *s T* command, then use the *s p* (`calc-permanent-variable`) command to save the value of `TimeZone` permanently.

The *t J* and `t U` commands with no numeric prefix arguments do the same thing as '`tzone()`'. If the current time zone is a generalized time zone, e.g., `EGT`, Calc examines the date being converted to tell whether to use standard or daylight savings time. But if the current time zone is explicit, e.g., `EST` or `EDT`, then that adjustment is used exactly and Calc's daylight savings algorithm is not consulted.

Some places don't follow the usual rules for daylight savings time. The state of Arizona, for example, does not observe daylight savings time. If you run Calc during the winter season in Arizona, the Unix `date` command will report `MST` time zone, which Calc will change to `MGT`. If you then convert a time that lies in the summer months, Calc will apply an incorrect daylight savings time adjustment. To avoid this, set your `TimeZone` variable explicitly to `MST` to force the use of standard, non-daylight-savings time.

By default Calc always considers daylight savings time to begin at 2 a.m. on the first Sunday of April, and to end at 2 a.m. on the last Sunday of October. This is the rule that has been in effect in North America since 1987. If you are in a country that uses different rules for computing daylight savings time, you have two choices: Write your own daylight savings hook, or control time zones explicitly by setting the `TimeZone` variable and/or always giving a time-zone argument for the conversion functions.

The Lisp variable `math-daylight-savings-hook` holds the name of a function that is used to compute the daylight savings adjustment for a given date. The default is `math-std-daylight-savings`, which computes an adjustment (either 0 or -1) using the North American rules given above.

The daylight savings hook function is called with four arguments: The date, as a floating-point number in standard Calc format; a six-element list of the date decomposed into year, month, day, hour, minute, and second, respectively; a string which contains the generalized time zone name in upper-case, e.g., `"WEGT"`; and a special adjustment to be applied to the hour value when converting into a generalized time zone (see below).

The Lisp function `math-prev-weekday-in-month` is useful for daylight savings computations. This is an internal version of the user-level `pwday` function described in the previous section. It takes four arguments: The floating-point date value, the corresponding six-element date list, the day-of-month number, and the weekday number (0-6).

The default daylight savings hook ignores the time zone name, but a more sophisticated hook could use different algorithms for different time zones. It would also be possible to use different algorithms depending on the year number, but the default hook always uses the algorithm for 1987 and later. Here is a listing of the default daylight savings hook:

```
(defun math-std-daylight-savings (date dt zone bump)
  (cond ((< (nth 1 dt) 4) 0)
        ((= (nth 1 dt) 4)
         (let ((sunday (math-prev-weekday-in-month date dt 7 0)))
                (cond ((< (nth 2 dt) sunday) 0)
                      ((= (nth 2 dt) sunday)
                       (if (>= (nth 3 dt) (+ 3 bump)) -1 0))
                      (t -1))))
            ((< (nth 1 dt) 10) -1)
            ((= (nth 1 dt) 10)
             (let ((sunday (math-prev-weekday-in-month date dt 31 0)))
               (cond ((< (nth 2 dt) sunday) -1)
                     ((= (nth 2 dt) sunday)
                      (if (>= (nth 3 dt) (+ 2 bump)) 0 -1))
                     (t 0))))
            (t 0))
      )
```

The `bump` parameter is equal to zero when Calc is converting from a date form in a generalized time zone into a GMT date value. It is -1 when Calc is converting in the other direction. The adjustments shown above ensure that the conversion behaves correctly and reasonably around the 2 a.m. transition in each direction.

There is a "missing" hour between 2 a.m. and 3 a.m. at the beginning of daylight savings time; converting a date/time form that falls in this hour results in a time value for the following hour, from 3 a.m. to 4 a.m. At the end of daylight savings time, the hour from 1 a.m. to 2 a.m. repeats itself; converting a date/time form that falls in in this hour results in a time value for the first manifestion of that time (*not* the one that occurs one hour later).

If `math-daylight-savings-hook` is `nil`, then the daylight savings adjustment is always taken to be zero.

In algebraic formulas, '`tzone(`*zone*`,` *date*`)`' computes the time zone adjustment for a given zone name at a given date. The *date* is ignored unless *zone* is a generalized time zone. If *date* is a date form, the daylight savings computation is applied to it as it appears. If *date* is a numeric date value, it is adjusted for the daylight-savings version of *zone* before being given to the daylight savings hook. This odd-sounding rule ensures that the daylight-savings computation is always done in local time, not in the GMT time that a numeric *date* is typically represented in.

The '`dsadj(`*date*`,` *zone*`)`' function computes the daylight savings adjustment that is appropriate for *date* in time zone *zone*. If *zone* is explicitly in or not in daylight savings time (e.g., `PDT` or `PST`) the *date* is ignored. If *zone* is a generalized time zone, the algorithms described above are used. If *zone* is omitted, the computation is done for the current time zone. `dsadj`

See *XXX* [Reporting Bugs], page *XXX*, for the address of Calc's author, if you should wish to contribute your improved versions of `math-tzone-names` and `math-daylight-savings-hook` to the Calc distribution.

7.6 Financial Functions

Calc's financial or business functions use the `b` prefix key followed by a shifted letter. (The `b` prefix followed by a lower-case letter is used for operations on binary numbers.)

Note that the rate and the number of intervals given to these functions must be on the same time scale, e.g., both months or both years. Mixing an annual interest rate with a time expressed in months will give you very wrong answers!

It is wise to compute these functions to a higher precision than you really need, just to make sure your answer is correct to the last penny; also, you may wish to check the definitions at the end of this section to make sure the functions have the meaning you expect.

7.6.1 Percentages

`M-%`
`%`
`percent`

The *M-%* (`calc-percent`) command takes a percentage value, say 5.4, and converts it to an equivalent actual number. For example, *5.4 M-%* enters 0.054 on the stack. (That's the *META* or *ESC* key combined with *%*.)

Actually, *M-%* creates a formula of the form '`5.4%`'. You can enter '`5.4%`' yourself during algebraic entry. The '`%`' operator simply means, "the preceding value divided by 100." The '`%`' operator has very high precedence, so that '`1+8%`' is interpreted as '`1+(8%)`', not as '`(1+8)%`'. (The '`%`' operator is just a postfix notation for the `percent` function, just like '`20!`' is the notation for '`fact(20)`', or twenty-factorial.)

The formula '`5.4%`' would normally evaluate immediately to 0.054, but the *M-%* command suppresses evaluation as it puts the formula onto the stack. However, the next Calc command that uses the formula '`5.4%`' will evaluate it as its first step. The net effect is that you get to look at '`5.4%`' on the stack, but Calc commands see it as '`0.054`', which is what they expect.

In particular, '`5.4%`' and '`0.054`' are suitable values for the *rate* arguments of the various financial functions, but the number '`5.4`' is probably *not* suitable—it represents a rate of 540 percent!

The key sequence *M-% ** effectively means "percent-of." For example, *68 RET 25 M-% ** computes 17, which is 25% of 68 (and also 68% of 25, which comes out to the same thing).

`c %`

The *c %* (`calc-convert-percent`) command converts the value on the top of the stack from numeric to percentage form. For example, if 0.08 is on the stack, *c %* converts it to '`8%`'. The quantity is the same, it's just represented differently. (Contrast this with *M-%*, which would convert this number to '`0.08%`'.) The `=` key is a convenient way to convert a formula like '`8%`' back to numeric form, 0.08.

To compute what percentage one quantity is of another quantity, use */ c %*. For example, *17 RET 68 / c %* displays '`25%`'.

`b %`
`relch`

The *b %* (`calc-percent-change`) [`relch`] command calculates the percentage change from one number to another. For example, *40 RET 50 b %* produces the answer '`25%`', since 50 is 25% larger than 40. A negative result represents a decrease: *50 RET 40 b %* produces '`-20%`', since 40 is 20% smaller than 50. (The answers are different in magnitude because, in the first case, we're increasing by 25% of 40, but in the second case, we're decreasing by 20% of 50.) The effect of *40 RET 50 b %* is to compute $(50-40)/40$, converting the answer to percentage form as if by *c %*.

7.6.2 Future Value

`b F`
`fv`

The *b F* (`calc-fin-fv`) [`fv`] command computes the future value of an investment. It takes three arguments from the stack: '`fv(`*rate*`,` *n*`,` *payment*`)`'. If you give payments of *payment* every year for *n* years, and the money you have paid earns interest at *rate* per year, then this function tells you what your investment would be worth at the end of the period. (The actual interval doesn't have to be years, as long as *n* and *rate* are expressed in terms of the same intervals.) This function assumes payments occur at the *end* of each interval.

The *I b F* [`fvb`] command does the same computation, but assuming your payments are at the beginning of each interval. Suppose you plan to deposit $1000 per year in a savings account earning 5.4% interest, starting right now. How much will be in the account after five years? `fvb(5.4%, 5, 1000) = 5870.73`. Thus you will have earned $870 worth of interest over the years. Using the stack, this calculation would have been *5.4 M-% 5 RET 1000 I b F*. Note that the rate is expressed as a number between 0 and 1, *not* as a percentage. `I b F` `fvb`

The *H b F* [`fvl`] command computes the future value of an initial lump sum investment. Suppose you could deposit those five thousand dollars in the bank right now; how much would they be worth in five years? `fvl(5.4%, 5, 5000) = 6503.89`. `H b F` `fvl`

The algebraic functions `fv` and `fvb` accept an optional fourth argument, which is used as an initial lump sum in the sense of `fvl`. In other words, `fv(`*rate*`, `*n*`, `*payment*`, `*initial*`) = fv(`*rate*`, `*n*`, `*payment*`) + fvl(`*rate*`, `*n*`, `*initial*`)`.

To illustrate the relationships between these functions, we could do the `fvb` calculation "by hand" using `fvl`. The final balance will be the sum of the contributions of our five deposits at various times. The first deposit earns interest for five years: `fvl(5.4%, 5, 1000) = 1300.78`. The second deposit only earns interest for four years: `fvl(5.4%, 4, 1000) = 1234.13`. And so on down to the last deposit, which earns one year's interest: `fvl(5.4%, 1, 1000) = 1054.00`. The sum of these five values is, sure enough, $5870.73, just as was computed by `fvb` directly.

What does `fv(5.4%, 5, 1000) = 5569.96` mean? The payments are now at the ends of the periods. The end of one year is the same as the beginning of the next, so what this really means is that we've lost the payment at year zero (which contributed $1300.78), but we're now counting the payment at year five (which, since it didn't have a chance to earn interest, counts as $1000). Indeed, $5569.96 = 5870.73 - 1300.78 + 1000$ (give or take a bit of roundoff error).

7.6.3 Present Value

The *b P* (`calc-fin-pv`) [`pv`] command computes the present value of an investment. Like `fv`, it takes three arguments: `pv(`*rate*`, `*n*`, `*payment*`)`. It computes the present value of a series of regular payments. Suppose you have the chance to make an investment that will pay $2000 per year over the next four years; as you receive these payments you can put them in the bank at 9% interest. You want to know whether it is better to make the investment, or to keep the money in the bank where it earns 9% interest right from the start. The calculation `pv(9%, 4, 2000)` gives the result 6479.44. If your initial investment must be less than this, say, $6000, then the investment is worthwhile. But if you had to put up $7000, then it would be better just to leave it in the bank. `b P` `pv`

Here is the interpretation of the result of `pv`: You are trying to compare the return from the investment you are considering, which is `fv(9%, 4, 2000) = 9146.26`, with the return from leaving the money in the bank, which is `fvl(9%, 4, `*x*`)` where x is the amount of money you would have to put up in advance. The `pv` function finds the break-even point, $x = 6479.44$, at which `fvl(9%, 4, 6479.44)` is also equal to 9146.26. This is the largest amount you should be willing to invest.

The *I b P* [`pvb`] command solves the same problem, but with payments occurring at the beginning of each interval. It has the same relationship to `fvb` as `pv` has to `fv`. For example `pvb(9%, 4, 2000) = 7062.59`, a larger number than `pv` produced because we get to start earning interest on the return from our investment sooner. `I b P` `pvb`

`H b P` `pvl` The *H b P* [pvl] command computes the present value of an investment that will pay off in one lump sum at the end of the period. For example, if we get our $8000 all at the end of the four years, `pvl(9%, 4, 8000) = 5667.40`. This is much less than `pv` reported, because we don't earn any interest on the return from this investment. Note that `pvl` and `fvl` are simple inverses: `fvl(9%, 4, 5667.40) = 8000`.

You can give an optional fourth lump-sum argument to `pv` and `pvb`; this is handled in exactly the same way as the fourth argument for `fv` and `fvb`.

`b N` `npv` The *b N* (`calc-fin-npv`) [npv] command computes the net present value of a series of irregular investments. The first argument is the interest rate. The second argument is a vector which represents the expected return from the investment at the end of each interval. For example, if the rate represents a yearly interest rate, then the vector elements are the return from the first year, second year, and so on.

Thus, `npv(9%, [2000,2000,2000,2000]) = pv(9%, 4, 2000) = 6479.44`. Obviously this function is more interesting when the payments are not all the same!

The `npv` function can actually have two or more arguments. Multiple arguments are interpreted in the same way as for the vector statistical functions like `vsum`. See *XXX* [Single-Variable Statistics], page *XXX*. Basically, if there are several payment arguments, each either a vector or a plain number, all these values are collected left-to-right into the complete list of payments. A numeric prefix argument on the *b N* command says how many payment values or vectors to take from the stack.

`I b N` `npvb` The *I b N* [npvb] command computes the net present value where payments occur at the beginning of each interval rather than at the end.

7.6.4 Related Financial Functions

The functions in this section are basically inverses of the present value functions with respect to the various arguments.

`b M` `pmt` The *b M* (`calc-fin-pmt`) [pmt] command computes the amount of periodic payment necessary to amortize a loan. Thus `pmt(`*rate*`, `*n*`, `*amount*`)` equals the value of *payment* such that `pv(`*rate*`, `*n*`, `*payment*`) = `*amount*.

`I b M` `pmtb` The *I b M* [pmtb] command does the same computation but using `pvb` instead of `pv`. Like `pv` and `pvb`, these functions can also take a fourth argument which represents an initial lump-sum investment.

`H b M` The *H b M* key just invokes the `fvl` function, which is the inverse of `pvl`. There is no explicit `pmtl` function.

`b #` `nper` The *b #* (`calc-fin-nper`) [nper] command computes the number of regular payments necessary to amortize a loan. Thus `nper(`*rate*`, `*payment*`, `*amount*`)` equals the value of *n* such that `pv(`*rate*`, `*n*`, `*payment*`) = `*amount*. If *payment* is too small ever to amortize a loan for *amount* at interest rate *rate*, the `nper` function is left in symbolic form.

`I b #` `nperb` The *I b #* [nperb] command does the same computation but using `pvb` instead of `pv`. You can give a fourth lump-sum argument to these functions, but the computation will be rather slow in the four-argument case.

The *H b #* [nperl] command does the same computation using `pvl`. By exchanging *payment* and *amount* you can also get the solution for `fvl`. For example, `nperl(8%, 2000, 1000) = 9.006`, so if you place $1000 in a bank account earning 8%, it will take nine years to grow to $2000. `H b #` `nperl`

The *b T* (`calc-fin-rate`) [rate] command computes the rate of return on an investment. This is also an inverse of `pv`: `rate(`*n*`, `*payment*`, `*amount*`)` computes the value of *rate* such that `pv(`*rate*`, `*n*`, `*payment*`) = `*amount*. The result is expressed as a formula like '`6.3%`'. `b T` `rate`

The *I b T* [rateb] and *H b T* [ratel] commands solve the analogous equations with `pvb` or `pvl` in place of `pv`. Also, `rate` and `rateb` can accept an optional fourth argument just like `pv` and `pvb`. To redo the above example from a different perspective, `ratel(9, 2000, 1000) = 8.00597%`, which says you will need an interest rate of 8% in order to double your account in nine years. `I b T` `H b T` `rateb` `ratel`

The *b I* (`calc-fin-irr`) [irr] command is the analogous function to `rate` but for net present value. Its argument is a vector of payments. Thus `irr(`*payments*`)` computes the *rate* such that `npv(`*rate*`, `*payments*`) = 0`; this rate is known as the *internal rate of return*. `b I` `irr`

The *I b I* [irrb] command computes the internal rate of return assuming payments occur at the beginning of each period. `I b I` `irrb`

7.6.5 Depreciation Functions

The functions in this section calculate *depreciation*, which is the amount of value that a possession loses over time. These functions are characterized by three parameters: *cost*, the original cost of the asset; *salvage*, the value the asset will have at the end of its expected "useful life"; and *life*, the number of years (or other periods) of the expected useful life.

There are several methods for calculating depreciation that differ in the way they spread the depreciation over the lifetime of the asset.

The *b S* (`calc-fin-sln`) [sln] command computes the "straight-line" depreciation. In this method, the asset depreciates by the same amount every year (or period). For example, '`sln(12000, 2000, 5)`' returns 2000. The asset costs $12000 initially and will be worth $2000 after five years; it loses $2000 per year. `b S` `sln`

The *b Y* (`calc-fin-syd`) [syd] command computes the accelerated "sum-of-years'-digits" depreciation. Here the depreciation is higher during the early years of the asset's life. Since the depreciation is different each year, *b Y* takes a fourth *period* parameter which specifies which year is requested, from 1 to *life*. If *period* is outside this range, the `syd` function will return zero. `b Y` `syd`

The *b D* (`calc-fin-ddb`) [ddb] command computes an accelerated depreciation using the double-declining balance method. It also takes a fourth *period* parameter. `b D` `ddb`

For symmetry, the `sln` function will accept a *period* parameter as well, although it will ignore its value except that the return value will as usual be zero if *period* is out of range.

For example, pushing the vector $[1,2,3,4,5]$ (perhaps with *v x 5*) and then mapping *V M ' [sln(12000,2000,5,$), syd(12000,2000,5,$), ddb(12000,2000,5,$)] RET* produces a matrix that allows us to compare the three depreciation methods:

```
[ [ 2000, 3333, 4800 ]
  [ 2000, 2667, 2880 ]
  [ 2000, 2000, 1728 ]
  [ 2000, 1333,  592 ]
  [ 2000,  667,   0  ] ]
```

(Values have been rounded to nearest integers in this figure.) We see that `sln` depreciates by the same amount each year, *`syd`* depreciates more at the beginning and less at the end, and *`ddb`* weights the depreciation even more toward the beginning.

Summing columns with *`V R : +`* yields [10000, 10000, 10000]; the total depreciation in any method is (by definition) the difference between the cost and the salvage value.

7.6.6 Definitions

For your reference, here are the actual formulas used to compute Calc's financial functions.

Calc will not evaluate a financial function unless the *rate* or *n* argument is known. However, *payment* or *amount* can be a variable. Calc expands these functions according to the formulas below for symbolic arguments only when you use the `a "` (`calc-expand-formula`) command, or when taking derivatives or integrals or solving equations involving the functions.

$$\mathtt{fv}(r,n,p) = p\frac{(1+r)^n - 1}{r}$$

$$\mathtt{fvb}(r,n,p) = p\frac{((1+r)^n - 1)(1+r)}{r}$$

$$\mathtt{fvl}(r,n,p) = p(1+r)^n$$

$$\mathtt{pv}(r,n,p) = p\frac{1-(1+r)^{-n}}{r}$$

$$\mathtt{pvb}(r,n,p) = p\frac{(1-(1+r)^{-n})(1+r)}{r}$$

$$\mathtt{pvl}(r,n,p) = p(1+r)^{-n}$$

$$\mathtt{npv}(r,[a,b,c]) = a(1+r)^{-1} + b(1+r)^{-2} + c(1+r)^{-3}$$

$$\mathtt{npvb}(r,[a,b,c]) = a + b(1+r)^{-1} + c(1+r)^{-2}$$

$$\mathtt{pmt}(r,n,a,x) = \frac{(a - x(1+r)^{-n})r}{1-(1+r)^{-n}}$$

$$\mathtt{pmtb}(r,n,a,x) = \frac{(a - x(1+r)^{-n})r}{(1-(1+r)^{-n})(1+r)}$$

$$\mathtt{nper}(r,p,a) = -\mathtt{log}(1 - \frac{ar}{p}, 1+r)$$

$$\mathtt{nperb}(r,p,a) = -\mathtt{log}(1 - \frac{ar}{p(1+r)}, 1+r)$$

$$\mathtt{nperl}(r,p,a) = -\mathtt{log}(\frac{a}{p}, 1+r)$$

$$\mathtt{ratel}(n,p,a) = \frac{p^{1/n}}{a^{1/n}} - 1$$

$$\mathtt{sln}(c, s, l) = \frac{c - s}{l}$$
$$\mathtt{syd}(c, s, l, p) = \frac{(c - s)(l - p + 1)}{l(l + 1)/2}$$
$$\mathtt{ddb}(c, s, l, p) = \frac{2(c - \text{depreciation so far})}{l}$$

In `pmt` and `pmtb`, $x = 0$ if omitted.

These functions accept any numeric objects, including error forms, intervals, and even (though not very usefully) complex numbers. The above formulas specify exactly the behavior of these functions with all sorts of inputs.

Note that if the first argument to the `log` in `nper` is negative, `nper` leaves itself in symbolic form rather than returning a (financially meaningless) complex number.

'`rate(num, pmt, amt)`' solves the equation '`pv(rate, num, pmt) = amt`' for '`rate`' using *H a R* (`calc-find-root`), with the interval '`[.01% .. 100%]`' for an initial guess. The `rateb` function is the same except that it uses `pvb`. Note that `ratel` can be solved directly; its formula is shown in the above list.

Similarly, '`irr(pmts)`' solves the equation '`npv(rate, pmts) = 0`' for '`rate`'.

If you give a fourth argument to `nper` or `nperb`, Calc will also use *H a R* to solve the equation using an initial guess interval of '`[0 .. 100]`'.

A fourth argument to `fv` simply sums the two components calculated from the above formulas for `fv` and `fvl`. The same is true of `fvb`, `pv`, and `pvb`.

The *ddb* function is computed iteratively; the "book" value starts out equal to *cost*, and decreases according to the above formula for the specified number of periods. If the book value would decrease below *salvage*, it only decreases to *salvage* and the depreciation is zero for all subsequent periods. The `ddb` function returns the amount the book value decreased in the specified period.

The Calc financial function names were borrowed mostly from Microsoft Excel and Borland's Quattro. The `ratel` function corresponds to '`@CGR`' in Borland's Reflex. The `nper` and `nperl` functions correspond to '`@TERM`' and '`@CTERM`' in Quattro, respectively. Beware that the Calc functions may take their arguments in a different order than the corresponding functions in your favorite spreadsheet.

7.7 Binary Number Functions

The commands in this chapter all use two-letter sequences beginning with the *b* prefix.

The "binary" operations actually work regardless of the currently displayed radix, although their results make the most sense in a radix like 2, 8, or 16 (as obtained by the *d 2*, *d 8*, or *d 6* commands, respectively). You may also wish to enable display of leading zeros with *d z*. See *XXX* [Radix Modes], page *XXX*.

The Calculator maintains a current *word size* w, an arbitrary positive or negative integer. For a positive word size, all of the binary operations described here operate modulo 2^w. In particular, negative arguments are converted to positive integers modulo 2^w by all binary functions.

If the word size is negative, binary operations produce 2's complement integers from -2^{-w-1} to $2^{-w-1} - 1$ inclusive. Either mode accepts inputs in any range; the sign of w affects only the results produced.

`b c` `clip` The *b c* (`calc-clip`) [`clip`] command can be used to clip a number by reducing it modulo 2^w. The commands described in this chapter automatically clip their results to the current word size. Note that other operations like addition do not use the current word size, since integer addition generally is not "binary." (However, see *XXX* [Simplification Modes], page *XXX*, `calc-bin-simplify-mode`.) For example, with a word size of 8 bits *b c* converts a number to the range 0 to 255; with a word size of −8 *b c* converts to the range −128 to 127.

`b w` The default word size is 32 bits. All operations except the shifts and rotates allow you to specify a different word size for that one operation by giving a numeric prefix argument: *C-u 8 b c* clips the top of stack to the range 0 to 255 regardless of the current word size. To set the word size permanently, use *b w* (`calc-word-size`). This command displays a prompt with the current word size; press *RET* immediately to keep this word size, or type a new word size at the prompt.

When the binary operations are written in symbolic form, they take an optional second (or third) word-size parameter. When a formula like '`and(a,b)`' is finally evaluated, the word size current at that time will be used, but when '`and(a,b,-8)`' is evaluated, a word size of −8 will always be used. A symbolic binary function will be left in symbolic form unless the all of its argument(s) are integers or integer-valued floats.

If either or both arguments are modulo forms for which M is a power of two, that power of two is taken as the word size unless a numeric prefix argument overrides it. The current word size is never consulted when modulo-power-of-two forms are involved.

`b a` `and` The *b a* (`calc-and`) [`and`] command computes the bitwise AND of the two numbers on the top of the stack. In other words, for each of the w binary digits of the two numbers (pairwise), the corresponding bit of the result is 1 if and only if both input bits are 1: '`and(2#1100, 2#1010) = 2#1000`'.

`b o` `or` The *b o* (`calc-or`) [`or`] command computes the bitwise inclusive OR of two numbers. A bit is 1 if either of the input bits, or both, are 1: '`or(2#1100, 2#1010) = 2#1110`'.

`b x` `xor` The *b x* (`calc-xor`) [`xor`] command computes the bitwise exclusive OR of two numbers. A bit is 1 if exactly one of the input bits is 1: '`xor(2#1100, 2#1010) = 2#0110`'.

`b d` `diff` The *b d* (`calc-diff`) [`diff`] command computes the bitwise difference of two numbers; this is defined by '`diff(a,b) = and(a,not(b))`', so that '`diff(2#1100, 2#1010) = 2#0100`'.

`b n` `not` The *b n* (`calc-not`) [`not`] command computes the bitwise NOT of a number. A bit is 1 if the input bit is 0 and vice-versa.

`b l` `lsh` The *b l* (`calc-lshift-binary`) [`lsh`] command shifts a number left by one bit, or by the number of bits specified in the numeric prefix argument. A negative prefix argument performs a logical right shift, in which zeros are shifted in on the left. In symbolic form, '`lsh(a)`' is short for '`lsh(a,1)`', which in turn is short for '`lsh(a,n,w)`'. Bits shifted "off the end," according to the current word size, are lost.

`H b l` `H b r` ... The *H b l* command also does a left shift, but it takes two arguments from the stack (the value to shift, and, at top-of-stack, the number of bits to shift). This version interprets the prefix argument just like the regular binary operations, i.e., as a word size. The Hyperbolic flag has a similar effect on the rest of the binary shift and rotate commands.

`b r` `rsh` The *b r* (`calc-rshift-binary`) [`rsh`] command shifts a number right by one bit, or by the number of bits specified in the numeric prefix argument: '`rsh(a,n) = lsh(a,-n)`'.

`b L` `ash` The *b L* (`calc-lshift-arith`) [`ash`] command shifts a number left. It is analogous to `lsh`, except that if the shift is rightward (the prefix argument is negative), an arithmetic shift is performed as described below.

The *b R* (`calc-rshift-arith`) [`rash`] command performs an “arithmetic” shift to the right, in which the leftmost bit (according to the current word size) is duplicated rather than shifting in zeros. This corresponds to dividing by a power of two where the input is interpreted as a signed, twos-complement number. (The distinction between the ‘`rsh`’ and ‘`rash`’ operations is totally independent from whether the word size is positive or negative.) With a negative prefix argument, this performs a standard left shift. `b R` `rash`

The *b t* (`calc-rotate-binary`) [`rot`] command rotates a number one bit to the left. The leftmost bit (according to the current word size) is dropped off the left and shifted in on the right. With a numeric prefix argument, the number is rotated that many bits to the left or right. `b t` `rot`

See *XXX* [Set Operations], page *XXX*, for the *b p* and *b u* commands that pack and unpack binary integers into sets. (For example, *b u* unpacks the number ‘`2#11001`’ to the set of bit-numbers ‘`[0, 3, 4]`’.) Type *b u V #* to count the number of “1” bits in a binary integer.

Another interesting use of the set representation of binary integers is to reverse the bits in, say, a 32-bit integer. Type *b u* to unpack; type *31 TAB -* to replace each bit-number in the set with 31 minus that bit-number; type *b p* to pack the set back into a binary integer.

8 Scientific Functions

The functions described here perform trigonometric and other transcendental calculations. They generally produce floating-point answers correct to the full current precision. The *H* (Hyperbolic) and *I* (Inverse) flag keys must be used to get some of these functions from the keyboard.

`P` `H P` `I P` `H I P`

One miscellanous command is shift-*P* (`calc-pi`), which pushes the value of π (at the current precision) onto the stack. With the Hyperbolic flag, it pushes the value e, the base of natural logarithms. With the Inverse flag, it pushes Euler's constant γ (about 0.5772). With both Inverse and Hyperbolic, it pushes the "golden ratio" ϕ (about 1.618). (At present, Euler's constant is not available to unlimited precision; Calc knows only the first 100 digits.) In Symbolic mode, these commands push the actual variables '`pi`', '`e`', '`gamma`', and '`phi`', respectively, instead of their values; see *XXX* [Symbolic Mode], page *XXX*.

`Q` `I Q` `sqr`

The *Q* (`calc-sqrt`) [`sqrt`] function is described elsewhere; see *XXX* [Basic Arithmetic], page *XXX*. With the Inverse flag [`sqr`], this command computes the square of the argument.

See *XXX* [Prefix Arguments], page *XXX*, for a discussion of the effect of numeric prefix arguments on commands in this chapter which do not otherwise interpret a prefix argument.

8.1 Logarithmic Functions

`L` `ln`

The shift-*L* (`calc-ln`) [`ln`] command computes the natural logarithm of the real or complex number on the top of the stack. With the Inverse flag it computes the exponential function instead, although this is redundant with the *E* command.

`E` `exp`

The shift-*E* (`calc-exp`) [`exp`] command computes the exponential, i.e., e raised to the power of the number on the stack. The meanings of the Inverse and Hyperbolic flags follow from those for the `calc-ln` command.

`H L` `H E` `log10` `exp10`

The *H L* (`calc-log10`) [`log10`] command computes the common (base-10) logarithm of a number. (With the Inverse flag [`exp10`], it raises ten to a given power.) Note that the common logarithm of a complex number is computed by taking the natural logarithm and dividing by $\ln 10$.

`B` `I B` `log` `alog`

The *B* (`calc-log`) [`log`] command computes a logarithm to any base. For example, ***1024 RET 2 B*** produces 10, since $2^{10} = 1024$. In certain cases like '`log(3,9)`', the result will be either 1:2 or 0.5 depending on the current Fraction Mode setting. With the Inverse flag [`alog`], this command is similar to `^` except that the order of the arguments is reversed.

`f I` `ilog`

The *f I* (`calc-ilog`) [`ilog`] command computes the integer logarithm of a number to any base. The number and the base must themselves be positive integers. This is the true logarithm, rounded down to an integer. Thus ***ilog(x,10)*** is 3 for all x in the range from 1000 to 9999. If both arguments are positive integers, exact integer arithmetic is used; otherwise, this is equivalent to '`floor(log(x,b))`'.

`f E` `expm1`

The *f E* (`calc-expm1`) [`expm1`] command computes $e^x - 1$, but using an algorithm that produces a more accurate answer when the result is close to zero, i.e., when e^x is close to one.

`f L` `lnp1`

The *f L* (`calc-lnp1`) [`lnp1`] command computes $\ln(x+1)$, producing a more accurate answer when x is close to zero.

8.2 Trigonometric/Hyperbolic Functions

The shift-*S* (`calc-sin`) [`sin`] command computes the sine of an angle or complex number. If the input is an HMS form, it is interpreted as degrees-minutes-seconds; otherwise, the input is interpreted according to the current angular mode. It is best to use Radians mode when operating on complex numbers. `S` `sin`

Calc's "units" mechanism includes angular units like `deg`, `rad`, and `grad`. While '`sin(45 deg)`' is not evaluated all the time, the *u s* (`calc-simplify-units`) command will simplify '`sin(45 deg)`' by taking the sine of 45 degrees, regardless of the current angular mode. See *XXX* [Basic Operations on Units], page *XXX*.

Also, the symbolic variable `pi` is not ordinarily recognized in arguments to trigonometric functions, as in '`sin(3 pi / 4)`', but the *a s* (`calc-simplify`) command recognizes many such formulas when the current angular mode is radians *and* symbolic mode is enabled; this example would be replaced by '`sqrt(2) / 2`'. See *XXX* [Symbolic Mode], page *XXX*. Beware, this simplification occurs even if you have stored a different value in the variable '`pi`'; this is one reason why changing built-in variables is a bad idea. Arguments of the form x plus a multiple of $\pi/2$ are also simplified. Calc includes similar formulas for `cos` and `tan`.

The *a s* command knows all angles which are integer multiples of $\pi/12$, $\pi/10$, or $\pi/8$ radians. In degrees mode, analogous simplifications occur for integer multiples of 15 or 18 degrees, and for arguments plus multiples of 90 degrees.

With the Inverse flag, `calc-sin` computes an arcsine. This is also available as the `calc-arcsin` command or `arcsin` algebraic function. The returned argument is converted to degrees, radians, or HMS notation depending on the current angular mode. `I S` `arcsin`

With the Hyperbolic flag, `calc-sin` computes the hyperbolic sine, also available as `calc-sinh` [`sinh`]. With the Hyperbolic and Inverse flags, it computes the hyperbolic arcsine (`calc-arcsinh`) [`arcsinh`]. `H S` `sinh` `H I S` `arcsinh`

The shift-*C* (`calc-cos`) [`cos`] command computes the cosine of an angle or complex number, and shift-*T* (`calc-tan`) [`tan`] computes the tangent, along with all the various inverse and hyperbolic variants of these functions. `C` `cos` `...`

The *f T* (`calc-arctan2`) [`arctan2`] command takes two numbers from the stack and computes the arc tangent of their ratio. The result is in the full range from −180 (exclusive) to +180 (inclusive) degrees, or the analogous range in radians. A similar result would be obtained with */* followed by *I T*, but the value would only be in the range from −90 to +90 degrees since the division loses information about the signs of the two components, and an error might result from an explicit division by zero which `arctan2` would avoid. By (arbitrary) definition, '`arctan2(0,0)=0`'. `f T` `arctan2`

The `calc-sincos` [`sincos`] command computes the sine and cosine of a number, returning them as a vector of the form '`[`*cos*`,` *sin*`]`'. With the Inverse flag [`arcsincos`], this command takes a two-element vector as an argument and computes `arctan2` of the elements. (This command does not accept the Hyperbolic flag.) `sincos` `arc...`

8.3 Advanced Mathematical Functions

Calc can compute a variety of less common functions that arise in various branches of mathematics. All of the functions described in this section allow arbitrary complex arguments and, except as noted, will work to arbitrarily large precisions. They can not at present handle error forms or intervals as arguments.

NOTE: These functions are still experimental. In particular, their accuracy is not guaranteed in all domains. It is advisable to set the current precision comfortably higher than you actually need when using these functions. Also, these functions may be impractically slow for some values of the arguments.

`f g`
`gamma`

The *f g* (`calc-gamma`) [`gamma`] command computes the Euler gamma function. For positive integer arguments, this is related to the factorial function: '`gamma(n+1) = fact(n)`'. For general complex arguments the gamma function can be defined by the following definite integral: $\Gamma(a) = \int_0^\infty t^{a-1} e^t dt$. (The actual implementation uses far more efficient computational methods.)

`f G`
`gammaP`
`...`

The *f G* (`calc-inc-gamma`) [`gammaP`] command computes the incomplete gamma function, denoted '`P(a,x)`'. This is defined by the integral, $P(a,x) = \left(\int_0^x t^{a-1} e^t dt\right) / \Gamma(a)$. This implies that '`gammaP(a,inf) = 1`' for any a (see the definition of the normal gamma function).

Several other varieties of incomplete gamma function are defined. The complement of $P(a,x)$, called $Q(a,x) = 1 - P(a,x)$ by some authors, is computed by the *I f G* [`gammaQ`] command. You can think of this as taking the other half of the integral, from x to infinity.

The functions corresponding to the integrals that define $P(a,x)$ and $Q(a,x)$ but without the normalizing $1/\Gamma(a)$ factor are called $\gamma(a,x)$ and $\Gamma(a,x)$, respectively. You can obtain these using the *H f G* [`gammag`] and *I H f G* [`gammaG`] commands.

`f b`
`beta`

The *f b* (`calc-beta`) [`beta`] command computes the Euler beta function, which is defined in terms of the gamma function as $B(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$, or by $B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1} dt$.

`f B`
`H f B`
`betaI`
`betaB`

The *f B* (`calc-inc-beta`) [`betaI`] command computes the incomplete beta function $I(x,a,b)$. It is defined by $I(x,a,b) = \left(\int_0^x t^{a-1}(1-t)^{b-1} dt\right) / B(a,b)$. Once again, the *H* (hyperbolic) prefix gives the corresponding un-normalized version [`betaB`].

`f e`
`I f e`
`erf`
`erfc`

The *f e* (`calc-erf`) [`erf`] command computes the error function $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$. The complementary error function *I f e* (`calc-erfc`) [`erfc`] is the corresponding integral from '`x`' to infinity; the sum $\mathrm{erf}(x) + \mathrm{erfc}(x) = 1$.

`f j`
`f y`
`besJ`
`besY`

The *f j* (`calc-bessel-J`) [`besJ`] and *f y* (`calc-bessel-Y`) [`besY`] commands compute the Bessel functions of the first and second kinds, respectively. In '`besJ(n,x)`' and '`besY(n,x)`' the "order" parameter n is often an integer, but is not required to be one. Calc's implementation of the Bessel functions currently limits the precision to 8 digits, and may not be exact even to that precision. Use with care!

8.4 Branch Cuts and Principal Values

All of the logarithmic, trigonometric, and other scientific functions are defined for complex numbers as well as for reals. This section describes the values returned in cases where the general result is a family of possible values. Calc follows section 12.5.3 of Steele's *Common Lisp, the Language*, second edition, in these matters. This section will describe each function briefly; for a more detailed discussion (including some nifty diagrams), consult Steele's book.

Note that the branch cuts for `arctan` and `arctanh` were changed between the first and second editions of Steele. Versions of Calc starting with 2.00 follow the second edition.

The new branch cuts exactly match those of the HP-28/48 calculators. They also match those of Mathematica 1.2, except that Mathematica's `arctan` cut is always in the right half of the complex plane, and its `arctanh` cut is always in the top half of the plane. Calc's cuts are continuous with quadrants I and III for `arctan`, or II and IV for `arctanh`.

Note: The current implementations of these functions with complex arguments are designed with proper behavior around the branch cuts in mind, *not* efficiency or accuracy. You may need to increase the floating precision and wait a while to get suitable answers from them.

For '`sqrt(a+bi)`': When $a < 0$ and b is small but positive or zero, the result is close to the $+i$ axis. For b small and negative, the result is close to the $-i$ axis. The result always lies in the right half of the complex plane.

For '`ln(a+bi)`': The real part is defined as '`ln(abs(a+bi))`'. The imaginary part is defined as '`arg(a+bi) = arctan2(b,a)`'. Thus the branch cuts for `sqrt` and `ln` both lie on the negative real axis.

The following table describes these branch cuts in another way. If the real and imaginary parts of z are as shown, then the real and imaginary parts of $f(z)$ will be as shown. Here `eps` stands for a small positive value; each occurrence of `eps` may stand for a different small value.

```
      z           sqrt(z)      ln(z)
----------------------------------------
   +,   0          +,  0       any, 0
   -,   0          0,  +       any, pi
   -, +eps       +eps, +      +eps, +
   -, -eps       +eps, -      +eps, -
```

For '`z1^z2`': This is defined by '`exp(ln(z1)*z2)`'. One interesting consequence of this is that '`(-8)^1:3`' does not evaluate to -2 as you might expect, but to the complex number $(1., 1.732)$. Both of these are valid cube roots of -8 (as is $(1., -1.732)$); Calc chooses a perhaps less-obvious root for the sake of mathematical consistency.

For '`arcsin(z)`': This is defined by '`-i*ln(i*z + sqrt(1-z^2))`'. The branch cuts are on the real axis, less than -1 and greater than 1.

For '`arccos(z)`': This is defined by '`-i*ln(z + i*sqrt(1-z^2))`', or equivalently by '`pi/2 - arcsin(z)`'. The branch cuts are on the real axis, less than -1 and greater than 1.

For '`arctan(z)`': This is defined by '`(ln(1+i*z) - ln(1-i*z)) / (2*i)`'. The branch cuts are on the imaginary axis, below $-i$ and above i.

For '`arcsinh(z)`': This is defined by '`ln(z + sqrt(1+z^2))`'. The branch cuts are on the imaginary axis, below $-i$ and above i.

For '`arccosh(z)`': This is defined by '`ln(z + (z+1)*sqrt((z-1)/(z+1)))`'. The branch cut is on the real axis less than 1.

For '`arctanh(z)`': This is defined by '`(ln(1+z) - ln(1-z)) / 2`'. The branch cuts are on the real axis, less than −1 and greater than 1.

The following tables for `arcsin`, `arccos`, and `arctan` assume the current angular mode is radians. The hyperbolic functions operate independently of the angular mode.

z	arcsin(z)	arccos(z)
(-1..1), 0	(-pi/2..pi/2), 0	(0..pi), 0
(-1..1), +eps	(-pi/2..pi/2), +eps	(0..pi), -eps
(-1..1), -eps	(-pi/2..pi/2), -eps	(0..pi), +eps
<-1, 0	-pi/2, +	pi, -
<-1, +eps	-pi/2 + eps, +	pi - eps, -
<-1, -eps	-pi/2 + eps, -	pi - eps, +
>1, 0	pi/2, -	0, +
>1, +eps	pi/2 - eps, +	+eps, -
>1, -eps	pi/2 - eps, -	+eps, +

z	arccosh(z)	arctanh(z)
(-1..1), 0	0, (0..pi)	any, 0
(-1..1), +eps	+eps, (0..pi)	any, +eps
(-1..1), -eps	+eps, (-pi..0)	any, -eps
<-1, 0	+, pi	-, pi/2
<-1, +eps	+, pi - eps	-, pi/2 - eps
<-1, -eps	+, -pi + eps	-, -pi/2 + eps
>1, 0	+, 0	+, -pi/2
>1, +eps	+, +eps	+, pi/2 - eps
>1, -eps	+, -eps	+, -pi/2 + eps

z	arcsinh(z)	arctan(z)
0, (-1..1)	0, (-pi/2..pi/2)	0, any
0, <-1	-, -pi/2	-pi/2, -
+eps, <-1	+, -pi/2 + eps	pi/2 - eps, -
-eps, <-1	-, -pi/2 + eps	-pi/2 + eps, -
0, >1	+, pi/2	pi/2, +
+eps, >1	+, pi/2 - eps	pi/2 - eps, +
-eps, >1	-, pi/2 - eps	-pi/2 + eps, +

Finally, the following identities help to illustrate the relationship between the complex trigonometric and hyperbolic functions. They are valid everywhere, including on the branch cuts.

```
sin(i*z)  = i*sinh(z)       arcsin(i*z)  = i*arcsinh(z)
cos(i*z)  =   cosh(z)       arcsinh(i*z) = i*arcsin(z)
tan(i*z)  = i*tanh(z)       arctan(i*z)  = i*arctanh(z)
sinh(i*z) = i*sin(z)        cosh(i*z)    =   cos(z)
```

The "advanced math" functions (gamma, Bessel, etc.) are also defined for general complex arguments, but their branch cuts and principal values are not rigorously specified at present.

8.5 Random Numbers

The *k r* (`calc-random`) [`random`] command produces random numbers of various sorts. `k r` `random`

Given a positive numeric prefix argument M, it produces a random integer N in the range $0 \le N < M$. Each of the M values appears with equal probability.

With no numeric prefix argument, the *k r* command takes its argument from the stack instead. Once again, if this is a positive integer M the result is a random integer less than M. However, note that while numeric prefix arguments are limited to six digits or so, an M taken from the stack can be arbitrarily large. If M is negative, the result is a random integer in the range $M < N \le 0$.

If the value on the stack is a floating-point number M, the result is a random floating-point number N in the range $0 \le N < M$ or $M < N \le 0$, according to the sign of M.

If M is zero, the result is a Gaussian-distributed random real number; the distribution has a mean of zero and a standard deviation of one. The algorithm used generates random numbers in pairs; thus, every other call to this function will be especially fast.

If M is an error form m `+/-` σ where m and σ are both real numbers, the result uses a Gaussian distribution with mean m and standard deviation σ.

If M is an interval form, the lower and upper bounds specify the acceptable limits of the random numbers. If both bounds are integers, the result is a random integer in the specified range. If either bound is floating-point, the result is a random real number in the specified range. If the interval is open at either end, the result will be sure not to equal that end value. (This makes a big difference for integer intervals, but for floating-point intervals it's relatively minor: with a precision of 6, '`random([1.0..2.0))`' will return any of one million numbers from 1.00000 to 1.99999; '`random([1.0..2.0])`' may additionally return 2.00000, but the probability of this happening is extremely small.)

If M is a vector, the result is one element taken at random from the vector. All elements of the vector are given equal probabilities.

The sequence of numbers produced by *k r* is completely random by default, i.e., the sequence is seeded each time you start Calc using the current time and other information. You can get a reproducible sequence by storing a particular "seed value" in the Calc variable `RandSeed`. Any integer will do for a seed; integers of from 1 to 12 digits are good. If you later store a different integer into `RandSeed`, Calc will switch to a different pseudo-random sequence. If you "unstore" `RandSeed`, Calc will re-seed itself from the current time. If you store the same integer that you used before back into `RandSeed`, you will get the exact same sequence of random numbers as before.

The `calc-rrandom` command (not on any key) produces a random real number between zero and one. It is equivalent to '`random(1.0)`'.

The *k a* (`calc-random-again`) command produces another random number, re-using the most recent value of M. With a numeric prefix argument n, it produces n more random numbers using that value of M. `k a`

The *k h* (`calc-shuffle`) command produces a vector of several random values with no duplicates. The value on the top of the stack specifies the set from which the random values are drawn, and may be any of the M formats described above. The numeric prefix argument gives the length of the desired list. (If you do not provide a numeric prefix argument, the length of the list is taken from the top of the stack, and M from second-to-top.) `k h` `shuffle`

If M is a floating-point number, zero, or an error form (so that the random values are being drawn from the set of real numbers) there is little practical difference between using `k h` and using `k r` several times. But if the set of possible values consists of just a few integers, or the elements of a vector, then there is a very real chance that multiple `k r`'s will produce the same number more than once. The `k h` command produces a vector whose elements are always distinct. (Actually, there is a slight exception: If M is a vector, no given vector element will be drawn more than once, but if several elements of M are equal, they may each make it into the result vector.)

One use of `k h` is to rearrange a list at random. This happens if the prefix argument is equal to the number of values in the list: `[1, 1.5, 2, 2.5, 3] 5 k h` might produce the permuted list '`[2.5, 1, 1.5, 3, 2]`'. As a convenient feature, if the argument n is negative it is replaced by the size of the set represented by M. Naturally, this is allowed only when M specifies a small discrete set of possibilities.

To do the equivalent of `k h` but with duplications allowed, given M on the stack and with n just entered as a numeric prefix, use `v b` to build a vector of copies of M, then use `V M k r` to "map" the normal `k r` function over the elements of this vector. See *XXX* [Matrix Functions], page *XXX*.

8.5.1 Random Number Generator

Calc's random number generator uses several methods to ensure that the numbers it produces are highly random. Knuth's *Art of Computer Programming*, Volume II, contains a thorough description of the theory of random number generators and their measurement and characterization.

If `RandSeed` has no stored value, Calc calls Emacs' built-in `random` function to get a stream of random numbers, which it then treats in various ways to avoid problems inherent in the simple random number generators that many systems use to implement `random`.

When Calc's random number generator is first invoked, it "seeds" the low-level random sequence using the time of day, so that the random number sequence will be different every time you use Calc.

Since Emacs Lisp doesn't specify the range of values that will be returned by its `random` function, Calc exercises the function several times to estimate the range. When Calc subsequently uses the `random` function, it takes only 10 bits of the result near the most-significant end. (It avoids at least the bottom four bits, preferably more, and also tries to avoid the top two bits.) This strategy works well with the linear congruential generators that are typically used to implement `random`.

If `RandSeed` contains an integer, Calc uses this integer to seed an "additive congruential" method (Knuth's algorithm 3.2.2A, computing $X_{n-55} - X_{n-24}$). This method expands the seed value into a large table which is maintained internally; the variable `RandSeed` is changed from, e.g., 42 to the vector [42] to indicate that the seed has been absorbed into this table. When `RandSeed` contains a vector, `k r` and related commands continue to use the same internal table as last time. There is no way to extract the complete state of the random number generator so that you can restart it from any point; you can only restart it from the same initial seed value. A simple way to restart from the same seed is to type `s r RandSeed` to get the seed vector, `v u` to unpack it back into a number, then `s t RandSeed` to reseed the generator with that number.

Calc uses a "shuffling" method as described in algorithm 3.2.2B of Knuth. It fills a table with 13 random 10-bit numbers. Then, to generate a new random number, it uses the previous number to index into the table, picks the value it finds there as the new random number, then replaces that table entry with a new value obtained from a call to the base random number generator (either the additive congruential generator or the `random` function supplied by the system). If there are any flaws in the base generator, shuffling will tend to even them out. But if the system provides an excellent `random` function, shuffling will not damage its randomness.

To create a random integer of a certain number of digits, Calc builds the integer three decimal digits at a time. For each group of three digits, Calc calls its 10-bit shuffling random number generator (which returns a value from 0 to 1023); if the random value is 1000 or more, Calc throws it out and tries again until it gets a suitable value.

To create a random floating-point number with precision p, Calc simply creates a random p-digit integer and multiplies by 10^{-p}. The resulting random numbers should be very clean, but note that relatively small numbers will have few significant random digits. In other words, with a precision of 12, you will occasionally get numbers on the order of 10^{-9} or 10^{-10}, but those numbers will only have two or three random digits since they correspond to small integers times 10^{-12}.

To create a random integer in the interval '`[0 .. m)`', Calc counts the digits in m, creates a random integer with three additional digits, then reduces modulo m. Unless m is a power of ten the resulting values will be very slightly biased toward the lower numbers, but this bias will be less than 0.1%. (For example, if m is 42, Calc will reduce a random integer less than 100000 modulo 42 to get a result less than 42. It is easy to show that the numbers 40 and 41 will be only 2380/2381 as likely to result from this modulo operation as numbers 39 and below.) If m is a power of ten, however, the numbers should be completely unbiased.

The Gaussian random numbers generated by '`random(0.0)`' use the "polar" method described in Knuth section 3.4.1C. This method generates a pair of Gaussian random numbers at a time, so only every other call to '`random(0.0)`' will require significant calculations.

8.6 Combinatorial Functions

Commands relating to combinatorics and number theory begin with the *k* key prefix.

The *k g* (`calc-gcd`) [`gcd`] command computes the Greatest Common Divisor of two integers. It also accepts fractions; the GCD of two fractions is defined by taking the GCD of the numerators, and the LCM of the denominators. This definition is consistent with the idea that '`a / gcd(a,x)`' should yield an integer for any '`a`' and '`x`'. For other types of arguments, the operation is left in symbolic form. `k g` `gcd`

The *k l* (`calc-lcm`) [`lcm`] command computes the Least Common Multiple of two integers or fractions. The product of the LCM and GCD of two numbers is equal to the product of the numbers. `k l` `lcm`

The *k E* (`calc-extended-gcd`) [`egcd`] command computes the GCD of two integers x and y and returns a vector $[g, a, b]$ where $g = \gcd(x, y) = ax + by$. `k E` `egcd`

The *!* (`calc-factorial`) [`fact`] command computes the factorial of the number at the top of the stack. If the number is an integer, the result is an exact integer. If the number is an integer-valued float, the result is a floating-point approximation. If the number is a non-integral real number, the generalized factorial is used, as defined by the Euler Gamma function. Please note that computation of large factorials can be slow; using floating-point format will help since fewer digits must be maintained. The same is true of many of the commands in this section. `!` `fact` `k d` `dfact` The *k d* (`calc-double-factorial`) [`dfact`] command computes the "double factorial" of an integer. For an even integer, this is the product of even integers from 2 to N. For an odd integer, this is the product of odd integers from 3 to N. If the argument is an integer-valued float, the result is a floating-point approximation. This function is undefined for negative even integers. The notation $N!!$ is also recognized for double factorials.

`k c` `choose` The *k c* (`calc-choose`) [`choose`] command computes the binomial coefficient N-choose-M, where M is the number on the top of the stack and N is second-to-top. If both arguments are integers, the result is an exact integer. Otherwise, the result is a floating-point approximation. The binomial coefficient is defined for all real numbers by $\frac{N!}{M!(N-M)!}$.

`H k c` `perm` The *H k c* (`calc-perm`) [`perm`] command computes the number-of-permutations function $\frac{N!}{(N-M)!}$.

`k b` `H k b` `bern` The *k b* (`calc-bernoulli-number`) [`bern`] command computes a given Bernoulli number. The value at the top of the stack is a nonnegative integer n that specifies which Bernoulli number is desired. The *H k b* command computes a Bernoulli polynomial, taking n from the second-to-top position and x from the top of the stack. If x is a variable or formula the result is a polynomial in x; if x is a number the result is a number.

`k e` `H k e` `euler` The *k e* (`calc-euler-number`) [`euler`] command similarly computes an Euler number, and *H k e* computes an Euler polynomial. Bernoulli and Euler numbers occur in the Taylor expansions of several functions.

`k s` `H k s` `stir1` `stir2` The *k s* (`calc-stirling-number`) [`stir1`] command computes a Stirling number of the first kind $\left[{n \atop m}\right]$, given two integers n and m on the stack. The *H k s* [`stir2`] command computes a Stirling number of the second kind $\left\{{n \atop m}\right\}$. These are the number of m-cycle permutations of n objects, and the number of ways to partition n objects into m non-empty sets, respectively.

`k p` The *k p* (`calc-prime-test`) command checks if the integer on the top of the stack is prime. For integers less than eight million, the answer is always exact and reasonably fast. For larger integers, a probabilistic method is used (see Knuth vol. II, section 4.5.4, algorithm P). The number is first checked against small prime factors (up to 13). Then, any number of iterations of the algorithm are performed. Each step either discovers that the number is non-prime, or substantially increases the certainty that the number is prime. After a few steps, the chance that a number was mistakenly described as prime will be less than one percent. (Indeed, this is a worst-case estimate of the probability; in practice even a single iteration is quite reliable.) After the *k p* command, the number will be reported as definitely prime or non-prime if possible, or otherwise "probably" prime with a certain probability of error.

`prime` The normal *k p* command performs one iteration of the primality test. Pressing *k p* repeatedly for the same integer will perform additional iterations. Also, *k p* with a numeric prefix performs the specified number of iterations. There is also an algebraic function '`prime(n)`' or '`prime(n,iters)`' which returns 1 if n is (probably) prime and 0 if not.

`k f` `prfac` The *k f* (`calc-prime-factors`) [`prfac`] command attempts to decompose an integer into its prime factors. For numbers up to 25 million, the answer is exact although it may take some time. The result is a vector of the prime factors in increasing order. For larger inputs, prime factors above 5000 may not be found, in which case the last number in the vector will be an unfactored integer greater than 25 million (with a warning message). For negative integers, the first element of the list will be -1. For inputs -1, 0, and 1, the result is a list of the same number.

The *k n* (`calc-next-prime`) [`nextprime`] command finds the next prime above a given number. Essentially, it searches by calling `calc-prime-test` on successive integers until it finds one that passes the test. This is quite fast for integers less than eight million, but once the probabilistic test comes into play the search may be rather slow. Ordinarily this command stops for any prime that passes one iteration of the primality test. With a numeric prefix argument, a number must pass the specified number of iterations before the search stops. (This only matters when searching above eight million.) You can always use additional *k p* commands to increase your certainty that the number is indeed prime. `k n` `nextpr...`

The *I k n* (`calc-prev-prime`) [`prevprime`] command analogously finds the next prime less than a given number. `I k n` `prevpr...`

The *k t* (`calc-totient`) [`totient`] command computes the Euler "totient" function $\phi(n)$, the number of integers less than n which are relatively prime to n. `k t` `totient`

The *k m* (`calc-moebius`) [`moebius`] command computes the Möbius μ function. If the input number is a product of k distinct factors, this is $(-1)^k$. If the input number has any duplicate factors (i.e., can be divided by the same prime more than once), the result is zero. `k m` `moebius`

8.7 Probability Distribution Functions

The functions in this section compute various probability distributions. For continuous distributions, this is the integral of the probability density function from x to infinity. (These are the "upper tail" distribution functions; there are also corresponding "lower tail" functions which integrate from minus infinity to x.) For discrete distributions, the upper tail function gives the sum from x to infinity; the lower tail function gives the sum from minus infinity up to, but not including, x.

To integrate from x to y, just use the distribution function twice and subtract. For example, the probability that a Gaussian random variable with mean 2 and standard deviation 1 will lie in the range from 2.5 to 2.8 is '`utpn(2.5,2,1) - utpn(2.8,2,1)`' ("the probability that it is greater than 2.5, but not greater than 2.8"), or equivalently '`ltpn(2.8,2,1) - ltpn(2.5,2,1)`'.

The *k B* (`calc-utpb`) [`utpb`] function uses the binomial distribution. Push the parameters n, p, and then x onto the stack; the result ('`utpb(x,n,p)`') is the probability that an event will occur x or more times out of n trials, if its probability of occurring in any given trial is p. The *I k B* [`ltpb`] function is the probability that the event will occur fewer than x times. `k B` `I k B` `utpb` `ltpb`

The other probability distribution functions similarly take the form *k X* (`calc-utpx`) [`utpx`] and *I k X* [`ltpx`], for various letters x. The arguments to the algebraic functions are the value of the random variable first, then whatever other parameters define the distribution. Note these are among the few Calc functions where the order of the arguments in algebraic form differs from the order of arguments as found on the stack. (The random variable comes last on the stack, so that you can type, e.g., *2 RET 1 RET 2.5 k N M-RET DEL 2.8 k N -*, using *M-RET DEL* to recover the original arguments but substitute a new value for x.)

The '`utpc(x,v)`' function uses the chi-square distribution with ν degrees of freedom. It is the probability that a model is correct if its chi-square statistic is x. `k C` `utpc`

The '`utpf(F,v1,v2)`' function uses the F distribution, used in various statistical tests. The parameters ν_1 and ν_2 are the degrees of freedom in the numerator and denominator, respectively, used in computing the statistic F. `k F` `utpf` ...

`k N`
`utpn`

The '`utpn(x,m,s)`' function uses a normal (Gaussian) distribution with mean m and standard deviation σ. It is the probability that such a normal-distributed random variable would exceed x.

`k P`
`utpp`

The '`utpp(n,x)`' function uses a Poisson distribution with mean x. It is the probability that n or more such Poisson random events will occur.

`k T`
`utpt`
...

The '`utpt(t,v)`' function uses the Student's "t" distribution with ν degrees of freedom. It is the probability that a t-distributed random variable will be greater than t. (Note: This computes the distribution function $A(t|\nu)$ where $A(0|\nu) = 1$ and $A(\infty|\nu) \to 0$. The `UTPT` operation on the HP-48 uses a different definition which returns half of Calc's value: '`UTPT(t,v) = .5*utpt(t,v)`'.)

While Calc does not provide inverses of the probability distribution functions, the `a R` command can be used to solve for the inverse. Since the distribution functions are monotonic, `a R` is guaranteed to be able to find a solution given any initial guess. See *XXX* [Numerical Solutions], page *XXX*.

9 Vector/Matrix Functions

Many of the commands described here begin with the *v* prefix. (For convenience, the shift-*V* prefix is equivalent to *v*.) The commands usually apply to both plain vectors and matrices; some apply only to matrices or only to square matrices. If the argument has the wrong dimensions the operation is left in symbolic form.

Vectors are entered and displayed using '`[a,b,c]`' notation. Matrices are vectors of which all elements are vectors of equal length. (Though none of the standard Calc commands use this concept, a three-dimensional matrix or rank-3 tensor could be defined as a vector of matrices, and so on.)

9.1 Packing and Unpacking

Calc's "pack" and "unpack" commands collect stack entries to build composite objects such as vectors and complex numbers. They are described in this chapter because they are most often used to build vectors.

The *v p* (`calc-pack`) [`pack`] command collects several elements from the stack into a matrix, complex number, HMS form, error form, etc. It uses a numeric prefix argument to specify the kind of object to be built; this argument is referred to as the "packing mode." If the packing mode is a nonnegative integer, a vector of that length is created. For example, *C-u 5 v p* will pop the top five stack elements and push back a single vector of those five elements. (*C-u 0 v p* simply creates an empty vector.) `v p`

The same effect can be had by pressing *[* to push an incomplete vector on the stack, using *TAB* (`calc-roll-down`) to sneak the incomplete object up past a certain number of elements, and then pressing *]* to complete the vector.

Negative packing modes create other kinds of composite objects:

-1	Two values are collected to build a complex number. For example, *5 RET 7 C-u -1 v p* creates the complex number $(5, 7)$. The result is always a rectangular complex number. The two input values must both be real numbers, i.e., integers, fractions, or floats. If they are not, Calc will instead build a formula like '`a + (0, 1) b`'. (The other packing modes also create a symbolic answer if the components are not suitable.)
-2	Two values are collected to build a polar complex number. The first is the magnitude; the second is the phase expressed in either degrees or radians according to the current angular mode.
-3	Three values are collected into an HMS form. The first two values (hours and minutes) must be integers or integer-valued floats. The third value may be any real number.
-4	Two values are collected into an error form. The inputs may be real numbers or formulas.
-5	Two values are collected into a modulo form. The inputs must be real numbers.

−6	Two values are collected into the interval '`[a .. b]`'. The inputs may be real numbers, HMS or date forms, or formulas.
−7	Two values are collected into the interval '`[a .. b)`'.
−8	Two values are collected into the interval '`(a .. b]`'.
−9	Two values are collected into the interval '`(a .. b)`'.
−10	Two integer values are collected into a fraction.
−11	Two values are collected into a floating-point number. The first is the mantissa; the second, which must be an integer, is the exponent. The result is the mantissa times ten to the power of the exponent.
−12	This is treated the same as −11 by the *v p* command. When unpacking, −12 specifies that a floating-point mantissa is desired.
−13	A real number is converted into a date form.
−14	Three numbers (year, month, day) are packed into a pure date form.
−15	Six numbers are packed into a date/time form.

With any of the two-input negative packing modes, either or both of the inputs may be vectors. If both are vectors of the same length, the result is another vector made by packing corresponding elements of the input vectors. If one input is a vector and the other is a plain number, the number is packed along with each vector element to produce a new vector. For example, *C-u -4 v p* could be used to convert a vector of numbers and a vector of errors into a single vector of error forms; *C-u -5 v p* could convert a vector of numbers and a single number *M* into a vector of numbers modulo *M*.

If you don't give a prefix argument to *v p*, it takes the packing mode from the top of the stack. The elements to be packed then begin at stack level 2. Thus *1 RET 2 RET 4 n v p* is another way to enter the error form '`1 +/- 2`'.

If the packing mode taken from the stack is a vector, the result is a matrix with the dimensions specified by the elements of the vector, which must each be integers. For example, if the packing mode is '`[2, 3]`', then six numbers will be taken from the stack and returned in the form '`[[a, b, c], [d, e, f]]`'.

If any elements of the vector are negative, other kinds of packing are done at that level as described above. For example, '`[2, 3, -4]`' takes 12 objects and creates a 2×3 matrix of error forms: '`[[a +/- b, c +/- d ... ]]`'. Also, '`[-4, -10]`' will convert four integers into an error form consisting of two fractions: '`a:b +/- c:d`'.

`pack` There is an equivalent algebraic function, '`pack(`*mode*`,` *items*`)`' where *mode* is a packing mode (an integer or a vector of integers) and *items* is a vector of objects to be packed (re-packed, really) according to that mode. For example, '`pack([3, -4], [a,b,c,d,e,f])`' yields '`[a +/- b, c +/- d, e +/- f]`'. The function is left in symbolic form if the packing mode is illegal, or if the number of data items does not match the number of items required by the mode.

The *v u* (`calc-unpack`) command takes the vector, complex number, HMS form, or other composite object on the top of the stack and "unpacks" it, pushing each of its elements onto the stack as separate objects. Thus, it is the "inverse" of *v p*. If the value at the top of the stack is a formula, *v u* unpacks it by pushing each of the arguments of the top-level operator onto the stack. `v u`

You can optionally give a numeric prefix argument to *v u* to specify an explicit (un)packing mode. If the packing mode is negative and the input is actually a vector or matrix, the result will be two or more similar vectors or matrices of the elements. For example, given the vector '`[a +/- b, c^2, d +/- 7]`', the result of *C-u -4 v u* will be the two vectors '`[a, c^2, d]`' and '`[b, 0, 7]`'.

Note that the prefix argument can have an effect even when the input is not a vector. For example, if the input is the number −5, then *c-u -1 v u* yields −5 and 0 (the components of −5 when viewed as a rectangular complex number); *C-u -2 v u* yields 5 and 180 (assuming degrees mode); and *C-u -10 v u* yields −5 and 1 (the numerator and denominator of −5, viewed as a rational number). Plain *v u* with this input would complain that the input is not a composite object.

Unpacking mode −11 converts a float into an integer mantissa and an integer exponent, where the mantissa is not divisible by 10 (except that 0.0 is represented by a mantissa and exponent of 0). Unpacking mode −12 converts a float into a floating-point mantissa and integer exponent, where the mantissa (for non-zero numbers) is guaranteed to lie in the range [1 .. 10). In both cases, the mantissa is shifted left or right (and the exponent adjusted to compensate) in order to satisfy these constraints.

Positive unpacking modes are treated differently than for *v p*. A mode of 1 is much like plain *v u* with no prefix argument, except that in addition to the components of the input object, a suitable packing mode to re-pack the object is also pushed. Thus, *C-u 1 v u* followed by *v p* will re-build the original object.

A mode of 2 unpacks two levels of the object; the resulting re-packing mode will be a vector of length 2. This might be used to unpack a matrix, say, or a vector of error forms. Higher unpacking modes unpack the input even more deeply.

There are two algebraic functions analogous to *v u*. The '`unpack(`*mode*`, `*item*`)`' function unpacks the *item* using the given *mode*, returning the result as a vector of components. Here the *mode* must be an integer, not a vector. For example, '`unpack(-4, a +/- b)`' returns '`[a, b]`', as does '`unpack(1, a +/- b)`'. `unpack`

The `unpackt` function is like `unpack` but instead of returning a simple vector of items, it returns a vector of two things: The mode, and the vector of items. For example, '`unpackt(1, 2:3 +/- 1:4)`' returns '`[-4, [2:3, 1:4]]`', and '`unpackt(2, 2:3 +/- 1:4)`' returns '`[[-4, -10], [2, 3, 1, 4]]`'. The identity for re-building the original object is '`apply(pack, unpackt(`*n*`, `*x*`)) = `*x*'. (The `apply` function builds a function call given the function name and a vector of arguments.) `unpackt`

Subscript notation is a useful way to extract a particular part of an object. For example, to get the numerator of a rational number, you can use '`unpack(-10, `*x*`)_1`'.

9.2 Building Vectors

Vectors and matrices can be added, subtracted, multiplied, and divided; see *XXX* [Basic Arithmetic], page *XXX*.

`|`

The `|` (`calc-concat`) command "concatenates" two vectors into one. For example, after *[1 , 2] [3 , 4]* `|`, the stack will contain the single vector '`[1, 2, 3, 4]`'. If the arguments are matrices, the rows of the first matrix are concatenated with the rows of the second. (In other words, two matrices are just two vectors of row-vectors as far as `|` is concerned.)

If either argument to `|` is a scalar (a non-vector), it is treated like a one-element vector for purposes of concatenation: *1 [2 , 3]* `|` produces the vector '`[1, 2, 3]`'. Likewise, if one argument is a matrix and the other is a plain vector, the vector is treated as a one-row matrix.

`H |`
`append`

The *H* `|` (`calc-append`) [`append`] command concatenates two vectors without any special cases. Both inputs must be vectors. Whether or not they are matrices is not taken into account. If either argument is a scalar, the `append` function is left in symbolic form. See also `cons` and `rcons` below.

`I |`
`H I |`

The *I* `|` and *H I* `|` commands are similar, but they use their two stack arguments in the opposite order. Thus *I* `|` is equivalent to *TAB* `|`, but possibly more convenient and also a bit faster.

`v d`
`diag`

The *v d* (`calc-diag`) [`diag`] function builds a diagonal square matrix. The optional numeric prefix gives the number of rows and columns in the matrix. If the value at the top of the stack is a vector, the elements of the vector are used as the diagonal elements; the prefix, if specified, must match the size of the vector. If the value on the stack is a scalar, it is used for each element on the diagonal, and the prefix argument is required.

To build a constant square matrix, e.g., a 3×3 matrix filled with ones, use *0 M-3 v d 1 +*, i.e., build a zero matrix first and then add a constant value to that matrix. (Another alternative would be to use *v b* and *v a*; see below.)

`v i`
`idn`

The *v i* (`calc-ident`) [`idn`] function builds an identity matrix of the specified size. It is a convenient form of *v d* where the diagonal element is always one. If no prefix argument is given, this command prompts for one.

In algebraic notation, '`idn(a,n)`' acts much like '`diag(a,n)`', except that a is required to be a scalar (non-vector) quantity. If n is omitted, '`idn(a)`' represents a times an identity matrix of unknown size. Calc can operate algebraically on such generic identity matrices, and if one is combined with a matrix whose size is known, it is converted automatically to an identity matrix of a suitable matching size. The *v i* command with an argument of zero creates a generic identity matrix, '`idn(1)`'. Note that in dimensioned matrix mode (see *XXX* [Matrix Mode], page *XXX*), generic identity matrices are immediately expanded to the current default dimensions.

`v x`
`index`

The *v x* (`calc-index`) [`index`] function builds a vector of consecutive integers from 1 to n, where n is the numeric prefix argument. If you do not provide a prefix argument, you will be prompted to enter a suitable number. If n is negative, the result is a vector of negative integers from n to -1.

With a prefix argument of just *C-u*, the *v x* command takes three values from the stack: n, *start*, and *incr* (with *incr* at top-of-stack). Counting starts at *start* and increases by *incr* for successive vector elements. If *start* or n is in floating-point format, the resulting vector elements will also be floats. Note that *start* and *incr* may in fact be any kind of numbers or formulas.

When *start* and *incr* are specified, a negative n has a different interpretation: It causes a geometric instead of arithmetic sequence to be generated. For example, '`index(-3, a, b)`' produces '`[a, a b, a b^2]`'. If you omit *incr* in the algebraic form, '`index(`*n*`,` *start*`)`', the default value for *incr* is one for positive n or two for negative n.

The *v b* (`calc-build-vector`) [`cvec`] function builds a vector of *n* copies of the value on the top of the stack, where *n* is the numeric prefix argument. In algebraic formulas, '`cvec(x,n,m)`' can also be used to build an *n*-by-*m* matrix of copies of *x*. (Interactively, just use *v b* twice: once to build a row, then again to build a matrix of copies of that row.) `v b` `cvec`

The *v h* (`calc-head`) [`head`] function returns the first element of a vector. The *I v h* (`calc-tail`) [`tail`] function returns the vector with its first element removed. In both cases, the argument must be a non-empty vector. `v h` `I v h` `head` `tail`

The *v k* (`calc-cons`) [`cons`] function takes a value *h* and a vector *t* from the stack, and produces the vector whose head is *h* and whose tail is *t*. This is similar to `|`, except if *h* is itself a vector, `|` will concatenate the two vectors whereas `cons` will insert *h* at the front of the vector *t*. `v k` `cons`

Each of these three functions also accepts the Hyperbolic flag [`rhead`, `rtail`, `rcons`] in which case *t* instead represents the *last* single element of the vector, with *h* representing the remainder of the vector. Thus the vector '`[a, b, c, d] = cons(a, [b, c, d]) = rcons([a, b, c], d)`'. Also, '`head([a, b, c, d]) = a`', '`tail([a, b, c, d]) = [b, c, d]`', '`rhead([a, b, c, d]) = [a, b, c]`', and '`rtail([a, b, c, d]) = d`'. `H v h` `rhead` …

9.3 Extracting Vector Elements

The *v r* (`calc-mrow`) [`mrow`] command extracts one row of the matrix on the top of the stack, or one element of the plain vector on the top of the stack. The row or element is specified by the numeric prefix argument; the default is to prompt for the row or element number. The matrix or vector is replaced by the specified row or element in the form of a vector or scalar, respectively. `v r` `mrow`

With a prefix argument of *C-u* only, *v r* takes the index of the element or row from the top of the stack, and the vector or matrix from the second-to-top position. If the index is itself a vector of integers, the result is a vector of the corresponding elements of the input vector, or a matrix of the corresponding rows of the input matrix. This command can be used to obtain any permutation of a vector.

With *C-u*, if the index is an interval form with integer components, it is interpreted as a range of indices and the corresponding subvector or submatrix is returned.

Subscript notation in algebraic formulas ('`a_b`') stands for the Calc function `subscr`, which is synonymous with `mrow`. Thus, '`[x, y, z]_k`' produces x, y, or z if k is one, two, or three, respectively. A double subscript ('`M_i_j`', equivalent to '`subscr(subscr(M, i), j)`') will access the element at row i, column j of a matrix. The `a _` (`calc-subscript`) command creates a subscript formula '`a_b`' out of two stack entries. (It is on the `a` "algebra" prefix because subscripted variables are often used purely as an algebraic notation.) `a _` `subscr` `-`

Given a negative prefix argument, *v r* instead deletes one row or element from the matrix or vector on the top of the stack. Thus *C-u 2 v r* replaces a matrix with its second row, but *C-u -2 v r* replaces the matrix with the same matrix with its second row removed. In algebraic form this function is called `mrrow`. `mrrow`

Given a prefix argument of zero, *v r* extracts the diagonal elements of a square matrix in the form of a vector. In algebraic form this function is called `getdiag`. `getdiag`

`v c`
`mcol`
`mrcol`

The *v c* (`calc-mcol`) [`mcol` or `mrcol`] command is the analogous operation on columns of a matrix. Given a plain vector it extracts (or removes) one element, just like *v r*. If the index in *C-u v c* is an interval or vector and the argument is a matrix, the result is a submatrix with only the specified columns retained (and possibly permuted in the case of a vector index).

To extract a matrix element at a given row and column, use *v r* to extract the row as a vector, then *v c* to extract the column element from that vector. In algebraic formulas, it is often more convenient to use subscript notation: '`m_i_j`' gives row i, column j of matrix m.

`v s`
`subvec`

The *v s* (`calc-subvector`) [`subvec`] command extracts a subvector of a vector. The arguments are the vector, the starting index, and the ending index, with the ending index in the top-of-stack position. The starting index indicates the first element of the vector to take. The ending index indicates the first element *past* the range to be taken. Thus, '`subvec([a, b, c, d, e], 2, 4)`' produces the subvector '`[b, c]`'. You could get the same result using '`mrow([a, b, c, d, e], [2 .. 4))`'.

If either the start or the end index is zero or negative, it is interpreted as relative to the end of the vector. Thus '`subvec([a, b, c, d, e], 2, -2)`' also produces '`[b, c]`'. In the algebraic form, the end index can be omitted in which case it is taken as zero, i.e., elements from the starting element to the end of the vector are used. The infinity symbol, `inf`, also has this effect when used as the ending index.

`I v s`
`rsubvec`

With the Inverse flag, *I v s* [`rsubvec`] removes a subvector from a vector. The arguments are interpreted the same as for the normal *v s* command. Thus, '`rsubvec([a, b, c, d, e], 2, 4)`' produces '`[a, d, e]`'. It is always true that `subvec` and `rsubvec` return complementary parts of the input vector.

See *XXX* [Selecting Subformulas], page *XXX*, for an alternative way to operate on vectors one element at a time.

9.4 Manipulating Vectors

`v l`
`vlen`

The *v l* (`calc-vlength`) [`vlen`] command computes the length of a vector. The length of a non-vector is considered to be zero. Note that matrices are just vectors of vectors for the purposes of this command.

`H v l`
`mdims`

With the Hyperbolic flag, *H v l* [`mdims`] computes a vector of the dimensions of a vector, matrix, or higher-order object. For example, '`mdims([[a,b,c],[d,e,f]])`' returns '`[2, 3]`' since its argument is a 2×3 matrix.

`v f`
`find`

The *v f* (`calc-vector-find`) [`find`] command searches along a vector for the first element equal to a given target. The target is on the top of the stack; the vector is in the second-to-top position. If a match is found, the result is the index of the matching element. Otherwise, the result is zero. The numeric prefix argument, if given, allows you to select any starting index for the search.

`v a`
`arrange`

The *v a* (`calc-arrange-vector`) [`arrange`] command rearranges a vector to have a certain number of columns and rows. The numeric prefix argument specifies the number of columns; if you do not provide an argument, you will be prompted for the number of columns. The vector or matrix on the top of the stack is *flattened* into a plain vector. If the number of columns is nonzero, this vector is then formed into a matrix by taking successive groups of n elements. If the number of columns does not evenly divide the number of elements in the vector, the last row will be short and the result will not be suitable for use as a matrix. For example, with the matrix '`[[1, 2], [3, 4]]`' on the stack, *`v a 4`* produces '`[[1, 2, 3, 4]]`' (a 1×4 matrix), *`v a 1`* produces '`[[1], [2], [3], [4]]`' (a 4×1 matrix), *`v a 2`* produces '`[[1, 2], [3, 4]]`' (the original 2×2 matrix), *`v a 3`* produces '`[[1, 2, 3], [4]]`' (not a matrix), and *`v a 0`* produces the flattened list '`[1, 2, 3, 4]`'.

The *`V S`* (`calc-sort`) [`sort`] command sorts the elements of a vector into increasing order. Real numbers, real infinities, and constant interval forms come first in this ordering; next come other kinds of numbers, then variables (in alphabetical order), then finally come formulas and other kinds of objects; these are sorted according to a kind of lexicographic ordering with the useful property that one vector is less or greater than another if the first corresponding unequal elements are less or greater, respectively. Since quoted strings are stored by Calc internally as vectors of ASCII character codes (see *XXX* [Strings], page *XXX*), this means vectors of strings are also sorted into alphabetical order by this command. `V S` `I V S` `sort` `rsort`

The *`I V S`* [`rsort`] command sorts a vector into decreasing order.

The *`V G`* (`calc-grade`) [`grade`, `rgrade`] command produces an index table or permutation vector which, if applied to the input vector (as the index of *`C-u v r`*, say), would sort the vector. A permutation vector is just a vector of integers from 1 to n, where each integer occurs exactly once. One application of this is to sort a matrix of data rows using one column as the sort key; extract that column, grade it with *`V G`*, then use the result to reorder the original matrix with *`C-u v r`*. Another interesting property of the `V G` command is that, if the input is itself a permutation vector, the result will be the inverse of the permutation. The inverse of an index table is a rank table, whose kth element says where the kth original vector element will rest when the vector is sorted. To get a rank table, just use *`V G V G`*. `V G` `I V G` `grade` `rgrade`

With the Inverse flag, *`I V G`* produces an index table that would sort the input into decreasing order. Note that *`V S`* and *`V G`* use a "stable" sorting algorithm, i.e., any two elements which are equal will not be moved out of their original order. Generally there is no way to tell with *`V S`*, since two elements which are equal look the same, but with *`V G`* this can be an important issue. In the matrix-of-rows example, suppose you have names and telephone numbers as two columns and you wish to sort by phone number primarily, and by name when the numbers are equal. You can sort the data matrix by names first, and then again by phone numbers. Because the sort is stable, any two rows with equal phone numbers will remain sorted by name even after the second sort.

The *`V H`* (`calc-histogram`) [`histogram`] command builds a histogram of a vector of numbers. Vector elements are assumed to be integers or real numbers in the range $[0..n)$ for some "number of bins" n, which is the numeric prefix argument given to the command. The result is a vector of n counts of how many times each value appeared in the original vector. Non-integers in the input are rounded down to integers. Any vector elements outside the specified range are ignored. (You can tell if elements have been ignored by noting that the counts in the result vector don't add up to the length of the input vector.) `V H` `histo...`

With the Hyperbolic flag, *`H V H`* pulls two vectors from the stack. The second-to-top vector is the list of numbers as before. The top vector is an equal-sized list of "weights" to attach to the elements of the data vector. For example, if the first data element is 4.2 and the first weight is 10, then 10 will be added to bin 4 of the result vector. Without the hyperbolic flag, every element has a weight of one. `H V H`

The *`v t`* (`calc-transpose`) [`trn`] command computes the transpose of the matrix at the top of the stack. If the argument is a plain vector, it is treated as a row vector and transposed into a one-column matrix. `v t` `trn`

`v v`
`rev`
The *v v* (`calc-reverse-vector`) [`vec`] command reverses a vector end-for-end. Given a matrix, it reverses the order of the rows. (To reverse the columns instead, just use *v t v v v t*. The same principle can be used to apply other vector commands to the columns of a matrix.)

`v m`
`vmask`
The *v m* (`calc-mask-vector`) [`vmask`] command uses one vector as a mask to extract elements of another vector. The mask is in the second-to-top position; the target vector is on the top of the stack. These vectors must have the same length. The result is the same as the target vector, but with all elements which correspond to zeros in the mask vector deleted. Thus, for example, '`vmask([1, 0, 1, 0, 1], [a, b, c, d, e])`' produces '`[a, c, e]`'. See *XXX* [Logical Operations], page *XXX*.

`v e`
`vexp`
The *v e* (`calc-expand-vector`) [`vexp`] command expands a vector according to another mask vector. The result is a vector the same length as the mask, but with nonzero elements replaced by successive elements from the target vector. The length of the target vector is normally the number of nonzero elements in the mask. If the target vector is longer, its last few elements are lost. If the target vector is shorter, the last few nonzero mask elements are left unreplaced in the result. Thus '`vexp([2, 0, 3, 0, 7], [a, b])`' produces '`[a, 0, b, 0, 7]`'.

`H v e`
With the Hyperbolic flag, *H v e* takes a filler value from the top of the stack; the mask and target vectors come from the third and second elements of the stack. This filler is used where the mask is zero: '`vexp([2, 0, 3, 0, 7], [a, b], z)`' produces '`[a, z, c, z, 7]`'. If the filler value is itself a vector, then successive values are taken from it, so that the effect is to interleave two vectors according to the mask: '`vexp([2, 0, 3, 7, 0, 0], [a, b], [x, y])`' produces '`[a, x, b, 7, y, 0]`'.

Another variation on the masking idea is to combine '`[a, b, c, d, e]`' with the mask '`[1, 0, 1, 0, 1]`' to produce '`[a, 0, c, 0, e]`'. You can accomplish this with *V M a &*, mapping the logical "and" operation across the two vectors. See *XXX* [Logical Operations], page *XXX*. Note that the `? :` operation also discussed there allows other types of masking using vectors.

9.5 Vector and Matrix Arithmetic

Basic arithmetic operations like addition and multiplication are defined for vectors and matrices as well as for numbers. Division of matrices, in the sense of multiplying by the inverse, is supported. (Division by a matrix actually uses LU-decomposition for greater accuracy and speed.) See *XXX* [Basic Arithmetic], page *XXX*.

The following functions are applied element-wise if their arguments are vectors or matrices: `change-sign`, `conj`, `arg`, `re`, `im`, `polar`, `rect`, `clean`, `float`, `frac`. See *XXX* [Function Index], page *XXX*.

`V J`
`ctrn`
The *V J* (`calc-conj-transpose`) [`ctrn`] command computes the conjugate transpose of its argument, i.e., '`conj(trn(x))`'.

`A`
`abs`
The *A* (`calc-abs`) [`abs`] command computes the Frobenius norm of a vector or matrix argument. This is the square root of the sum of the squares of the absolute values of the elements of the vector or matrix. If the vector is interpreted as a point in two- or three-dimensional space, this is the distance from that point to the origin.

`v n`
`rnorm`
The *v n* (`calc-rnorm`) [`rnorm`] command computes the row norm, or infinity-norm, of a vector or matrix. For a plain vector, this is the maximum of the absolute values of the elements. For a matrix, this is the maximum of the row-absolute-value-sums, i.e., of the sums of the absolute values of the elements along the various rows.

The *V N* (`calc-cnorm`) [`cnorm`] command computes the column norm, or one-norm, of a vector or matrix. For a plain vector, this is the sum of the absolute values of the elements. For a matrix, this is the maximum of the column-absolute-value-sums. General k-norms for k other than one or infinity are not provided. `V N` `cnorm`

The *V C* (`calc-cross`) [`cross`] command computes the right-handed cross product of two vectors, each of which must have exactly three elements. `V C` `cross`

The *&* (`calc-inv`) [`inv`] command computes the inverse of a square matrix. If the matrix is singular, the inverse operation is left in symbolic form. Matrix inverses are recorded so that once an inverse (or determinant) of a particular matrix has been computed, the inverse and determinant of the matrix can be recomputed quickly in the future. `&` `inv`

If the argument to *&* is a plain number x, this command simply computes $1/x$. This is okay, because the '`/`' operator also does a matrix inversion when dividing one by a matrix.

The *V D* (`calc-mdet`) [`det`] command computes the determinant of a square matrix. `V D` `det`

The *V L* (`calc-mlud`) [`lud`] command computes the LU decomposition of a matrix. The result is a list of three matrices which, when multiplied together left-to-right, form the original matrix. The first is a permutation matrix that arises from pivoting in the algorithm, the second is lower-triangular with ones on the diagonal, and the third is upper-triangular. `V L` `lud`

The *V T* (`calc-mtrace`) [`tr`] command computes the trace of a square matrix. This is defined as the sum of the diagonal elements of the matrix. `V T` `tr`

9.6 Set Operations using Vectors

Calc includes several commands which interpret vectors as *sets* of objects. A set is a collection of objects; any given object can appear only once in the set. Calc stores sets as vectors of objects in sorted order. Objects in a Calc set can be any of the usual things, such as numbers, variables, or formulas. Two set elements are considered equal if they are identical, except that numerically equal numbers like the integer 4 and the float 4.0 are considered equal even though they are not "identical." Variables are treated like plain symbols without attached values by the set operations; subtracting the set '`[b]`' from '`[a, b]`' always yields the set '`[a]`' even though if the variables '`a`' and '`b`' both equalled 17, you might expect the answer '`[]`'.

If a set contains interval forms, then it is assumed to be a set of real numbers. In this case, all set operations require the elements of the set to be only things that are allowed in intervals: Real numbers, plus and minus infinity, HMS forms, and date forms. If there are variables or other non-real objects present in a real set, all set operations on it will be left in unevaluated form.

If the input to a set operation is a plain number or interval form *a*, it is treated like the one-element vector '`[`*a*`]`'. The result is always a vector, except that if the set consists of a single interval, the interval itself is returned instead.

See *XXX* [Logical Operations], page *XXX*, for the `in` function which tests if a certain value is a member of a given set. To test if the set A is a subset of the set B, use '`vdiff(A, B) = []`'.

The *V +* (`calc-remove-duplicates`) [`rdup`] command converts an arbitrary vector into set notation. It works by sorting the vector as if by *V S*, then removing duplicates. (For example, *[a, 5, 4, a, 4.0]* is sorted to '`[4, 4.0, 5, a, a]`' and then reduced to '`[4, 5, a]`'). Overlapping intervals are merged as necessary. You rarely need to use *V +* explicitly, since all the other set-based commands apply *V +* to their inputs before using them. `V +` `rdup`

`V V`
`vunion`

The *V V* (`calc-set-union`) [`vunion`] command computes the union of two sets. An object is in the union of two sets if and only if it is in either (or both) of the input sets. (You could accomplish the same thing by concatenating the sets with `|`, then using *V +*.)

`V ^`
`vint`

The *V ^* (`calc-set-intersect`) [`vint`] command computes the intersection of two sets. An object is in the intersection if and only if it is in both of the input sets. Thus if the input sets are disjoint, i.e., if they share no common elements, the result will be the empty vector '`[]`'. Note that the characters *V* and `^` were chosen to be close to the conventional mathematical notation for set union ($A \cup B$) and intersection ($A \cap B$).

`V -`
`vdiff`

The *V -* (`calc-set-difference`) [`vdiff`] command computes the difference between two sets. An object is in the difference $A - B$ if and only if it is in A but not in B. Thus subtracting '`[y,z]`' from a set will remove the elements '`y`' and '`z`' if they are present. You can also think of this as a general *set complement* operator; if A is the set of all possible values, then $A - B$ is the "complement" of B. Obviously this is only practical if the set of all possible values in your problem is small enough to list in a Calc vector (or simple enough to express in a few intervals).

`V X`
`vxor`

The *V X* (`calc-set-xor`) [`vxor`] command computes the "exclusive-or," or "symmetric difference" of two sets. An object is in the symmetric difference of two sets if and only if it is in one, but *not* both, of the sets. Objects that occur in both sets "cancel out."

`V ~`
`vcompl`

The *V ~* (`calc-set-complement`) [`vcompl`] command computes the complement of a set with respect to the real numbers. Thus '`vcompl(x)`' is equivalent to '`vdiff([-inf .. inf], x)`'. For example, '`vcompl([2, (3 .. 4]])`' evaluates to '`[[-inf .. 2), (2 .. 3], (4 .. inf]]`'.

`V F`
`vfloor`

The *V F* (`calc-set-floor`) [`vfloor`] command reinterprets a set as a set of integers. Any non-integer values, and intervals that do not enclose any integers, are removed. Open intervals are converted to equivalent closed intervals. Successive integers are converted into intervals of integers. For example, the complement of the set '`[2, 6, 7, 8]`' is messy, but if you wanted the complement with respect to the set of integers you could type *V ~ V F* to get '`[[-inf .. 1], [3 .. 5], [9 .. inf]]`'.

`V E`
`venum`

The *V E* (`calc-set-enumerate`) [`venum`] command converts a set of integers into an explicit vector. Intervals in the set are expanded out to lists of all integers encompassed by the intervals. This only works for finite sets (i.e., sets which do not involve '`-inf`' or '`inf`').

`V :`
`vspan`

The *V :* (`calc-set-span`) [`vspan`] command converts any set of reals into an interval form that encompasses all its elements. The lower limit will be the smallest element in the set; the upper limit will be the largest element. For an empty set, '`vspan([])`' returns the empty interval '`[0 .. 0)`'.

`V #`
`vcard`

The *V #* (`calc-set-cardinality`) [`vcard`] command counts the number of integers in a set. The result is the length of the vector that would be produced by *V E*, although the computation is much more efficient than actually producing that vector.

Another representation for sets that may be more appropriate in some cases is binary numbers. If you are dealing with sets of integers in the range 0 to 49, you can use a 50-bit binary number where a particular bit is 1 if the corresponding element is in the set. See *XXX* [Binary Functions], page *XXX*, for a list of commands that operate on binary numbers. Note that many of the above set operations have direct equivalents in binary arithmetic: *b o* (`calc-or`), *b a* (`calc-and`), *b d* (`calc-diff`), *b x* (`calc-xor`), and *b n* (`calc-not`), respectively. You can use whatever representation for sets is most convenient to you.

`b p`
`b u`
`vpack`
`vunpack`

The *b u* (`calc-unpack-bits`) [`vunpack`] command converts an integer that represents a set in binary into a set in vector/interval notation. For example, '`vunpack(67)`' returns '`[[0 .. 1], 6]`'. If the input is negative, the set it represents is semi-infinite: '`vunpack(-4) = [2 .. inf)`'. Use *V E* afterwards to expand intervals to individual values if you wish. Note that this command uses the *b* (binary) prefix key.

The *b p* (`calc-pack-bits`) [`vpack`] command converts the other way, from a vector or interval representing a set of nonnegative integers into a binary integer describing the same set. The set may include positive infinity, but must not include any negative numbers. The input is interpreted as a set of integers in the sense of *V F* (`vfloor`). Beware that a simple input like '`[100]`' can result in a huge integer representation (2^{100}, a 31-digit integer, in this case).

9.7 Statistical Operations on Vectors

The commands in this section take vectors as arguments and compute various statistical measures on the data stored in the vectors. The references used in the definitions of these functions are Bevington's *Data Reduction and Error Analysis for the Physical Sciences*, and *Numerical Recipes* by Press, Flannery, Teukolsky and Vetterling.

The statistical commands use the `u` prefix key followed by a shifted letter or other character.

See *XXX* [Manipulating Vectors], page *XXX*, for a description of *V H* (`calc-histogram`).

See *XXX* [Curve Fitting], page *XXX*, for the `a` *F* command for doing least-squares fits to statistical data.

See *XXX* [Probability Distribution Functions], page *XXX*, for several common probability distribution functions.

9.7.1 Single-Variable Statistics

These functions do various statistical computations on single vectors. Given a numeric prefix argument, they actually pop *n* objects from the stack and combine them into a data vector. Each object may be either a number or a vector; if a vector, any sub-vectors inside it are "flattened" as if by *v* `a` *0*; see *XXX* [Manipulating Vectors], page *XXX*. By default one object is popped, which (in order to be useful) is usually a vector.

If an argument is a variable name, and the value stored in that variable is a vector, then the stored vector is used. This method has the advantage that if your data vector is large, you can avoid the slow process of manipulating it directly on the stack.

These functions are left in symbolic form if any of their arguments are not numbers or vectors, e.g., if an argument is a formula, or a non-vector variable. However, formulas embedded within vector arguments are accepted; the result is a symbolic representation of the computation, based on the assumption that the formula does not itself represent a vector. All varieties of numbers such as error forms and interval forms are acceptable.

Some of the functions in this section also accept a single error form or interval as an argument. They then describe a property of the normal or uniform (respectively) statistical distribution described by the argument. The arguments are interpreted in the same way as the *M* argument of the random number function *k r*. In particular, an interval with integer limits is considered an integer distribution, so that '`[2 .. 6)`' is the same as '`[2 .. 5]`'. An interval with at least one floating-point limit is a continuous distribution: '`[2.0 .. 6.0)`' is *not* the same as '`[2.0 .. 5.0]`'!

`u #`
`vcount`

The *u #* (`calc-vector-count`) [`vcount`] command computes the number of data values represented by the inputs. For example, '`vcount(1, [2, 3], [[4, 5], [], x, y])`' returns 7. If the argument is a single vector with no sub-vectors, this simply computes the length of the vector.

`u +`
`u *`
`vsum`
`vprod`

The *u +* (`calc-vector-sum`) [`vsum`] command computes the sum of the data values. The *u ** (`calc-vector-prod`) [`vprod`] command computes the product of the data values. If the input is a single flat vector, these are the same as *V R +* and *V R ** (see *XXX* [Reducing and Mapping], page *XXX*).

`u X`
`u N`
`vmax`
`vmin`

The *u X* (`calc-vector-max`) [`vmax`] command computes the maximum of the data values, and the *u N* (`calc-vector-min`) [`vmin`] command computes the minimum. If the argument is an interval, this finds the minimum or maximum value in the interval. (Note that '`vmax([2..6)) = 5`' as described above.) If the argument is an error form, this returns plus or minus infinity.

`u M`
`vmean`

The *u M* (`calc-vector-mean`) [`vmean`] command computes the average (arithmetic mean) of the data values. If the inputs are error forms x `+/-` σ, this is the weighted mean of the x values with weights $1/\sigma^2$.

$$\mu = \frac{\sum \frac{x_i}{\sigma_i^2}}{\sum \frac{1}{\sigma_i^2}}$$

If the inputs are not error forms, this is simply the sum of the values divided by the count of the values.

Note that a plain number can be considered an error form with error $\sigma = 0$. If the input to *u M* is a mixture of plain numbers and error forms, the result is the mean of the plain numbers, ignoring all values with non-zero errors. (By the above definitions it's clear that a plain number effectively has an infinite weight, next to which an error form with a finite weight is completely negligible.)

This function also works for distributions (error forms or intervals). The mean of an error form 'a `+/-` b' is simply a. The mean of an interval is the mean of the minimum and maximum values of the interval.

`I u M`
`vmeane`

The *I u M* (`calc-vector-mean-error`) [`vmeane`] command computes the mean of the data points expressed as an error form. This includes the estimated error associated with the mean. If the inputs are error forms, the error is the square root of the reciprocal of the sum of the reciprocals of the squares of the input errors. (I.e., the variance is the reciprocal of the sum of the reciprocals of the variances.)

$$\sigma_\mu^2 = \frac{1}{\sum \frac{1}{\sigma_i^2}}$$

If the inputs are plain numbers, the error is equal to the standard deviation of the values divided by the square root of the number of values. (This works out to be equivalent to calculating the standard deviation and then assuming each value's error is equal to this standard deviation.)

$$\sigma_\mu^2 = \frac{\sigma^2}{N}$$

`H u M`
`vmedian`

The *H u M* (`calc-vector-median`) [`vmedian`] command computes the median of the data values. The values are first sorted into numerical order; the median is the middle value after sorting. (If the number of data values is even, the median is taken to be the average of the two middle values.)

The median function is different from the other functions in this section in that the arguments must all be real numbers; variables are not accepted even when nested inside vectors. (Otherwise it is not possible to sort the data values.) If any of the input values are error forms, their error parts are ignored.

The median function also accepts distributions. For both normal (error form) and uniform (interval) distributions, the median is the same as the mean.

The *H I u M* (`calc-vector-harmonic-mean`) [`vhmean`] command computes the harmonic mean of the data values. This is defined as the reciprocal of the arithmetic mean of the reciprocals of the values. `H I u M` `vhmean`

$$\frac{N}{\sum \frac{1}{x_i}}$$

The **u** *G* (`calc-vector-geometric-mean`) [`vgmean`] command computes the geometric mean of the data values. This is the Nth root of the product of the values. This is also equal to the `exp` of the arithmetic mean of the logarithms of the data values. `u G` `vgmean`

$$\exp\left(\sum \ln x_i\right) = \left(\prod x_i\right)^{1/N}$$

The *H* **u** *G* [`agmean`] command computes the "arithmetic-geometric mean" of two numbers taken from the stack. This is computed by replacing the two numbers with their arithmetic mean and geometric mean, then repeating until the two values converge. `H u G` `agmean`

$$a_{i+1} = \frac{a_i + b_i}{2}, \qquad b_{i+1} = \sqrt{a_i b_i}$$

Another commonly used mean, the RMS (root-mean-square), can be computed for a vector of numbers simply by using the *A* command.

The **u** *S* (`calc-vector-sdev`) [`vsdev`] command computes the standard deviation σ of the data values. If the values are error forms, the errors are used as weights just as for **u** *M*. This is the *sample* standard deviation, whose value is the square root of the sum of the squares of the differences between the values and the mean of the N values, divided by $N-1$. `u S` `vsdev`

$$\sigma^2 = \frac{1}{N-1} \sum (x_i - \mu)^2$$

This function also applies to distributions. The standard deviation of a single error form is simply the error part. The standard deviation of a continuous interval happens to equal the difference between the limits, divided by $\sqrt{12}$. The standard deviation of an integer interval is the same as the standard deviation of a vector of those integers.

The *I* **u** *S* (`calc-vector-pop-sdev`) [`vpsdev`] command computes the *population* standard deviation. It is defined by the same formula as above but dividing by N instead of by $N-1$. The population standard deviation is used when the input represents the entire set of data values in the distribution; the sample standard deviation is used when the input represents a sample of the set of all data values, so that the mean computed from the input is itself only an estimate of the true mean. `I u S` `vpsdev`

$$\sigma^2 = \frac{1}{N} \sum (x_i - \mu)^2$$

For error forms and continuous intervals, `vpsdev` works exactly like `vsdev`. For integer intervals, it computes the population standard deviation of the equivalent vector of integers.