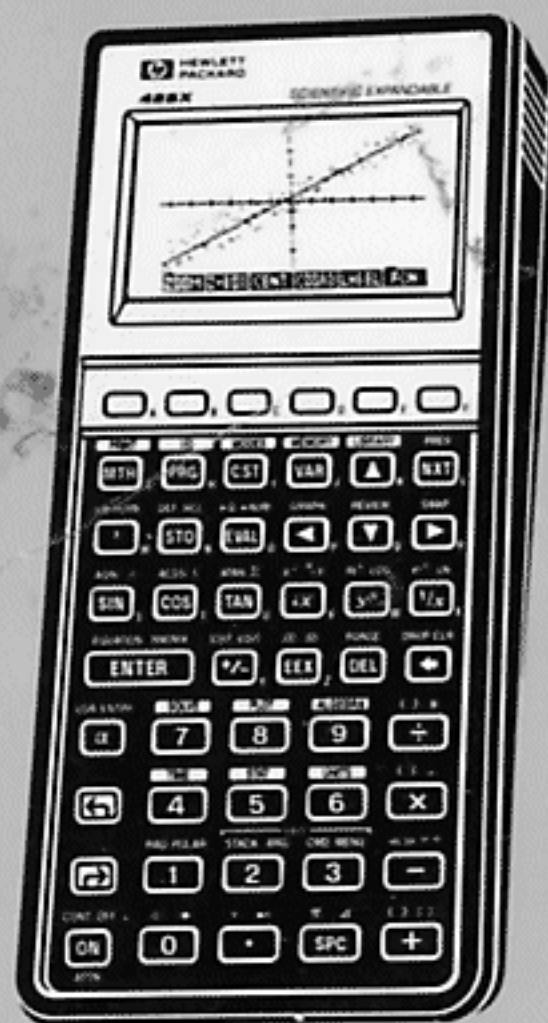
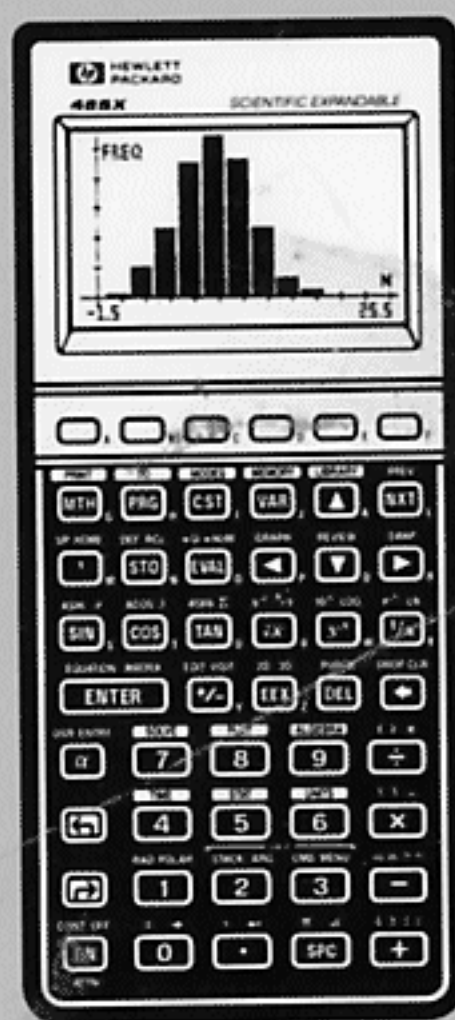
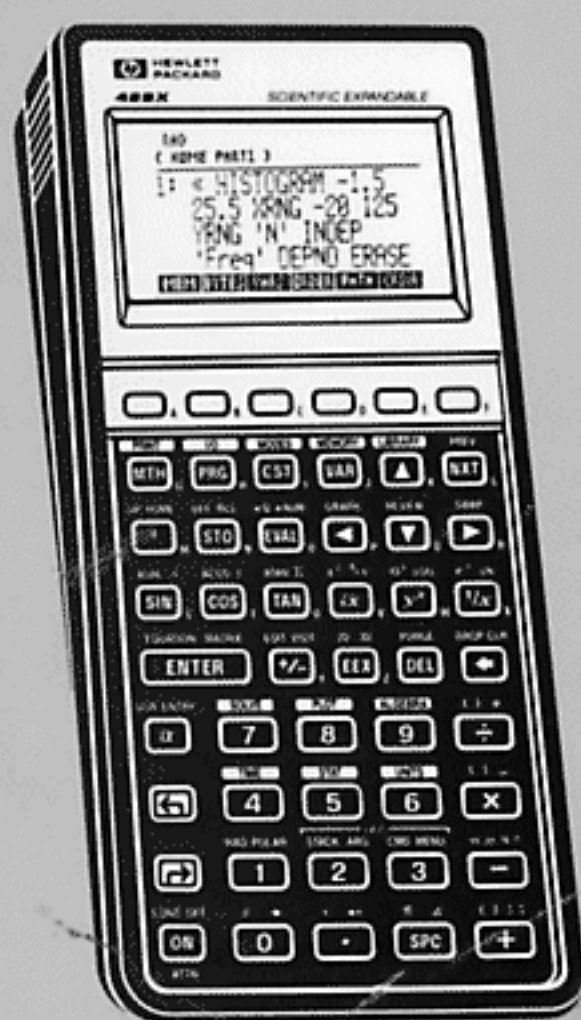


~~SECRET~~ COPY

HP 48

INSIGHTS

PART II: Problem-Solving Resources



William C. Wickes

HP 48 Insights II Program Disk

As a convenience for readers of *HP 48 Insights II* who use or have access to an IBM-compatible or MacIntosh personal computer, Larken Publications is offering a disk containing all of the HP 48 programs and examples described in the book. By downloading the programs individually or collectively from your computer to your HP 48, you avoid the effort and errors of entering the programs manually from the calculator keyboard.

To order one or more of these disks, remove this page from your book, fill out the ordering information below, and send it with your payment to:

Larken Publications
Department PC
4517 NW Queens Ave.
Corvallis OR 97330 USA

Make checks payable to *Larken Publications* (no charge or C.O.D. orders). Foreign orders must be paid in U.S. Funds through a U.S. bank or via international postal money order.

Name _____

Address _____

City _____

State _____

Zip _____

Country _____

Quantity		Unit Price	Price
	<i>HP 48 Insights II Program Disk</i>	\$10.00	
	(Optional) Airmail postage outside of USA, Canada, Mexico	1.00	

TOTAL

\$ _____

Disk Type (*check one*): ☐ IBM 5.25" ☐ IBM 3.5" ☐ MacIntosh 3.5"

HP 48 Insights

II. Problem-Solving Resources

William C. Wickes

William C Wickes
8/30/98

Larken Publications
4517 NW Queens Avenue
Corvallis, Oregon 97330

Copyright © William C. Wickes 1992

All rights reserved. No part of this book may be reproduced, transmitted, or stored in a retrieval system in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the author.

First Edition

First Printing, January 1992

ISBN 0-9625258-4-7

Acknowledgements

I thank my wife, Susan, and my children, Kenneth and Lara, for their help in the preparation of this manuscript.

*Dedicated to those mathematics teachers with the vision and
courage to adopt calculators like the HP 48 for their classrooms,
to improve the understanding and enthusiasm of their students.*

CONTENTS

13. Introduction to Part II.	401
13.1 About This Book	402
13.2 Notation	403
13.3 Terminology	408
14. HP Solve	409
14.1 The Equation Entry Menu	412
14.2 Basic HP Solve Operation	414
14.2.1 Independent, Dependent, and Unknown Variables	415
14.2.2 An Example	417
14.2.3 Equation Management	420
14.2.3.1 Using Subdirectories	420
14.2.3.2 The Equation Catalog	421
14.3 Solving With Units	423
14.3.1 Faster Solving with Units	425
14.3.2 Monitoring Convergence	426
14.4 Interpreting Results	427
14.4.1 Qualifying Messages	428
14.5 First Guesses	429
14.5.1 How Many Guesses?	430
14.5.2 Examples Using $x(x-2)(x+2) = 0$	431
14.6 Obtaining Guesses	433
14.7 Finding Critical Points	436
14.8 Using ISOL with HP Solve	439
14.9 Programmable Solving	440
14.10 Secondary Results	441
14.11 Modifying the Solver Menu	443
15. Plotting	447
15.1 The Plot Menus	447
15.2 Plotting Essentials	450
15.2.1 DRAW	452
15.2.1.1 Autoscaling	453
15.2.2 EQ and Σ DAT	453
15.2.3 PPAR	454
15.2.3.1 Saving A Graph	455
15.2.4 The Plot Scale	456
15.2.4.1 Redimensioning the Graph Screen	458
15.2.5 The Independent Variable	459
15.2.6 The Dependent Variable	460

15.2.7	Resolution	460
15.2.8	Axes and Labels	461
15.2.8.1	Tick Marks	463
15.2.9	Plot Type	464
15.3	The Plot Environment	465
15.3.1	The Plot Cursor	466
15.3.1.1	The Mark	468
15.3.2	Recentering and Zooming	469
15.3.2.1	Programmable Zooming	472
15.3.3	Drawing on the Graph Screen	473
15.3.4	Working with Graphic Objects	476
15.4	Function Plots	477
15.4.1	Plotting Programs	480
15.4.2	The Function Menu	483
15.5	Conic Sections	488
15.6	Polar Plots	491
15.6.1	Examples	493
15.6.2	Varying the Angle Increment	495
15.7	Parametric Plots	497
15.8	Truth Plots	501
15.8.1	Julia Sets	504
15.9	Scatter Plots	507
15.9.1	Plotting Curve Fits	509
15.10	Bar Charts	509
15.11	Histograms	511
16.	Symbolic Objects and Solutions	515
16.1	Motivations	515
16.2	General Symbolic Problem Solving	520
16.3	Symbolic vs. Numerical Solutions	524
16.3.1	$\rightarrow Q$ and $\rightarrow Q\pi$	525
16.4	Automated Symbolic Solutions: QUAD and ISOL	526
16.4.1	ISOL	527
16.4.2	SHOW	527
16.4.3	QUAD	528
16.5	Multiple Roots	529
16.5.1	Using the Solver Menu to Select Roots	532
16.6	Algebraic Objects as Programs	534
16.7	Refining User-Defined Functions	535
16.7.1	Preventing Evaluation: QUOTE	535
16.7.2	Applying Functions without Evaluation: APPLY	536
16.7.3	Preserving Local Variables' Values:	538

17.	Expression Manipulations	541
17.1	Extensive Manipulations	541
17.1.1	COLCT 545	
17.1.2	EXPAN 547	
17.2	The EquationWriter Subexpression Mode	550
17.2.1	Navigating the Expression 552	
17.2.2	Editing Subexpressions 554	
17.2.3	RULES 556	
17.2.3.1	Repeated Operations 561	
17.2.3.2	Condensed RULES Notation 562	
17.2.3.3	Moving Terms 562	
17.2.3.4	Commutation 564	
17.2.3.5	Association 565	
17.2.3.6	Distribution 566	
17.2.3.7	Merging 567	
17.2.3.8	Prefix Operations 567	
17.2.3.9	Unit Identities 568	
17.2.3.10	Adding Fractions 569	
17.2.3.11	Logarithms 570	
17.2.3.12	Exponentials 570	
17.2.3.13	Definition Expansions 570	
17.2.3.14	Addition Angle Formulae 570	
17.2.3.15	Collecting Terms 572	
17.3	Pattern Matching and Substitution	572
17.4	Simplifying Polynomials	575
17.4.1	Polynomial Programs 580	
18.	Calculus	587
18.1	Differentiation	587
18.1.1	Calculus with Trigonometric Functions 589	
18.1.1.1	User-Defined Derivatives 590	
18.1.2	Formal Derivatives 592	
18.2	Taylor's Polynomials	594
18.3	Summations	595
18.3.1	Summation Patterns 596	
18.4	Integration	598
18.4.1	Symbolic Integration 598	
18.4.1.1	Integration Patterns 601	
18.4.1.2	Derivative and Integral 602	
18.4.1.3	Adding Integration Patterns 604	
18.4.2	Numerical Integration 608	
18.4.3	Integration Strategies 611	

18.4.4	Programs as Integrands	615
18.4.5	Multiple Integrals	616
18.4.6	Polynomial Approximations	617
19.	The Time System	621
19.1	The Clock	621
19.1.1	Setting and Reading the Time	622
19.1.2	Time Arithmetic	625
19.1.3	Date Arithmetic	626
19.2	The Alarm System	627
19.2.1	Setting Alarms Manually	628
19.2.1.1	Cancelling Alarm Repeats	630
19.2.2	Appointment Alarms	630
19.2.2.1	Unacknowledged Repeating Alarms	635
19.2.3	Control Alarms	635
19.2.4	Alarm Commands	636
19.2.5	The Alarm Catalog	638
19.2.6	Automatic Alarm Deletion	640
20.	Statistics	641
20.1	Data Entry	641
20.2	One-Variable Sample Statistics	644
20.3	Two-Variable Statistics	646
20.3.1	Correlations	648
20.3.2	Regressions	649
20.3.2.1	Pseudo-Linear Regressions	650
20.3.3	Best Fit	653
20.3.4	Scatter Plots with Error Bars	653
20.3.5	Summary Statistics	655
20.4	General Least-Squares Fitting	656
20.4.1	Utilities	660
20.4.2	Polynomial Fits	662
20.4.3	Non-Linear Least Squares Fits	664
20.5	Probability Commands	669
20.5.1	Combinations and Permutations	669
20.5.1.1	Factorial and Gamma Function	671
20.5.2	Random Numbers	671
20.6	Upper-Tail Probability Distributions	673
20.6.1	UTPN	674
20.6.2	UTPT	674
20.6.3	UTPC	675
20.6.4	UTPF	675
20.6.5	Probability Density Functions	677

21. Unit Management	679
21.1 Types of Units	680
21.1.1 Prefixes 681	
21.1.2 Built-in Units 682	
21.1.3 User-Defined Units 683	
21.1.4 Unit Object Mechanics 683	
21.2 Unit Conversions	684
21.2.1 The Unit Menus 685	
21.2.2 Using ? 687	
21.2.3 Units in Custom Menus 687	
21.3 Unit Object Mathematics	688
21.3.1 Unit Operations Requiring Dimensional Consistency 688	
21.3.2 Unit Functions with Simplification 689	
21.3.3 Operations on the Unit Magnitude 690	
21.3.4 Trigonometric Unit Functions 691	
21.3.5 Examples of Calculations with Units. 691	
21.3.6 HP Solve 692	
21.3.7 Plotting 692	
21.3.8 Differentiation 693	
21.3.9 Integration 694	
21.4 Unit Management Idiosyncrasies	695
21.4.1 Non-integer Unit Powers 695	
21.4.2 Temperature 696	
21.4.3 Angle Units 698	
21.4.4 Photometric Units 698	
Program Index	701
Subject Index	705
Delimiters and Punctuation	717
Functions	717
RULES Operations	717

13. Introduction to Part II.

The HP48 scientific calculator is renowned for its remarkable concentration of mathematical computation features in a small, highly portable package. Even a quick glance at the keyboard suggests its power and range--everything from the sines and cosines that are traditional for scientific calculators to more exotic features associated with customization and communications. The HP48 represents the culmination of eighteen years of calculator development at Hewlett-Packard, incorporating virtually every useful feature of its predecessors, plus many that have never appeared on any other handheld calculator. The symbolic algebra system incorporated in the HP48 and its immediate forerunner the HP28 is actually the most widely used computer algebra system in the world.

By *HP48* we are referring to the HP48SX, which was introduced in 1990, and the HP48S, which followed in 1991. The lower-cost HP48S shares an identical feature set with the HP48SX except that it lacks plug-in memory ports. Throughout this book, it will generally not be necessary to distinguish between the two models, since all of the operations discussed here, and all of the example programs, execute identically on both types.

The HP48 contains several hundred programmable commands that form the basis of its programmability and customizability. You can supplement the built-in command set with *libraries*, which are software packages designed to integrate with the permanent ROM-based system, to provide additional programmable and non-programmable features. Furthermore, you can use the built-in and library commands to create an endless variety of programs that make the HP48 a very personalized tool. The straightforward and consistent design principles of the calculator ensure that this considerable array of programmed and preprogrammed operations is manageable in a straightforward and simple-to-use manner. These principles and methods are the subject matter of *HP48 Insights I: Principles and Programming*.

It is possible to accomplish most tasks for which the HP48 is suited by using commands or combinations of commands--programs. However, the HP48 goes well beyond providing a simple list of commands, to provide a higher level of convenience and power for certain common computation tasks. In particular, for five mathematical topics plus personal time management, the HP48 offers special menus that provide prompting and special operations to facilitate manual interaction, in addition to programmable commands. These six subjects are labeled by the single orange titles that appear above the [7], [8], [9], [4], [5], and [6] keys: SOLVE, PLOT, ALGEBRA, TIME, STAT, and UNITS. The interactive features associated with the ALGEBRA are actually accessed via

the EquationWriter ( **EQUATION**), but with that exception the menus are self-contained and focused on their respective subject areas.

Topics associated with these six menus form the principal subject matter of this book. The HP 48 *Owner's Manuals* refers to the menus as “applications.” We prefer to use that term for the uses of the calculator and its features rather than for the features themselves. The subtitle *Problem Solving Resources* is more descriptive of the nature and intent of the six menus--the six are at least *prime* resources for tasks related to the six areas, as part of the full problem-solving arsenal of the HP 48.

13.1 About This Book

HP 48 Insights II: Problem Solving Resources is the second volume of the two-part *Insights* series for the HP 48 calculator. For the sake of the master index provided at the end of this book, the page and section numbering is made continuous with that of *HP 48 Insights I*. Thus the first chapter of this book is Chapter 13 (apologies to triska-dekaphobes), and the first page is page 401. (The last numbered page in part I is 381, but starting part II at 401 makes it easier to remember for index page references, i.e. pages numbered higher than 400 are in part II).

To a considerable degree, this book stands alone from *HP 48 Insights I*. Assuming that you have reviewed at least as much of the *HP 48 Owner's Manuals* as is necessary to understand HP 48 rudiments such as keyboard and menu operations, object entry, etc., you should be able to read the material and exercise the examples here without reference to part I. There are, of course, numerous cross references between the two parts, including a few uses in part II of programs that are listed in part I, but for the most part these are not crucial to the material in part II. For the sake of readers who have not read part I, at the end of this chapter we reproduce the descriptions of notation conventions and important terminology from Chapter I that are used throughout this book.

The principal subjects of *HP 48 Insights II* are those associated with the six special menu keys mentioned in the previous section: numerical equation solving, plotting, symbolic algebra and calculus, the HP 48 time system, statistics and curve fitting, and physical unit management. There is no necessary order for presentation of these topics, so we will just follow the order in which the menus appear on the keyboard. You can actually study the chapters in any order; cross-references between the chapters will help you find other material related to any section you happen to be reading.

The following summarizes the chapter topics:

	Chapter	Topics
13.	Introduction	Introductory material, notation conventions.
14.	HP Solve	Interactive and programmable numerical root-finding.
15.	Plotting	Plotting basics, menus, and parameters; the plot environment; drawing; graphics objects; the eight plot types.
16.	Symbolic Objects and Solutions	Automated and interactive symbolic problem solving; $\rightarrow Q$; user-defined functions.
17.	Symbolic Manipulations	Extensive manipulations; the Equation-Writer subexpression mode; pattern-matching and substitution; polynomials.
18.	Calculus	Derivatives; Taylor's polynomials; sums; symbolic and numerical integrals.
19.	The Time System	The HP 48 clock; date and time arithmetic; alarms.
20.	Statistics	One- and two-variable statistics; regressions; probability commands; upper-tail probability distributions.
21.	Unit Management	Unit objects and conversions; the unit menus; unit functions; solving, plotting, and calculus with units; idiosyncrasies.

13.2 Notation

In order to help you recognize various calculator commands, keystroke sequences, and results, we use throughout this book certain notation conventions:

- All calculator commands and displayed results that appear in the text are printed in helvetica characters, e.g. DUP 1 2 SWAP. When you see characters like these, you are to understand that they represent specific HP 48 operations rather than any ordinary English-language meanings.

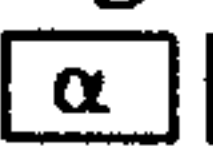
- Italics used within calculator operations sequences indicate varying inputs or results. For example, 123 'REG' STO means that 123 is stored in the specific variable REG, whereas 123 '*name*' STO indicates that the 123 is stored in a variable for which you may choose any name you want. Similarly, << *program* >> indicates an unspecified program object; { *numbers* } might represent a list object containing numbers as its elements.

Italics are also used for emphasis in ordinary text.

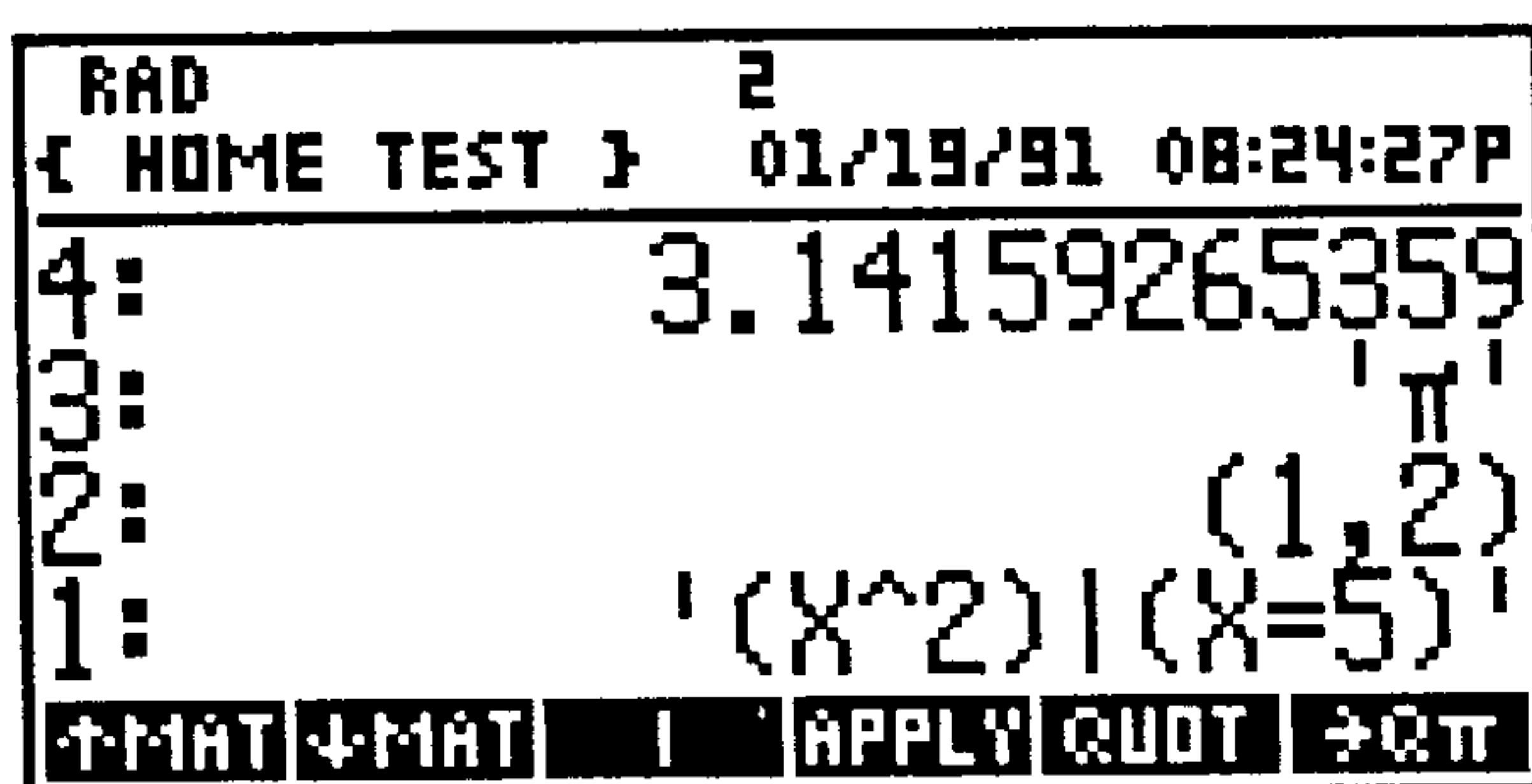
- HP 48 keys are displayed in helvetica characters surrounded by rectangular boxes, e.g. **ENTER**, **EVAL**, or **EEX**. The back-arrow key looks like this: , and the cursor keys like these: , , , and .
- A shifted key is shown with the key name in a box preceded by a left- or right-shift key picture,  or , e.g. **TIME**, or **PURGE**. A shifted key is identified by the orange or blue label above the key, rather than the label on the key itself--**SOLVE** rather than .
- Menu keys for operations available in the various menus are printed with the key labels surrounded by boxes drawn to suggest the reverse characters you see in the display, like these: **SIGN** or **-LIST**.

Examples of HP 48 operations take several forms. When appropriate, we will give step-by-step instructions that include specific keystrokes and show the relevant levels of the stack, with comments, as in the following sample:

Keystrokes:	Results:	Comments:
123 ENTER 456 +	1: 579	Adding 123 and 456 returns 579 to level 1.

For better legibility, we don't show individual letters and digits in key boxes--we just show 123 rather than , and ABC rather than . Key boxes are used for multi-letter keys on the keyboard and in menus.

In some cases, a printed listing of the stack contents isn't adequate, so we use an actual HP 48-generated picture of the calculator display, such as this picture from Chapter 6:



A large number of the examples, however, are given in a more compact format than the keystroke example shown above. These examples consist of a *sequence* of HP 48 commands and data that you are to execute, together with the stack objects that result from the execution. The “right hand” symbol $\rightarrow$ is used as a shorthand for “the HP 48 returns...” In the compact format, the addition example is written as

$$123 \quad 456 \quad + \quad \rightarrow \quad 579$$

The $\rightarrow$ means “enter the objects and commands on the left, in left-to-right order, and the HP 48 will give back--*return*--the objects on the right.” If there are multiple results, they are listed to the right of the $\rightarrow$ in the order in which they are returned. For example,

$$A \quad B \quad C \quad \text{ROT} \quad \text{SWAP} \quad \rightarrow \quad B \quad A \quad C$$

indicates that B is returned to level 3, A to level 2, and C to level 1.

Because of the flexibility of the HP 48, there are usually several ways you can accomplish any given sequence, so we often don’t specify precise keystrokes unless there are non-programmable operations in the sequence. If there are no key boxes in the left-side sequence, you can always obtain the right-side results by typing the left side as text into the command line, then pressing **ENTER** when you get to the $\rightarrow$ symbol.

The $\rightarrow$ symbol is also used in the *stack diagrams* that are part of most program listings. The stack diagrams show how to set up stack objects for execution of the program, where the objects to the left of the $\rightarrow$ are the “input” objects, and the objects following the $\rightarrow$ are the program outputs.

The most elaborate “examples” in this book are *programs*. Each program is listed in a box that includes a suggested program variable name, a stack diagram, the actual steps that make up the program, and comments to help you understand the steps. The

following sample listing illustrates the various features of the format:

SAMPLE	Sample Program Listing				checksum
	level 3	level 2	level 1		level 1
	"string"	[matrix]	n	↔	[matrix']
<pre><< A B → a b << IF C D THEN 1 2 → n m << START E F DO G UNTIL H END NEXT >> ELSE I J END >> >></pre>					Start of program. Start of local variable procedure Start of IF structure. Start of local variable procedure. Start of definite loop. DO loop. End of definite loop. End of local variable procedure. End of IF structure. End of local variable procedure. End of program.

1. The name of the program (SAMPLE) is listed first, followed by an expanded version of the name that is descriptive of its purpose. When you have entered the listed program, you should store it in a variable with the specified name. If no name is given, the program is just intended to illustrate some point in the text, and there's no need to give it any particular name.
2. The program's checksum is listed at the end of the name line, as a four-digit hexadecimal number. If you enter the program into your HP 48, you can verify that you have entered it correctly by comparing the listed checksum with the value returned by BYTES (section 12.5.1) for your program.
3. Below the program name is a *stack diagram*, that specifies the program's input and output on the stack. The program *arguments* are shown to the left of the ↔, and the *results* to the right. In the example, the stack diagram indicates that the program requires a string in level 3, a matrix in level 2, and a real number *n* in level 1, and returns a new matrix in level 1. The object symbols in the stack diagram are as descriptive as possible, showing not only the required object type but also the conceptual purpose of the objects. A stack diagram

length width height ↔ volume

shows that a program takes three real numbers (no object delimiters) representing length, width, and height, and returns another real number that is the volume.

4. The program listing is broken into lines, where each line has one or more program objects listed at the left, and explanatory comments on the right. There may be just one object on a line, or several whenever the collective effect of the objects is easy to follow. You do not have to use the same line breaks (or any at all) when you enter the program.
5. Lists, embedded programs, and program structures start on a new line unless they are short enough to fit entirely on one line. More frequently, each program or list delimiter or structure word starts a new line. The sequences between the structure words are indented, so that the structure words stand out. In the case of nested structures, each structure word of a particular structure is lined up vertically at the same indentation from the left margin. (The structure word $\rightarrow$ does not start a new line, but the local variable defining procedure that follows the $\rightarrow$ does start a new line.) Note that when you edit a program on the HP 48, the program display follows these same conventions, within the limitation of the 22-character display or printer width.
6. The comments at the right of the listing describe the purpose or results of the program lines at the left. If you are creating a program using a personal computer text editor, you can include similar comments in your program, setting them off from the program objects using the $@$ delimiter (section 6.4.3.1). An especially useful “comment” is a description of the contents of the stack that are obtained after the execution of a program line. In our listings, the stack contents are distinguished from ordinary comments by enclosing the stack objects between $| |$ symbols. The leftmost object in the series is in the highest stack level; the rightmost is in level 1. Thus

$| \ a \ b \ c \ d \ |$

indicates that the object a is in level 4, b in level 3, c in level 2, and d in level 1.

We recommend that you use similar conventions when developing and recording your own programs. Whether you write programs out by hand and type them into the HP 48, or use a personal computer to write programs and transfer them to the HP 48 via the serial port, program stack diagrams and comments are invaluable for later understanding and modification of the programs. Of course, there will be many occasions when you create a program directly in the HP 48 command line without benefit of any program listing. In these cases, we still recommend that you afterwards make a listing, or copy the program to a personal computer file, so that you can recover the program if you lose it for any reason.

13.3 Terminology

Finding useful terminology to describe a computer system like the HP 48 with new or unusual features can be a substantial problem. We have to use existing English words that are close to the meaning we wish to convey, but the dictionary definitions of the words usually differ from their meanings as applied to the HP 48. Consider the word *object*: for the HP 48, *object* means any of the mathematical or logical elements that constitute the data and procedural building blocks of the RPL language, but you won't find that meaning in a dictionary (although it is close to the definition used in mathematics).

Our solution to this difficulty is to provide precise definitions of any terms that we use that are specific to the HP 48, and then use those definitions consistently throughout. In some cases, the definitions we offer may differ from those used in the HP 48 manuals, usually because we need more careful definitions to get across a particular point. For example, the owners' manuals do not make a distinction between *execute* and *evaluate*. We find that such a distinction is useful (section 3.3) because it simplifies the descriptions of related subjects, such as the nature of global name objects (section 3.6.1).

Two other important terms that arise frequently are *mode* and *environment*. A mode is a calculator setting, often associated with one or more *flags* (section 7.1), that determines how a particular keystroke or command will behave. For example, in *polar mode*, complex numbers and vectors are displayed in polar coordinates rather than the usual rectangular coordinates. An *environment* is a glorified mode, which determines the entire calculator interface, including the display, key actions, and available operations.

The "home base" for the HP 48 is the *standard environment*. In this environment, the display shows the status area, stack, and menu key labels. All keys are active, with their ordinary labeled definitions. If you press  **GRAPH**, the HP 48 switches to the *plot environment*. Here the display is devoted to a graph or other picture, the menu keys are restricted to a menu of plotting operations, and the remaining keys are either assigned additional plot actions or are inactive altogether. Pressing **ATTN** returns to the standard environment. Other environments include the EquationWriter, the MatrixWriter, and the equation and statistics matrix catalogs.

While introducing and using this kind of specialized terminology, at the same time we will be using an informal style that takes some liberties with the language to avoid unnecessarily stilted descriptions. "You are in the program branch menu" is almost a non-sequitor when taken out of context, but it reads more easily than "the current HP 48 menu is the program branch menu," and its meaning is clear.

14. HP Solve

HP Solve is an interactive equation-solving system that has become an important feature on several advanced HP calculators, starting with the HP 18C in 1987. This system was developed as a generalization of the “time-value-of-money” (TVM) key system that originated with the HP 80 calculator. In HP Solve, variable menu keys replace the fixed TVM keys, and a general-purpose numerical root-finder replaces the TVM equation-solver. A root-finder first appeared as a calculator function on the HP 34C. That original algorithm has been refined several times in later calculators; the HP 48 is the first calculator to permit its application to equations containing physical units.

In brief, HP Solve combines a menu key interface with an iterative root-finder, for the numerical solution of any problem that can be expressed as a single equation containing one unknown variable. This powerful system relieves you of the need to rearrange or solve an equation by hand before entering it into the calculator, or to write programs to find solutions.

A “solution” to such a problem is the determination of a value of the unknown variable, called a *root*, for which the equation is satisfied--the left side equals the right side. The *root-finder* is a built-in program that finds a root of an equation by an iterative process. It starts with one or two estimates of the root that you supply, then adjusts the value of the unknown variable repeatedly to minimize the difference between the two sides of the equation.

The TVM system in HP calculators uses a highly refined solving algorithm that is customized for the time-value-of-money equation

$$PV + (1 + I \cdot p) \cdot PMT \cdot \left(\frac{1 - (1 + I)^{-N}}{I} \right) = -FV \cdot (1 + I)^{-N}$$

where

PV is the present value;

FV is the future value;

I is the periodic interest rate;

N is the number of compounding periods;

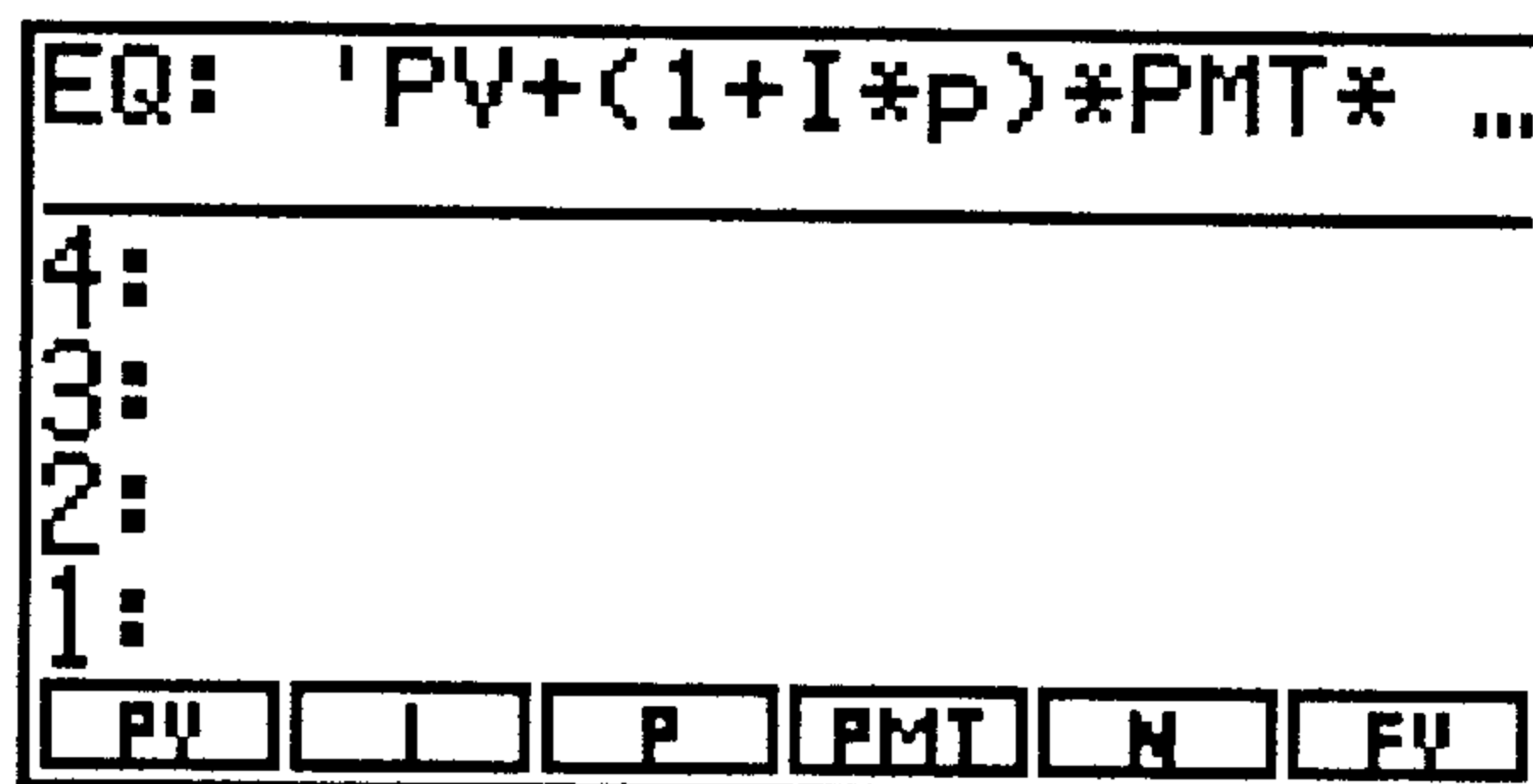
PMT is the periodic payment; and

p has the value 1 for “Begin” mode and 0 for “End” mode.

This equation can serve as an initial illustration of the HP48 implementation of HP Solve. Using the EquationWriter, you can enter the equation as it is shown above; or, you can use the command line to enter it in a linear format:

$$'PV + (1 + I * p) * PMT * ((1 - (1 + I)^{-N}) / I = -FV * (1 + I)^{-N}'$$

With the equation in level 1, press $\leftarrow$ **SOLVE** $\equiv$ **STEQ** $\equiv$ **SOLVR** $\equiv$:



The $\equiv$ **SOLVR** $\equiv$ key does two things:

- It displays the *current equation*, an algebraic object or program that is selected for solving, as specified by the contents of the variable EQ. In this case, the TVM equation was stored directly in EQ by $\equiv$ **STEQ** $\equiv$.
- It activates the *HP Solve variables menu*, or *solver menu* for short, which is derived from the current equation. This menu contains a menu key labeled with a variable name for each variable in the current equation.

The solver menu key labels are displayed in dark characters on a light background rather than the usual light-on-dark, to indicate that the menu keys have definitions that are effectively reversed from those of normal menu keys. That is, in the VAR or CST menus, pressing an unshifted menu key labeled by a global name executes the name object. For data class objects (section 3.4), this is equivalent to recalling the object stored in the corresponding variable. A left-shifted menu key stores a stack object into the named variable. In the solver menu, the primary and left-shift definitions are exchanged: a primary menu key *stores* a stack object, and the left-shifted menu key returns an object to the stack. In this case, the returned object is *not* the object currently stored in the variable, but instead is a number that is computed to be a solution of the current equation. This means that for this value of the specified variable, the current equation evaluates to zero or near zero, given the stored values of all of the other variables in the equation.

For example, use the TVM equation to compute the monthly payment on a 30-year loan of \$50,000, at an annual interest rate of 9%. Enter the values as follows:

$\leftarrow$ MODES	2	$\equiv$ FIX $\equiv$	Set display to two decimal places.
50000	$\equiv$ PV $\equiv$		Principle value of loan.
.09 12	$\div$	$\equiv$ I $\equiv$	Monthly interest rate.
30 12	$\times$	$\equiv$ N $\equiv$	Number of payment periods.
0	$\equiv$ P $\equiv$		End mode.
0	$\equiv$ FV $\equiv$		Future value (loan paid off).

Now use the $\leftarrow$ **REVIEW** key to check the entries:

EQ:	'PV+(1+I*p)*PMT* ...				
PV:	50,000.00				
I:	0.01				
p:	0.00				
PMT:	undefined				
N:	360.00				
FV:	0.00				
$\square$ PV $\square$	$\square$ I $\square$	$\square$ P $\square$	$\square$ PMT $\square$	$\square$ N $\square$	$\square$ FV $\square$

At this point, you have not specified a value for the payment, so it appears with the value undefined. For this problem, you don't need to specify any value (the calculator will start with the value zero), so you can proceed to solve for the payment. In general, even the unknown variable needs an initial value to guide the root-finder (see section 14.5). Press $\leftarrow$ $\equiv$ **PMT** $\equiv$:

Zero					
4:					
3:					
2:					
1:	PMT: -402.31				
$\square$ PV $\square$	$\square$ I $\square$	$\square$ P $\square$	$\square$ PMT $\square$	$\square$ N $\square$	$\square$ FV $\square$

The TVM equation uses the convention that a payment (out) is a negative amount, so the result is negative, returned as the tagged object PMT: -402.31. The status area of

the display also shows the result, plus a *qualifying message*, in this case Zero, that provides information on the nature of the solution (see section 14.4.1).

Underlying HP Solve is a numerical root-finder that iteratively adjusts the value of the unknown variable until the equation is satisfied to within the numerical precision limits of the calculator. Although the HP 48 root-finder is uncommonly sophisticated and robust, the unique power of HP Solve arises from its ability to analyze an equation automatically to create the solver menu, relieving you of any need to program beyond entering the equation as an HP 48 object. The menu is a great convenience, since it lets you enter and keep track of the variables' values, and solve for any of the variables. In particular, you can solve for one variable, then enter a new value for that variable and re-solve for another variable. In the current example, suppose that you can only afford a monthly payment of \$300. Then you can solve to see how large a loan you can obtain:

-300 $\equiv$ PMT $\leftarrow$ $\equiv$ PV $\equiv$ PV: 37284.56

Some other HP calculators that provide HP Solve do not require a shift key press to identify a variable for solving. Instead, solving occurs when you press two consecutive menu keys. This shortcut assumes that normally you will solve for a variable immediately after storing a new values for one or more of the others, which is a very natural way to use HP Solve. However, if you interrupt this flow by performing any non-store operation, you must then do an extra store with a menu key. The HP 48 instead uses the less ambiguous shift-to-solve method in order to preserve more flexibility in HP Solve calculations. For example, you can enter or compute values for several of the known variables together, then store them into the variables by pressing their respective solver menu keys. You could not do this if pressing two consecutive menu keys started the root-finder.

14.1 The Equation Entry Menu

Interactive use of HP Solve is centered on the HP Solve equation entry menu. When you press $\leftarrow$ to activate this menu, you see a display like this:

```

Current equation:
EQ: 'PV+(1+I*p)*PMT*(...
4:
3:
2:
1:
SOLVE ROOT NEW EDEQ STEQ CHT

```


The status area displays the current equation, preceded by the name of the variable in which it is stored. If there is no variable EQ in the current directory, the menu display looks like this:

```

No current equation.
Enter eqn, press NEW
4:
3:
2:
1:
SOLVR ROOT NEW EDEQ STEQ CAT

```

The message provides some guidance for new users on how to create a current equation. The menu contains the following menu keys:

- SOLVR for activating the solver menu.
- ROOT for entering ROOT, the command form of the root-finder.
- NEW for entering and naming new equations.
- EDEQ to copy the current equation to the command line for modification.
- STEQ for storing an object directly from the stack into the variable EQ.  **STEQ** recalls the current contents of EQ to level 1.
- CAT for reviewing and selecting potential current equations from memory.

NEW provides a convenient shortcut for activating a new HP Solve equation. After you have entered a new object into level 1, pressing **NEW** produces the following display:

α

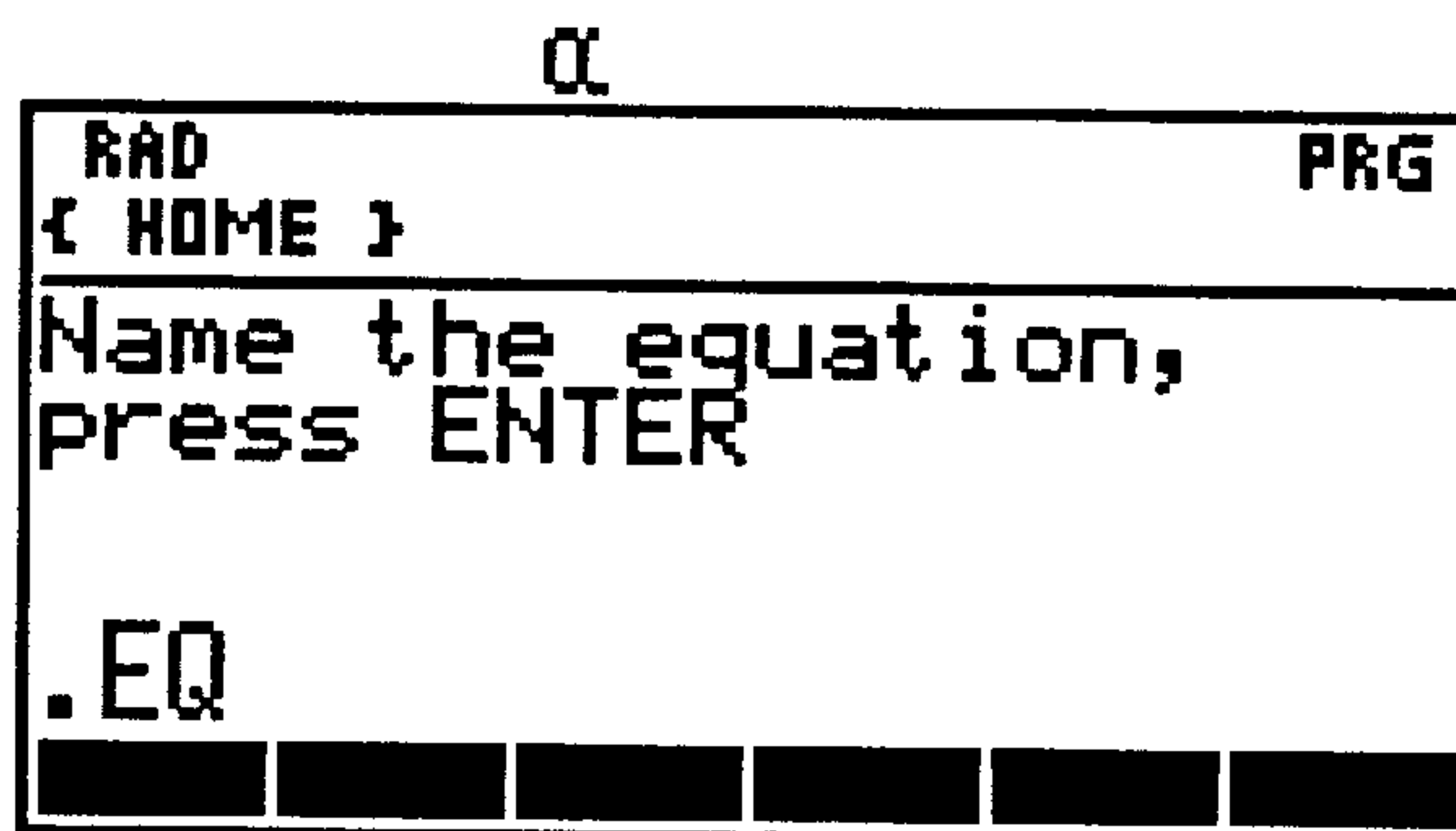
```

RAD PRG
{ HOME }
Name the equation,
press ENTER

```


You now type a name, and press **ENTER**. Notice that the α annunciator is on, and you may type alpha-keys directly. **ENTER** stores the level 1 object in a global variable with the name you supply, and also stores that name in EQ so that the new object becomes the current equation.

When the level 1 object is anything other than an algebraic object, the initial **NEW** display looks like this:



The .EQ (or ,EQ when flag -51 is set) is a suffix for the variable name that ensures that the variable will be included in the equation catalog (section 14.2.3.2). If you don't want to use the suffix, press **ATTN** to clear the command line before you enter the variable name.

When the current equation object is a program or algebraic object of substantial size, the time it takes to show the object every time you press **◀ SOLVE** or **▶ SOLVE** can be a nuisance. You can suppress the automatic displays associated with these and all other HP 48 menus by setting flag -58. With the flag set, you can still produce the special displays on demand by pressing **◀ REVIEW**.

14.2 Basic HP Solve Operation

The general procedure for using the HP Solve is as follows:

1. Enter or compute the algebraic object or program representing a problem. Following the terminology of the HP 48 Owner's Manual, we call this object the *current equation*, even when it is not literally an equation.
2. Store the problem object in a global variable, and store the variable's name in a global variable named EQ. The variable EQ in the current directory is always used by HP Solve to identify the current equation. You can also store the current equation object itself directly in EQ. This is most suitable when you have no further use for the equation after solving it, since it is overwritten when you next choose a new current equation. The NEW operation (in the **◀ SOLVE** menu)

provides a shortcut method of storing and selecting a new object.

3. Activate the solver menu. You can use $\equiv \text{SOLVR} \equiv$ in the $\boxed{\leftarrow} \text{SOLVE}$ menu, or in the equation catalog menu, or $\boxed{\rightarrow} \text{SOLVE}$ when any other menu is active. The solver menu contains a menu key for each independent variable in the current equation.
4. Enter values for each of the “known” variables--the variables for which you already know the values. Do this by entering a value into level 1, then pressing the appropriate menu key.
5. Store a guess for the solution value of the unknown variable, again by entering a value and pressing the variable’s menu key.
6. Solve for the unknown: press $\boxed{\leftarrow}$, then the menu key for the unknown variable. This starts the root-finder. When the root-finder is finished, it returns the solved value for the unknown to level 1, tagged with the variable’s name, and stores that (untagged) value in the unknown variable. You will also see a *qualifying message* in the display that can help you interpret the result.
7. Verify the result, using the evaluation key $\equiv \text{EXPR} = \equiv$ that appears at the end of the solver menu.
8. Repeat steps 2 through 6 with new values for the variables, and perhaps a new choice of the unknown variable.

A nice property of HP 48 Solve is that the current “equation” does not literally have to be an equation. You can use programs, expressions, and equations almost interchangeably for solving purposes. In effect, HP Solve always solves $f(x) = 0$. In the case of an HP 48 algebraic expression object, $f(x)$ is the expression represented by the object, where x is the name of the unknown variable. For an equation object representing $g(x) = h(x)$, HP Solve solves $f(x) = g(x) - h(x) = 0$. For a program, $f(x)$ is the expression that is equivalent to that (RPN) program. Programs used for this purpose must be equivalent to an algebraic expression, taking no arguments from the stack and returning one real number or unit object.

14.2.1 Independent, Dependent, and Unknown Variables

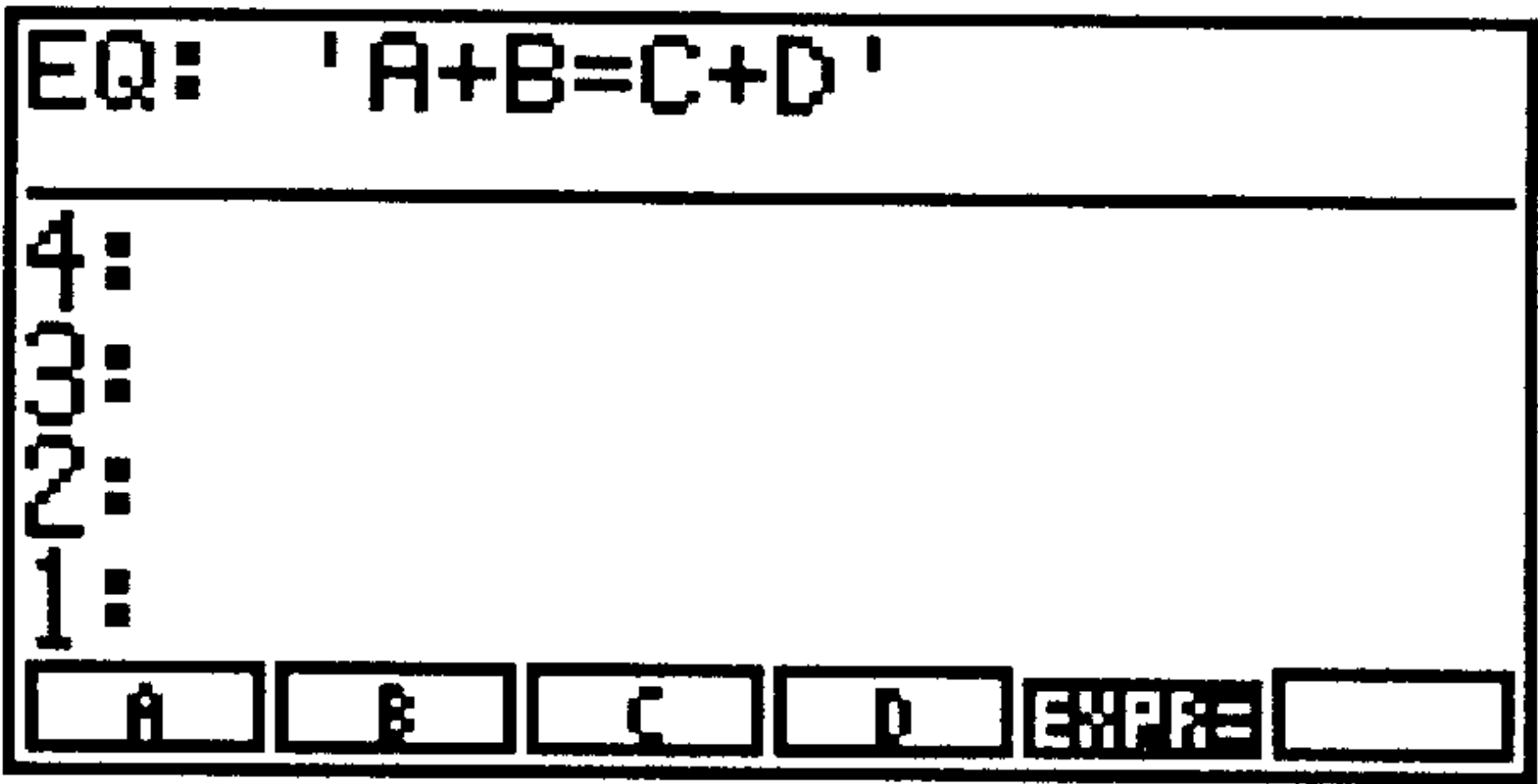
Consider applying HP Solve to an equation ‘ $A+B=C+D$ ’. If any of the four variables takes only numeric values, we call it an *independent* variable, because you can choose any values for it without regard to the other three variables. You can also choose any independent variable to be the *unknown* variable, and solve for its value rather than assigning it. The unknown variable is considered independent because you can at any point assign it a value and solve for one of the other variables.

Suppose now that you store the expression ‘ $B+C+F$ ’ as the value of the variable A.

This changes the role of A to that of a *dependent* variable--you can no longer set its value arbitrarily, but must compute it from other variables. By storing an expression in A, you are saying that the symbol A now is just an abbreviation for the expression.

Of all the variable names that appear in the current equation, only those that are independent variables appear in the solver menu. To see this, enter 'A+B=C+D' as the current equation:

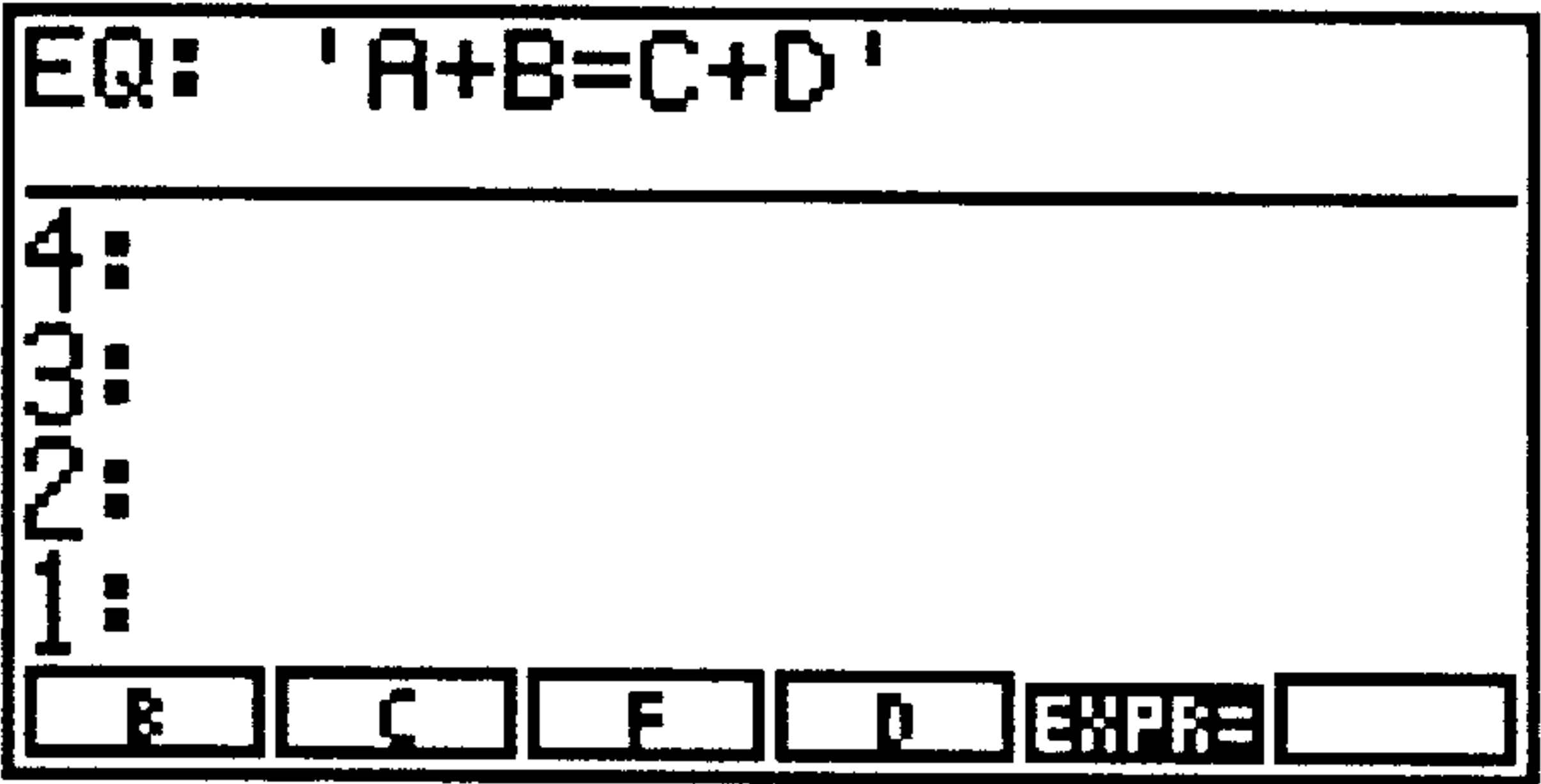
'A+B=C+D'  **SOLVE**  **STEQ**  **SOLVR** :



Here you see menu keys for A, B, C, and D, assuming that these variables do not have procedures already stored in them. This indicates that they are all independent variables. Now execute

'B+C+F' **ENTER**  **A**  ,

then rebuild the solver menu by pressing  **SOLVE** :



Notice that **F** now appears in the menu instead of **A**. **A** is no longer independent, since it is defined in terms of **B**, **C**, and **F**, so it has been removed from the menu. You are really trying to solve ' $B + C + F + B = C + D$ '; the name **A** now is effectively an abbreviation for the expression ' $B + C + F$ '.

In some cases, you may wish to assign a value to a current equation variable and also remove the variable from the menu. For example, a physical constant like the speed of light might appear in the equation as a symbol, but there is no reason to include the symbol (name) in the solver menu, since you won't need to change the value. In such cases, instead of storing the value itself in the variable, store a program consisting only of the value, e.g. `<< 2.998E10 >>` for the speed of light. The equation will evaluate properly for HP Solve, but the solver menu will not include the variable. If you store the program by means of the variable's solver menu key, then press $\boxed{\rightarrow}$ **SOLVE** to rebuild the solver menu, and the menu key will disappear.

In the context of the mathematical function plotting performed by **DRAW**, the term "independent variable" is used to refer to the variable corresponding to the horizontal axis (section 15.7.5). That is, the vertical coordinate represents the value of the current equation as a function of the independent variable. The values of all of the other variables in the current equation are held constant during the plot.

When you use **DRAW** in conjunction with HP Solve, you should use **INDEP** (section 15.2.5) to select HP Solve unknown variable as the independent plot variable corresponding to the horizontal coordinate. Then the plot will represent the value of the current equation as a function of the unknown. In particular, the roots of an expression are shown by the intersections of the curve and the horizontal zero axis. For an equation, **DRAW** plots two curves, which intersect at the equation's roots.

14.2.2 An Example

All of the steps involved in HP Solve are demonstrated in the following problem, which has appeared as a programming exercise in several HP calculator manuals.

■ *Example.* Solve for t in the equation

$$h = 5000(1 - e^{-t/20}) - 200t = 0.$$

(This is imagined as the equation of motion of a "ridget," where h is the height in meters of the ridget above the ground, and t is the time since it was hurled into the air. Thus you are to solve for the time at which the ridget strikes the ground.)

■ *Solution.*

1. Start by entering the equation:

$\leftarrow$ **MODES** $\equiv$ **STD** $\equiv$ 'H=5000*(1-EXP(-T/20))-200*T' **ENTER**

2. Name the equation, and select it as the current equation. Before doing this, however, you might want to save the TVM equation from the previous example:

'EQ' RCL 'TVM' STO

Now press $\equiv$ **NEW** $\equiv$, then **RIDGET** **ENTER**. The new equation is stored in the variable **RIDGET**, and the name **RIDGET** is stored in **EQ**.

3. Activate the solver menu. You can do this by pressing $\equiv$ **SOLVR** $\equiv$ in the $\leftarrow$ **SOLVE** menu or in the equation catalog (section 14.2.3.2), or by pressing $\rightarrow$ **SOLVE**. In this example, you see the following:

RIDGET: 'H=5000*(1-EXP					
4:					
3:					
2:					
1:					
	H	T	EXPR=		

Notice the menu keys for the two variables in the problem, T and H, and the $\equiv$ **EXPR=** $\equiv$ key. The latter appears as the last entry in solver menus, and is used for verifying solutions (section 14.4). **EXPR=** is present on the second page of the TVM menu shown earlier, but you must press **NXT** to see it.

4. Enter values for the known variables. For this example, there is only one known variable, H, which is to have the value zero. Enter 0 $\equiv$ **H** $\equiv$.

5. Enter an initial guess for the solution, which is necessary as a starting point for the root-finder's iteration. You can take your chances and enter any number that seems reasonable, but usually you can do better with a quick analysis. In this case, you can observe from the equation that

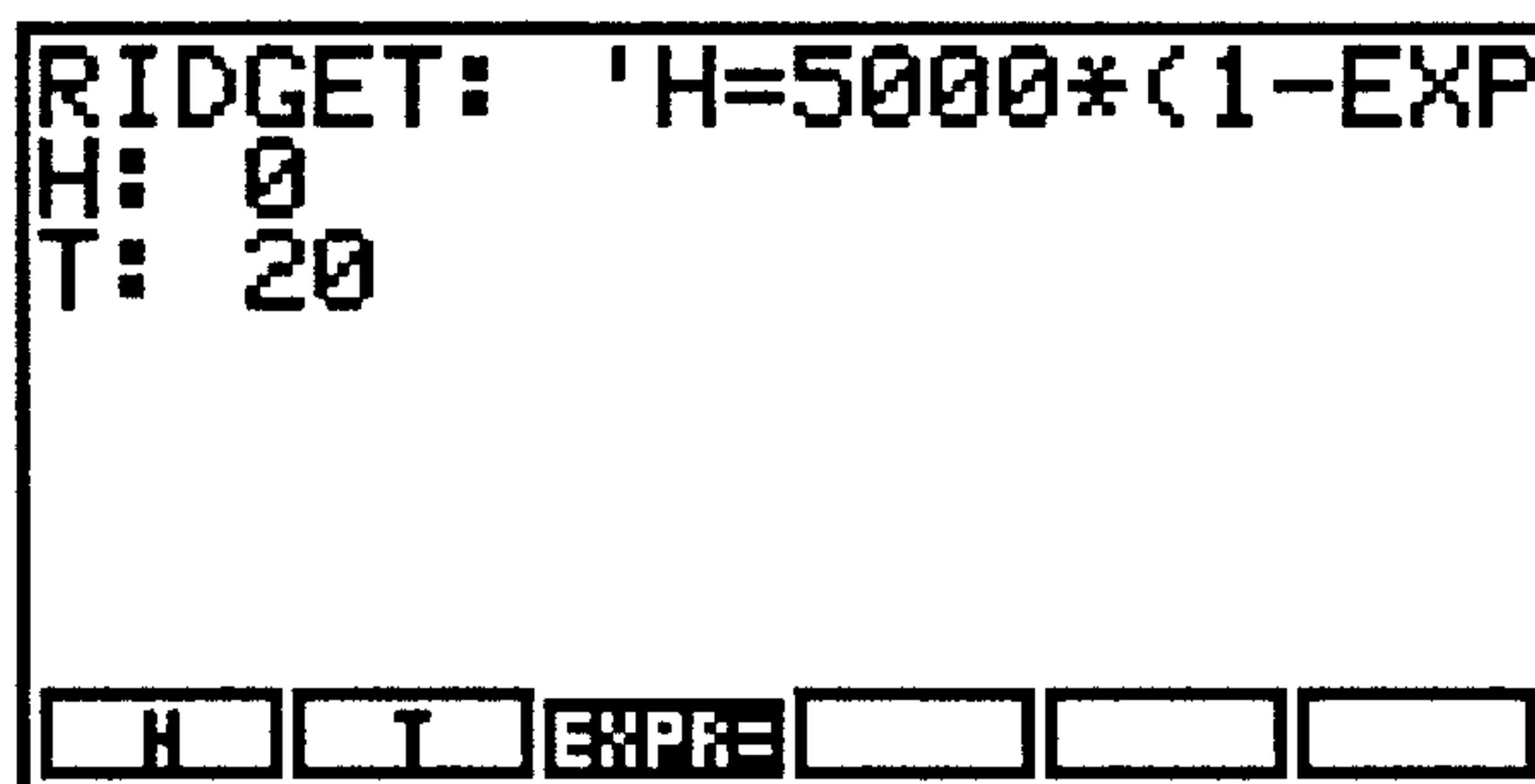
- There is a solution for $h = 0$ at $t = 0$; that solution is of no interest, so you should choose a positive guess away from zero.
- When t is large, you can neglect the exponential term, approximating the equation as

$$5000 - 200t = 0,$$

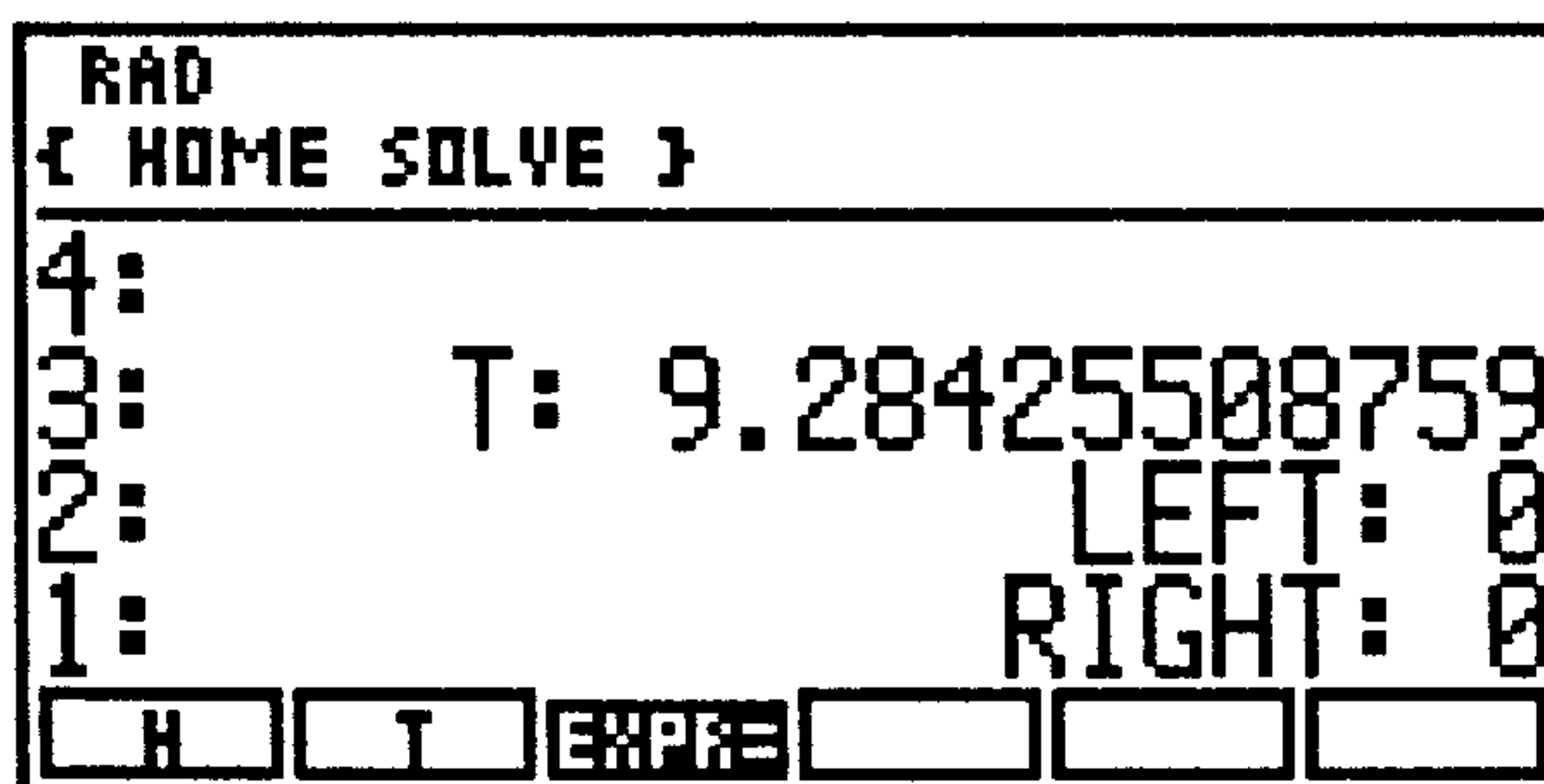
which has the solution $t = 25$.

Since the exponential term is negative, the actual root must be less than 25. (The HP 41 Advantage Pac manual uses guesses of 5 and 6, which do return the right answer, but guesses closer to 25 and farther away from 0 are safer--that is, they are less likely to lead to the $t=0$ solution. Try a guess of 20, by entering $20 \equiv T \equiv$.

Pressing  **REVIEW** shows the values of both variables:

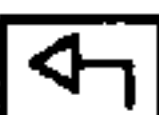


6. Solve for the unknown, by pressing $\boxed{\leftarrow}$ followed by the solver menu key for that variable. Here you are solving for T, so you press $\boxed{\leftarrow} \boxed{\equiv} \boxed{T} \boxed{\equiv}$ to start the root-finder. After a few seconds, the HP 48 returns the solution 9.28425508759, plus the qualifying message Zero. The message indicates that the two sides of the equation are equal (difference zero) to 12 decimal places, when evaluated for this value of T. You can verify the accuracy of the result by pressing $\boxed{\equiv} \boxed{\text{EXPR} =} \boxed{\equiv}$:



This shows that both sides of the equation evaluate to the same value, suggesting that the result is a valid root.

7. Vary the parameters, and re-solve the equation as desired. For example, you might like to know how high the ridget is at $T=5$:

5     105.99608464

As long as you don't make any extreme changes in any of the variables' values after finding one solution, you should not have to enter new guesses for any variable you select as an unknown.

14.2.3 Equation Management

Although it is possible to use HP Solve entirely by storing one equation at a time in EQ, you can accumulate any number of equations stored in global variables, then pick and choose among them as needed. Several HP 48 features help you organize their storage and use.

14.2.3.1 Using Subdirectories

When working through any of the examples in the preceding sections, you may have noticed that an HP Solve equation's variables appear in the VAR menu along with the equation itself and EQ. This is consistent with the usual HP 48 convention that storing into any variable is always done in the current directory (section 5.2). However, the convention is also that recall and evaluation of variables by name can take place from any subdirectory of the directory containing the variables. For HP Solve, this means that you can separate an equation from its variables by placing the variables in a subdirectory. There are several advantages:

- The directory containing the equation can remain relatively uncluttered.
- You can have multiple sets of stored values for an equation's variables, storing each set in its own subdirectory.
- By using CLVAR, you can easily purge all of an equation's variables at once, leaving the equation itself undisturbed.

To carry out an HP Solve exercise using a subdirectory, store the equation object in the parent directory and then switch to the subdirectory. The solver menu is built from the object specified by EQ in the current directory, so you must store the equation's name in EQ in the subdirectory. Then activate the solver menu as usual, using  **SOLVR** or  **SOLVE**. The menu works the same as it does when the equation object is in the

current directory, but as you store values using the menu keys, the variables are created in the current directory rather than in the parent directory with the equation.

You do not even need to keep all of the variables together. For example, when using the TVM equation from section 14.0, you might want to solve several different problems all using the same interest rate. In that case, you can store in the interest in a variable *I* in the same directory as the TVM equation, then create a new subdirectory for each problem. As long as you don't store a new value for *I* in any of the subdirectories, the root-finder will use the value stored in the parent directory. (As described in section 14.2.1, you can even remove *I* from the solver menu by storing the program `<< rate >>` in *I* instead of *rate* by itself.)

14.2.3.2 The Equation Catalog

As you accumulate stored equation objects for solving and plotting, the *equation catalog* is a valuable tool for finding and managing the equations. The catalog is activated by the `≡CAT≡` key in either the `◀◀ SOLVE` or `◀◀ PLOT` menus, or, from any other menu, by `▶▶ ALGEBRA`. For example, assuming that you have created the TVM and the RIDGET variables from the preceding examples in the current directory, then `≡CAT≡` gives the following display:

```

{ HOME }
┌───────────┐
│▶RIDGET: 'H=5000*(1-E...
│TVM: 'PV+(1+I*p)*PMT...
└───────────┘
PLOT SOLVE EQ+ EDIT ÷STK VIEW

```

The equation catalog is an *environment* (section 13.3), meaning that the display and keyboard are dedicated to its use. The primary feature of the display is a vertical listing of variables, each in the format *name: content*. Only the first eight or fewer characters of a name are shown, to leave room for displaying the stored object on the same line. The variables are shown in the same order as they appear in the VAR menu. Up to five variables are shown, along with a *selection arrow*--the triangle in the leftmost column. If the catalog lists more than five, you can move the arrow through the list with `▶` or `▼`. `◀▶` `▶` and `◀▶` `▼` move the arrow up and down one "page" of five variables at a time. `▶▶` `▶` selects the first variable in the catalog, and `▶▶` `▼` selects the last.

The catalog does not show all of the variables in the current directory, but restricts its

listing to the following:

- Variables containing algebraic objects. These are presumed to be the most common entries for solving and plotting, so they are always included in the catalog.
- Variables with names ending in .EQ or ,EQ. This lets you include objects of any type in the catalog, especially programs and lists.
- EQ, regardless of its content. This reminds you of which object is the current equation.
- Directories. These are listed in the shortened format *name: dir*.

The catalog display can be slow when the stored objects are large, because of the time required to build the display form of each object. To move more quickly around the catalog, press **≡FAST≡** (in the second page of the catalog menu). This option suppresses the object display and shows only the variable names. The HP 48 remembers this mode between catalog sessions by means of flag -59, which is set for the fast display, and cleared for the full display. The white square in the key label indicates that the fast display is selected; pressing **≡FAST□≡** again restores the full display.

You can move the catalog through the directory system without leaving the catalog environment. Moving the selection arrow to a directory, then pressing **ENTER**, switches to that directory and shows the catalog derived from its variables. **◀ UP** and **▶ HOME** are also active in the catalog environment, for moving upwards through the directory structure.

The selection arrow specifies a particular variable for the operations represented by the catalog menu keys. For example, moving the arrow to a particular variable and pressing **≡SOLVR≡** stores that variable's name in EQ and activates the solver menu. **≡PLOT≡** similarly activates the **▶ PLOT** menu. Other options:

- **≡VIEW≡** displays the selected object in its command-line form (up to four lines). The display persists while you hold the key down.
- **≡EDIT≡** copies the selected object to the command line, where you can edit it. As usual, **ENTER** stores the edited version back in the variable, and **ATTN** cancels the changes and returns to the catalog display.
- **≡-STK≡** copies the selected object to the stack, tagged with its variable name. For variables that are not directories, **ENTER** has the same effect as **≡-STK≡**.
- **≡ORDER≡** (second page) moves the selected object to the beginning of the current directory, so that it appears as the first entry in the catalog. Pressing **≡ORDER≡** with a variable *name* selected is equivalent to executing { *name* } ORDER (section 5.1).

- **PURG** purges the selected variable from memory.

EQ+ provides a means for applying certain catalog operations to a list of names. The first time you press **EQ+**, it creates a list containing the selected variable name. The list is displayed in the status area. Then you can move the selection arrow to other entries; each subsequent use of **EQ+** adds the selected name to the list. While such a list is displayed, pressing **SOLVR** stores the list in EQ and activates the solver menu; **PLOTR** works similarly. Also:

- **EDIT** copies the list to the command line for editing.
- **-STK** copies the list to the stack.
- **ENTER** copies the list to the stack, unless the selection arrow points to a directory. In that case, the selected directory becomes current and the pending list remains active.
- **ORDER** applies the command ORDER to the pending list, and clears the pending list.
- **ATTN** returns to normal catalog operation, discarding the list.

VIEW and **PURG** apply as usual to the selected variable, ignoring any pending list.

14.3 Solving With Units

The current equation and any of its variables may include unit objects, and the result returned by HP Solve will be a unit object when appropriate. The general approach to solving with or without units is the same, but there are a few points to bear in mind.

- If the unknown variable requires units to satisfy the current equation, then the initial guess for the unknown must have units. The HP Solve result will be expressed in the same units as the original guess. It is not possible for the HP 48 to provide a default guess, nor can it determine even the dimensions of the result. (To do so would require a symbolic solution, which is not within the scope of HP Solve. See section 14.8.)
 - You can use the solver menu to store unit objects as values for the variables. Once you have established the units of a variable by storing a unit object, you can change the magnitude of the unit object by entering a real number and pressing the menu key; this leaves the unit part unchanged. However, if you later want to use that variable in another HP Solve problem where it has no units, you can store a dimensionless number in the variable by enclosing the number in a list, or by purging the variable first.
- *Example.* A railroad car weighing 5 tons rolls with an initial speed of 4 ft/s down a

225 foot ramp with a vertical drop of 3 feet, then across a level track 175 feet long, terminated by a spring-loaded bumper. The rolling friction of the car is 50 pounds force. What must the spring constant be in order to stop the car by compressing the spring 2 feet?

■ *Solution.* This example is most easily solved as a conservation of energy problem. That is, the initial kinetic plus potential energy of the car must equal the energy dissipated by friction plus the energy associated with compressing the spring by 2 feet. Expressing this as an equation, we have

$$'M*V^2/2+M*g*H=F*(R+L+X)+K*X^2/2',$$

where M is the car mass, V is the initial speed, g is the acceleration of gravity, H is the ramp height, F is the frictional force, R is the length of the ramp, L is the length of the level track, X is the compression of the spring, and K is the (unknown) spring constant.

To use HP Solve, start by storing the above equation as the current equation:

'M*V^2/2+M*g*H=F*(R+L+X)+K*X^2/2' ⏮ SOLVE ≡NEW≡ RRCAR ENTER

The solver menu (≡SOLVR≡) then looks like this:

RRCAR: 'M*V^2/2+M*g*						
4:						
3:						
2:						
1:						
	M	V	G	H	F	R

Since there are more than six variables in the equation, you must press NXT to see the remaining three variable keys plus ≡EXPR=≡.

Now enter the parameters from the problem:

5_ton ≡M≡
 4_ft/s ≡V≡

1_ga	<u>G</u>
3_ft	<u>H</u>
50_lbf	<u>F</u>
225_ft	<u>R</u>
175_ft	<u>L</u>
2_ft	<u>X</u>

Since the equation is linear in K, any initial guess will do, as long as it includes the appropriate units:

0_lbf/ft K $\leftarrow$ K $\rightarrow$ 6193.23_lbf/ft

14.3.1 Faster Solving with Units

Evaluation of algebraic objects containing units is intrinsically slower than ordinary numerical evaluation due to the unit conversions that take place as the evaluation proceeds. Since using a root-finder requires repeated evaluation of the current equation, solving an equation with units can sometimes take a long time. For faster solutions, you can of course remove all of the units from an equation and solve with dimensionless quantities, but then you have to do all of the unit conversions manually before and after solving. Short of this, you can often obtain substantial speed improvements by replacing the current equation with a modified version in which as many of the equation's calculations as possible have already been performed. (This will speed up solving with any equation, but the effects are most dramatic for equations containing units.)

The easiest way to preprocess an equation or expression to reduce its number of operations is to use **SHOW** (section 16.4.2). This command can selectively resolve some variables while leaving others unevaluated; for efficient solving the idea is to evaluate all names except that of the unknown variable. For example, after entering values for all of the variables in the railroad car problem, you can reduce the equation as follows:

RRCAR { K } SHOW $\rightarrow$ '16.2432380069_ton*ga*ft=20100_lbf*ft+(K*4_ft^2)/2'

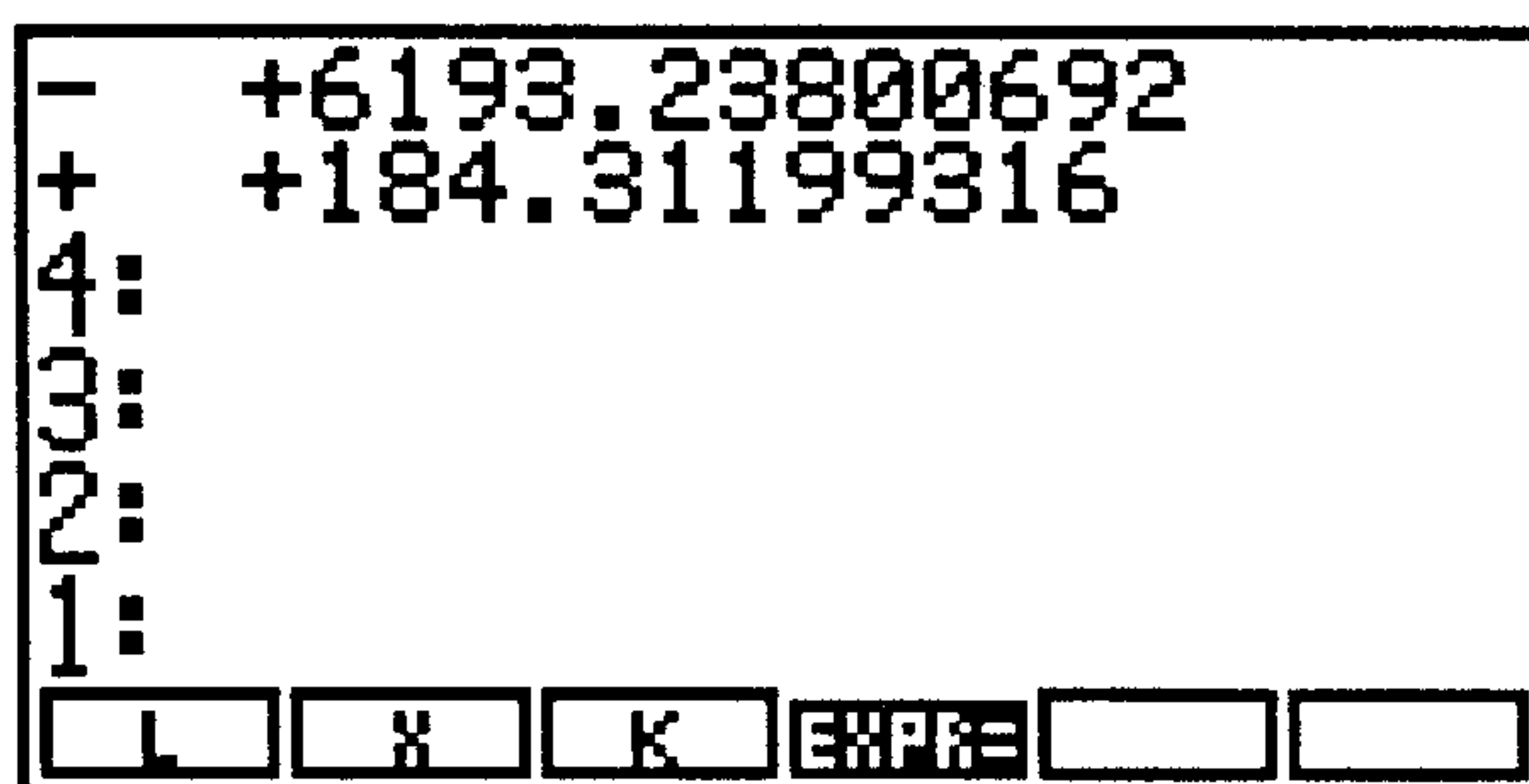
Now only K remains in the equation, and many of the arithmetic operations have been carried out. If you make this result the current equation, then solve for K, you will see that the solution is obtained much more rapidly. However, you can do even better by changing the equation into an expression, since one more unit subtraction is eliminated:

EQ $\rightarrow$ - COLCT $\rightarrow$ '12386.4760137_lbf*ft+(-2_ft^2)*K'

Using this expression as the current equation gives the fastest possible solution for K, using HP Solve or ROOT.

14.3.2 Monitoring Convergence

When a particular HP Solve solution appears to be taking a long time, you can elect to watch the root-search in progress, so that you can decide whether to continue or to terminate the process. Pressing any key while the root-finder is active creates a display like this:



The status area shows two numbers, each preceded by a + or - sign; new values are displayed at each root-finder iteration. The numbers are the values of the unknown variable that bound the region in which the root-finder is searching; the + or - symbols indicate the sign of the current equation evaluated at the two points.

There are several circumstances that can cause the root-finder to iterate for a long time. For example:

- The current equation has the same sign at both edges of the region defined by the initial guesses. If the region includes a root, the root-finder can usually home in on it quickly. However, if there is no root in the region, the root-finder has to search outside of the region. The search may take many iterations before finding a sign change. In that case, when you monitor the search you will see the numbers changing while the two signs remain the same.
- The current equation is relatively “flat” around a root. In this case, the two signs are opposite, but it takes many iterations to narrow the search region to isolate the root. You will see the two numbers slowly get closer in value, with one or the other changing by a few decimal places at each iteration. The example $\text{IFTE}(X>0,1,-1)=0$ discussed in the next section is an extreme case; it can take several hundred iterations to refine the successive values to 0 and $1\text{E}-499$.
- The root-finder is searching in a region where the current equation is constant.

14.4 Interpreting Results

Not all HP Solve problems work out as nicely as the ridget example. An equation may have multiple solutions, no solution, discontinuities, infinities, etc. It is important when you use HP Solve that you do not accept the “solution” at face value, but take some steps to interpret the result to assure yourself that it is a meaningful solution. This is the purpose of EXPR= and of the *qualifying message*.

The EXPR= key is provided in the solver menu as a ready means for evaluating the current equation to verify a solution:

- When the current equation is an expression or a program, EXPR= evaluates it, tagging the result with EXPR:. Following a successful HP Solve solution, the result should be zero or near-zero.
- When the current equation is an actual equation, EXPR= evaluates the two sides of the equation separately, returning two results tagged with LEFT: and RIGHT:. The equality or near-equality of these two results is a measure of the validity of the solution.

EXPR= is handy for “table-filling” evaluation with symbolic values as well as numerical. Consider, for example, filling in the following table:

A	B	C	A+B+C
12.5	13.7	50.1	?
14.9	6.4	75.0	?
11.7	22.2	64.3	?

The task is to sum the each row of entries for A, B, and C. No root-finding is necessary, but the solver menu is still helpful. After storing 'A+B+C' as the current equation, each sum can be computed like this:

12.5 A 13.7 B 50.1 C EXPR=  EXPR:76.3

You can also store an equation like 'SUM=A+B+C' in EQ, and use  SUM in the solver menu to find each sum, but using the root-finder is always slower than a straight-forward evaluation with EXPR=.

In some cases, EXPR= may not return numerical results. It uses EVAL rather than →NUM, so that only one level of evaluation is performed (section 3.5.2). You can always

force a numerical result by setting flag -3 before using `EXPR=`, or by executing `→NUM` afterwards.

14.4.1 Qualifying Messages

The message **Zero** obtained in the ridget example is the most welcome of the possible qualifying messages. **Zero** means that the HP 48 has found a value of the unknown variable for which the current equation numerically evaluates to exactly zero. “Exactly” in this sense means exact to 12-digit precision, the numerical accuracy of the HP 48. When you see the **Zero** message, you are assured that `≡EXPR≡` will return a single value of zero, or two equal values for the left and right sides of an equation.

When HP Solve returns the qualifying message **Sign Reversal**, it means that it is unable to find a value of the unknown that exactly satisfies the current equation. But it did find two values of the unknown that differ only in the twelfth digit, for which the corresponding values of the equation have *opposite signs*. This means that the equation crosses the zero axis somewhere between the two values, and so either value may be a good approximation of a solution. However, the calculator can’t tell for sure that there is a solution between the two values. For example, HP Solve returns the value 1.E-499 as the “solution” for the equation `'IFTE(X>0,1,-1)=0'`, which has a discontinuity at $X=0$. The **Sign Reversal** message warns you to check the solution.

The most immediate method of testing the result is to use `≡EXPR≡` as described in the previous section. In the case of `'IFTE(X>0,1,-1)=0'`, `≡EXPR≡` returns **LEFT: 1** and **RIGHT: 0**. The disparity between these two results indicates that this may not be a proper solution. To check further, you should plot the current equation to get a visual indication of its behavior.

If you solve the equation `'X^2=2'`, you obtain the result 1.41421356237, with the **Sign Reversal** message. In this case pressing `≡EXPR≡` returns **LEFT: 1.99999999999** and **RIGHT: 2.00000000000**. The near-equality of these two values indicates that you have a good solution.

Other possible qualifying messages are as follows:

Bad Guess(es) The root-finder returns this error when it is unable to proceed because the guess or guesses that you supplied yield equation values that are not real numbers. For example, you will see this message if you solve `'√X=2'`, and start with a guess of $X=-5$ (for which $\sqrt{X}$ is imaginary).

- Constant?** This error occurs when the equation returns the same value for every value of the unknown tried by the root-finder. The equation is either a constant, or the guesses are in a region where the equation varies so slowly that the root-finder can't make any progress towards finding a root.
- Extremum** The root-finder has found a local minimum or maximum instead of a root. See section 14.7.

14.5 First Guesses

You may find it disappointing that a system as sophisticated as HP Solve requires you to supply a "first guess" of the answer to start the root-finder, so that it will return the "right" answer. After all, the TVM system on HP financial calculators, which is a specialized solver, doesn't require a first guess, yet it always returns the right result.

The need for an initial guess arises because generally equations can have more than one solution (or no solution at all). For example, $x^2 + x - 12 = 0$ has solutions at $x = 3$ and $x = -4$; the equation $\cos(\sin x) = y$ has infinitely many solutions for x . Furthermore:

- It is not practical for the calculator to attempt to determine how many solutions a particular equation has.
- Of all the possible solutions to an equation, there is no way for the calculator to know which one *you* want, based only on the equation itself.

The first point is true because the calculator can find solutions only by searching for them. Roots may occur anywhere between plus and minus infinity. To cover this range, of course, would require an infinite number of steps--or at least a very large number for the finite range of the calculator (from -10^{500} to 10^{500}). This is obviously not practical. On the other hand, if HP Solve took relatively few steps to cover the real number domain, it could easily skip right over a region containing a root, and never find it. In short, an automatic solver can never know when it's finished finding roots, no matter how many it finds. The only reasonable thing for it to do is search until it finds one root, then quit.

The second point is a statement that a choice among multiple roots of an equation cannot be represented within the equation itself--it has to come from external information, often from some physical situation. For example, if the TVM equation somehow gave a negative solution for the interest rate, you could reject that solution as meaningless. But the information that the equation is only supposed to be valid for positive interest is not contained in the equation itself. As another example, consider the equation $(x-2)(x-3) = 0$, which has roots at $x=2$ and $x=3$. Which root is "correct?"

These considerations lead to the requirement that you must specify an initial guess for the root-finder. Your guess tells the root-finder where to start looking for a root, and guides it to a particular root among multiple roots.

14.5.1 How Many Guesses?

The HP48 root-finder begins execution by trying to find a region of values of the unknown variable in which the value of the current equation changes signs. That is, at the two boundaries of the region, evaluating the equation must give values with opposite signs. (The “sign” of an *equation* in this sense is the sign of the difference of the left and right sides.) Then it narrows the region until it contains just one point at which the equation is exactly satisfied (Zero message). Failing that, it finds two neighboring points where the equation has opposite signs (Sign Reversal). You will obtain the fastest results from HP Solve by specifying a good initial search region by means of the initial guess.

The HP 48 gives you the option of making single, double, or triple initial guesses for HP Solve. A *single guess* is a number, a *double guess* is a list of two numbers, and a *triple guess* is a list of three numbers. Any of the numbers can be complex--only the real parts are used (this is a convenience provided to let you use the coordinates of points digitized from plots, which are returned as complex number objects). The best choice is usually the double guess; it is more reliable than a single guess, and the extra certainty provided by the triple guess is seldom necessary. A good double guess contains two values of the independent variable that

- a. define a region in which the equation is well-behaved (no discontinuities, non-real values, or infinities) and which contains the root you want, and only that root; and
- b. yield values of the current equation with opposite signs.

With such a double guess, HP Solve will always home in quickly on the correct root.

It is often sufficient to supply only a single guess that is closer to your desired root than to any other root or extremum. HP Solve takes your single guess and makes its own second guess by duplicating the value and adding a small amount. Unless there actually is a root between the two guesses, the search starts to look outside of the initial region. Then, it is a matter of chance whether it finds the root you want, or some other root or extremum first. It is never guaranteed that the root-finder will find a particular root from a single guess, unless the equation has only one root (and no extrema). However, a single guess usually suffices in cases where you are using HP Solve repeatedly on the same equation, where each time you vary one or more of the independent variables by small amounts. The last solved value of the unknown variable is likely to be close enough to the new value to be a good single first guess. Since that value is already

stored in the unknown variable, it will be used as the first guess unless you explicitly replace it with another guess.

When you interrupt HP Solve by pressing **ATTN**, it returns a list of three numbers. This list is a record of where the root-finder was searching when it was halted, and you can use it as a triple guess to tell the root-finder where to resume its iteration. If you want to restart the root-finder, store the list in the unknown variable, then solve for the unknown. You can also supply your own triple guess as the fastest way to make HP Solve find a particular root. Choose the first number in the triple guess list as your best estimate for the root, and choose the second and third numbers to bound the search region as in the double guess case.

Table 14.1 summarizes the meaning and application of the single, double, and triple guess options for HP Solve.

Table 14.1. HP Solve Guesses

Type of Guess	Meaning	When to Use
Single	One value close to a root.	To solve equations with only one root; to re-solve after adjusting the values of the independent variables.
Double	Two values on opposite sides of a root, where the values of EQ have opposite signs.	To guarantee that the root found will be the one between the two guesses, for equations with multiple roots and/or extrema.
Triple	First guess in the list is a best guess of the root; the 2nd and 3rd surround the root as in the 2-guess case.	To resume an interrupted root-search. Also is faster in general than the 2-guess case.

14.5.2 Examples Using $x(x-2)(x+2) = 0$

To illustrate the effect of different guesses, we will use various types of guesses in solving the equation $x(x-2)(x+2) = 0$ repeatedly. This cubic equation obviously has roots at $x = -2$, $x = +2$, and $x = 0$, so you know what to expect from HP Solve. To get an idea of how quickly the root-finder finds a root in each case, we will count the number of iterations the root-finder makes, by recording each execution of the current equation. This is achieved by the following:

'(X-2)*(X+2)*X' 'CUBEX' STO

<< CUBEX 1 'N' STO+ >> STEQ

When you solve for X by pressing $\boxed{\leftarrow} \boxed{X}$, the value of N is incremented by 1 each time the latter program is executed. The difference in N before and after solving is the number of root-finder iterations. $\boxed{N}$ appears in the solver menu, where you can use it to reset N to zero before each trial, but it has no effect on the root search. Here are the results of various trials:

Initial Guess	Result	Iterations	Remarks
-10	-2	17	Found the most negative root.
+10	+2	17	Found the most positive root.
0.9	0	11	Found the root closest to the guess.
1.1	0	13	Did <i>not</i> find the root closest to the guess.
{ 1.1 10 }	2	14	Found the root between the two guesses. Fewer iterations than with the single guess 10.
{ -.9 3 }	-2	9	Did not find the root between the two guesses--the equation has the same sign at both guesses.
{ 1.9 1.1 10 }	2	8	Found the root between the two guesses; faster than the {1.1 10} case because of the additional "best guess."

To specify multiple guesses for a problem involving units, you can use a list of two or more guesses as in the dimensionless case. One of the list elements should be a unit object with the correct units for that variable. If more than one element is a unit object, the units of the last element with units are used for the root-finder--the other elements are presumed to have the same units, and only their magnitudes are used. If you interrupt the root-finder with $\boxed{ATTN}$, the first element of the three-element list returned will be a unit object.

14.6 Obtaining Guesses

In the preceding sections we discussed the need for initial guesses for the root-finder, without much explanation of how to obtain good guesses to supply. There are three general approaches you can use:

- Use the *default* guess.
- Use mathematical approximation.
- Use guesses obtained from a plot of the current equation. This is the easiest method, and generally the most reliable.

If you don't explicitly store a guess in the unknown variable before solving, HP Solve uses a default value to start the root-finder iteration. The default value is simply the current number stored in the unknown variable, or the number zero if the unknown variable has no current value. When you use the default guess, you are trusting that HP Solve will happen to find the correct root. This will certainly be the case if the equation has only one solution, and no extrema or other properties that might prevent the root-finder from converging on that value. If the current value of the unknown happens to be sufficiently close to a root, it is likely but not certain that the root-finder will return that root.

Using mathematical approximation to obtain a guess consists of studying the equation and trying to estimate the root from the mathematics of the equation. We did this in the problem of the ridget flinger in section 14.2.2. As another example, consider solving $\cos x = x$. For small $|x|$, $\cos x \approx 1 - \frac{1}{2}x^2$. Substituting this approximation in the equation, you obtain the quadratic equation $2 - x^2 = 2x$, which has the solutions $x = \pm\sqrt{3} - 1$. The choice of the negative root gives a value of x too large for the approximation to be valid, leaving the positive root $x = \sqrt{3} - 1 \approx .732$. With this value as a first guess for X for a current equation ' $\text{COS}(X)=X$ ', HP Solve returns the result 0.7391 for X , which is quite close to the guess. The approximation method evidently can provide very good first guesses, but it does require some mathematical skill and intuition.

The preferred method of obtaining guesses is to *plot* the current equation and view the resulting curve to identify roots. Furthermore, you can invoke the root-finder directly while in the plot environment, using the plot cursor to specify an initial guess. Usually, you can place the cursor so close to a root that the single guess is sufficient to obtain the desired solution. You can use a plot to obtain single, double, or triple guesses and ensure that HP Solve finds the root you want. This application of plotting is an important reason for the existence of the plotting feature in the HP 48.

The key point to remember when you plot an expression is that the roots of the

expression are the values of the independent (abscissa) variable for which the plotted curve intersects the horizontal axis. Thus you can literally *see* the roots in the plot (assuming that the drawn horizontal axis passes through the origin, which is the usual case). For an equation, the HP 48 plots the left and right sides of the equation independently. The roots of the equation are the independent variable values for which the two sides have the same value. The roots of equations appear as the intersections of the two curves.

To illustrate the plot/solve process, consider the following problem.

■ *Example.* A rocket makes a round trip to α -Centauri, 4 light-years from the earth. On both legs of the voyage, the rocket accelerates uniformly to the half-way point, then turns around and decelerates at the same rate until it arrives at its destination. What acceleration is required so that the trip can be accomplished in 20 years of earth time? By uniform acceleration we mean that the astronauts experience a constant acceleration in their own frame of reference, so that they feel as if they were in a uniform gravitational field.

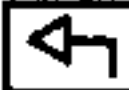
■ *Solution.* Since the trip divides into four kinematically identical 2 light-year segments, we only need to consider one segment, which must be traversed in 5 years. Looking at the first segment, let X be the distance from the earth and T the elapsed time, both measured on earth clocks. These are related by the formula

$$X = \frac{c^2}{A} \left[\sqrt{1 + \left(\frac{AT}{c}\right)^2} - 1 \right].$$

where A is the unknown acceleration, and c is the speed of light. Enter this formula using the EquationWriter, or enter the linear form in the command line:

$$'X - c^2/A * (\sqrt{1 + (A*T/c)^2} - 1) + 0\_lyr'$$

and store it as the current equation:

 **SOLVE**  **NEW** SPACE  **ENTER** .

The 0_lyr at the end of the equation ensures that both sides of the equation have the same units. Without this, a plot of the equation would have a misleading appearance, since the units are dropped when plotting (see section 21.3.7).

Now use the solver menu ( **SOLVR**) to enter the various parameters:

2_lyr	<u>X</u>
1_c	<u>C</u>
5_yr	<u>T</u>
0_ga	<u>A</u>

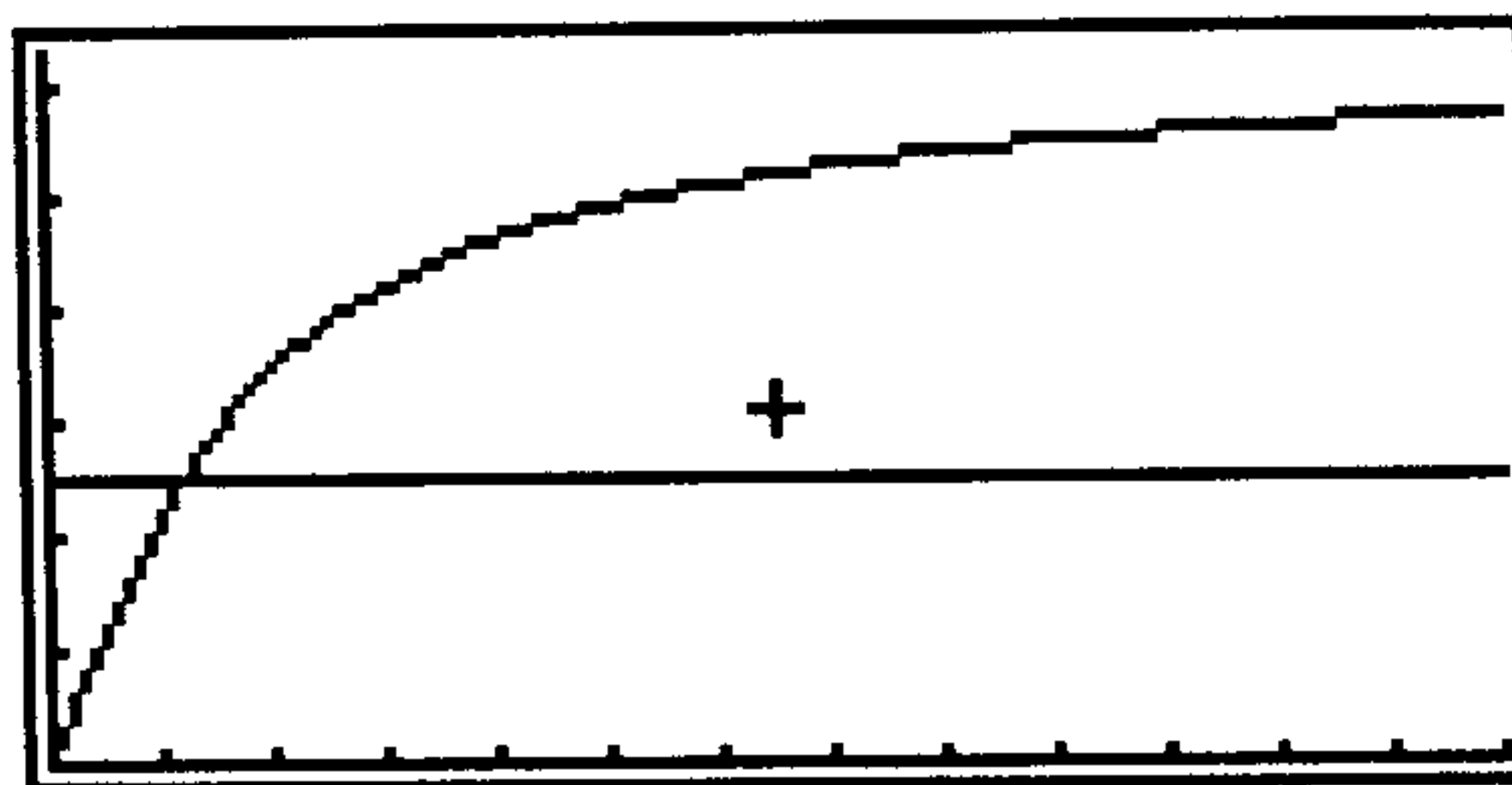
The last entry serves to establish the units for A, which are required to make a plot. Continue as follows:

← **PLOT** PTYPE FUNC → MODES **NXT** -30 SF

It is necessary to set flag -30 so that both sides of the equation will be plotted (see section 15.4).

→ **PLOT** 'A' INDEP .001 2 XRNG 0 5 YRNG

The units of the vertical coordinate are determined by the dimensions of X. The right-hand side of the equation is the distance traveled in 5 years, which can not be greater than 5 light years, so we choose a range from 0 to 5. The horizontal plot limits are set from 0 to 2 (ga), which is a range of accelerations that might be tolerable to human beings--if there is no solution in this range, then the space trip is certainly impractical. Now make the plot by pressing ERASE DRAW :



Since the plotting is fairly slow, you might stop the plotting by pressing **ATTN** as soon as you see the curve representing the right-hand side intersect the horizontal line at X=2. To find the solution, move the cursor over to the vicinity of the intersection, and press FCN ISECT , which returns the coordinates of the intersection. The computed acceleration, A=0.185 ga, is comparable to the acceleration of gravity on the surface of the moon.

Once you have used the interactive plot environment to find a solution for a problem, you can return to the solver menu to find additional solutions as you vary the problem's parameters. Generally, you won't change any of the parameters enough from trial to trial to require replotting for determining new first guesses. In the current example, you might ask how long the trip might take if the acceleration were increased to 1 *ga*:

[R→] [SOLVE] 1 **[≡] [A] [≡]** **[←] [≡] [T] [≡]** **[⇐]** T: 2.81_yr.

The round trip time is four times this result, or 11.2 years.

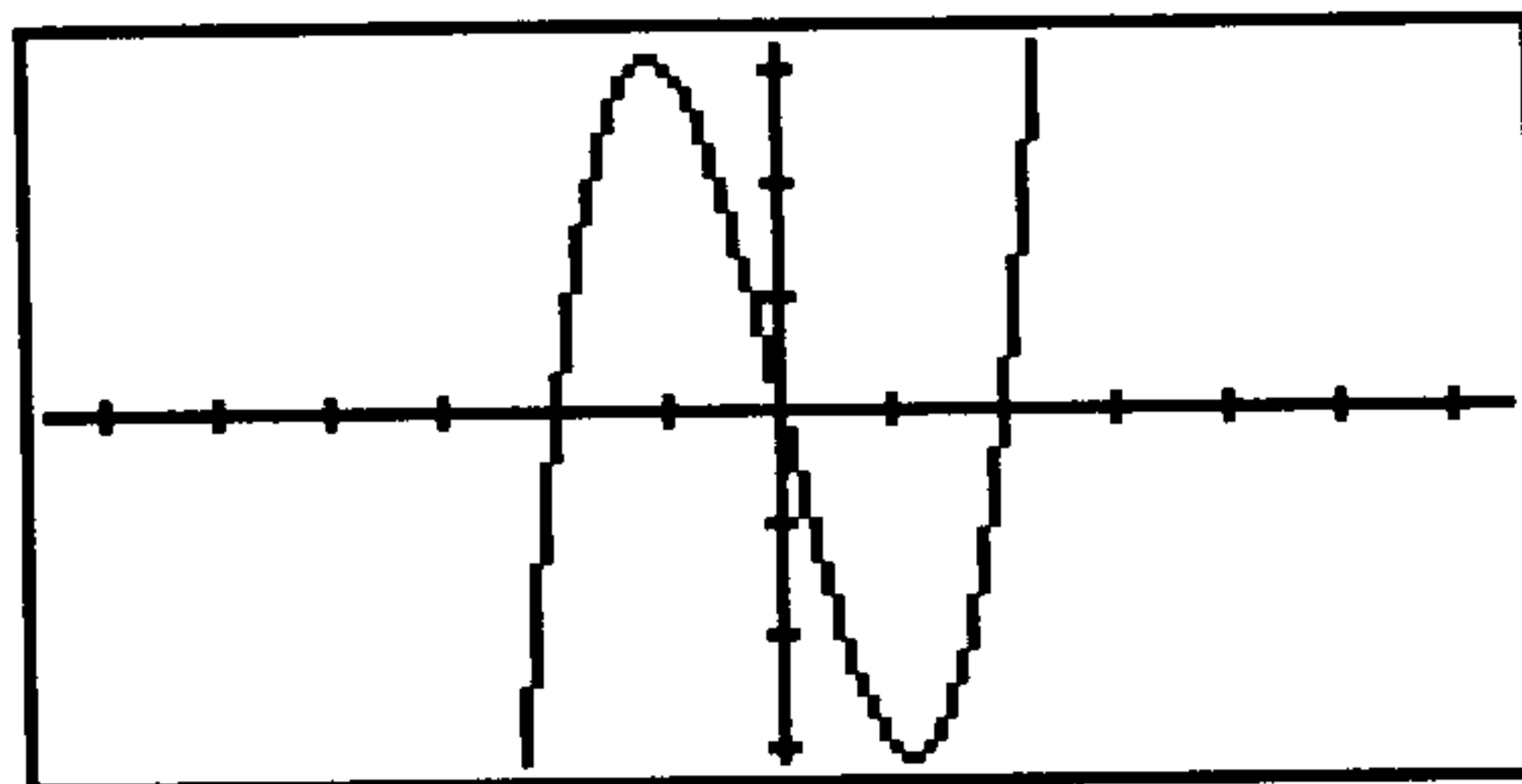
You can also use a plot to obtain guess values, without immediately using **[≡] [ROOT] [≡]** or **[≡] [ISECT] [≡]** to find a solution. As described in section 15.3.1, pressing **[ENTER]** returns the cursor coordinates to the stack as a complex number object. If the unknown variable is dimensionless, you can store the complex number directly in the variable as a guess--HP Solve uses only the real part. If the unknown requires units, you should not store the complex number. Use **RE**, or **[←] [2D] [DROP]**, to extract the real part, and store that in the unknown.

14.7 Finding Critical Points

Although HP Solve is designed for finding roots of expressions, you can also use it to find critical points. These are points at which the derivative of a function is zero--that is, the critical points of $f(x)$ are the roots of $df/dx = 0$. Therefore, if the current equation can be differentiated, the most straightforward method of finding its extrema is to differentiate it, make the derivative the current equation, and solve for the roots of the derivative. The roots of the derivative have the same abscissa values as the extrema of the original expression. Note that this method will also find inflection points, which are points at which the first and second derivatives are zero (at maxima, the second derivative is negative; at minima it is positive).

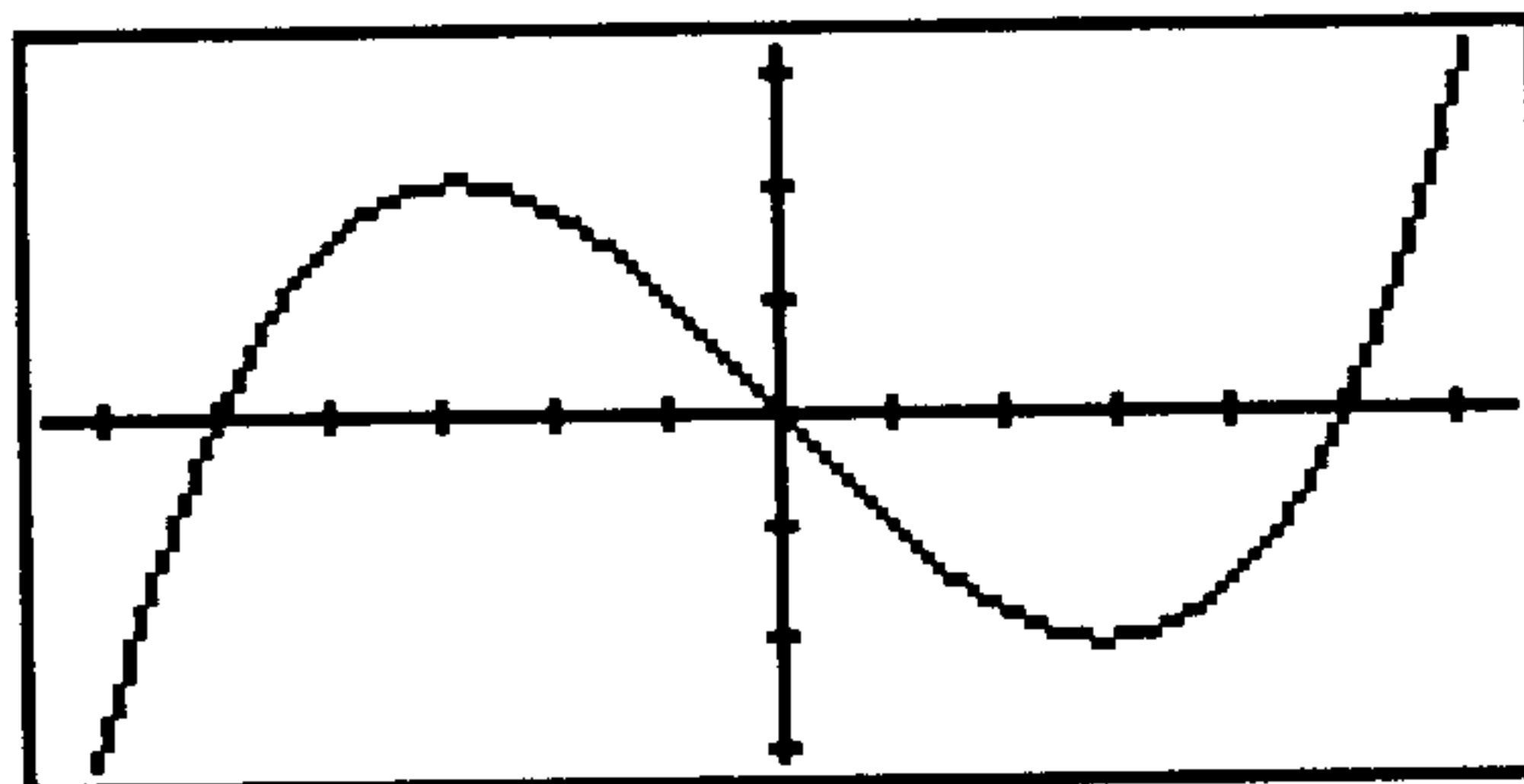
This process is automated by the **EXTR** operation, in the **FCN** sub-menu of the interactive plot menu. To illustrate the process, reactivate **CUBEX** (section 14.5.2) as the current equation, and plot it:

[←] [MODES] 3 **[≡] [FIX] [≡]**
 'CUBEX' **[←] [PLOT]** **[≡] [STEQ] [≡]**
[R→] [PLOT] **[≡] [NXT] [≡]** **[≡] [RESET] [≡]**
[←] [≡] [PREV] [≡] 'X' **[R→] [PURGE]** 'X' **[≡] [INDEP] [≡]** **[≡] [ERASE] [≡]** **[≡] [DRAW] [≡]** :



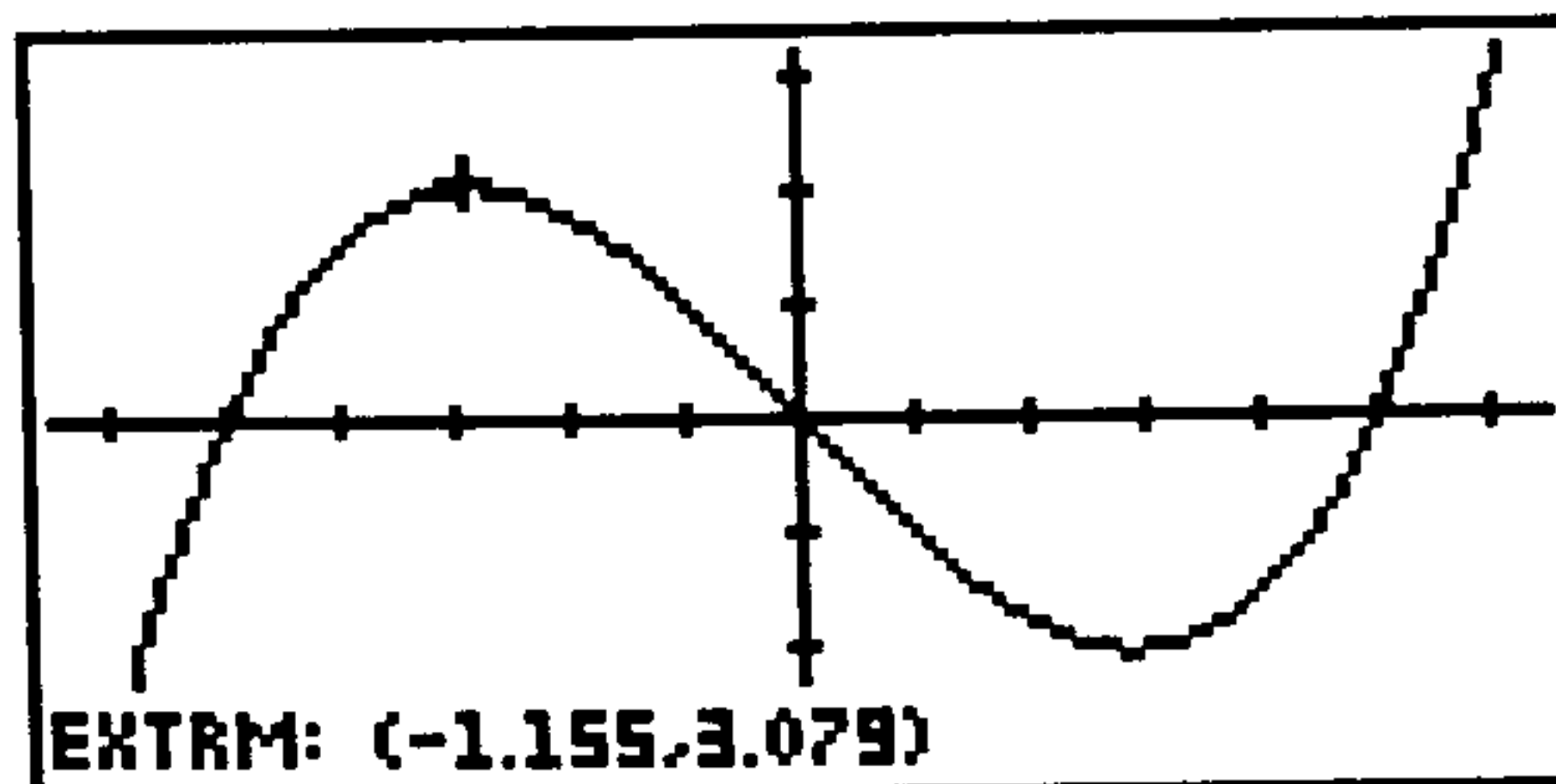
You can make better use of the plot screen by using the zoom operation:

ZOOM X .4 ENTER
ZOOM Y 1.5 ENTER :



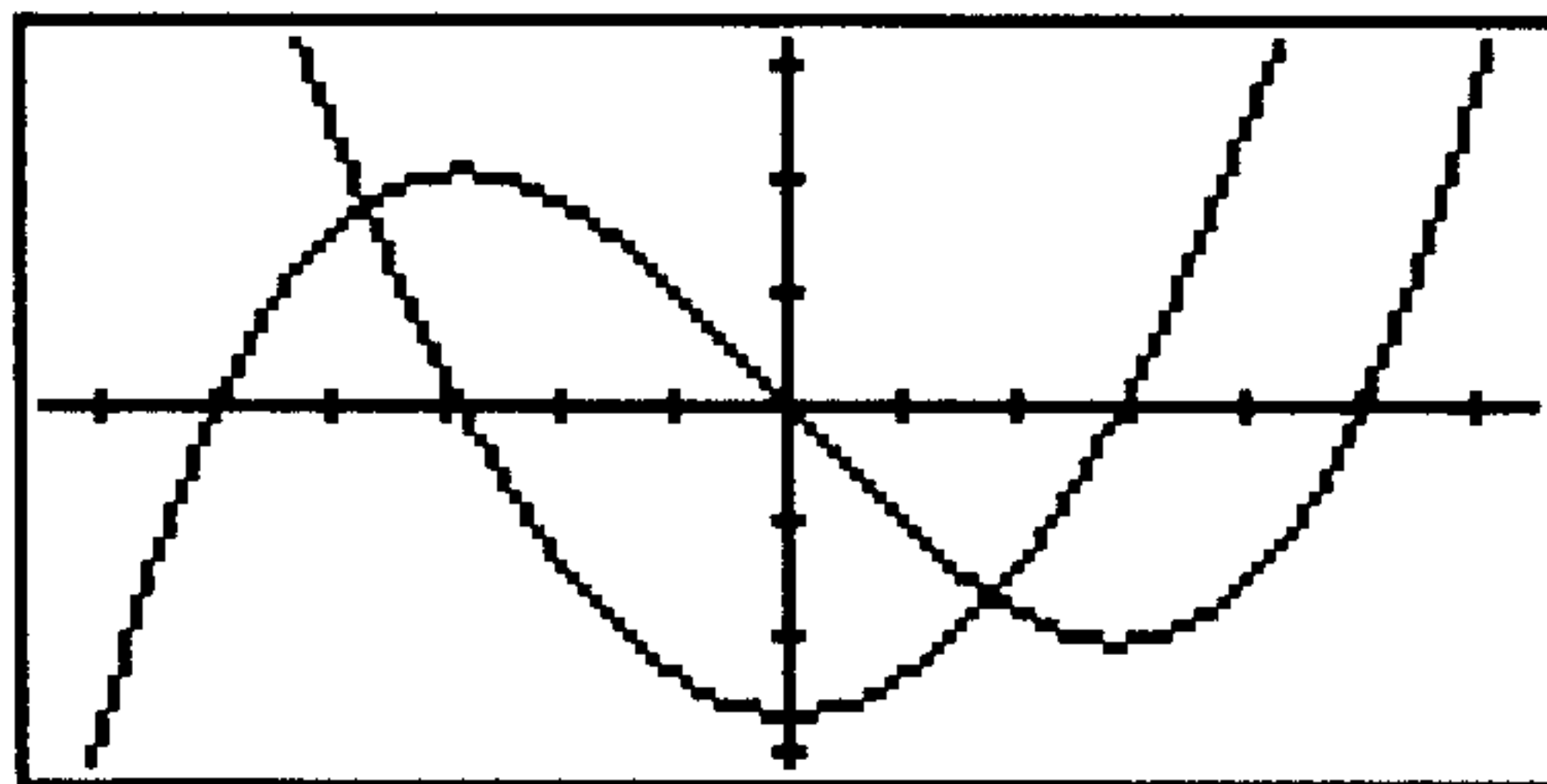
(You don't have to wait for the first zoom to finish replotting before starting the second. You can press ATTN immediately after the first ENTER, then ZOOM again to enter the parameter for the second zoom.)

To determine the coordinates of the relative maximum, move the cursor to the left to near the x -position of the maximum, then press FCN EXTR :



Notice that the cursor moves to the maximum point; this will always happen unless the maximum is not visible. In that case, (OFF SCREEN) is displayed along with the qualifying message.

It is instructive to combine the plot of the current equation with that of its derivative. When the current equation is an expression or an equation, this is easy to achieve by using $\overline{\overline{F'}}$ (in the second page of the FCN sub-menu):



The first curve drawn is the derivative, which for this example is a quadratic. The second curve is the original (cubic) current equation. Notice that the extrema of the cubic occur at the same values of x as the zeros of the quadratic.

When the current equation is not differentiable for any reason, you can still use HP Solve to find its extrema. Since the root-finder searches for a minimum of absolute value, it will stop when it encounters a local minimum that is *above* the horizontal axis, since movement in either direction produces an increase in the current equation value. A similar effect occurs for a local maximum that is *below* the axis. In either case, the value returned by HP Solve or ROOT reflects the position of an extremum rather than a root (HP Solve shows the qualifying message Extremum).

In the current example, the maximum is above the axis, and the minimum is below, so the root-finder will not find either extremum. However, you can still find the extrema without differentiating by adding a constant to the original equation such that a maximum is pushed below the axis, or a minimum is pushed above. For example, to find the maximum near $X = -1.15$, add -4 to the equation, then solve:

```
CUBEX 4 - <[SOLVE] <[NEW] CUBEX2 <[ENTER]
<[SOLVR] -1.15 <[X] <[X] <[X] X: -1.15469997367
```


14.8 Using ISOL with HP Solve

Since $\equiv \text{EXPR} = \equiv$ is faster than numerical solving, it is often advantageous to attempt to solve the current equation symbolically as a preliminary to using HP Solve. This is particularly true in cases where you only intend to solve for one variable in the current equation. By using ISOL (section 16.4.1) once to solve the equation, you can reduce the problem from a root-finding task to one of straightforward evaluation using $\equiv \text{EXPR} = \equiv$.

Consider, for example, the simple travel cost problem:

$$\text{'COST=DIST*PPG/MPG'}$$

where DIST is the distance traveled in miles, PPG is the price per gallon of gasoline, and MPG is the number of miles per gallon. Imagine that you want to construct a table of distance as a function of cost. If you enter the above equation as the current equation, you have to solve numerically for DIST after every new entry for COST. To save time, you can instead solve the equation for DIST symbolically. First, purge the variables, to ensure a fully symbolic solution:

{ COST DIST PPG MPG } PURGE.

Then solve for DIST:

'COST=DIST*PPG/MPG' 'DIST' ISOL $\rightarrow$ 'DIST=COST*MPG/PPG'

If you store the result as the current equation, you can then use the solver menu to enter the constant values for PPG, and MPG, and then successive values for COST. After each COST entry, pressing $\equiv \text{EXPR} = \equiv$ returns LEFT: DIST, and RIGHT: *value*, where *value* is the distance computed for the latest COST value.

When you use ISOL as a preliminary to HP Solve, keep in mind that ISOL works only at the "top level" of an expression or equation. It does not evaluate any of the names that appear within the object, so that implied references to the unknown variable are not made explicit. For example, consider the equation 'X+Y=Z', where Z has the value 'X+Y'. Solving the equation for X using ISOL, you obtain the result 'X=Z-Y'. But this result evaluates to 'X=X-Y+Y', showing that the "solution" is meaningless. To guard against this type of problem, it is wise to use SHOW (section 16.4.2) before applying ISOL to make explicit all references to a specified name. In the example, 'X+Y=Z' 'X' SHOW returns 'X+Y=X+Y', making it obvious that the equation is trivial. SHOW is preferred over EVAL for this purpose because a) you only need to execute it once, whereas EVAL may have to be used repeatedly; and b) SHOW only evaluates names that reference the argument name at some level, keeping the result expression as compact as

possible.

HP Solve on some other HP calculators automatically attempts an ISOL-like solution whenever you solve for a variable. In many cases this can speed up the solution by avoiding the use of the iterative root-finder, and in some cases may find a solution where the root-finder can not. This is not done on the HP 48 for several reasons:

- HP Solve is oriented toward finding a single solution, selected by you by means of an initial guess. A symbolic solution would have to be a principal value (section 16.5), which might prevent you from obtaining the particular root you want. Or it could be a general solution with arbitrary signs and integers, which would be inconvenient for repeated solving.
- A symbolic solution is only possible in certain problems, but the calculator would have to attempt it every time you solved, which could waste a lot of time. It would have to execute SHOW as well as ISOL, to insure that all references to the unknown variable are made explicit before solving.
- Since ISOL is an available command, you can choose whether to use it or HP Solve in a particular situation, rather than have the HP 48 impose its choice on you.

14.9 Programmable Solving

The root-finder used by HP Solve is available for program use as the command ROOT. This command requires as arguments the same elements that are needed for an HP Solve exercise:

- A program or an algebraic object (level 3) that acts as the current equation. The variable EQ is not used by ROOT.
- The name (level 2) of the unknown variable.
- A number or list of one, two, or three numbers (level 1), that acts as an initial guess for the root-finder.

All of the same considerations of the number and nature of the initial guesses discussed in section 14.5 apply to program root-finding as well. Of course, since a program can't make visual inferences from a plot, it must include specific guesses, or logic to determine appropriate guesses.

ROOT returns its result for the unknown variable to the stack, and stores it in the variable. You can not monitor the root-finder during execution of ROOT as you can during an HP Solve exercise.

■ *Example.* Find the positive roots of $\cos nx = e^x$, for $n = 1, 2$, and 3 .

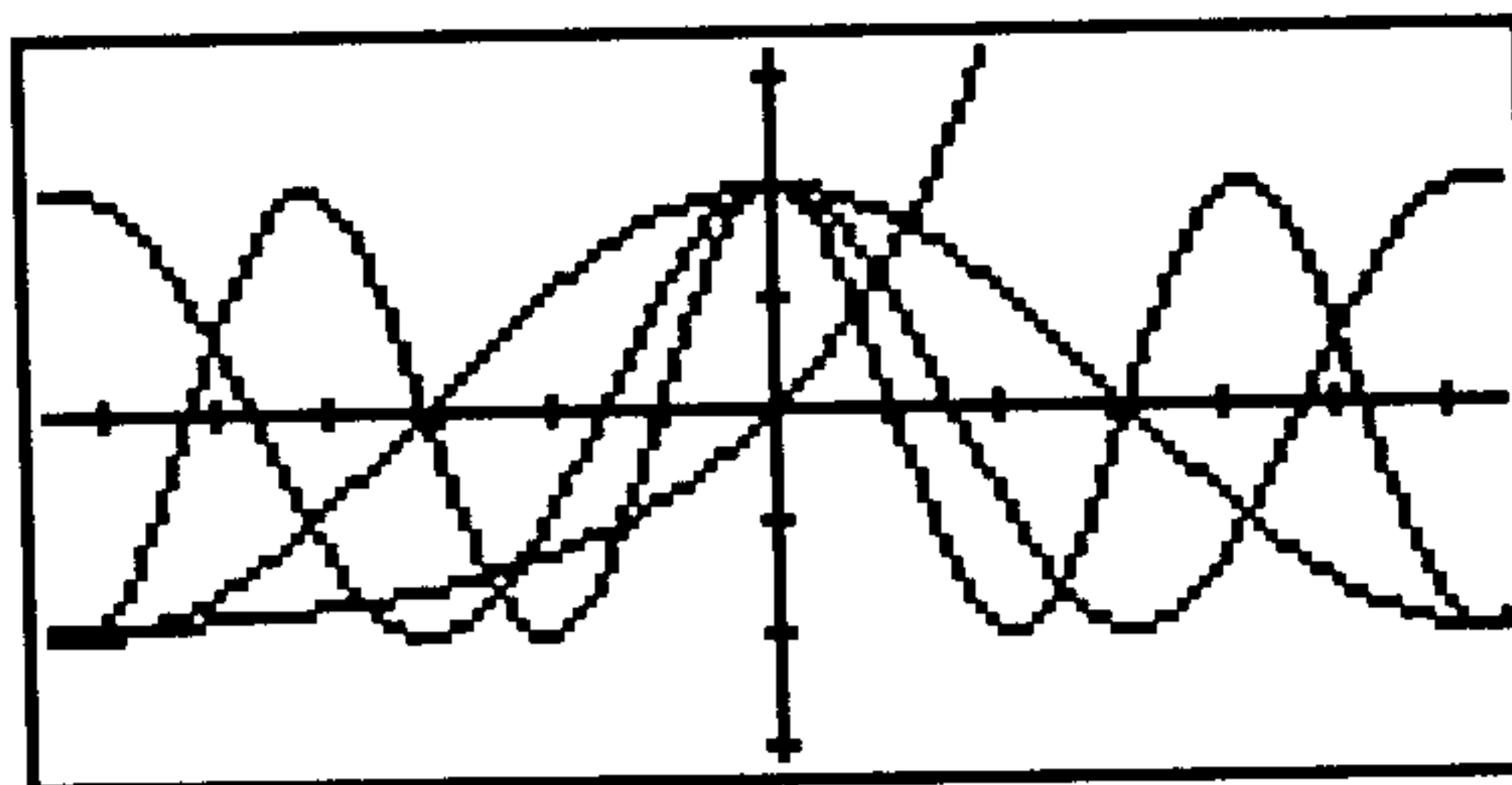
■ *Solution.* Executing the program below returns these results:

0.601346767726 0.466819325092 0.369256413055

<pre> << RCLF RAD → f << 1 3 FOR n 'COS(n*X)=EXP(X)-1' 'X' { 0 2 } ROOT NEXT f STOF >> >> </pre>	Set radians mode. For n=1,2,3: Equation to solve. Unknown variable. Bracket the positive root. Find the root. Restore trigonometric mode.

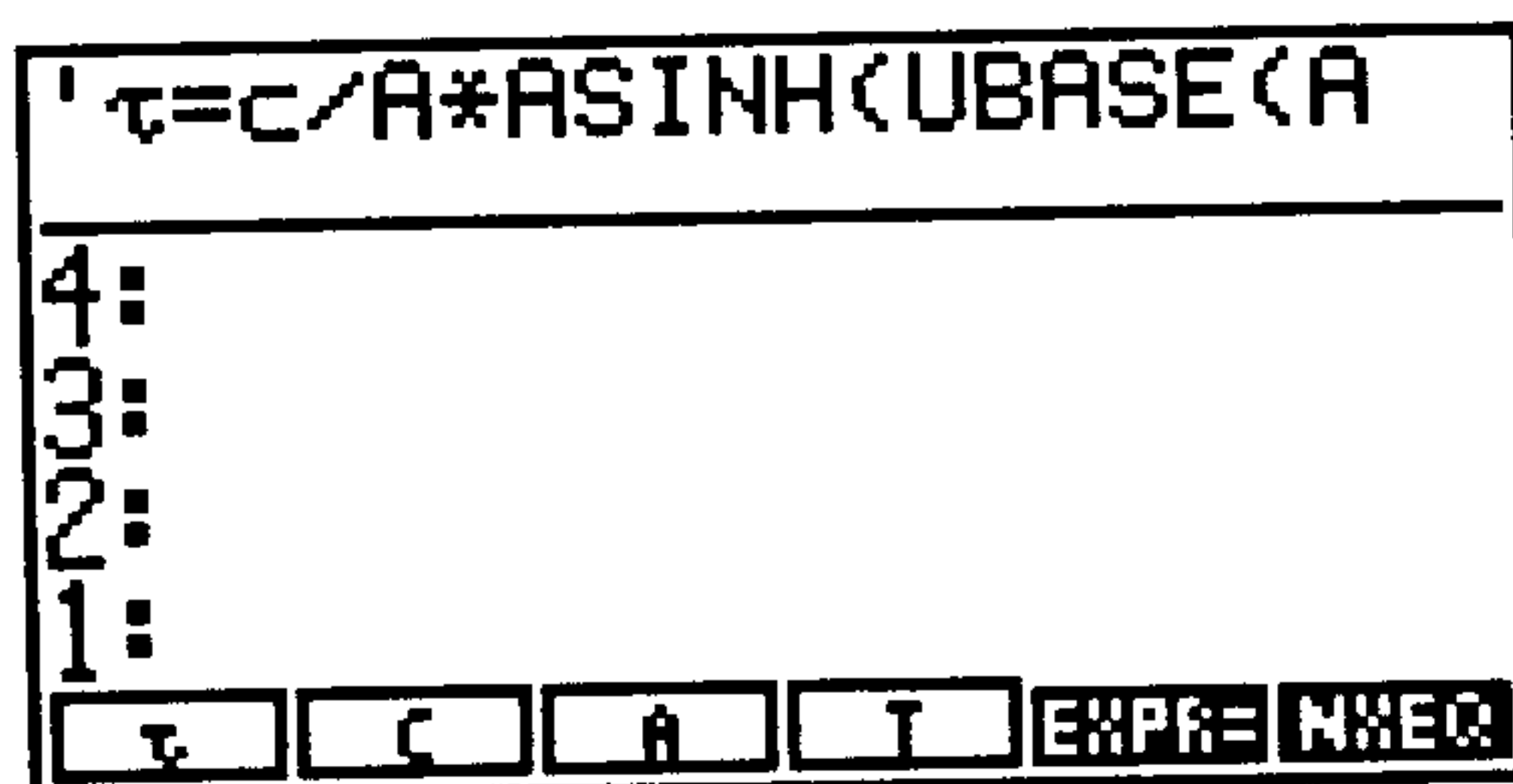
The HP 48 plot below shows the exponential curve intersecting the three cosine curves. The current equation for the plot is the list (see section 15.2.2)

{ 'COS(X)' 'COS(2*X)' 'COS(3*X)' 'EXP(X)-1' }.



14.10 Secondary Results

HP Solve is designed primarily to solve one equation for one unknown. However, the system does allow you to work with more than one equation, helping you to find a value for an unknown in one equation and use it as a known variable in the next (this process is automated further by the multiple equation solver in the HP 82211A Solve Equation Library Application Card). In place of the single object stored as the current equation, you may use a list of objects--any mixture of programs, algebraic objects, and the names of variables containing programs or algebraic objects.



You can use the value of T as a first guess for τ :

$\boxed{\rightarrow} \boxed{T} \boxed{=} \boxed{\tau} \boxed{\leftarrow} \boxed{T} \boxed{=}$ $\tau: 1.72948030729\_yr.$

The astronauts age four times this amount, or 6.91 years, during the complete round trip. Now you can determine the half-way speed, using c as an initial guess:

$\boxed{=NXEQ} \boxed{\rightarrow} \boxed{c} \boxed{=} \boxed{v} \boxed{\leftarrow} \boxed{v} \boxed{=}$ $V: .945261928211\_c.$

When EQ contains a list, the solver menu is built from the first object in the list. Pressing $\boxed{=NXEQ}$ changes the stored list by rotating its first object to the end of the list, so that the second object moves to the head of the list and is then used to build the new menu. If you look now at the SPACE variable, you will see that the equation for V is the first object in the list. To restore the list to its original order, press $\boxed{=NXEQ}$ again.

The equation catalog operation $\boxed{=EQ+}$ provides a convenient means for making a list of objects for HP Solve. See section 14.2.3.2.

14.11 Modifying the Solver Menu

One of the advantages of HP Solve is that it automatically analyzes the current equation and builds the solver menu for you. This can, however, sometimes lead to a less-than-ideal menu. Accordingly, the HP 48 gives you the option of defining the menu in a manner similar to ordinary custom menu construction (section 7.3).

Any object in EQ, or any entry in a list of objects in EQ, can be replaced by a list containing the original object plus a custom menu list:

$\{ \text{object} \{ \text{custom-menu} \} \}$

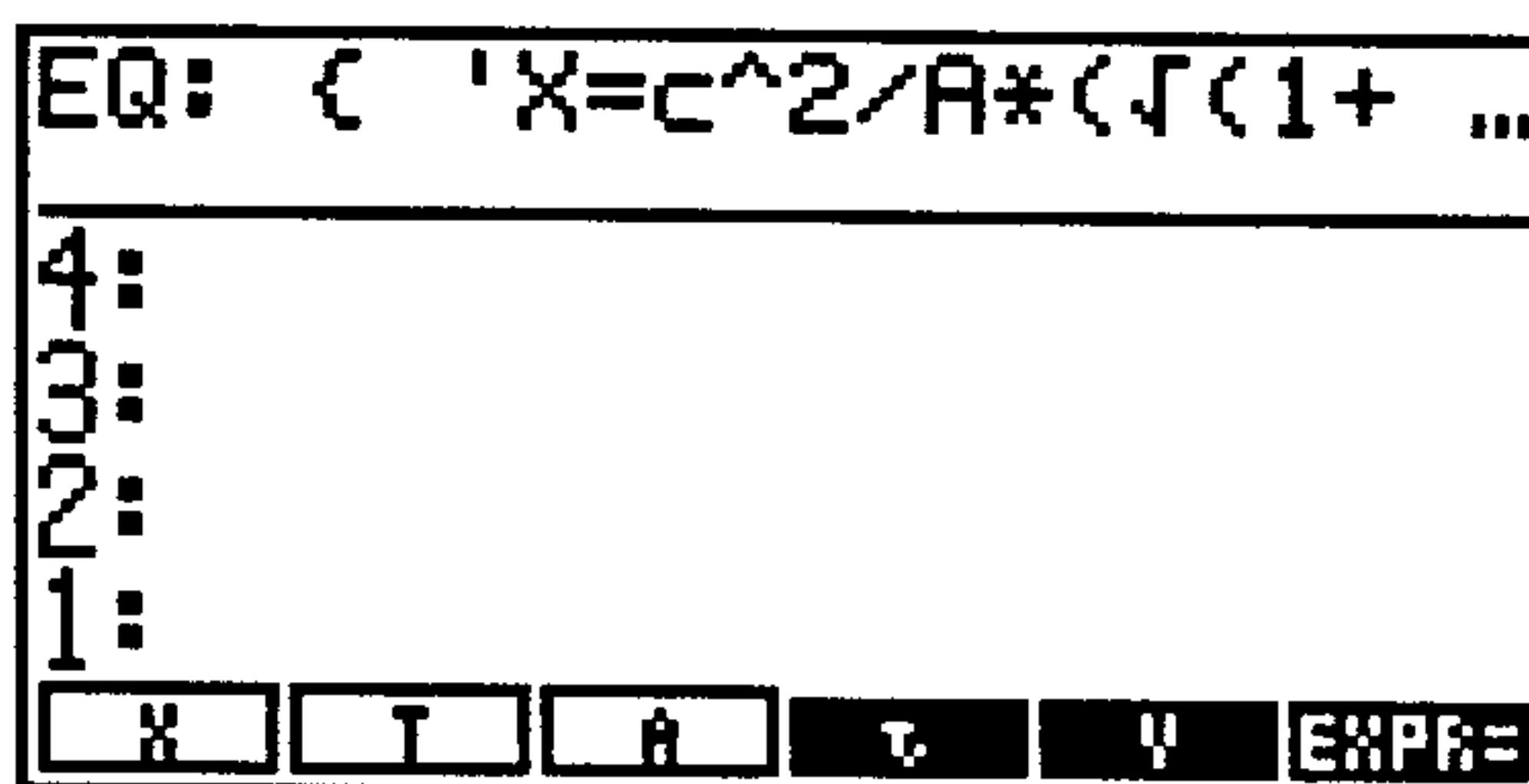
The *object* is used as the source for the root-finder as usual, but the normal solver menu is replaced by a custom menu defined by the inner list. The menu can have any of the features of a custom menu, with two modifications:

- Menu key labels corresponding to name objects in the *custom-menu* list appear as black-on-white labels, and the corresponding key definitions are the usual solver menu actions.
- **EXPR=** appears at the end of the menu.

In section 14.10 we showed how to use HP Solve with a list of related equations. In the example there, we computed the proper time τ and the speed v from separate equations included in the current equation list. Since those computations required only evaluation rather than root-finding, another approach is to add menu keys for evaluating τ and v to the original menu. For example, consider the following list:

$\{ 'X=c^2/A*(\sqrt{1+(A*T/c)^2}-1)'$	<i>Original equation.</i>
$\{$	<i>Start of custom menu list.</i>
X	X key.
T	T key.
A	A key.
{ "τ"	
<< 'c/A*ASINH(UBASE(A*T/c))' →NUM	
'1_yr' CONVERT	
>>	
}	τ key.
{ "v"	
<< 'c*TANH(UBASE(A*τ/c))' →NUM	
>>	
}	v key.
}	<i>End of custom menu list.</i>
}	

With this list as the current equation, the solver menu looks like this:



Here you can observe several things:

- The custom menu list lets you place the menu keys in the order in which they are likely to be used, rather than the order defined by their appearance in the original equation.
- No key is included for c , the speed of light. This quantity is a constant, so there is little point in including the corresponding variable name in the menu. (Of course, you must store a value for c initially, using STO.)
- The first three entries in the menu list are the names X , A , and T ; these define ordinary solver menu key actions.
- The next two menu list entries are lists themselves. In each case, the first element is a string that is used for the menu key label. If the element were a name, the label would appear as dark letters on a white background. Using a string instead yields a label with white letters on dark, indicating that the menu key is not an ordinary solver menu key.

The second element in each individual key list specifies the unshifted menu key action. For τ and V , the actions are programs that evaluate to the values of these variables derived from previously stored or calculated variables. You can also specify left- and right-shifted menu key actions by adding additional objects to the key list (section 7.3.3).

- **EXPR=** appears at the end of the menu. Pressing this key evaluates the main equation, as described in section 14.4.

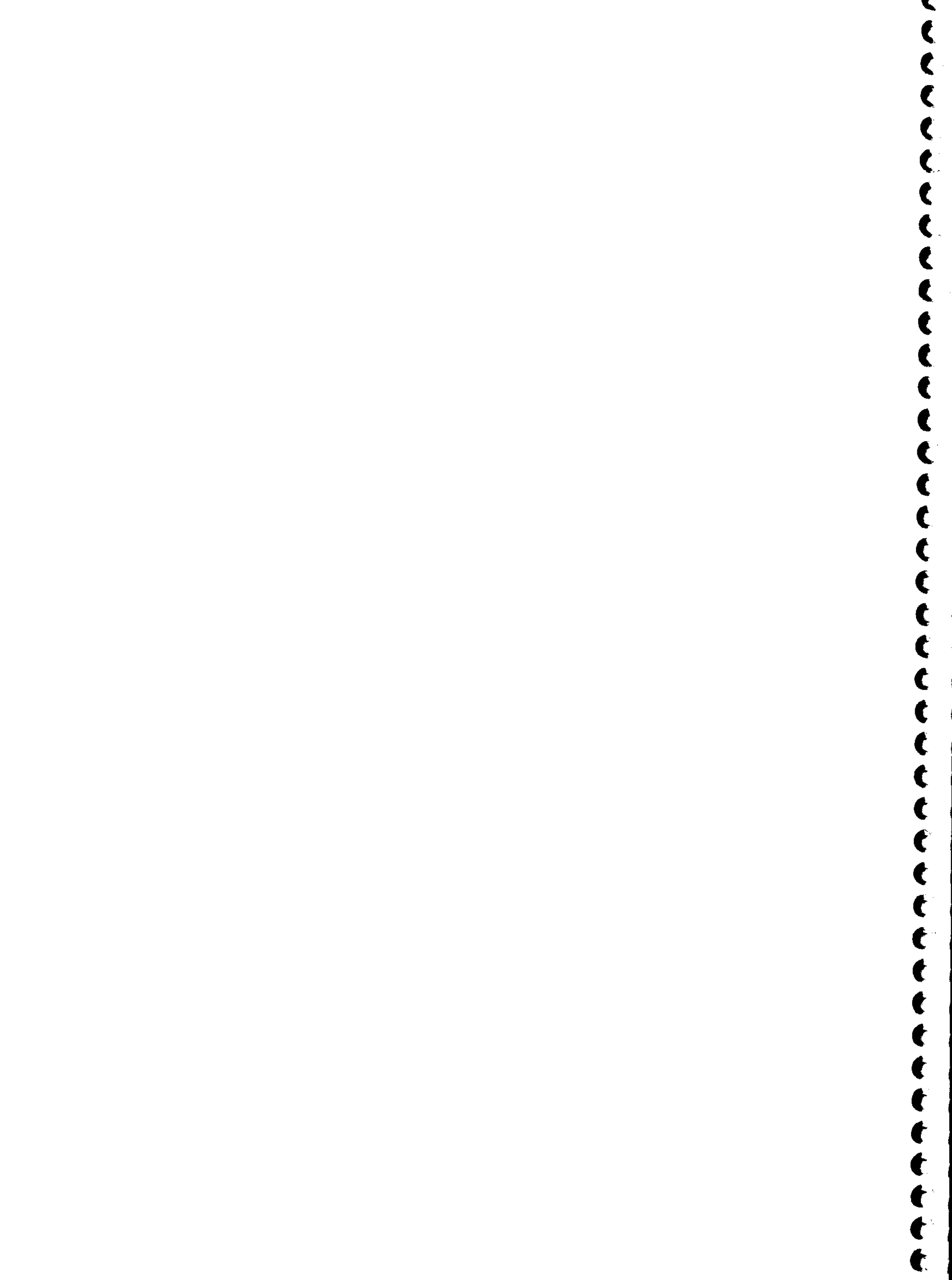
If necessary, re-enter the values from the original problem:

2_lyr $\equiv X$ 5_yr $\equiv T$

then

$\leftarrow$ $\equiv A$	$\rightarrow$ A: .184521099199_ga
τ	$\rightarrow$ 4.44831376704_yr
V	$\rightarrow$.318993857812_c.

As a final note to this example, we should mention that while the *kinematics* of the problem--speed, time, and distance--look attractive, the *dynamics*--the fuel required, radiation shielding, etc.--make such a trip effectively impossible.



15. Plotting

Much of the current interest in the use of calculators and computers in mathematics education centers around plotting. A graph of a function or of data points provides a view that is often more revealing than a symbolic expression or a table of data. Accordingly, the HP 48 provides an extensive set of automated plotting facilities that make it easy to create and work with plots. In Chapter 10 of Part I, we studied general purpose HP 48 graphics and display control, with an emphasis on the creation of special displays for program input and output. This chapter concentrates on mathematically oriented plotting, specifically the capabilities associated with the DRAW command and the plot environment.

A note on terminology: following common usage, we will use x and y as generic names for the variables associated with the horizontal and vertical directions on the graph screen. The HP 48 similarly uses X and Y as default names. However, you are by no means restricted to those choices--you can substitute any other names.

15.1 The Plot Menus

The commands and operations associated with automatic plotting are found in two menus, activated by the $\leftarrow$ [PLOT] and $\rightarrow$ [PLOT] keys. $\leftarrow$ [PLOT] activates the plot equation entry menu, which is a near copy of the $\leftarrow$ [SOLVE] menu. The NEW, EDEQ, STEQ, and CAT operations are the same for plotting and solving, and both systems share the same current equation identified by the variable EQ. The uses of these operations are described in section 14.1.

After pressing $\leftarrow$ [PLOT], you will see a display like this:

```
Plot type: POLAR
EQ: -2*COS(4*8)+SIN(...
4:
3:
2:
1:
PLOT: PTYPE NEW EDEQ STEQ CAT
```

The current equation display in the second line is the same as that described in section 14.2. The top line shows the current plot type. If the type is one of the three discrete plot types, the current equation display is replaced by a current matrix display:



The matrix is represented by its dimensions in the form $[m \times n]$, where m is the number of rows and n the number of columns.

The plot menu replaces the solve menu's ROOT key with PTYPE. This key activates a menu that contains eight commands for selecting the eight automatic plot types. Pressing any of the PTYPE menu keys changes the current plot type to that indicated on the key label, and changes the menu back to the plot menu. The plot types are:

<i>Function</i>	A curve $y(x)$ is plotted by computing the vertical coordinates y for each value of the horizontal coordinate x , as x is varied from the left side of the screen to the right.
<i>Conic</i>	Both branches of the solution for y of a second-order equation in x and y are plotted in the manner of a function plot, producing a conic section figure.
<i>Polar</i>	Points (r, θ) specified in polar coordinates are plotted, as the polar angle θ is varied over a specified range.
<i>Parametric</i>	A curve $(x(t), y(t))$ is plotted as a parameter t is varied over a specified range.
<i>Truth</i>	A pixel is turned on at every point (x, y) for which the evaluation of a function of x and y returns <i>true</i> .
<i>Bar</i>	A bar chart is plotted that represents the values in one column of the current statistics matrix.
<i>Histogram</i>	A histogram is generated from a column of data values sorted into equal-width bins.
<i>Scatter</i>	Scattered points are plotted corresponding to coordinates specified by two columns of the current statistics matrix.

The first five plot types are derived from the continuous variation of a particular

variable; we will refer to these collectively as the *continuous* plot types. We will call the last three *discrete* plot types, because they use discrete individual data points.

The first key in the equation entry menu is **≡PLOT≡** (for “plotter”). This key activates the plot execution menu, which is also generally available with **▢PLOT**. This menu contains the actual plotting commands, ERASE, DRAW, and AUTO, plus all of the commands related to the various plot parameters--plot scale, axes, labels, etc. The relationship between the **◀PLOT** and **▢PLOT** menus is similar to that of the **◀SOLVE** and **▢SOLVE** menus. You use the **◀** menu to set up the current equation (and the plot type, for plotting), then activate the **▢** menu to do the actual plotting.

The plot execution menu has an automatic prompting display that shows several of the current plotting parameters, e.g.

```

Plot type: POLAR
EQ: '-2*COS(4*θ)+SIN(...
Indep: 'θ'
x:      -6.5      6.5
y:      -1.55     1.6
ERASE DRAW AUTO XRNG YRNG INDEP

```

- The top display row shows the current plot type.
- The second row shows the name and partial contents of the current equation object (for continuous plots), or the name and dimensions of the current statistics matrix (for discrete plots).
- For continuous plot types, the next row shows the name of the independent variable. For conic and truth plots, the dependent variable name is also listed. For discrete plot types, the *x*- and *y*-columns (section 15.9) and the current pseudo-linear fit model (section 20.3.2.1) are displayed on a single line.
- The current horizontal and vertical plot ranges are shown, in the format

```

X:  xmin  xmax
Y:  ymin  ymax

```

The menu commands and other operations are described in the rest of this chapter.

15.2 Plotting Essentials

When you set out to make a plot of any type on the HP 48, you must specify a number of elements that determine the nature and appearance of the graph. Obviously, you must provide either a set of coordinates for points to be plotted, or a mathematical expression from which the coordinates may be computed. But, in addition, you must determine:

1. The type of plot--the manner in which the coordinates or the expression is to be interpreted. For example, an expression might represent y as a function of x , or it might specify a radial coordinate r as a function of a polar angle θ .
2. Which variables in the expression correspond to the two coordinate axes, and the labels, if any, that are to be applied to the axes.
3. The range of an independent variable, or the portion of a discrete data set, that is to be used in a plot.
4. The plot scale, i.e. the mapping of the logical coordinates associated with a graph onto the physical display.
5. Whether axes are to be drawn, and if so, where.
6. The plot *resolution*, i.e. the frequency with which an expression is sampled to determine plotted points. Also, whether the calculator should "connect the dots" by drawing straight line segments between computed points.

Making a graph of $\sin x$ is a simple example that illustrates the elementary HP 48 plotting process. Step 1 is to enter the expression, and store it in the variable EQ:

'SIN(X)' $\boxed{\rightarrow}$ $\boxed{\text{PLOT}}$ $\boxed{\text{STEQ}}$

As shown in the example in section 14.6, the plotting command DRAW uses EQ in the same manner as HP Solve, to select an object for plotting. Since you are plotting a trigonometric function, the result will be sensitive to the current angle mode, which should be *radians* in this case (press $\boxed{\leftarrow}$ $\boxed{\text{RAD}}$ if necessary). Next, make sure that the HP 48 will produce an ordinary function graph (step 2):

$\boxed{\text{PTYPE}}$ $\boxed{\text{FUNC}}$

At this point, you should see a display like this:


```

Plot type: FUNCTION
EQ: 'SIN(X)'
4::
3::
2::
1::
PLOT PTYPE NEW EDEQ STEQ CAT

```

Now enter the following:

PLOT **NXT** **RESET** **←** **PREV**

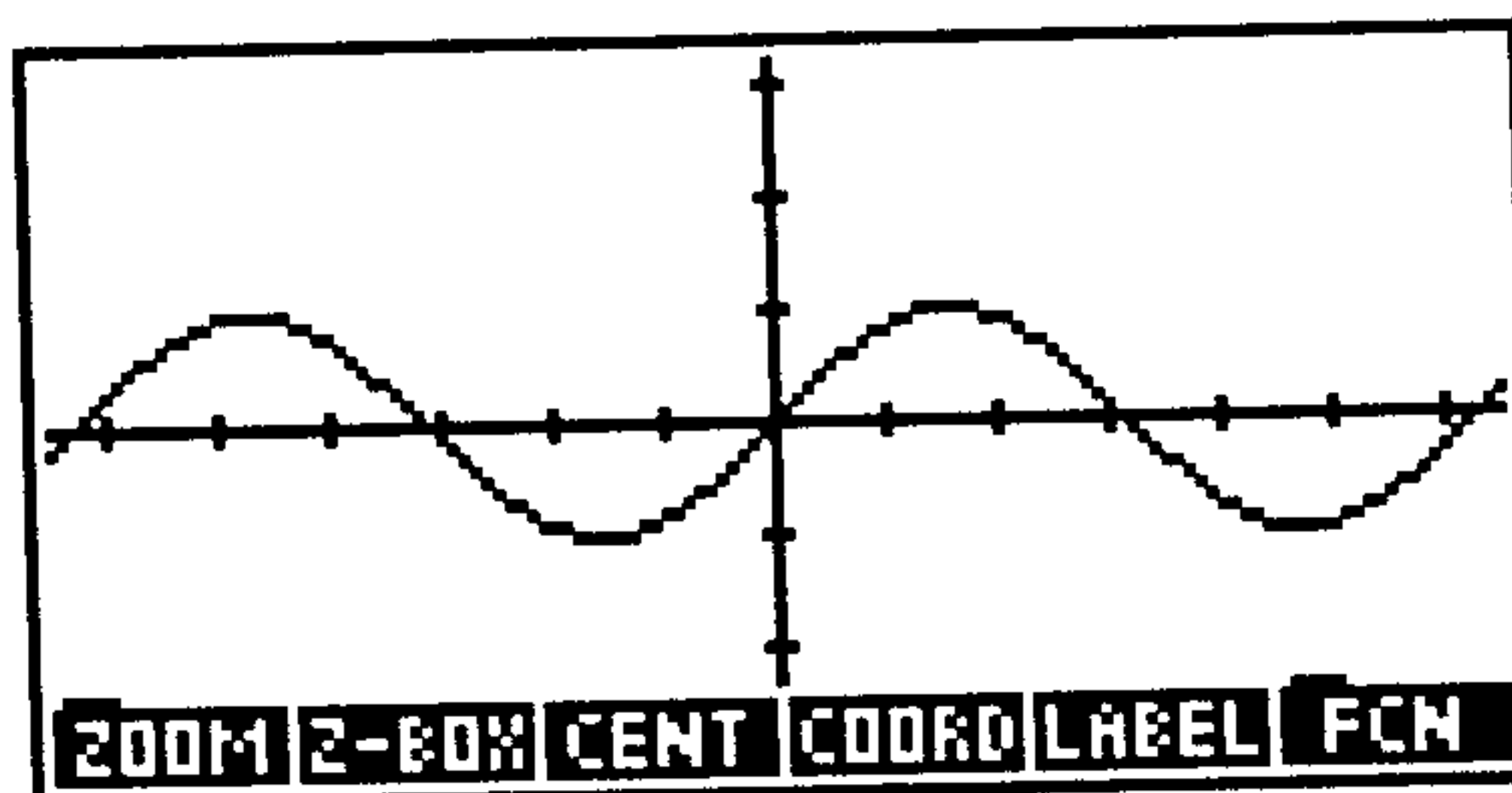
```

Plot type: FUNCTION
EQ: 'SIN(X)'
Indep: 'X'
x:      -6.5      6.5
y:      -3.1      3.2
ERASE DRAW AUTO XANG YANG INDEP

```

RESET takes care of steps 3-6 by supplying default values for all of the remaining parameters.

Now you are ready to draw the graph. Press **ERASE** **DRAW**:



After drawing the graph, the HP 48 automatically activates the *plot environment* (see section 15.3), where you can rescale the graph (zoom), read data from the graph, and for function plots, compute intercepts, slopes, and areas. To return to the standard

environment, press **ATTN** .

In the next sections, we will study in more detail each of the graphing steps illustrated in the preceding example. We will first use ordinary function graphs to illustrate general methods, then consider other types of graphs.

15.2.1 DRAW

The center of the automatic plotting system is the DRAW command, which translates an expression or statistical data into a graph. Like HP Solve, DRAW uses an implicit argument, which may be either the object stored in the variable EQ, or the matrix stored in Σ DAT, according to the current plot type. However, DRAW also requires a number of other implicit arguments:

- The variable PPAR contains a list of parameters that determine the plot type, scale, labels, axes, and variables.
- Flag -30 determines whether DRAW plots one curve (clear) or two (set) for function and polar plots with a current equation of the form $y = f(x)$. See section 15.4.
- Flag -31 determines whether DRAW connects successive computed points with straight line segments (flag clear), or plots unconnected points (set). The **CNCT** key in the second page of the **MODES** menu toggles this flag: when the white box appears in the menu label, the flag is clear and DRAW connects points.

(The program DRAWPIX in section 10.3.5 emulates DRAW for function plots; studying the program can help you understand the detailed behavior of DRAW.)

The **DRAW** key (**PLOT** menu) is designed for interactive plotting, and so executes three commands in sequence: DRAX to draw axes, DRAW to plot the specified data, and finally GRAPH to activate the plot environment. In a program, you can use the three commands separately or together. The **DRAW** key also provides access to the current equation: **DRAW** executes STEQ, and **DRAW** executes RCEQ.

Note that DRAW does not erase the graph screen, which makes it easy to superimpose various graphs. ERASE is a separate command; as its name suggests, it blanks the graph screen, making a clean slate for subsequent plotting. You can interrupt DRAW with **ATTN** ; this terminates plotting but does not abort other program execution (e.g. **DRAW** still turns on the plot environment). This is helpful when DRAW is still executing, but has already plotted the useful part of a graph.

15.2.1.1 Autoscaling

When the default plot range parameters are inappropriate for a particular plot, and yet you do not know in advance what ranges might be better, you can use **AUTO** as a substitute for **DRAW**. **AUTO** attempts to find a vertical range such that at least a portion of the plot is visible; for some plot types, **AUTO** also adjusts the horizontal plot range. The algorithm is different for each plot type:

- **Function Plots.** **AUTO** samples the current equation at 40 points over the current x -range, discards points that return $\pm\infty$, then sets the y -range to display the maximum and minimum of the sample y values, plus the vertical origin.
- **Parametric Plots.** **AUTO** samples the current equation at 40 points over the independent variable range, then sets the horizontal and vertical ranges to include the minima and maxima in both directions, plus the origin.
- **Polar Plots.** Same as parametric.
- **Conic Section Plots.** **AUTO** only sets the vertical scale to be the same as the horizontal scale.
- **Truth Plots.** **AUTO** does not change the plot ranges, executing the same as **DRAW**.
- **Scatter Plots.** **AUTO** sets the x -range to the minimum and maximum values in the independent variable column (set by **XCOL**) in **ΣDAT**, and the y -range to the minimum and maximum values in the dependent variable column (set by **YCOL**).
- **Bar Plots.** **AUTO** sets the x -range from zero to the number of elements in **ΣDAT**, and the y -range to the minimum and maximum of the elements in the independent column.
- **Histograms.** **AUTO** sets the x -range to the minimum and maximum of **ΣDAT** elements, and the y -range from zero to the number of rows in **ΣDAT**.

In all cases except conic section plots and truth plots, the y -range is stretched in the negative direction by an additional 15% over that described above, to provide room for the plot environment menu labels. More details of autoscaling are provided in the sections below discussing the individual plot types.

15.2.2 EQ and ΣDAT

The five continuous plot types *function*, *conic*, *polar*, *parametric*, and *truth*, all use the contents of a variable **EQ** in the current directory to specify the plotting “source,” in much the same manner as **HP Solve**. Following the terminology of Chapter 14, we call the source object the *current equation*, even when it is not literally an equation (which is even less likely in general for plotting than for solving). The precise interpretation of the current equation varies according to the plot type, and is described in the

corresponding sections below. The rules for indirection are the same as for HP Solve--if EQ contains a name (or a list of names), then the actual source object(s) is that stored in the named global variable(s).

When EQ contains a list of objects, each is plotted, one after another. Each object must be suitable for the current plot type. If any of the objects is a name, then the object stored in the global variable that matches the name is plotted rather than the name.

15.2.3 PPAR

With the exception of the modes established by flags -30 and -31, all of the information required by DRAW to plot the current equation is recorded in a parameter list stored in a variable PPAR (*Plot PARameters*) in the current directory. If PPAR does not exist when DRAW (or any other command that affects the graph screen) is executed, it is created with the following default values:

$$\{ (-6.5, -3.1) \quad (6.5, 3.1) \quad X \quad 0 \quad (0,0) \quad \text{FUNCTION} \quad Y \}.$$

The commands XRNG, YRNG, INDEP, DEPN, RES, CENT, SCALE, AXES, *H, and *W change the objects in the PPAR list. If the list does not exist when one of these commands is executed, the default list is stored, then modified accordingly by the command.

The list above is a particular example of the general form of PPAR:

$$\{ (x_{\min}, y_{\min}) \quad (x_{\max}, y_{\max}) \quad indep \quad res \quad (x_{\text{axes}}, y_{\text{axes}}) \quad ptype \quad depn \},$$

The entries are as follows:

$(x_{\min}, y_{\min})$ is a complex number representing the logical coordinates (section 15.2.4) of the lower-leftmost pixel on the graph screen.

$(x_{\max}, y_{\max})$ specifies the logical coordinates of the upper-rightmost pixel;

indep is the name of the independent variable. *indep* may also be a list $\{ name \quad t_{\min} \quad t_{\max} \}$, where *name* is the name of the independent variable, and $t_{\min}$ and $t_{\max}$ are the range of that variable over which the graph is to be plotted.

res is the plot *resolution*--the increment between successive values of the independent variable.

$(x_{\text{axes}}, y_{\text{axes}})$ specifies the coordinates of the intersection of the axes.

ptype is the plot *type*, actually one of the eight plot type selection commands.

depn is the name of the dependent variable used by conic, parametric, and truth plots. For truth plots, *depn* may also be a list $\{ name \quad y_{\min} \quad y_{\max} \}$, where *name* is the name of the dependent variable, and $y_{\min}$ and $y_{\max}$ are the range of that variable over which a graph is plotted.

The default values supplied for PPAR select the *function* plot type, and logical coordinates with uniform scales of 1 per 10 pixels in both directions, with the origin and axes at the center of the screen. X and Y are chosen as the independent and dependent variables, respectively. The value of 0 for the resolution (section 15.2.7) is a default for that parameter, which indicates that one point is to be plotted for every pixel column. You can restore PPAR to its default values (except for the plot type) by pressing **RESET** in the second page of the plot execution menu.

The variable Σ DAT is the analog of EQ for the discrete plot types scatter, bar, and histogram. This variable may contain a matrix, or the name of a matrix. DRAW uses some or all of the data in that matrix to create its plot. Because superimposing several discrete plots is not very useful, Σ DAT may only contain a single matrix or name.

15.2.3.1 Saving A Graph

The **GRAPH** key (section 10.1) makes it easy to switch between the plot environment and the standard environment, when you want to perform various calculator operations and then view a graph again. However, if you clear the graph screen or plot another graph, you must redraw the original graph if you want to view it again. One alternative is to save the picture with a sequence like PICT RCL *name* STO (section 10.3.3). However, this method only saves the picture itself, without recording any of the contextual information such as the plot parameters or the current equation. The programs listed below provide a mechanism for the complete recording and restoration of a graph, including the picture, the current equation, all of the plot parameters, the plotting flags -30 and -31, and the trigonometric angle mode flags -17 and -18. The latter are included because trigonometric functions are common in plotting expressions.

PREST Plot Restore 3DC4	
level 1	level 1
'name'	
<pre><< RCL OBJ- DROP IF THEN STOΣ ELSE STEQ END RCLF 64 STWS 1 GET #FFFFFFF9FFCFFF AND + STOF 'PPAR' STO PICT STO >></pre>	Recall the stored list. Extract all of the objects. Restore the current matrix or equation. Set full wordsize. Restore plot flags. Restore plot parameters. Restore picture.

PSAVE Plot Save D720	
<i>level 1</i>	<i>level 1</i>
<i>'name'</i>	<i>CR</i>
<pre> << PICT RCL PPAR RCLF DUP 64 STWS 1 GET #60030000h AND SWAP STOF { SCATTER HISTOGRAM BAR } 3 PICK 6 GET IF POS THEN RCLΣ 1 ELSE RCEQ 0 END 5 →LIST SWAP STO >> </pre>	Recall the current picture. Recall PPAR. Recall current flags, set full wordsize. Record the plot flags. Restore wordsize. Get the plot type. Get the current matrix... or current equation. Combine into a list. Save in a variable.

PSAVE takes a name as its argument, and saves the current picture and its associated parameters in a variable with that name. PREST also takes a name, and restores the plot state that is stored with that name.

15.2.4 The Plot Scale

When dealing with mathematical plots, it is most convenient to describe positions on the graph screen in terms of *logical coordinates* rather than pixel numbers. As described in section 10.3.4, logical coordinates are derived from a coordinate system imposed on the graph screen, to reflect the ranges of values associated with the variables that correspond to the horizontal and vertical directions. The translation between pixel numbers and logical coordinates is determined by the first two entries in the PPAR list, $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$. These specify the logical coordinates of the lower-leftmost pixel and the upper-rightmost pixel, respectively.

There are several ways to set or change the coordinate ranges. You can, of course, edit or otherwise modify PPAR directly, but this is usually less convenient than using specialized commands.

- PMIN takes a complex number and stores it as the new first element in PPAR, i.e. as $(x_{\min}, y_{\min})$. PMAX similarly stores a new $(x_{\max}, y_{\max})$ replacing the second element in PPAR. With these two commands, you can completely specify the logical coordinate system. However, the commands listed next are usually easier to use, and PMIN and

PMAX are provided primarily for compatibility with the HP 28S (they are not available in any menu).

- When you are preparing to make a graph, the most natural way to establish the coordinate system is to enter $x_{\min}$ and $x_{\max}$ together for the horizontal range, and then $y_{\min}$ and $y_{\max}$ together for the vertical range. XRNG and YRNG perform these tasks; each takes two real numbers from the stack, in the order *minimum maximum*, and stores them as the new real and imaginary parts of the first two PPAR elements. If you enter the parameters manually with the XRNG and YRNG keys, you get immediate confirmation since the plot menu display shows the new ranges that you just entered. You can also recall the current ranges using those keys:

<u>▢</u> <u>XRNG</u>	<u>▢</u> $x_{\min}$ $x_{\max}$
<u>▢</u> <u>YRNG</u>	<u>▢</u> $y_{\min}$ $y_{\max}$

- You can specify the coordinates of the center pixel of the graph screen with CENTR, and the plotting scale using SCALE. Using these two commands fixes the plot ranges as effectively as using XRNG and YRNG. CENTR takes a complex number representing logical coordinates from the stack, and readjusts the plot parameters so that the “center” pixel will have those coordinates. The center pixel on the default graph screen has the pixel coordinates { #65d #32d }.

SCALE takes two real numbers as arguments. The first (level 2) specifies the new plotting scale along the x-direction, in logical units per 10 pixels (i.e. per tick mark). The second argument similarly specifies the y-scale.

Executing

(0,0) CENTR 1 1 SCALE

establishes the default plot parameters, the same as executing RESET.

The ▢ CENTR and ▢ SCALE keys return the current stored values of the center and the scale.

- *H and *W are programmable “zoom” commands that magnify or demagnify the plot scales in the y- and x-directions, respectively. See section 15.3.2.1.
- The plot environment menu (section 15.3) provides a number of interactive operations for changing the plotting ranges, using the cursor to provide the necessary coordinates.

15.2.4.1 Redimensioning the Graph Screen

When you want to extend the range of a plot without increasing the scale, you can resize the graph screen to larger dimensions than the default 131×64 . In the plot environment, you can use the cursor arrow keys to scroll the display so that you can see hidden portions of the larger graph screen.

All plotting and drawing commands work normally with large graph screens, with their effects scaled to the larger size. `DRAW`, for example, takes longer to execute a function plot, simply because there are more columns for which to compute points than in the default case. The first and second elements in `PPAR` always specify the logical coordinates of the lower-left and upper-right pixels, so the conversion between logical coordinates and pixel numbers depends on the size of the graph screen.

The graph screen size can be changed by `PDIM` (*Plot DIMensions*). This command takes two arguments that specify the new horizontal and vertical graph screen dimensions. If the arguments are two binary integers `#dimx` (level 2) and `#dimy`, the resulting screen is dimensioned to $dim_x \times dim_y$ pixels, leaving the `PPAR` range parameters unchanged. The maximum horizontal width is 2048, but there is no limit other than available memory on the vertical height. You can also change the screen size while keeping the plotting scale unchanged, by using `PDIM` with complex number arguments $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$. These arguments are stored in `PPAR` as the new range parameters, and the graph screen is expanded or shrunk keep the plot scales (logical units/pixel) constant: given w and h as the original graph screen height and width in pixels, the new screen will be $w' \times h'$, where

$$w' = \frac{x'_{\max} - x'_{\min}}{x_{\max} - x_{\min}}(w - 1) + 1$$

$$h' = \frac{y'_{\max} - y'_{\min}}{y_{\max} - y_{\min}}(h - 1) + 1$$

In the plot environment, there are two ways to move the display window around on the larger graph screen:

- Move the plot cursor to an edge of the screen, then continue moving the cursor in the same direction. The cursor “drags” the window along with it. If you press `▢▶` before any cursor arrow key, the cursor moves to the edge of the window in the arrow direction. If the cursor is already at the window edge, it will jump to the edge of the graph screen, moving the window accordingly.
- Press `◀◀ GRAPH`. This turns off the cursor and the menu labels. The cursor arrow keys then scroll the window in the arrow directions. Pressing `◀◀ GRAPH` again restores the cursor and the labels.

15.2.5 The Independent Variable

The third element in PPAR specifies the *independent variable*. This variable is the one that is varied automatically in continuous plots to produce the successive points that constitute the graphs. (For discrete plots, the independent variable is only used to label the horizontal axis.) Specifically:

- For function, conic section, and truth plots, the independent variable is associated with the horizontal axis; plotting proceeds from right to left as the independent variable is incremented.
- For polar plots, the independent variable is the polar angle.
- For parametric plots, the independent variable is not directly associated with either axis. The horizontal and vertical coordinates of plotted points are computed as separate functions of the independent variable.

The HP48 uses the name *X* as the default independent variable name. INDEP allows you to change the name to any other; for example, ' θ ' INDEP selects θ as the independent variable, as might be appropriate for a polar plot. A common error when you work with several different current equations and plot types is to forget that changing either does not automatically change the independent variable; you should always check its name before using DRAW. The $\boxed{\rightarrow}$ **PLOT** menu helps here by displaying the independent variable name as part of its prompting display. Also, $\boxed{\rightarrow}$ **INDEP** returns the current independent variable object from PPAR.

When the independent variable is associated with the horizontal axis, its limits are obvious: its value is varied from that at the left edge of the screen to that at the right edge. DRAW uses these values (obtained from $x_{\min}$ and $x_{\max}$ in PPAR) as the default independent variable range for function, conic section, and truth plots. For polar plots, the default is one "circle," i.e. 0° to 360° in degrees mode, 0 to 2π in radians mode, and 0 to 400 in grads mode. For parametric plots, DRAW also uses the x -range by default, but this is seldom a useful choice.

To specify an alternate range, the independent variable entry in PPAR may be a list of the form $\{ \textit{name start end} \}$, where *name* is the independent variable name and *start* through *end* is the range of values of the independent variable over which the graph is plotted. However, you don't have to re-enter the entire list to change part of it; INDEP makes various changes according to the type of its argument:

- '*name*' INDEP replaces the entire independent variable entry with *name* alone (thus resetting to the default range).
- $\{ \textit{name} \}$ INDEP replaces the name stored in PPAR, but leaves any existing ranges unchanged. For example, if the existing entry is $\{ X 0 10 \}$, $\{ T \}$ INDEP changes the

entry to { T 0 10 }.

- *start end INDEP* or { *start end* } INDEP, where *start* and *end* are real numbers, replace the existing range (or add a range if one is not present), without changing the name of the independent variable. Thus if the existing entry is { T 0 10 } or just T, then 5 50 INDEP or { 5 50 } INDEP changes the entry to { T 5 50 }. Note: LAS-TARG returns a list even if two separate real numbers were entered.
- { *name start end* } INDEP replaces the independent variable entry with the argument list.

Even for function, conic section, and truth plots, it is sometimes useful to specify an independent variable plotting range that is different from the *x*-range defined by the screen limits. For example, if you are plotting ' $\sqrt{X}$ ' with the default *x*-range from -6.5 to +6.5, you might set the independent variable range from 0 to 6.5, to avoid unnecessary time spent "plotting" at negative values of *X* where the expression returns imaginary values.

15.2.6 The Dependent Variable

The seventh entry in the PPAR list specifies the *dependent variable*, which is used as the second variable required for conic section, parametric, and truth plots. For other plot types, the dependent variable name is used only to label the vertical axis. The dependent variable object is either a name, or a list { *name start end* }. *Start* and *end* are used by truth plots to specify the range of *y*-values for the plot.

DEPND establishes a new dependent variable entry from an argument that may be a name, a list, a complex number, or two real numbers, following the same logic as INDEP. $\boxed{\rightarrow} \equiv \text{DEPN} \equiv$ returns the current dependent variable object to the stack.

15.2.7 Resolution

The fourth entry in PPAR is a real number or a binary integer that specifies the plot *resolution*. In general terms, this parameter determines the increment in the independent variable between successive points that are plotted. If it is a real number, the increment is in logical units; a binary integer specifies an increment in pixels. The precise meaning varies according to the current plot type; we will postpone detailed explanations to the sections describing the individual plot types.

The initial default value of the resolution parameter is 0. This value is often the most convenient, since it gives reasonable results for all of the plot types, so you don't have to change it when you switch types. When necessary, you can change the resolution with the command RES, which takes a real number or a binary integer and stores it as the new fourth entry in PPAR. $\equiv \text{RES} \equiv$ is found in the second page of the plot execution

menu; $\boxed{\rightarrow} \equiv \text{RES}$ returns the current resolution parameter to the stack.

Table 15.1 below summarizes the interpretation of the resolution, and the meaning of the default value 0, for each of the eight plot types.

Table 15.1. Plot Resolution

Plot type	Resolution		
	<i>real step</i> = 0	<i>real step</i> $\neq$ 0	binary # <i>step</i>
Function	$\Delta x = scale_x/10$ (1 column)	$\Delta x = step$	$\Delta x = step \cdot scale_x/10$
Conic	$\Delta x = scale_x/10$ (1 column)	$\Delta x = step$	$\Delta x = step \cdot scale_x/10$
Polar	$\Delta \theta = 2$ (degrees/grads mode) or $\pi/90$ (radians)	$\Delta \theta = step$	$\Delta \theta = step \cdot scale_x/10$
Parametric	$\Delta t = (t_{end} - t_{start})/130$	$\Delta t = step$	$\Delta t = step \cdot scale_x/10$
Truth	$\Delta x = scale_x/10$ (1 column)	$\Delta x = step$	$\Delta x = step \cdot scale_x/10$
Scatter	Not used		
Histogram (bin width)	$\Delta x = (x_{max} - x_{min})/13$ (10 columns)	$\Delta x = step$	$\Delta x = step \cdot scale_x/10$
Bar (bar width)	$\Delta x = 1$	$\Delta x = step$	$\Delta x = step \cdot scale_x/10$

In the table, x is the horizontal coordinate (and the independent variable for function, truth, and conic plots); t is the independent variable for parametric plots; θ is the independent (polar angle) variable for polar plots. $scale_x$ is the plot scale in the x -direction, logical units per 10 pixels. An increment $\Delta x = scale_x/10$ is equivalent to moving one pixel width in the x -direction.

Leaving the resolution at its default value of zero is often a good choice, since then you don't have to worry about changing the resolution when you switch plot types or current equations.

15.2.8 Axes and Labels

When you execute DRAW by means of the $\equiv \text{DRAW} \equiv$ key, the HP 48 automatically draws axes on the graph screen to help you interpret the graph. $\equiv \text{DRAW} \equiv$ actually invokes DRAX (*DRaw AXes*); to reproduce the key's effect in programs, you must include DRAX as well as DRAW. The axes are drawn through the point specified by the fifth element in PPAR. Usually, this element is a complex number that represents the coordinates of the origin of the axes. It may also be a list of the form

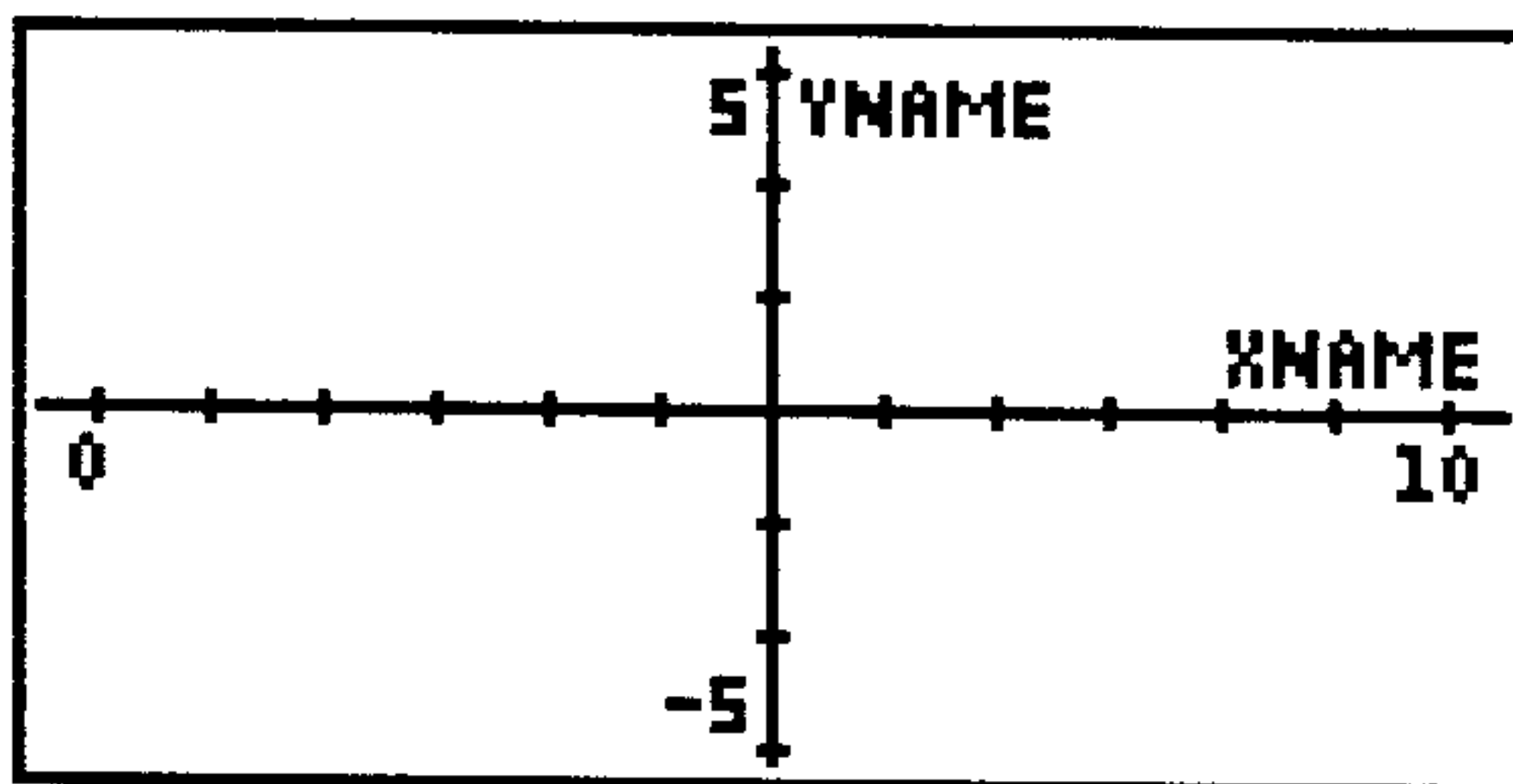
$$\{ (x_{axes}, y_{axes}) \text{ "label}_x" \text{ "label}_y" \}$$

Here the first element specifies the intersection of the axes, and the remaining two elements are strings that are used to label the x- and y-axes, respectively.

AXES allows you to move the axis origin to another position, or to add or change axis label strings to PPAR. This command takes a variety of arguments, much like **INDEP** or **DEPND**:

- If the argument is a single complex number, it replaces the current PPAR entry.
- If the argument is a list containing only a complex number, that number replaces the current origin coordinates without affecting any current label strings.
- If the argument is a list containing two strings, the strings replace the current label strings (the PPAR element becomes a list, if it is not already). The origin is unchanged.
- If the argument is a list containing a complex number and two strings, the list replaces the current PPAR entry.

The actual labeling is performed by **LABEL**, which you may execute as an ordinary command or by means of the **LABEL** keys in the third page of the plot execution menu and in the first page of the plot environment menu. **LABEL** labels the axes with variable names or strings, and with the values of the coordinates at the edges of the plot. Labels are placed on the graph screen like this:



This picture was generated by the following sequence:

```
STD 0 10 XRNG -5 5 YRNG (5,0) AXES
{"XNAME" "YNAME"} AXES DRAX LABEL
```

STD is included in the sequence because **LABEL** formats its coordinate values according to the current real number display mode.

LABEL will make axis labels even if you haven't stored label strings in PPAR. The horizontal axis label shown as XNAME in the picture is chosen in the following priority order:

1. The "*label_x*" string.
2. The independent variable name.

(The latter choice is usually not too meaningful for polar or parametric plots.) The vertical axis name is similarly chosen in the following priority order:

1. The "*label_y*" string.
2. If flag -30 is clear, and the current equation is of the form *name* = *expression*, where *name* is not the independent variable, then *name*.
3. The dependent variable name from PPAR.

15.2.8.1 Tick Marks

The *tick marks* that are drawn on the axes by DRAX are placed at ten-pixel intervals (measured from the axis origin). With the default plot scales, the marks are spaced one logical unit apart; with other scales they may or may not be helpful as position indicators other than to distinguish the axes from other lines.

Despite the modest pixel density of the HP 48 display, tick marks at other spacings can be useful, even if they end up with irregular spacing. The programs listed next let you specify arbitrary and independent tick spacings for the horizontal and vertical axes (SETTICKS), and to draw axes with the custom tick marks (TICKAXES). The tick spacings, which you may specify with real numbers (logical units) or binary integers (pixels), are stored as a two-element list in a variable TICPAR.

SETTICKS			Store Tick Spacings	1483
	level 2	level 1		
	<i>x</i>	<i>y</i>	☒	
	# <i>n_x</i>	# <i>n_y</i>	☒	
<< 2 →LIST 'TICPAR' STO >>				Combine parameters into a list. Store as TICPAR.

TICKAXES	Draw Axes with Ticks	1E4E
<pre><< (0,0) PIX? DROP PPAR OBJ→ DROP TICPAR IF DUP TYPE 6 SAME THEN 1 DUP SETTICKS RCL END OBJ→ DROP IF DUP TYPE 10 == THEN { #0 #0 } PX→C #0 ROT 2 →LIST PX→C - IM END SWAP IF DUP TYPE 10 == THEN #0 2 →LIST PX→C { #0 #0 } PX→C - RE END 9 ROLL C→R 10 ROLL C→R 9 ROLL IF DUP TYPE 5 == THEN 1 GET END C→R → dy dx xmin ymin xmax ymax xaxis yaxis << 4 DROPN xmin yaxis R→C xmax yaxis R→C LINE xaxis ymin R→C xaxis ymax R→C LINE xaxis yaxis R→C DUP C→PX OBJ→ DROP 1 - SWAP 1 + SWAP 2 →LIST PX→C DUP2 + 3 ROLLD - C→R ROT C→R → xm ym xp yp << xmin xaxis - dx / 0 RND dx * xaxis + xmax FOR x x yp R→C x ym R→C LINE dx STEP ymin yaxis - dy / 0 RND dy * yaxis + ymax FOR y xp y R→C xm y R→C LINE dy STEP >> >> >></pre>	Ensure PPAR exists. Take PPAR apart. If TICPAR doesn't exist, then create a default version. Take TICPAR apart. If y-increment is binary, convert to real. If x-spacing is binary, convert to real. $x_{\min}$ $y_{\min}$ $x_{\max}$ $y_{\max}$. Axes entry. Extract coordinates from list. Store in local variables. Discard extra parameters. Draw x-axis. draw y-axis. Compute x- and y-coordinates of points next to the axes. (x_{axis}, y_{axis}) $(x_{axis} + , y_{axis} +)$ $(x_{axis} + , y_{axis} +)$ $(x_{axis} - , y_{axis} -)$ Save coordinates in local variables. Position of first x tick. Draw the next x tick. Position of first y tick. Draw the next y tick.	

15.2.9 Plot Type

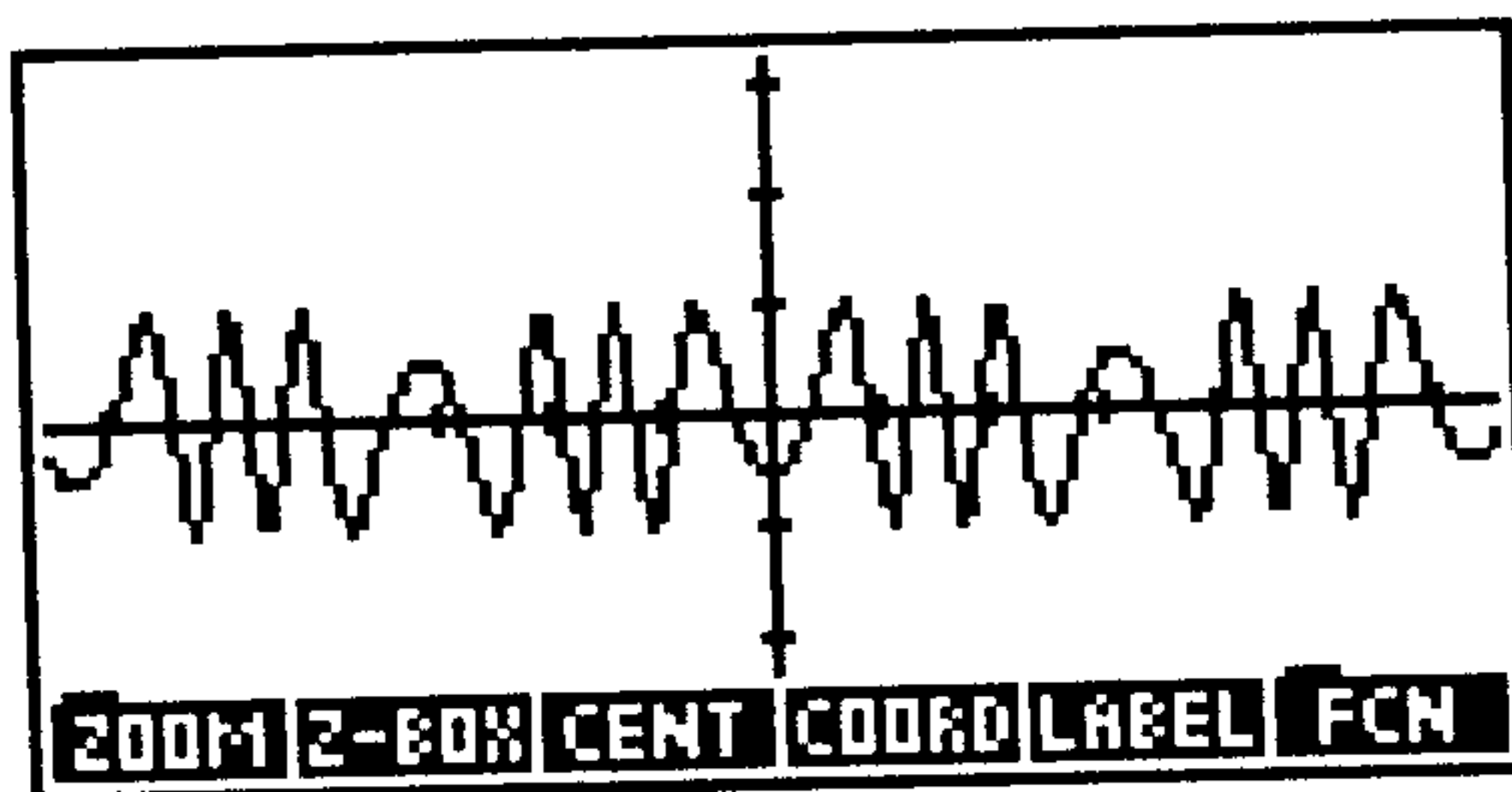
The sixth PPAR entry is one of the eight plot type commands FUNCTION, CONIC, POLAR, PARAMETRIC, TRUTH, SCATTER, BAR, and HISTOGRAM. When any of these commands is executed, it stores itself as the sixth entry in the PPAR list. There, the command name is used by DRAW to determine the type of plot to create.

15.3 The Plot Environment

When you press $\leftarrow$ GRAPH (or just $\leftarrow$ when no command line is present) or $\equiv$ DRAW $\equiv$, the HP 48 switches to the *plot environment* (see section 6.1), where the keyboard and display are dedicated to interaction with the graph screen. Here you can view graphs, add labels, points, lines, and other geometric figures, change the plotting ranges and scales, transfer coordinates and pictures to and from the stack, and perform various analytic operations on function graphs. As from other HP 48 environments, you can exit back to the standard environment by pressing $\equiv$ ATTN $\equiv$.

To illustrate the use of the plot environment, create a simple function plot:

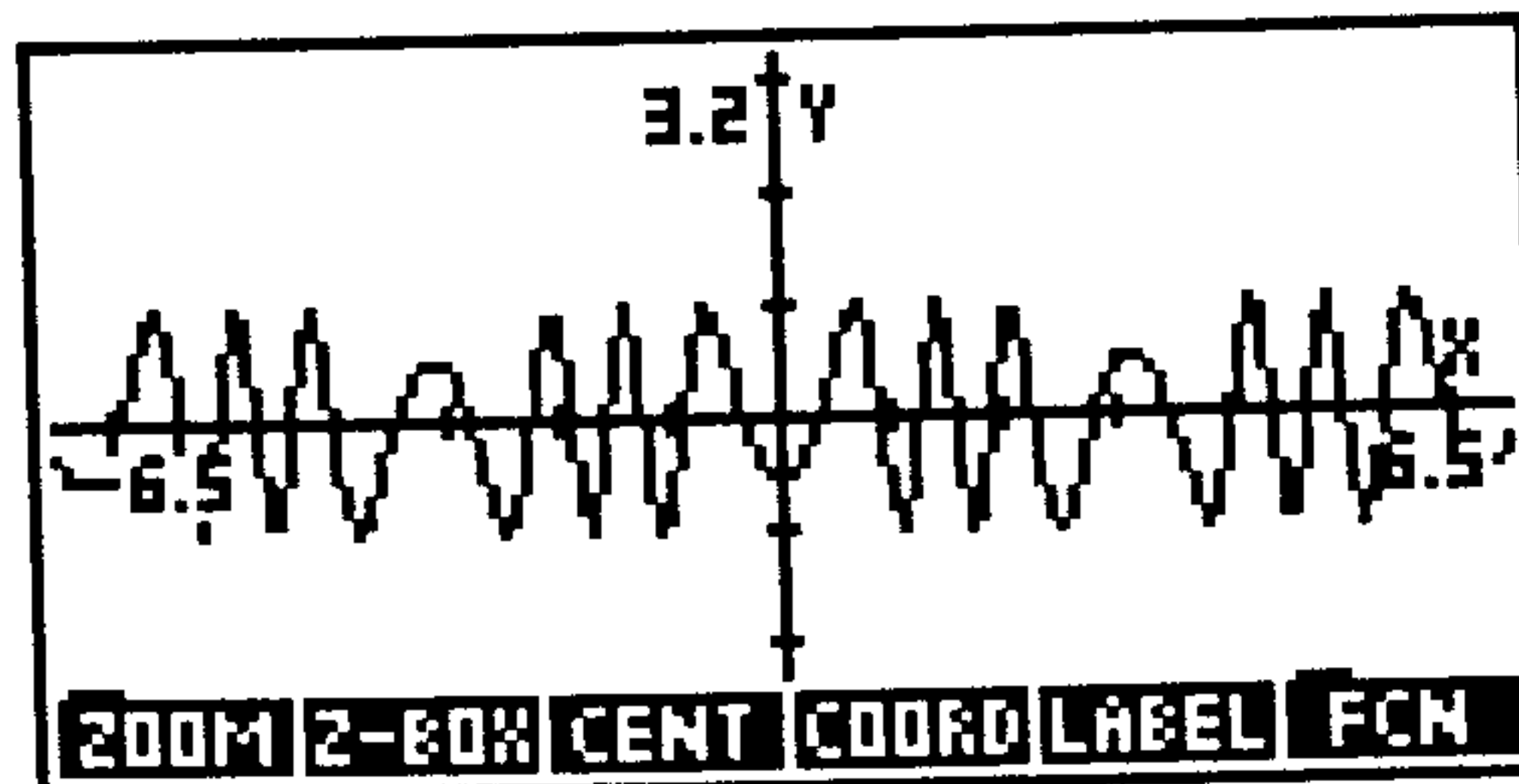
STD RAD 'SIN(10*COS(X))' $\leftarrow$ PLOT $\equiv$ NEW $\equiv$ ENVEX $\equiv$ ENTER $\equiv$
 $\equiv$ PTYPE $\equiv$ $\equiv$ FUNC $\equiv$ $\equiv$ PLOTR $\equiv$ NXT $\equiv$ RESET $\equiv$ $\leftarrow$ PREV $\equiv$ ERASE $\equiv$ $\equiv$ DRAW $\equiv$



The display is characteristic of the plot environment:

- The upper part of the display shows the graph screen, currently with the graph of $\sin(10 \cos x)$.
- The first page of the plot environment menu appears in the menu label area.

The menu labels actually overlay part of the graph screen. For example, if you press $\equiv$ LABEL $\equiv$, you may notice that you can only see three of the four coordinate-limit values:



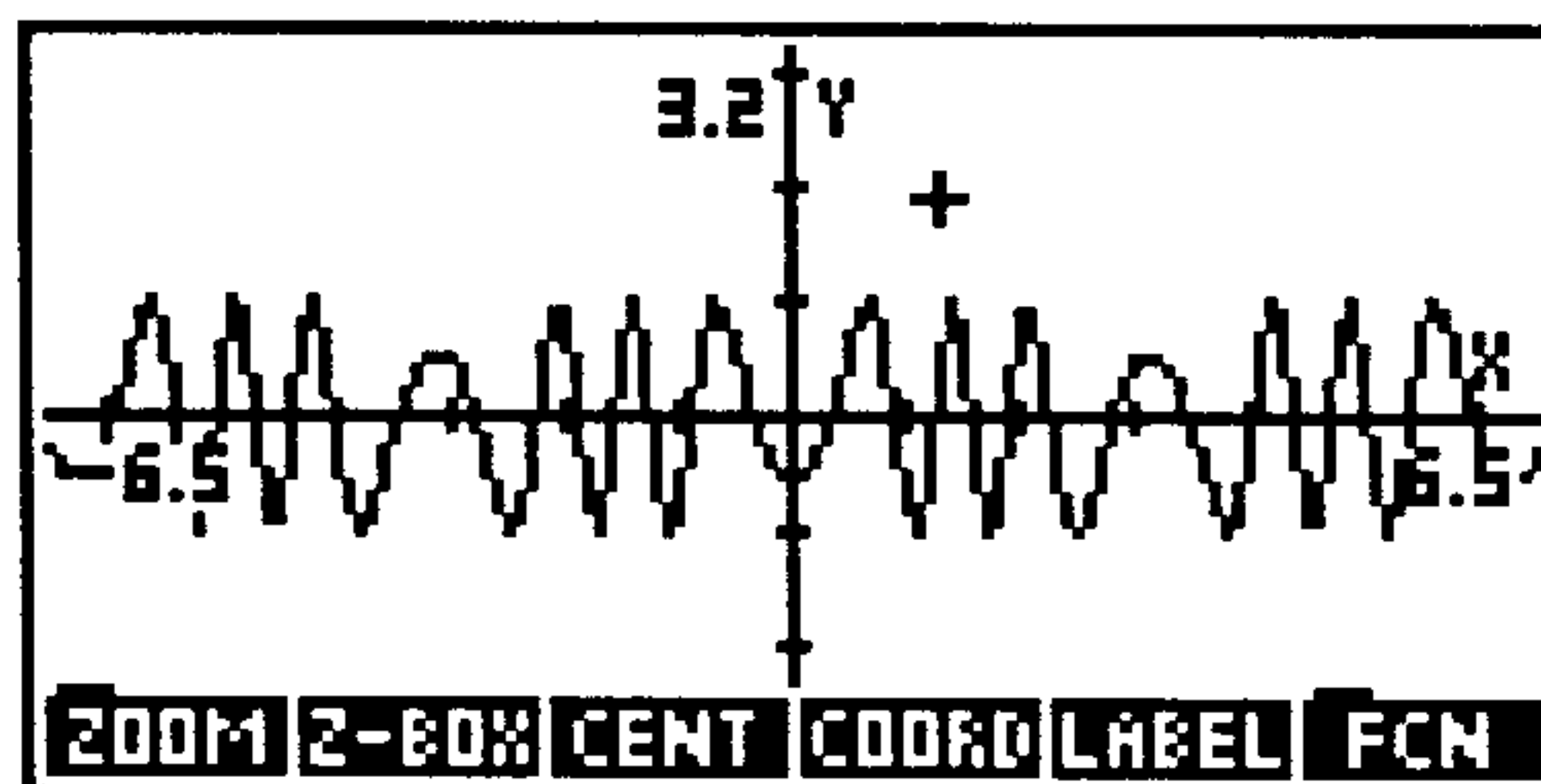
To uncover the hidden part of the graph, press $\boxed{-}$ (to “take away” the labels), or $\boxed{\equiv}\boxed{\text{KEYS}}\boxed{\equiv}$ in the third page of the menu. To restore the labels, press any menu key, or $\boxed{+}$ again. You can also move the cursor downwards with $\boxed{\nabla}$ or $\boxed{\rightarrow}\boxed{\nabla}$. When the cursor reaches the top of the menu labels, further downward motion causes the graph screen to scroll upwards, revealing the hidden part. Finally, you can press $\boxed{\leftarrow}\boxed{\text{GRAPH}}$, which removes the menu labels and the cursor. This is especially useful when you want to use $\boxed{\text{ON}} - \boxed{\text{PRINT}}$ to print an image of the graph screen without the extraneous features.

Along with the menu keys (including $\boxed{\text{NXT}}$, $\boxed{\leftarrow}\boxed{\text{PREV}}$ and $\boxed{\rightarrow}\boxed{\text{PREV}}$), the rest of the HP 48 keyboard is redefined for plot-related operations:

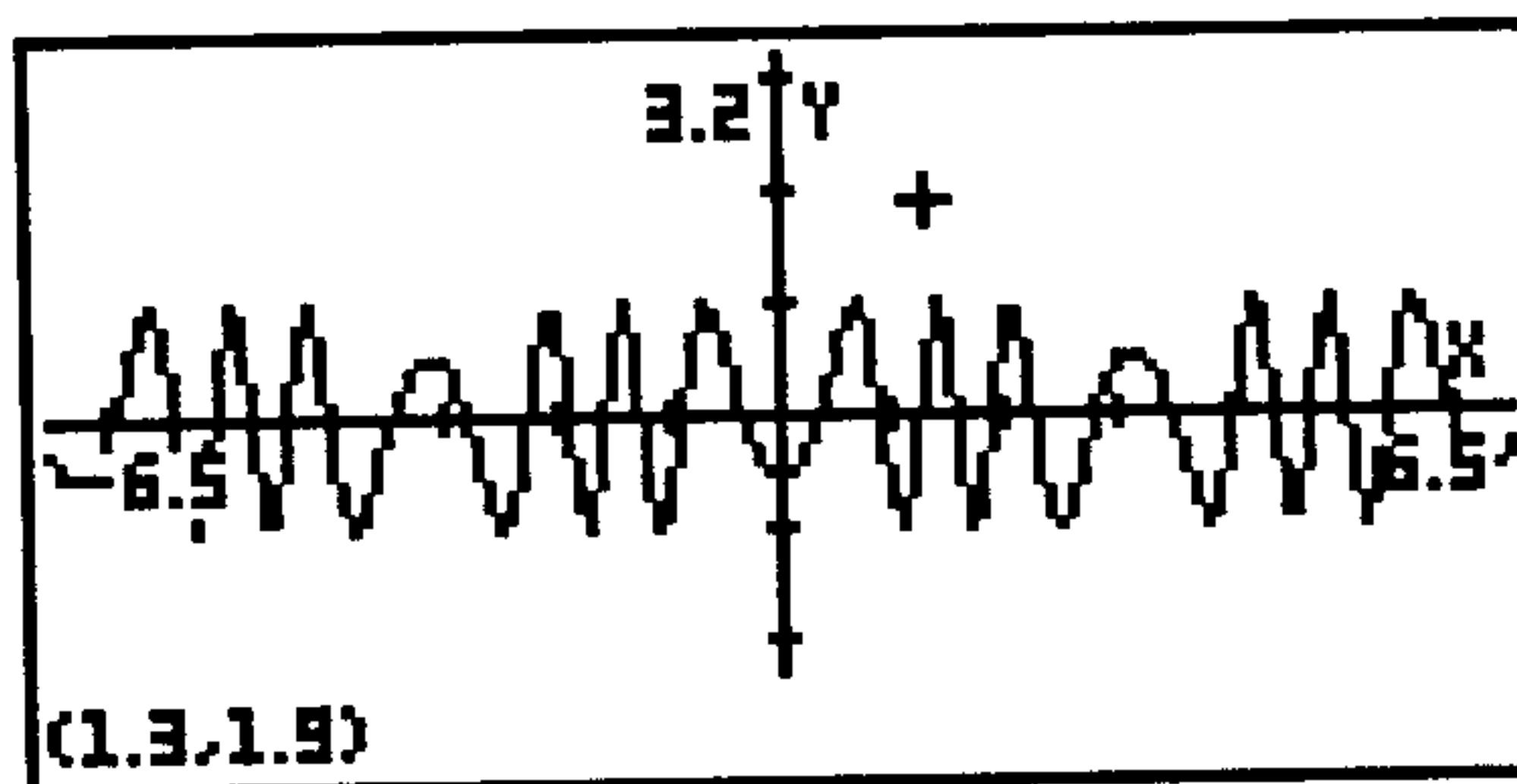
- $\boxed{\Delta}$, $\boxed{\triangleleft}$, $\boxed{\nabla}$, and $\boxed{\triangleright}$ move the plot cursor in the indicated directions. Pressing $\boxed{\rightarrow}$ before any of the arrow keys moves the cursor to the edge of the screen in that direction.
- $\boxed{\text{STO}}$ copies the graph screen to the stack as a graphics object.
- $\boxed{\text{ENTER}}$ returns the cursor position to the stack as a complex number object.
- $\boxed{+/-}$ switches the cursor type.
- $\boxed{\text{DEL}}$ erases a rectangular portion of the graph screen.
- $\boxed{\rightarrow}\boxed{\text{CLR}}$ erases the entire graph screen.
- $\boxed{\times}$ sets the plot mark.
- $\boxed{-}$ turns the menu labels on and off.
- $\boxed{+}$ turns the coordinate display on and off.

15.3.1 The Plot Cursor

Not immediately visible in the display picture above is the *plot cursor*, which is initially placed at the center of the screen. To reveal the cursor, you can move it away from the axes with $\boxed{\triangleright}$ and $\boxed{\Delta}$:

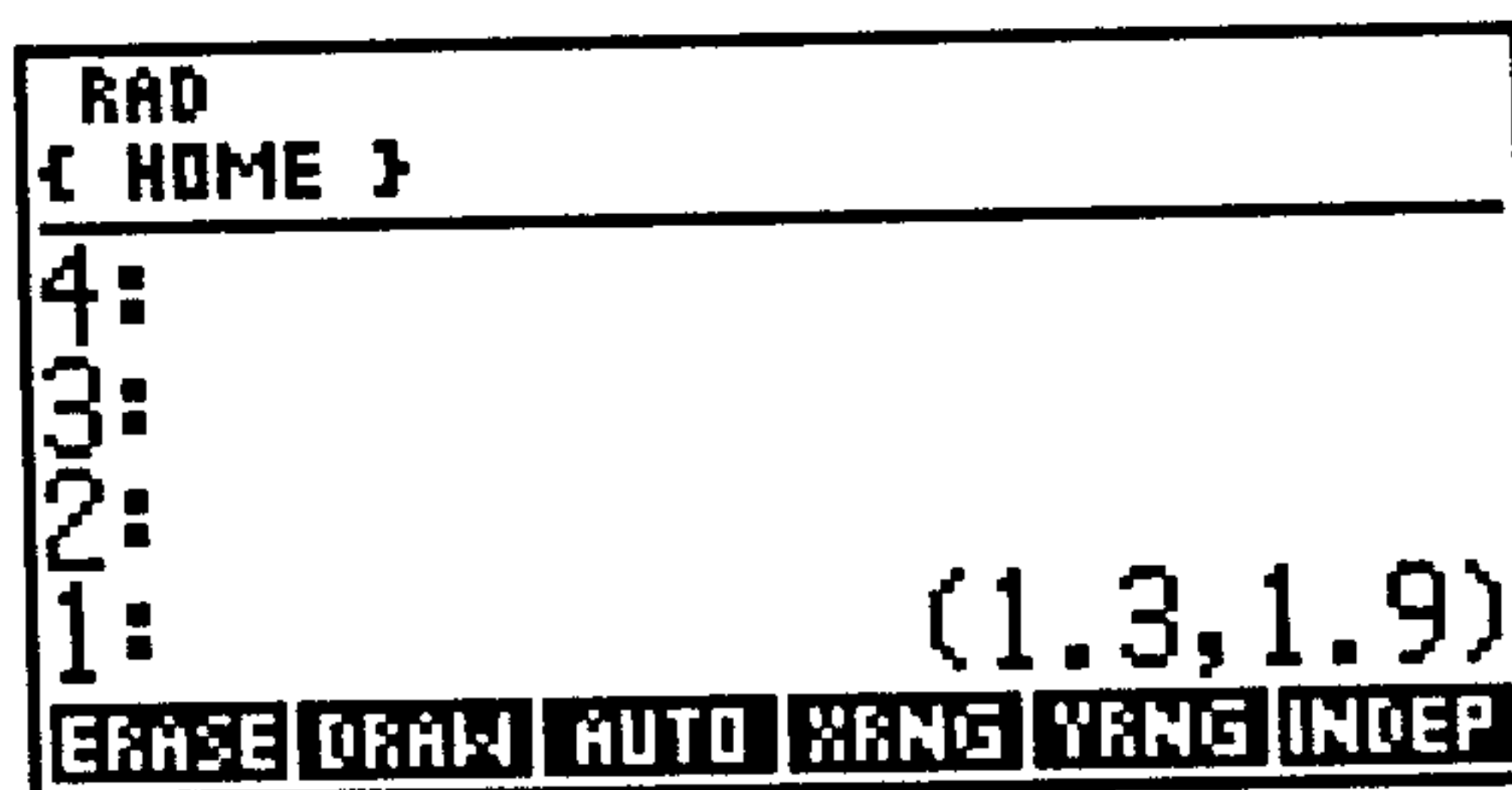


Now you can see that the cursor is a small "+". The cursor is used as a focal point for various operations. For example, if you press $\equiv$ COORD $\equiv$ or $\boxed{+}$, you can read the cursor position in the menu label area:



The coordinate display follows the current number display format, which we set to standard format at the start of this example. If you move the cursor while its coordinates are being displayed, the coordinates are changed along with the cursor position (this slows cursor movement down somewhat). You can restore the menu labels by pressing $\boxed{+}$, $\boxed{-}$, or any menu key.

For use in further computations, you can copy the cursor position to the stack by pressing $\boxed{\text{ENTER}}$. You will see the busy annunciator flash momentarily, but there is no other visible indication until you exit the environment by pressing $\boxed{\text{ATTN}}$:



Here you can see the cursor coordinates represented as a complex number. To return to the plot environment now, press $\boxed{\triangleleft}$.

Because the default plot cursor is drawn as a superimposed cross, it is invisible any time it is situated on a dark screen region. This makes the cursor particularly hard to use with bar charts and histograms. To remedy this situation, you can alter the nature of the cursor by pressing $\boxed{+/-}$ (or $\equiv+/-\equiv$ in the third page of the plot environment menu).

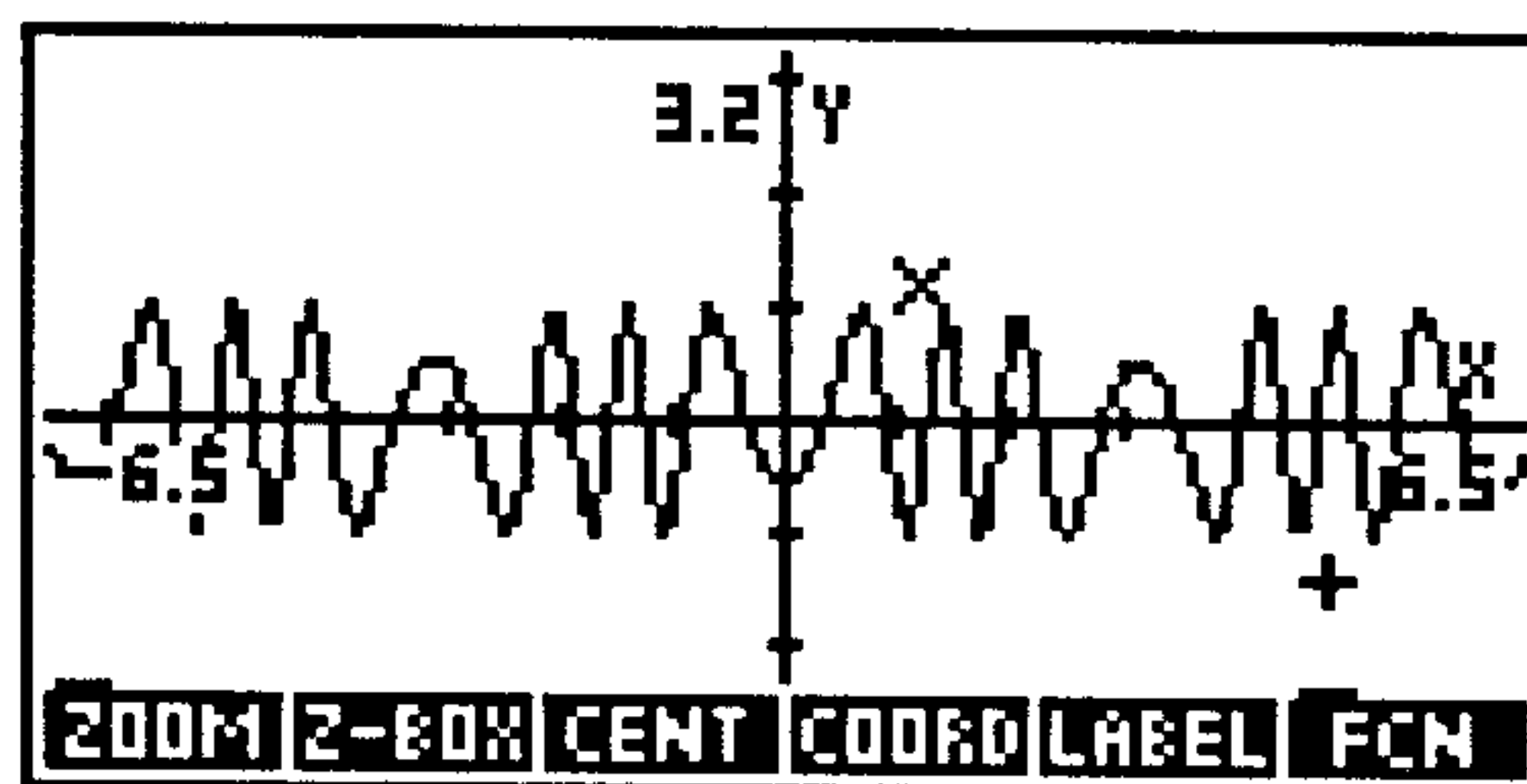
This changes the cursor from a simple superposition of the cross to a pixel-by-pixel reversal, in the manner of GXOR (see section 10.3.1). On a dark background, the cursor is white, whereas on a white background, the cursor is dark. In an area with a mixture of dark and white pixels, you can press $\boxed{+/-}$ several times to see where the cursor is. The cursor type is encoded by flag -32, which allows you to set the type in a program. The default superimposed cursor corresponds to flag -32 clear.

15.3.1.1 The Mark

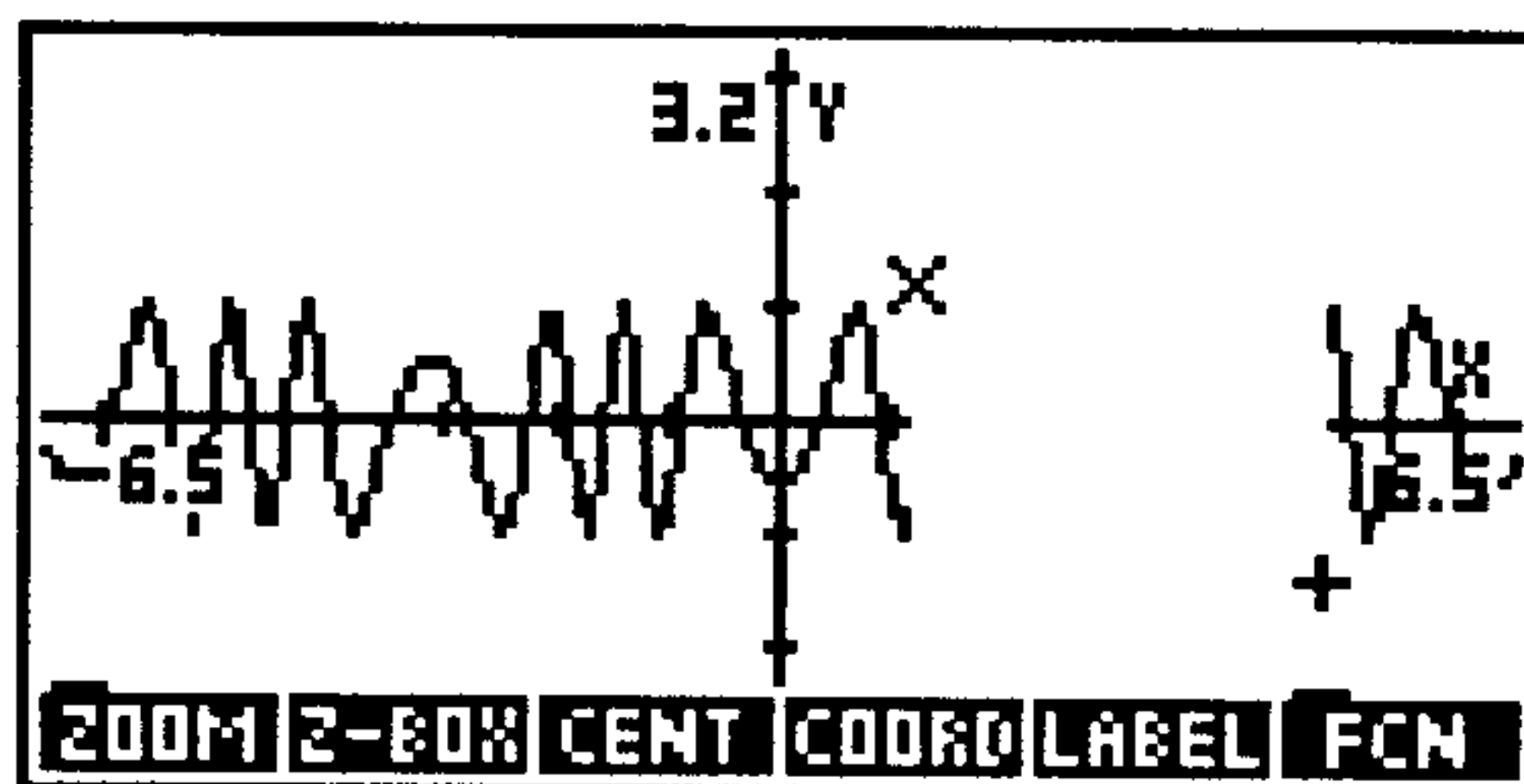
For certain operations, it is necessary to specify two points on the screen. The cursor is always available as one of those points; for the other, you set a *mark*. The mark is set by moving the cursor to a point, then pressing $\boxed{\times}$ or $\equiv \text{MARK} \equiv$ (in the third page of the plot environment menu). This draws an $\times$ mark, which remains when you move the cursor away. If a mark already exists, the old mark is erased and a new one drawn--unless the cursor is actually on the mark, which just erases the mark.

If you press any menu key that requires a mark when no mark exists, that key will set the mark rather than executing its labeled operation.

A simple example of a cursor/mark operation is $\boxed{\text{DEL}}$, which erases the graph screen in the region defined by the cursor and the mark. With the cursor and mark as follows,



$\boxed{\text{DEL}}$ erases like this:

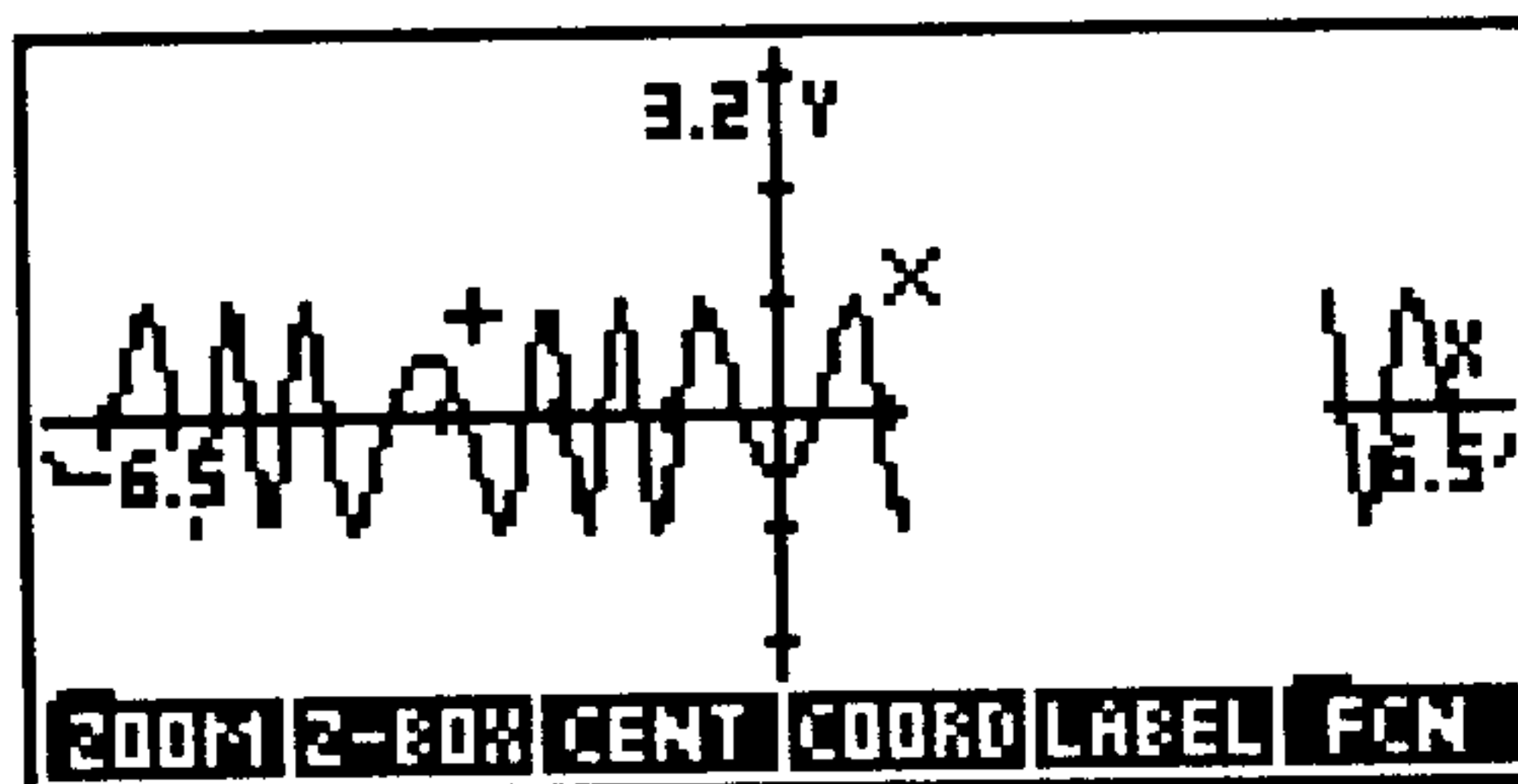


This operation is also available via the menu key DEL, in the third page of the plot environment menu.

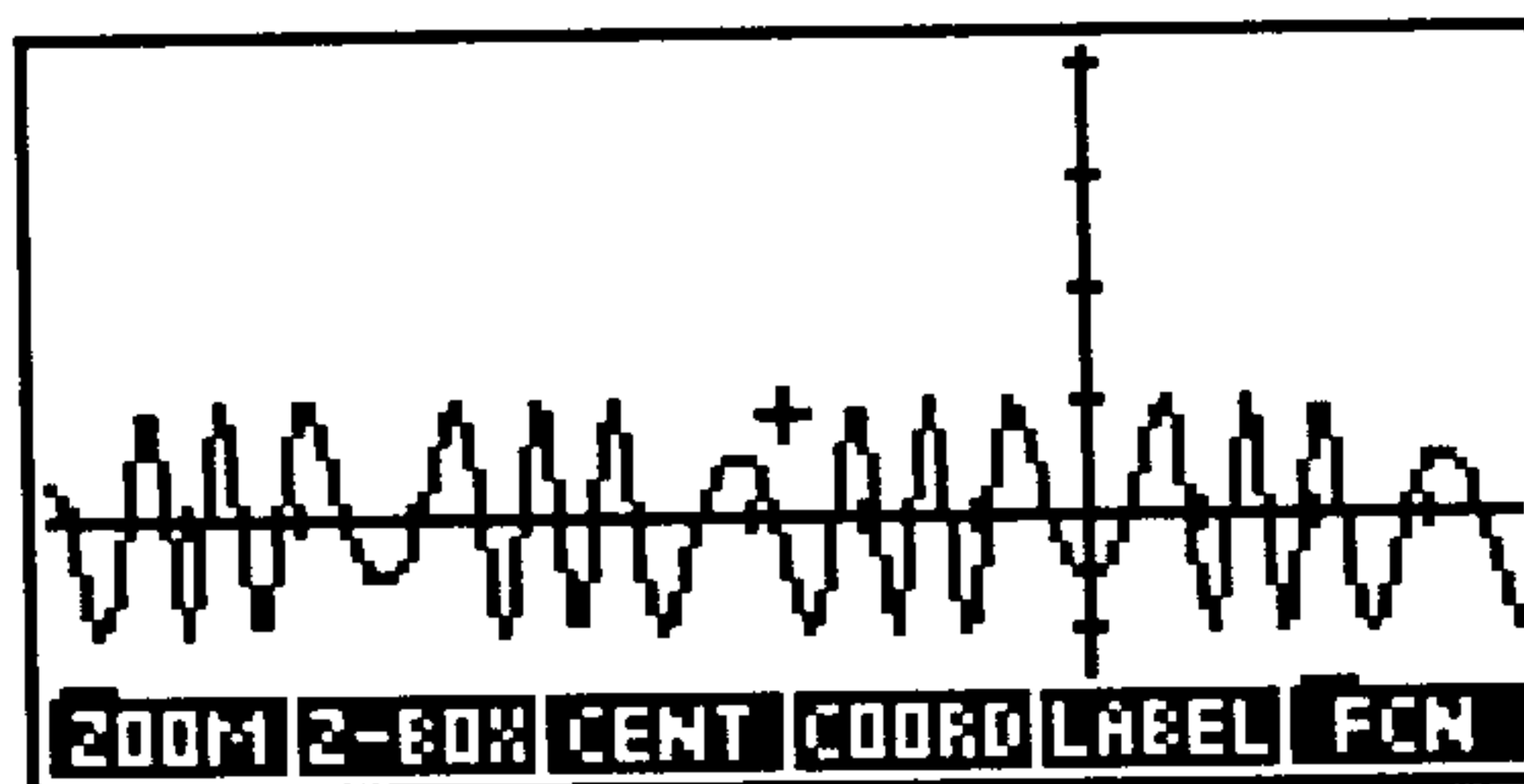
15.3.2 Recentering and Zooming

Unless you are clever in choosing the plot ranges before you make a graph, chances are the plot ranges will not be quite right. It is often easier to make a trial plot, then adjust the ranges, than it is to choose the ranges in advance. The key operations for this purpose are ZOOM and Z-BOX, found in the first page of the plot environment menu.

Recentering is a simple one-step operation by which you can move a graph relative to the screen so that some desired feature is centered on the screen. All you do is move the cursor to the point where you want the new screen center, then press CENT. For example, with the cursor as shown here



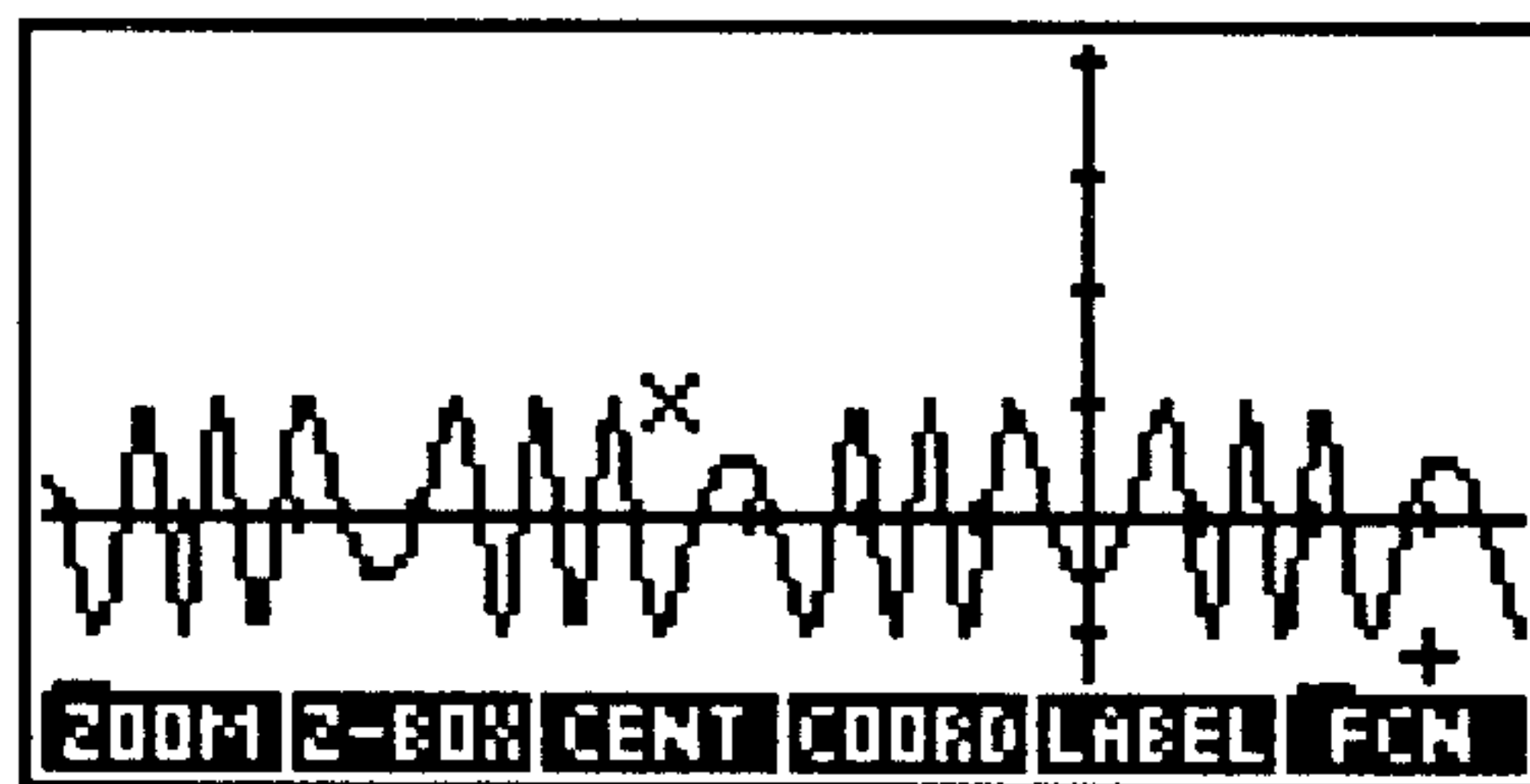
CENT redraws the picture like this:



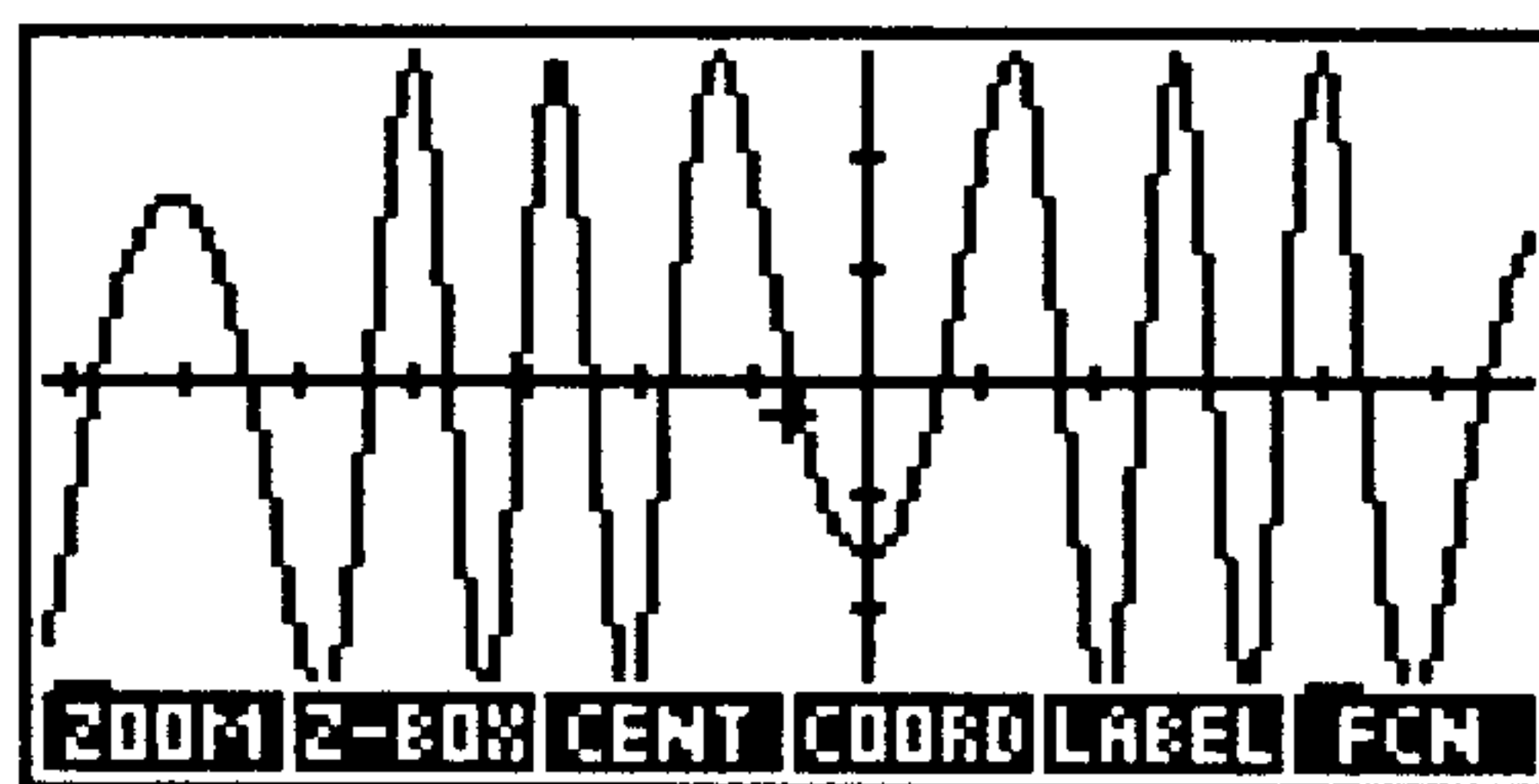
You could achieve the same effect by pressing ENTER (to enter the cursor coordinates), then ATTN CENTR ERASE DRAW, but for interactive use, performing the recentering from within the plot environment is more convenient.

The term “zooming” originated in photography, where use of variable focus lenses allows you to change the magnification of a scene. In computer graphics, it has come to mean a similar change of magnification of a plot, even though the speed implied by the term may not be possible. For a computed plot, there is seldom any advantage to changing the plot scales unless the plot is recomputed to add new data.

In the plot environment, you can zoom in on or out from a plot by specifying numerical magnification factors, or, for zooming in, by selecting a region that you want to expand to fill the screen. The latter is the most intuitive method; you position the mark and the cursor to define a rectangular region, then press ≡Z-BOX≡ (*Zoom-to-BOX*). The plot is redrawn so that the region fills the screen (the full 131×64). For example, with the cursor and mark as shown,



≡Z-BOX≡ yields this picture:

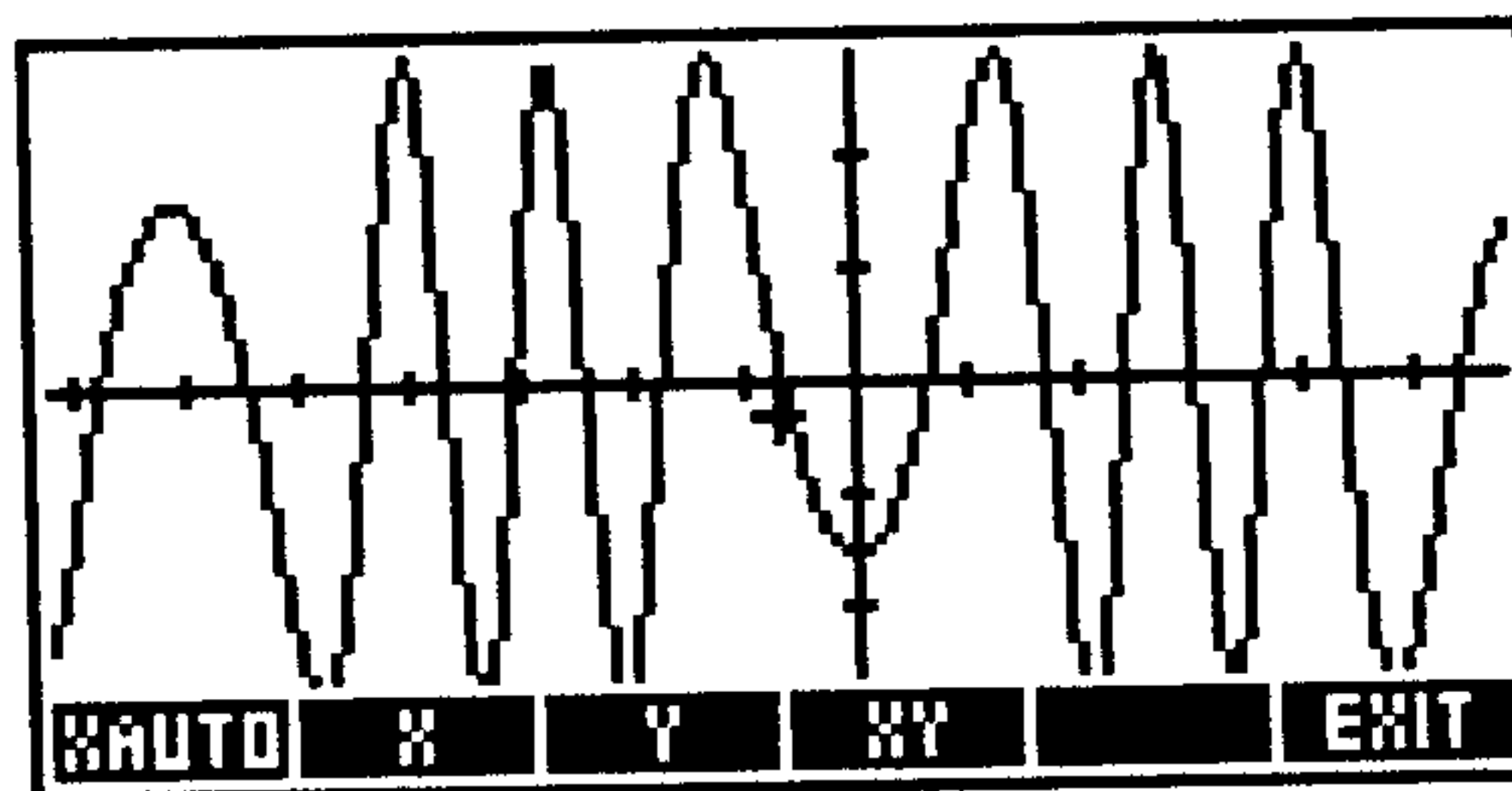


You can use ≡Z-BOX≡ to change the scale in only one direction by placing the mark and the cursor in the same row or column. That is, if the mark and the cursor are in the same column, ≡Z-BOX≡ increases the y-scale to stretch the marked range to fit the screen vertically, but leaves the x-scale unchanged. Similarly, using ≡Z-BOX≡ with the

cursor and the mark in the same row increases the x -scale but does not change the y -scale. (If the cursor is on the mark, Z-BOX does nothing.)

When you want to zoom *out*, i.e. increase the plot scale so that more of a curve or other structure fits on the screen, you must use the numerical ZOOM operation. Here you may specify a magnification factor by which to multiply the x - or y -scale, or both; a factor larger than 1 zooms *out*, shrinking plot features by that factor. A factor smaller than 1 zooms *in*, increasing the size of plot features.

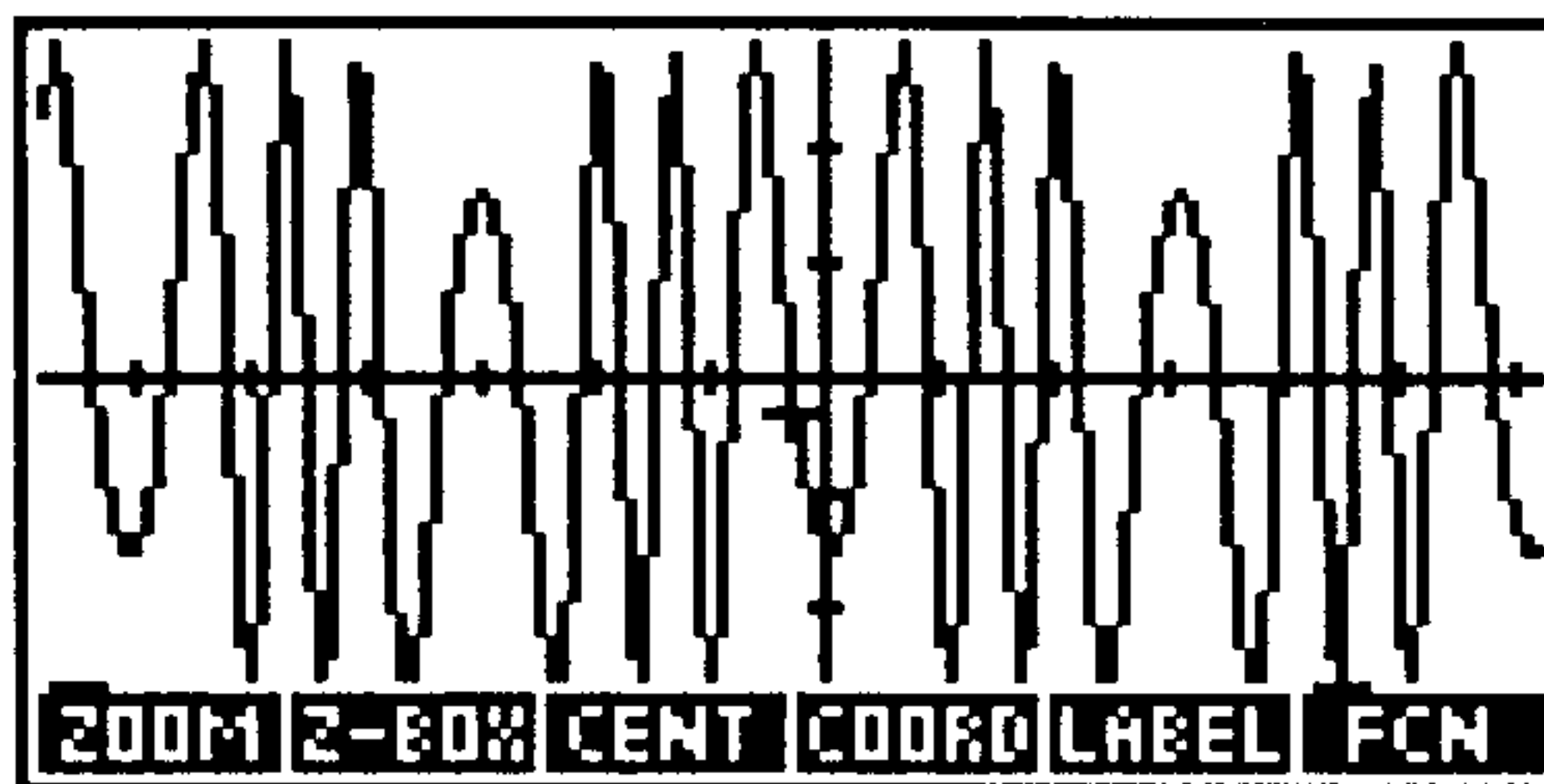
With the picture as shown above, pressing ZOOM changes the menu as shown here:



EXIT allows you to cancel the change, returning to the main plot environment menu without altering the plot. Pressing any of the other four labeled menu keys prompts you for a zoom magnification factor. For instance, pressing X produces this display:

RAD	PRG
{ HOME }	
x axis zoom.	
Enter value (zoom out	
if >1), press ENTER	
◆	

Now you may type in a zoom factor, or press ATTN to return to the zoom menu. Try 2 ENTER :



The plot is redrawn, with a doubled x -scale. The logical coordinates of the center of the screen are preserved, so that the curve appears to be compressed along the x -direction, around the screen center. An analogous behavior is produced by pressing Y in the zoom menu, with the y -scale varying according to the zoom factor and the x -scale held constant. XY produces a uniform rescaling in both directions by the entered factor. The latter is often the most useful when you are searching for a particular plot feature; it lets you zoom out, repeatedly if necessary, until you can see the feature. Then you can zoom back in to magnify that area of the graph.

The final option in the zoom menu is XAUTO. This operation rescales in the x -direction according to the zoom factor you enter, then autoscales in the y -direction, following the autoscale logic described in section 15.2.1.1.

15.3.2.1 Programmable Zooming

The commands `*H` and `*W` are programmable commands that can stretch or shrink a plot in height and width, respectively. For either command, you supply a real number argument that is used by the command to multiply the appropriate coordinates of $P_{\min}$ and $P_{\max}$. For example, to double the height of a plot (i.e. to flatten a curve to half its original height on the screen), you execute `2 *H`, then `DRAW`. To make a plot cover twice the horizontal range of the previous plot (compressing the plotted curve sideways), execute `2 *W`. The effects are the same as manual zooming using the `ZOOM` menu in the plot environment, with zoom factors of 2.

The effects on the plot range parameter of using `*H` and `*W` with arguments h and w are listed below. The primed quantities are the new values, and the unprimed quantities are the original values.

`*H`

$$y'_{\min} = \frac{(32 + 31h)}{31} y_{\min} + \frac{(31 - 31h)}{31} y_{\max}$$

$$y'_{\max} = \frac{(32 - 32h)}{31} y_{\min} + \frac{(31 + 32h)}{31} y_{\max}$$

*W

$$x'_{\min} = \frac{1}{2}(1 + w)x_{\min} + \frac{1}{2}(1 - w)x_{\max}$$

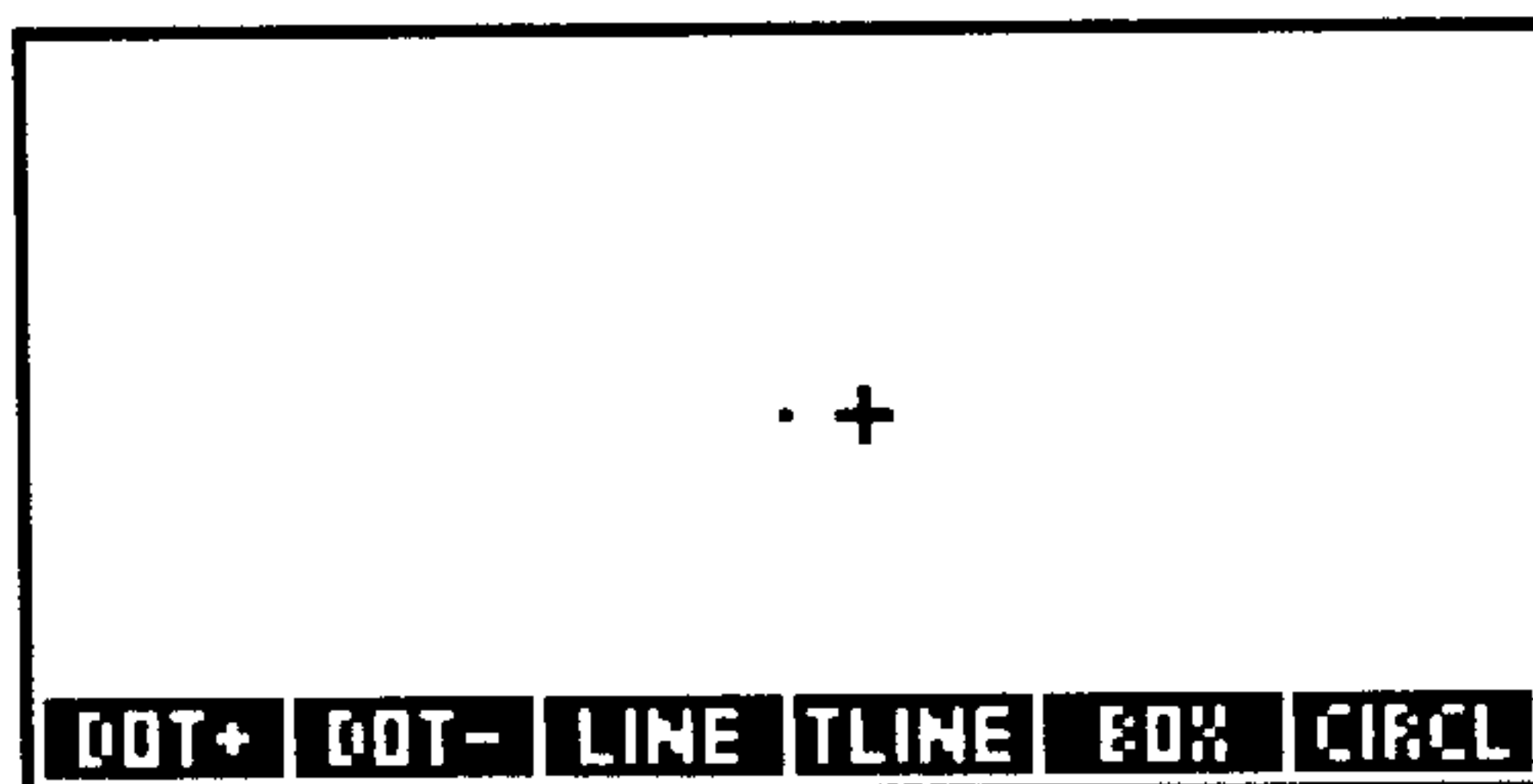
$$x'_{\max} = \frac{1}{2}(1 - w)x_{\min} + \frac{1}{2}(1 + w)x_{\max}$$

15.3.3 Drawing on the Graph Screen

In addition to the labels you can add automatically using LABEL (section 15.2.3), the plot menu provides several other tools for embellishing a graph and for making diagrams that may or may not be related to any mathematical plot. These tools are interactive versions of the drawing commands described in section 10.3.4, using the cursor and the mark to specify positions rather than stack-entered coordinates.

To illustrate these operations, we will start with a blank graph screen. To follow the examples, you should exit from the plot environment by pressing **ATTN** (if necessary), then **▢** **≡PLOT≡** **≡ERASE≡**, and **◀** to return to the plot environment. (You can also clear the screen without leaving the environment by pressing **▢** **CLR**, but the preceding method also returns the cursor to the center.)

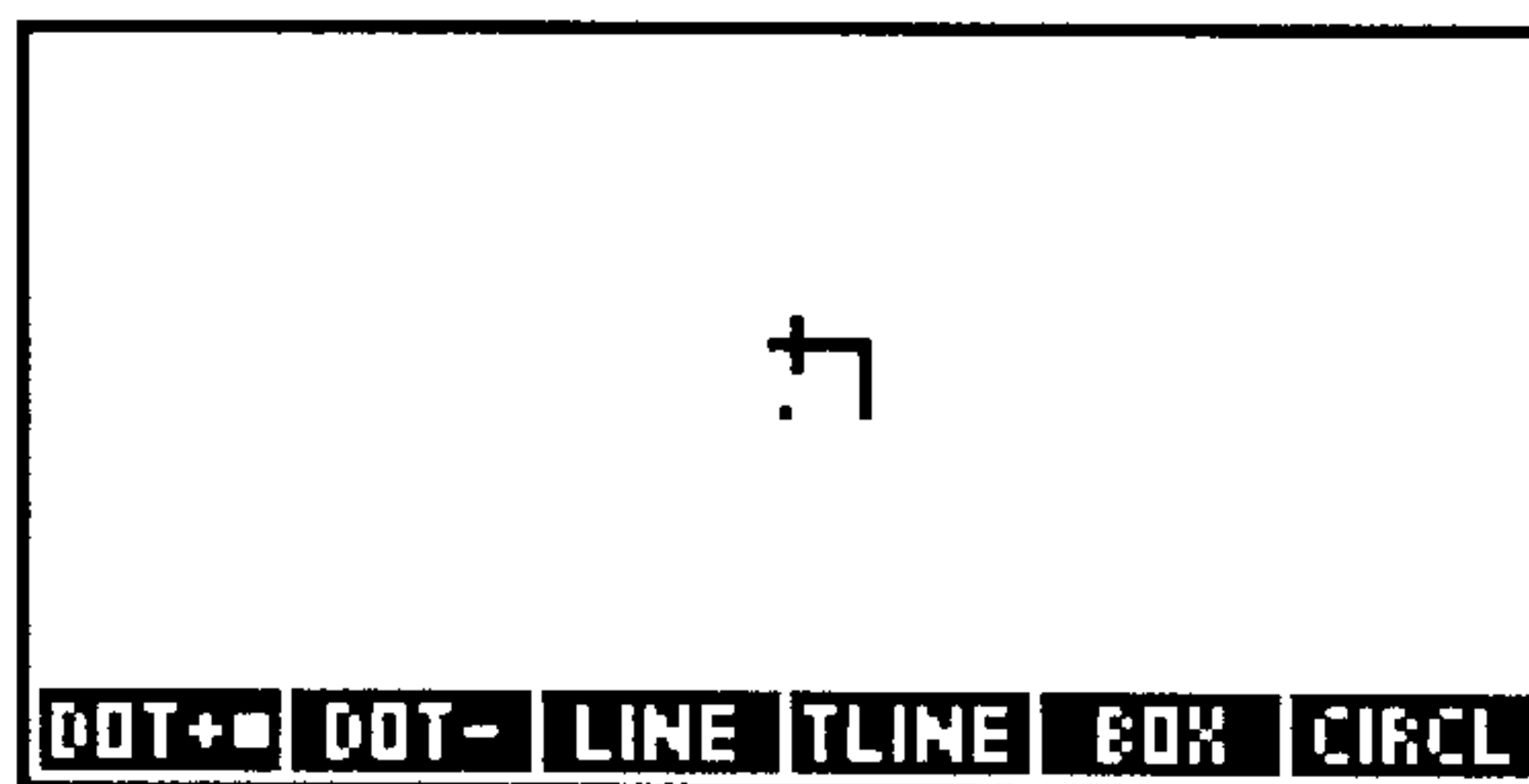
The simplest operation is turning “on” (dark) a pixel, as you might with PIXON. This is done by pressing **≡DOT+≡** (second page) *twice*, which turns on the pixel at the center of the cursor. If you do this then move the cursor a few pixels to the right, you will see the center pixel turned on:



Pressing **≡DOT+≡** just once activates a line-drawing mode in which the cursor leaves a trail behind as you move it. Thus

≡DOT+≡ **▢** **▢** **▢** **▢** **▢** **▢** **◀** **◀** **◀** **◀** **◀** **◀**

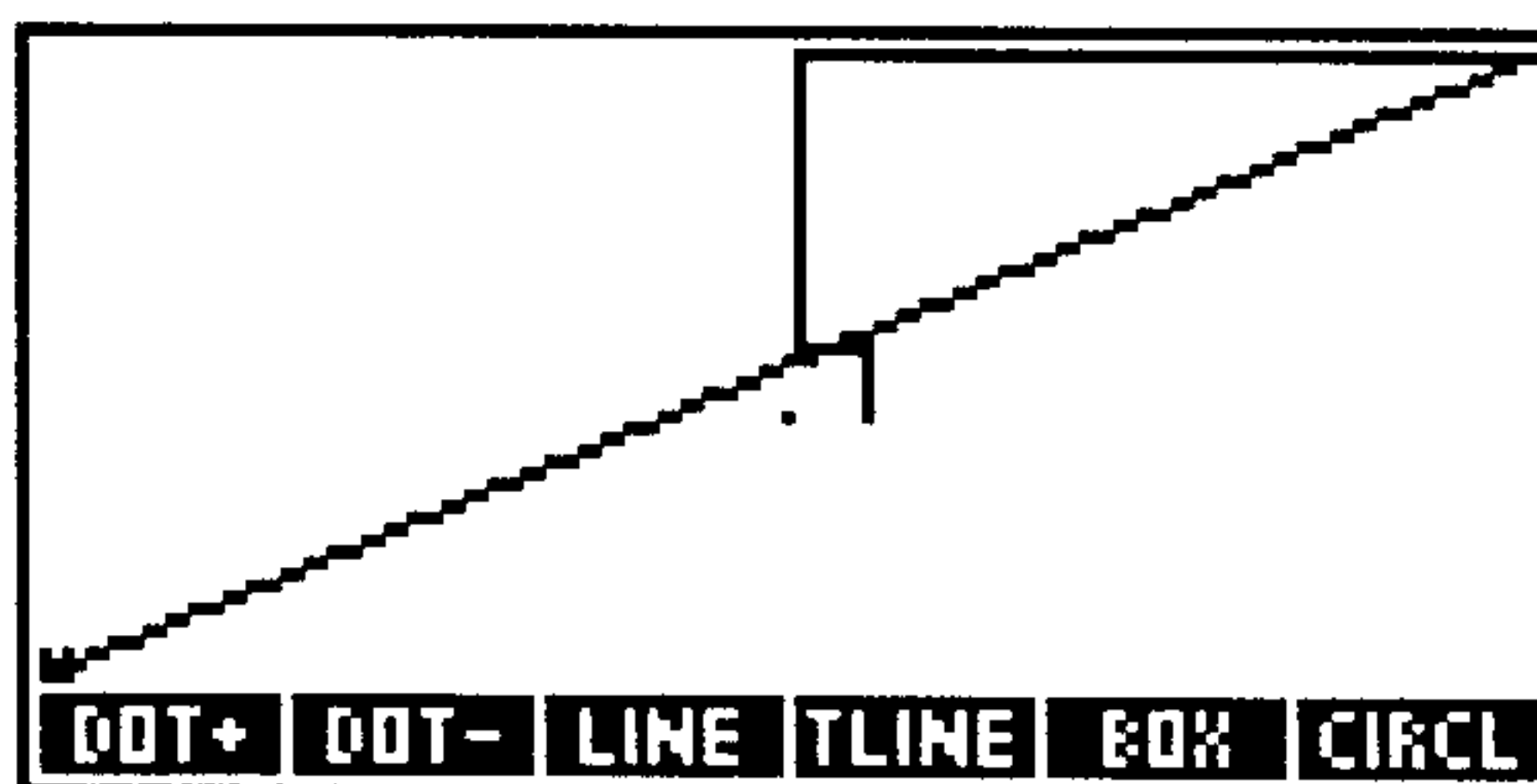
draws lines like this:



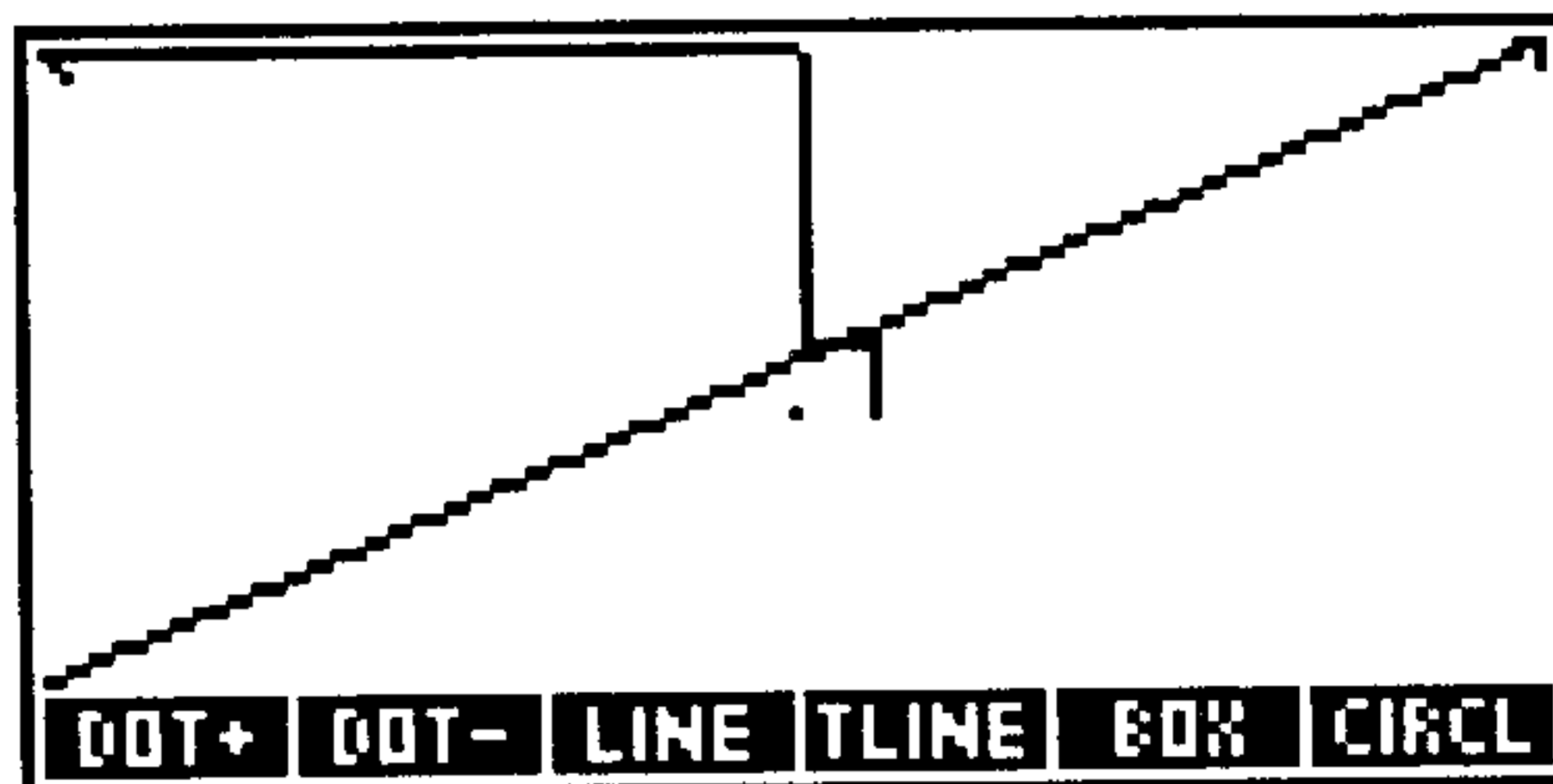
Notice that the DOT+ key label contains a white square when the line drawing mode is active. Pressing DOT+ when the square is visible turns the mode off and removes the square from the label.

DOT- turns the cursor into a pixel “eraser.” Pressing this key once erases the pixel at the cursor center, and turns on a white square in the key label (turning off the square in the DOT+, if present). While in this mode, the cursor erases pixels as you move it. Pressing DOT- a second time deactivates the eraser mode.

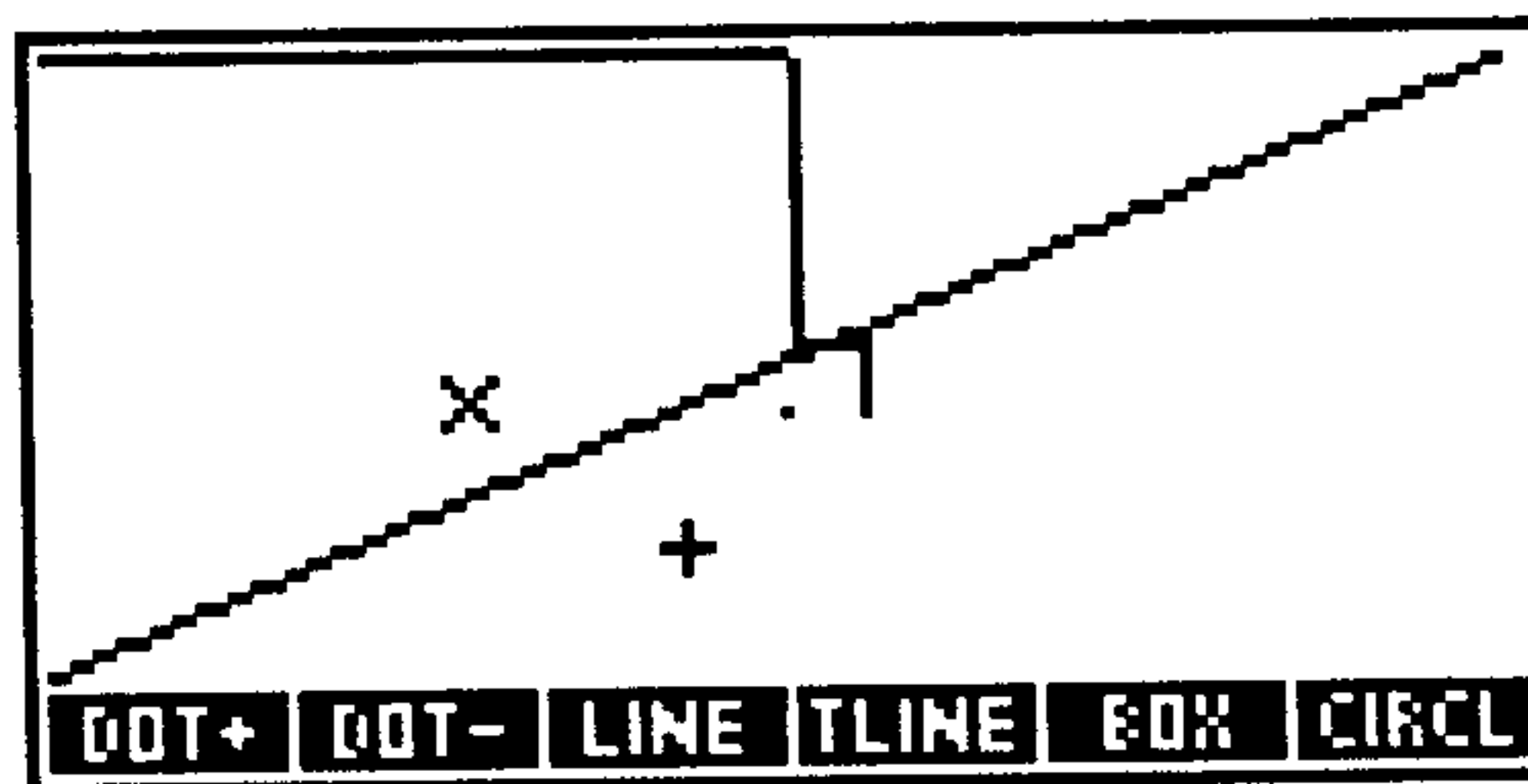
For drawing lines in arbitrary directions, LINE is faster and more accurate than using DOT+. LINE connects the cursor and the mark with a straight line. The mark is automatically moved to the cursor position, which makes it easy to draw a series of connected lines:



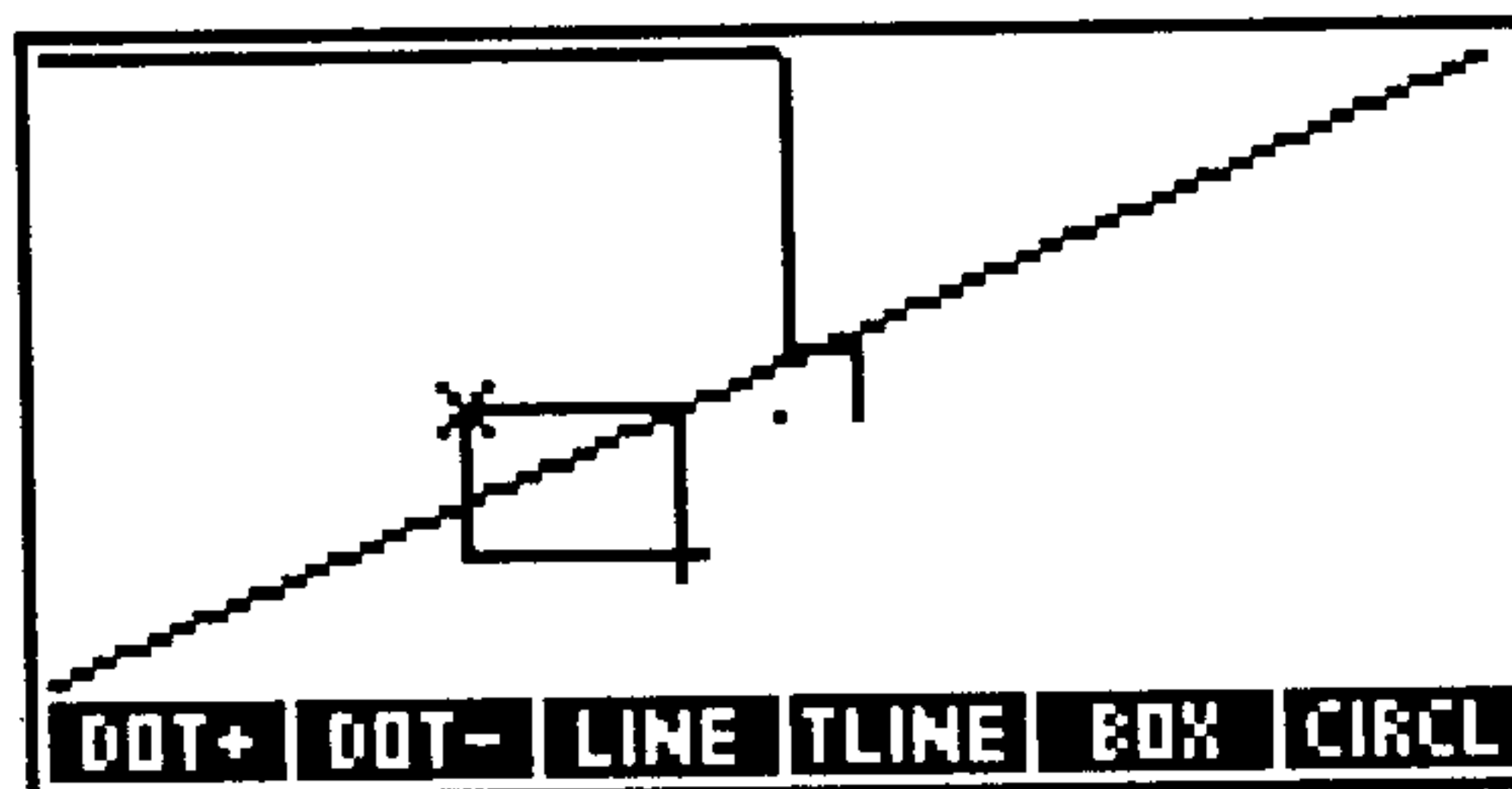
For drawing lines through already-dark regions, TLINE reverses the pixels along the line connecting the cursor and the mark:



BOX is the interactive counterpart of the programmable BOX, drawing a rectangle defined by the cursor and the mark at opposite corners. With the cursor and mark positioned like this:

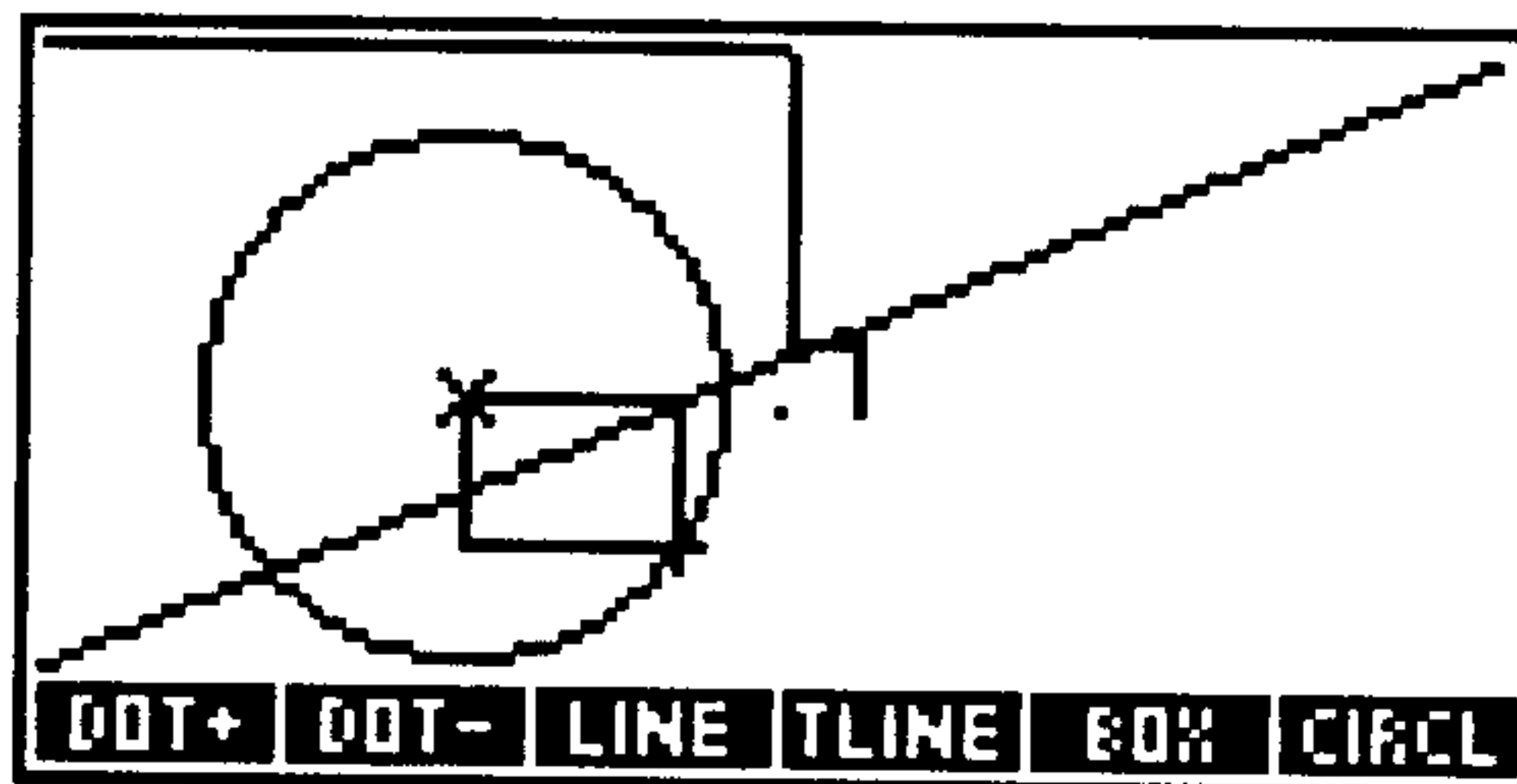


BOX draws a rectangle:



The final simple drawing tool is **CIRCL**, which is the interactive version of ARC. This

operation nominally draws a circle, with its center at the mark. For example, with the cursor and mark still on the corners of the box just drawn, CIRCL draws this circle:



To draw only a portion of a circle, place the mark at the center of the circle, and the mark at the start of the arc you wish to draw. Then press CIRCL; when the arc is drawn as far as you want, press ATTN to stop the drawing.

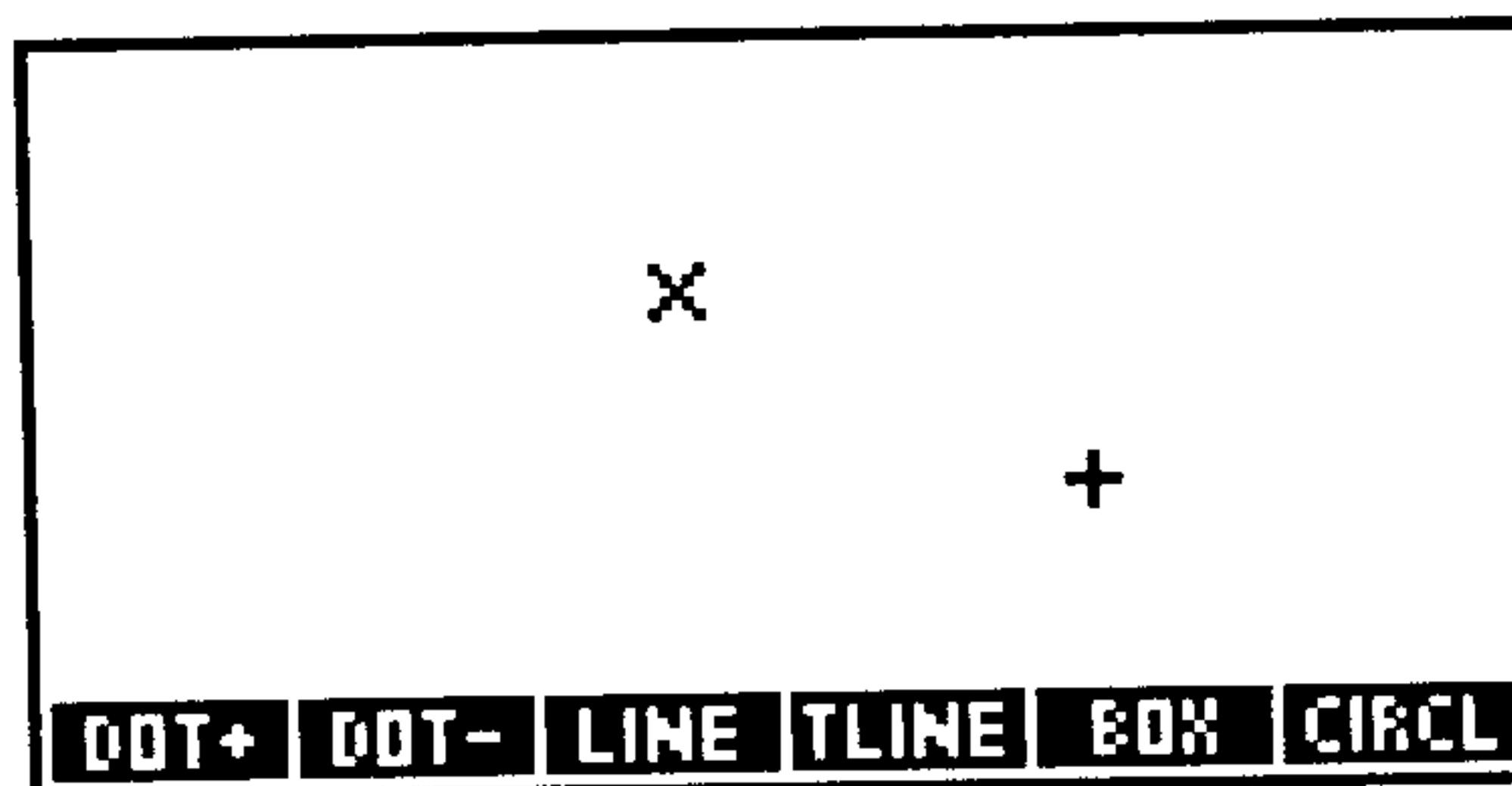
15.3.4 Working with Graphic Objects

The graphics operations embodied in the commands STO, SUB, and REPL, as they apply to the graph screen via PICT (section 10.3.3), are available as interactive operations in the plot environment menu. As for the pixel commands described in the previous section, the mark and cursor are used in lieu of the screen coordinates used as command arguments.

Pressing STO in the plot environment enters a copy of the graph screen into stack level 1 as a graphics object. The object contains the full graph screen picture, but without the menu labels, the cursor, or the mark. The action of STO in the plot environment is apparently reversed from its action using PICT in the standard environment. That is, plot environment STO is equivalent to the command sequence PICT RCL. This reversal arises from the “direction” of the store: in the standard environment, the stack object is the visible focus, and you store that object into the graph screen (PICT). In the plot environment, the graph screen is visible, and STO stores the picture onto the (invisible) stack.

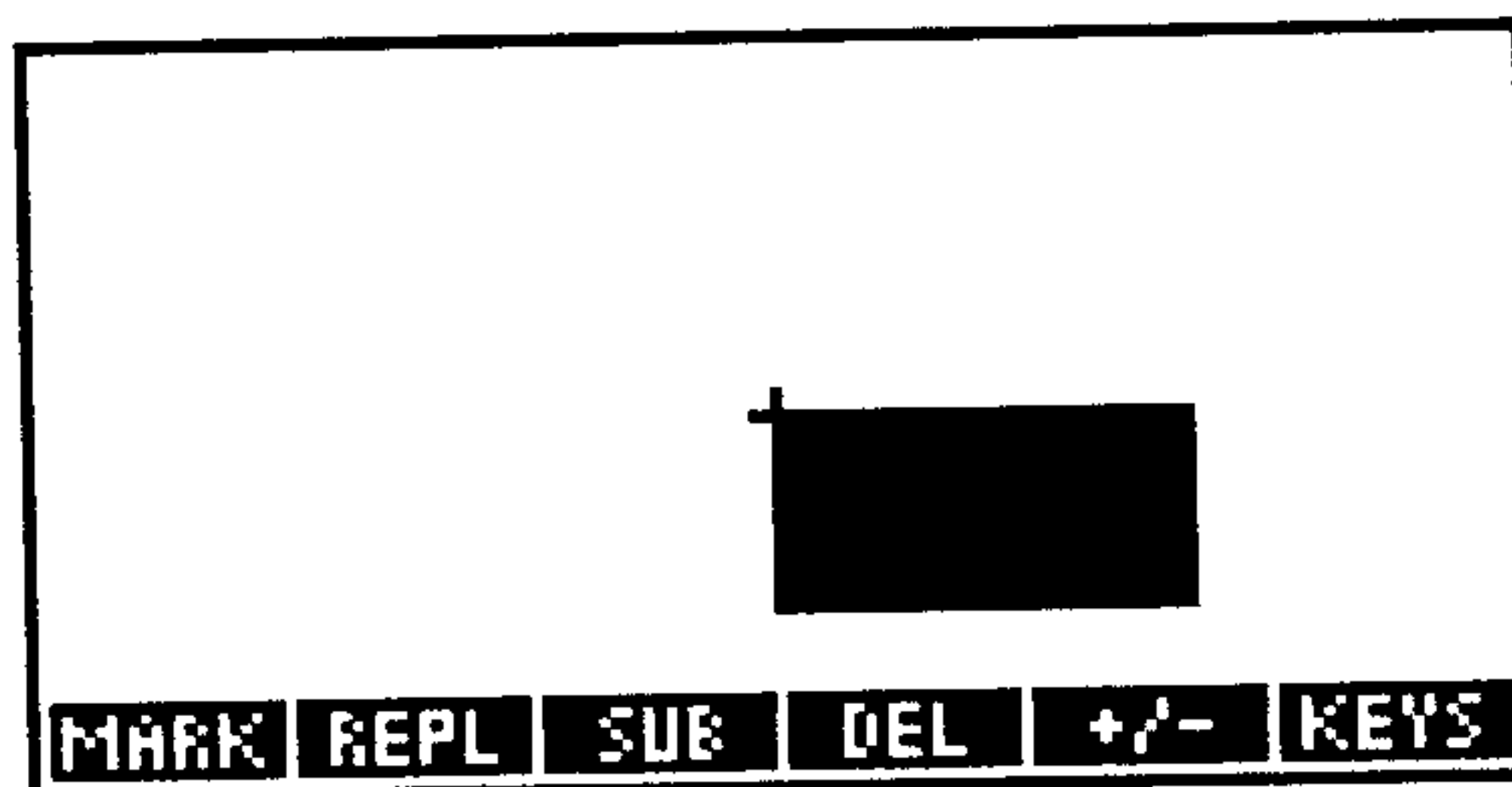
RCL does not work in the plot environment, but you can bring a graphics object from the stack to the graph screen by using REPL, in the third page of the menu. This operation takes a graphics object from the stack and writes it onto the graph screen, replacing the screen region with the cursor at its upper-left corner. SUB provides the reverse operation; it copies a rectangular region defined by the mark and the cursor to the stack as a graphic object. These operations are easily illustrated by a simple example. Starting in the plot environment with a blank screen (use CLR), place

the cursor and the mark as follows:



Then

SUB ATTN +/- < NXT NXT REPL :



SUB copies the region bounded by the mark and the cursor to the stack as a graphics object, where +/- (NEG) reverses the image. Then, back in the plot environment, REPL replaces a region below and to the right of the cursor, in this case with a dark rectangle. The region affected by REPL is the same size as the stack object, except that any rows or columns in the graphics object that would extend beyond the edges of the graph screen are truncated.

15.4 Function Plots

The plot of $\sin x$ used as an example in section 15.2 is a typical example of a *function plot*. In this type of plot, the y-coordinate is a *function* of the x-coordinate, where each value of x yields a single value of y . In particular, the x -coordinate is incremented from the left edge of the graph screen to the right, in steps determined by the resolution parameter (section 15.2.7); one point is plotted in each column corresponding to the

successive values of x . If the resolution is a real number, x is incremented by that amount; if it is a binary integer $\#n$, x is incremented by n pixel columns. (The default value of 0 is equivalent to a value of $\#1$, for plotting in every column.)

If the *curve-fill* flag (-31) is clear, computed points separated by more than one pixel in either direction are connected by a "straight" lines to make continuous plotted curves. Although plotting these *filled curves* is the default choice, you should use some care in using that option, since it causes many points to be plotted that are not computed directly from the current equation, which can be misleading.

The function $f(x)$ that determines the y -coordinates of plotted points is specified by the current equation. In its simplest form, this object is an expression in the independent variable x . However, the current equation may also be any of the following:

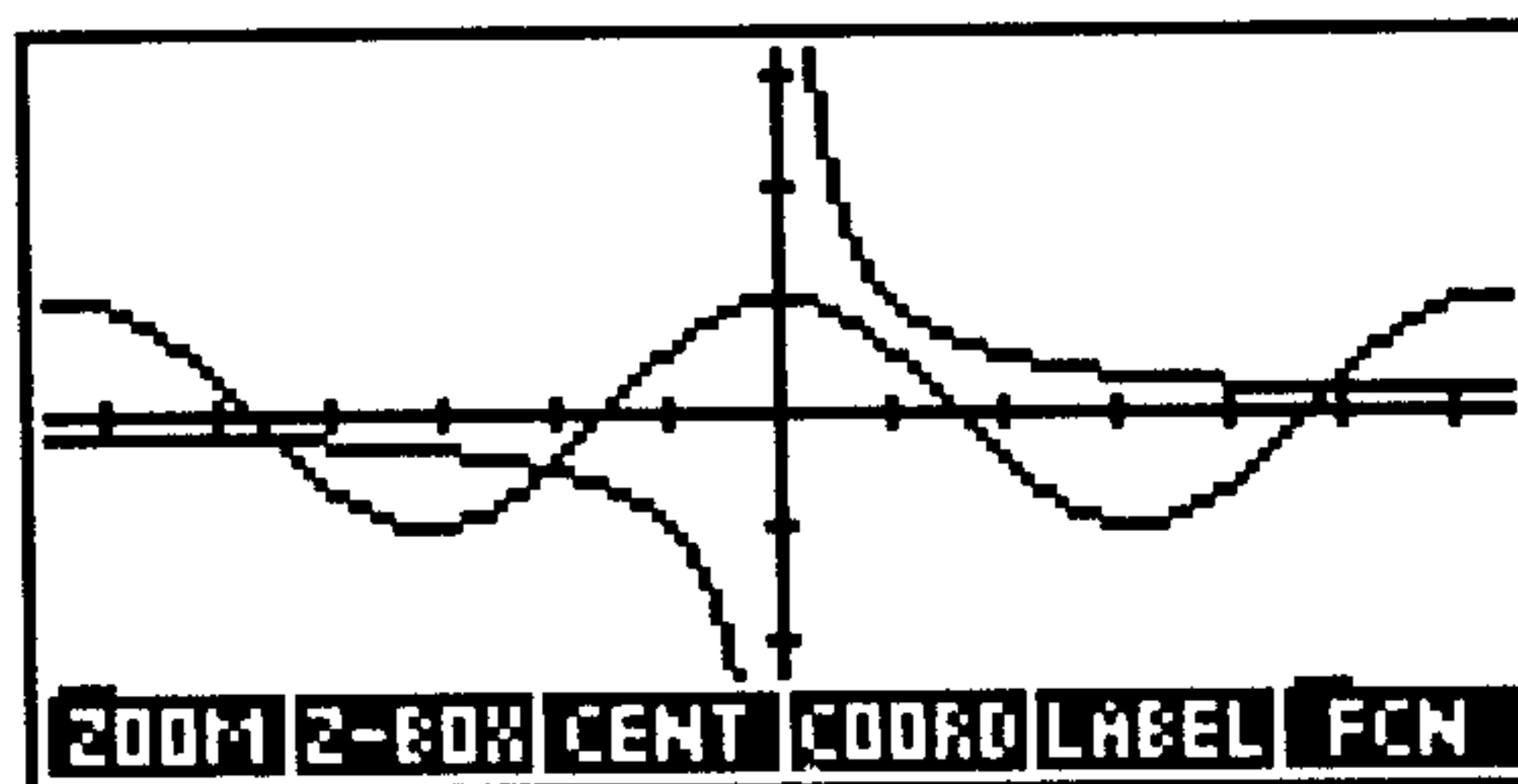
- *Multi-variable expressions.* HP48 expressions can contain any number of variables, e.g. ' $\text{SIN}(A+B*X)^C$ '. One variable is designated (by INDEP) as the independent variable; you must supply values for the remaining variables so that the expression evaluates to a number. Note that since HP Solve uses the same current equation in EQ, you can use the solver menu (section 14.0) to store values for all of the variables.
- *Programs.* Any program equivalent to an expression (takes no arguments from the stack, and returns one number) can be used instead of an expression.
- *Definition Equations.* When plotting functions, we commonly speak of plotting $y=f(x)$, where $f(x)$ is the function, and y is little more than a label for the vertical axis. To accommodate this convention, when the current equation object is an equation of the form ' $\text{name}=\text{expression}$ ', and flag -30 is clear, DRAW graphs as if the object were just ' expression '. Thus, plotting ' $F=\text{SIN}(X)$ ' or ' $G=\text{SIN}(X)$ ' produces the same curve. The only difference is that LABEL labels the vertical axis with F in the first case and G in the second. Setting flag -30 cancels this special treatment of defining equations, which are then plotted as two separate expressions, as described below.
- *Vertical lines.* When the current equation has the form ' $x=\text{constant}$ ', where x is the independent variable, and flag -30 is set, DRAW plots a straight vertical line. With flag -30 set, DRAW plots a horizontal line at $y=\text{constant}$, and a diagonal line $y=x$.
- *Other Equations.* When the current equation is a true *equation*, that is, two expressions combined with an "=" sign, the HP 48 graphs the two sides of the equation as separate curves, regardless of the state of flag -30.

The treatment of equations as separate expressions arises from the deliberate

connection between the HP 48's plotting and solving systems. The solutions obtained by HP Solve correspond to the intersection points of the two curves. This does provide a simple method of graphing two expressions simultaneously; you can form an equation by setting the two expressions equal to each other, then graph the equation. For example,

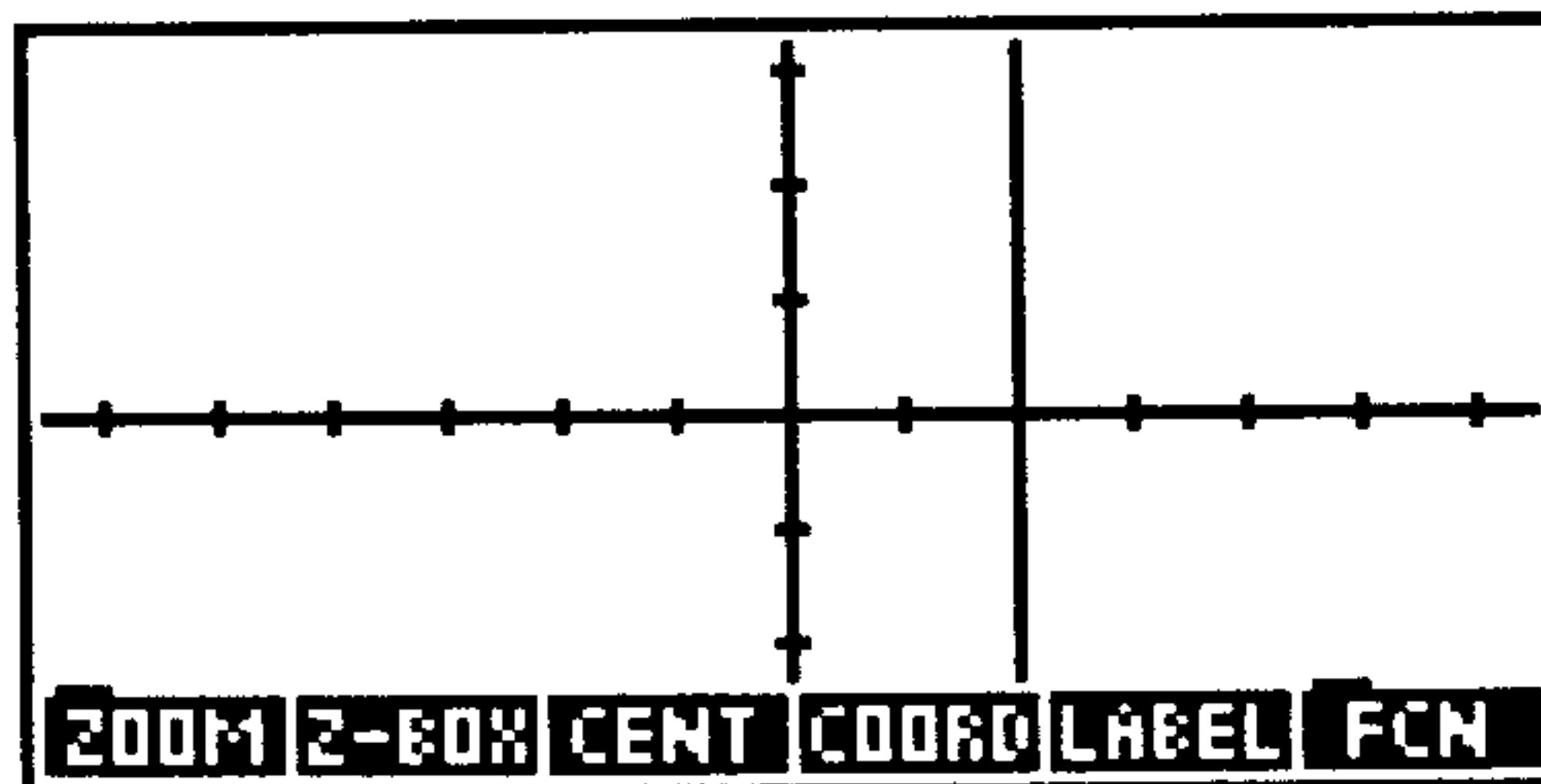
RAD $\boxed{\rightarrow}$ **PLOT** **NXT** $\equiv$ **RESET** $\boxed{\leftarrow}$ **PREV**
 'COS(X)=1/X' $\boxed{\leftarrow}$ $\equiv$ **DRAW** $\equiv$ **ERASE** $\equiv$ **DRAW**

yields the following display:

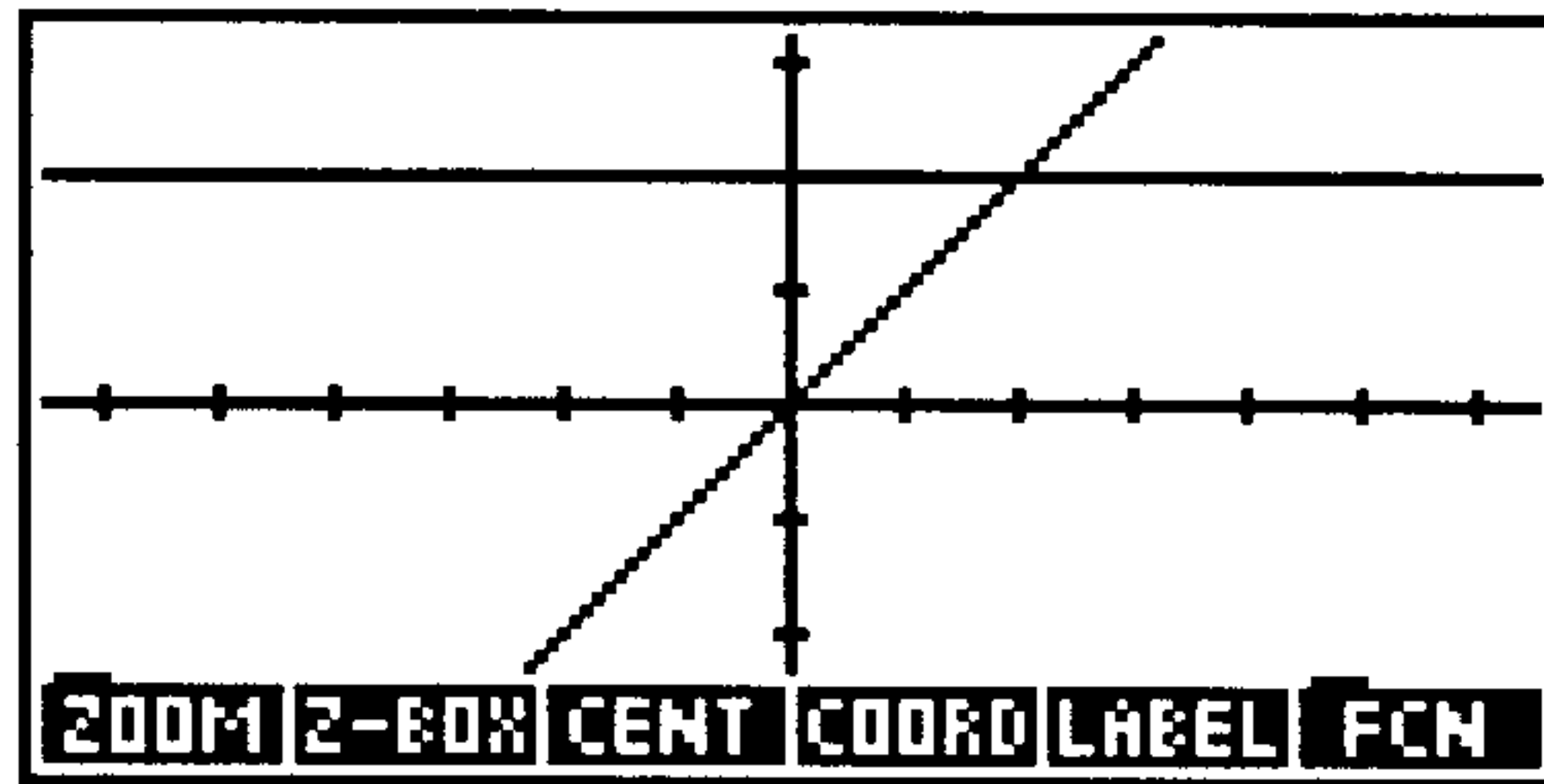


Note that the two expressions are plotted simultaneously, unlike the case when EQ contains a list of expressions, for which the curves are plotted sequentially.

Although function plots are nominally designed to plot y as a function of x , DRAW does make special provision for current equations of the form $x = \text{constant}$, where x is the independent variable. When flag -30 is clear, such equations are plotted as vertical lines with x -intercept at *constant*. Thus 'X=2' plots as



If flag -30 is set, the same equation plots as



The horizontal line is the right side of the equation, representing the line $y=2$. The diagonal line is the right side, corresponding to $y=x$.

15.4.1 Plotting Programs

As stated previously, a program used as a “current equation” should execute as if it were an algebraic object, taking no arguments and returning one real number. However, as long as the program follows this input/output convention, it can perform any number of additional operations on the stack or memory during its execution. There are two restrictions:

- The program can not halt using **HALT** or **PROMPT**. Executing either of these commands causes the **HALT Not Allowed** error.
- After completing its plot, **DRAW** restores the stack to the same number of objects that were present when it started (this is done as part of **DRAW**’s error-handling process). If a program is to return extra results that you want to keep after plotting, it must store them in a variable or use some other strategy to protect them from the stack reset.

You should note that a user-defined function (section 8.5) is a program, and moreover, one that uses stack arguments. Therefore, you can not plot a user-defined function by storing the program or its name in **EQ**. Instead, you must include it as part of an expression, with the independent variable as its argument. That is, if you have a user-defined function named **F**, you can plot '**F(X)**', but not '**F**' by itself.

The programs listed next give examples of what you can do by using a program as the current equation object. The first, **MANPLOT**, lets you plot points manually, one at a time. When you execute **DRAW** with **MANPLOT** in **EQ**, you will see the graph screen briefly, then a display like this:

RAD	PRG
{ HOME }	
X: -6.5	
:Y:	
ERASE	DRAW AUTO XANG YRNG INDEP

At this point, you type the y value corresponding to the x value in the prompt, and press **ENTER**. DRAW plots the specified point, then prompts you for the next one, and so on.

MANPLOT	Manual Plot Program	CE3F
<pre> << PPAR 3 GET IF DUP TYPE 5 == THEN 1 GET END -STR 2 OVER SIZE 1 - SUB ":" + X + ":" PPAR 7 GET IF DUP TYPE 5 == THEN 1 GET END -STR 2 OVER SIZE 1 - SUB OVER + + { V } + INPUT OBJ→ EVAL { #0 #0 } PVIEW >> </pre>		Get the independent variable name. Convert to a string. Append a colon. Append the current x value. Get the dependent variable name. Convert to a string. Surround with colons. Prompt for the next y. Convert string to a number. Show the graph screen

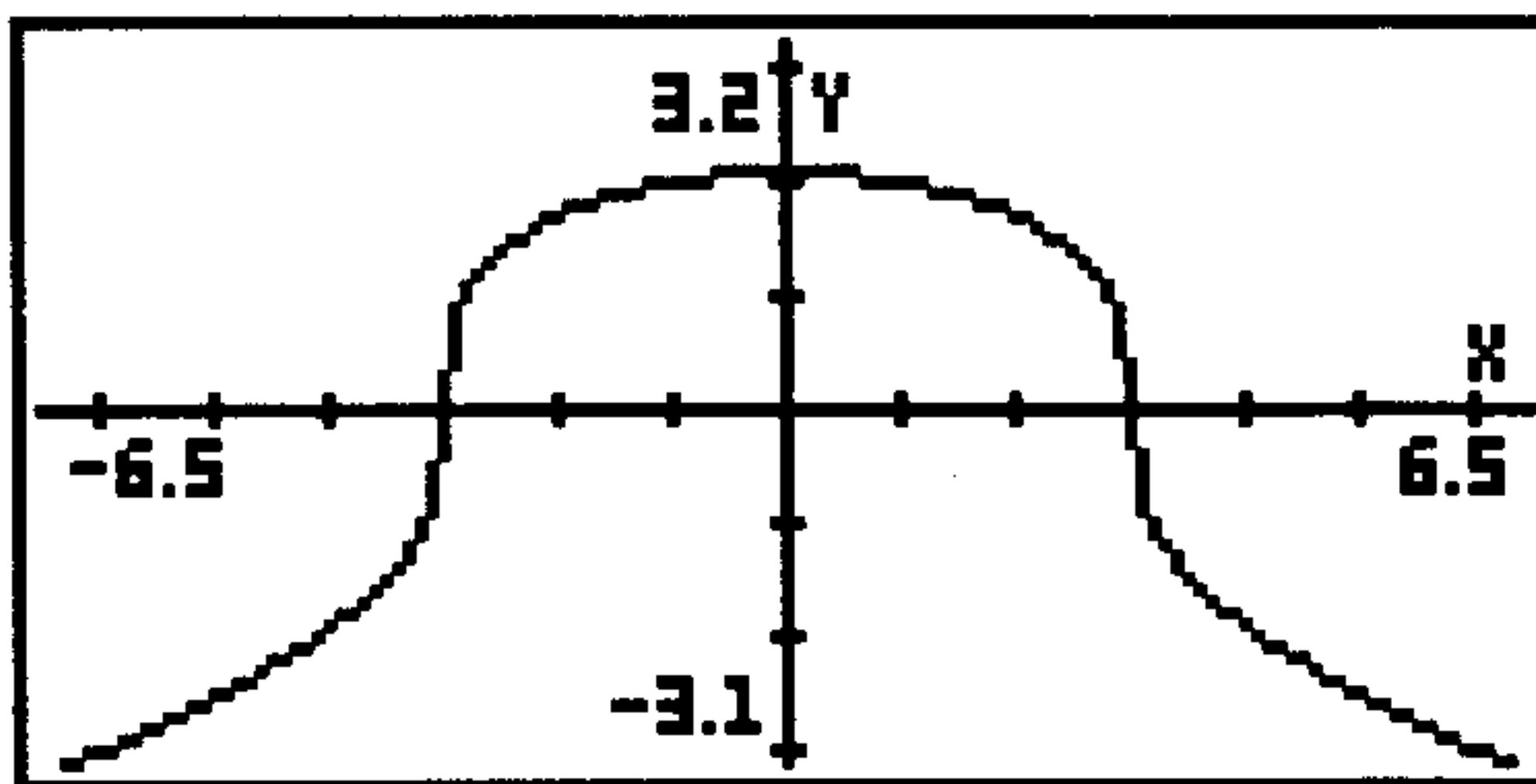
The next program, SCDRAW, is a replacement for DRAW that saves the coordinates of all of the computed points as it plots them. Upon completion, SCDRAW returns a list containing all of the points, in order. SCDRAW is designed for function plots where the current equation is an expression or a program. If the current equation is an equation, the y-coordinates returned by SCDRAW will be the differences between the y-coordinates computed from the the left and right sides of the equation.

SCDRAW <i>Save Coordinates and Draw</i> E5A3	
<i>level 1</i>	
☐ { <i>coordinates</i> }	
<pre> << (0,0) PIX? DROP PPAR 3 GET IF DUP TYPE 5 == THEN 1 GET END 'EQ' RCL → ind eq << {} IFERR << eq →NUM ind EVAL OVER R→C ROT SWAP + SWAP >> 'EQ' STO DRAX DRAW THEN 1 ELSE 0 END eq 'EQ' STO IF THEN ERRN DOERR END >> >> </pre>	Get plot parameters. Get independent variable name. Save in local variables. Start with an empty list of coordinates. Evaluate the current equation (y). Current x value. Combine x and y into a complex number. Add to coordinate list. Return y value for DRAW. Replace EQ with program. Make the graph. Signal that an error occurred. Restore EQ. Report any error.

SLVDRAW uses an approach similar to that of SCDRAW, in this case to let you plot the solutions of an equation, rather than treating an equation as two independent expressions. For each value of the independent variable, SLVDRAW uses ROOT to find a value of the dependent variable, and plots the corresponding point. Before plotting, you must store in the dependent variable an estimate of its value at the left end of the screen; subsequently, each solved value is used as an initial guess for the next solution. As an example,

```
'X^2+Y^3=9' STEQ -3 'Y' STO SLVDRAW
```

with the default plot ranges yields this plot (after adding labels):



SLVDRAW

Solve and Draw

99B1

```

<< RCLF -31 CF
  IFERR 'PPAR' RCL
  THEN 1 *H RCL
  END
  7 GET
  IF DUP TYPE 5 ==
  THEN 1 GET
  END
  'EQ' RCL → flags dep eq
  << IFERR
    << eq dep DUP EVAL ROOT
    >> 'EQ' STO
    DRAX DRAW
  THEN 1
  ELSE 0
  END
  eq 'EQ' STO flags STOF
  IF THEN ERRN DOERR END
  >>
>>

```

Activate argument recovery.

Get PPAR list.

Get dependent variable name.

Save in local variables.

Solve for the dependent variable.

Replace EQ with program.

Make the graph.

Signal that an error occurred.

Restore EQ and flags.

Report any error.

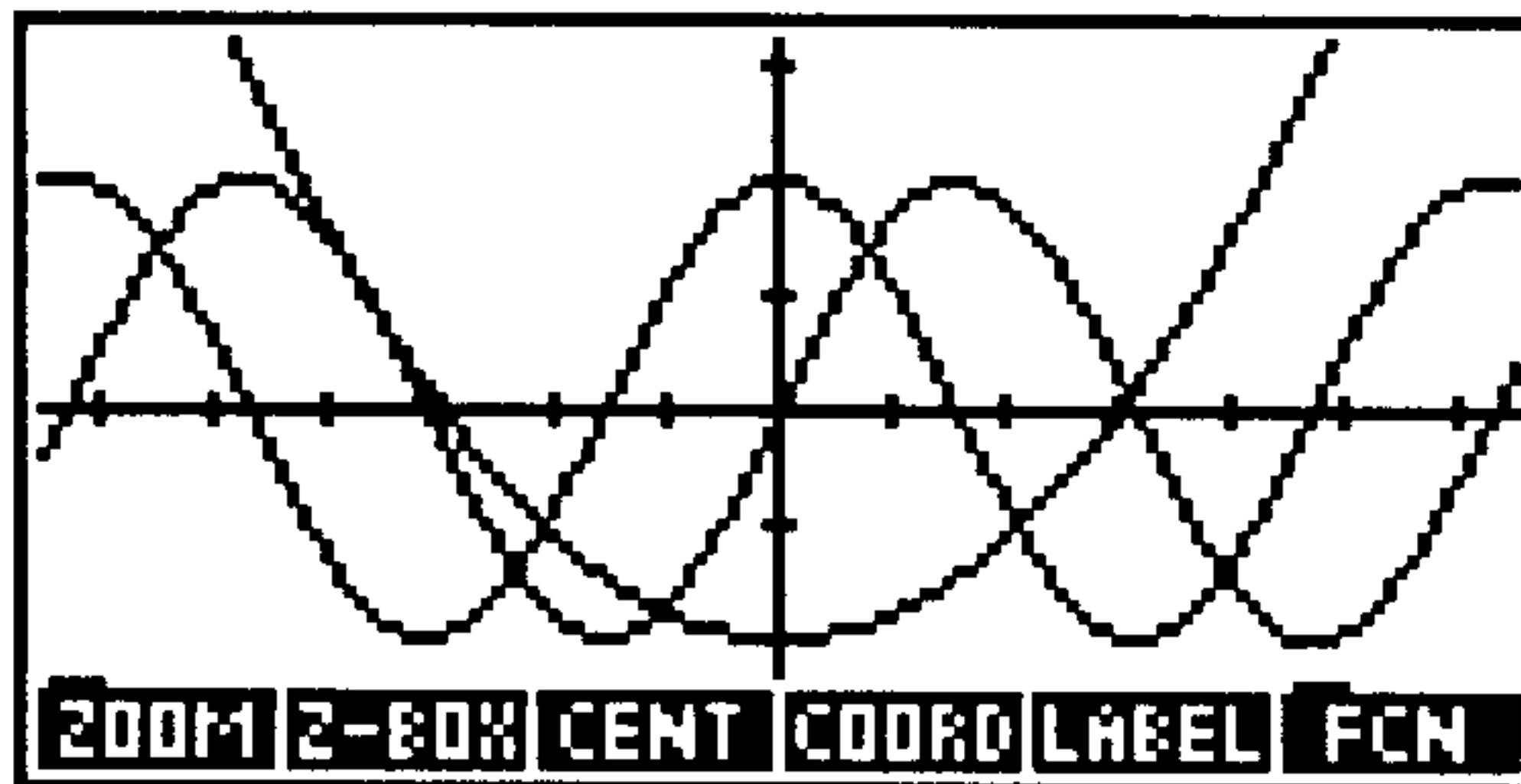
15.4.2 The Function Menu

The **FCN** menu key in the plot environment activates a sub-menu of operations that are applicable to function-type plots (for other plot types, **FCN** flashes Invalid PTYPE). In general, the operations are designed for extracting information from a function curve at one or more points, where the points in question are specified by means of the cursor.

To help in exploring the operations, we will use a list of expressions (stored as LIST.EQ) as the current equation:

{ 'SIN(X)' 'COS(X)' '(X/3)^2-1' } NEW LIST ENTER
 'PPAR' PURGE -1.55 1.6 YRNG RAD ERASE DRAW

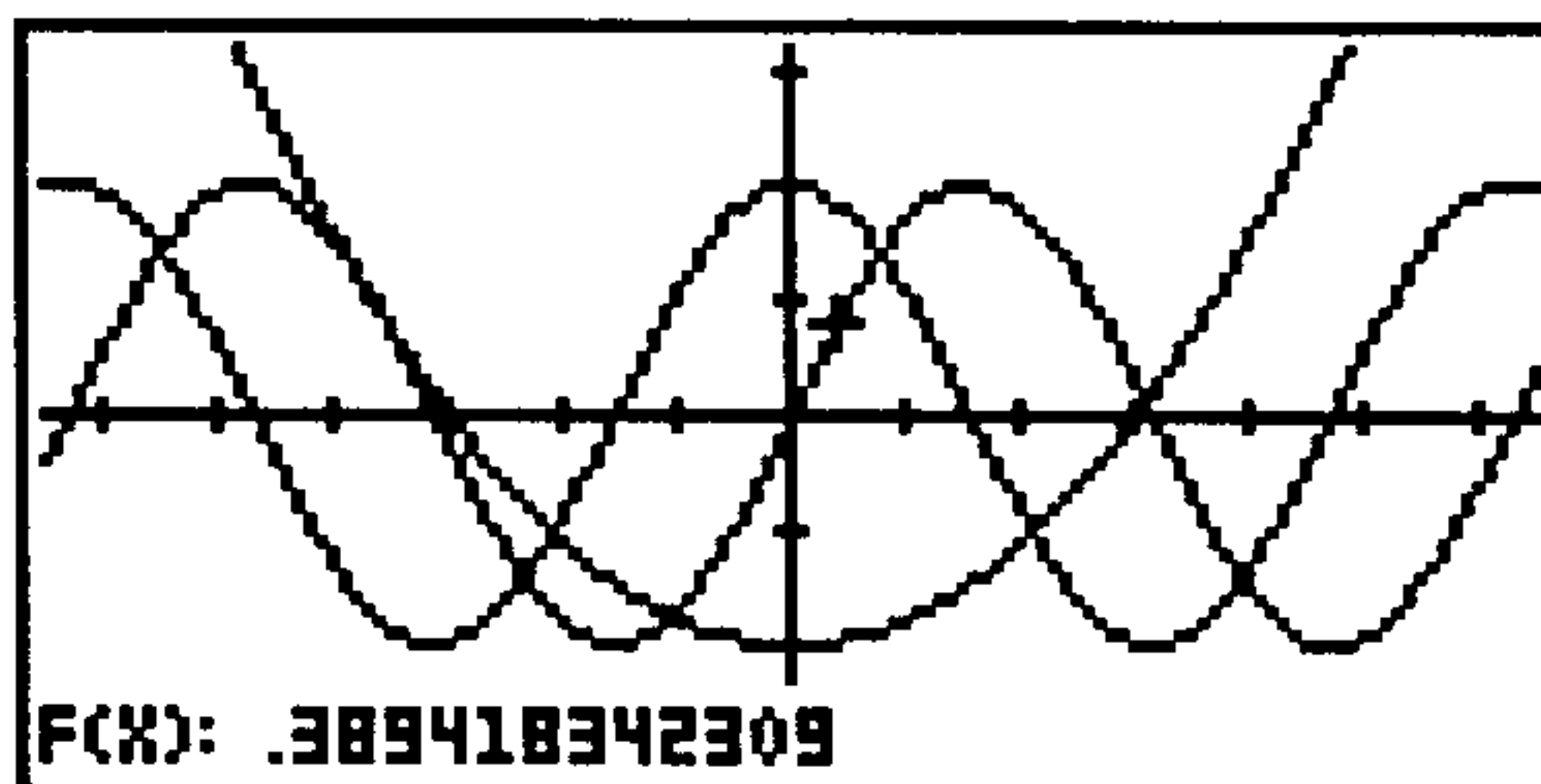
then produces this picture:



When the current equation is a single object, all of the function menu operations are directed to that object. If you are working with a list of objects, all of the operations select the first object in the list; ISECT may also use the second object. To determine which object is the first in the list, you can either

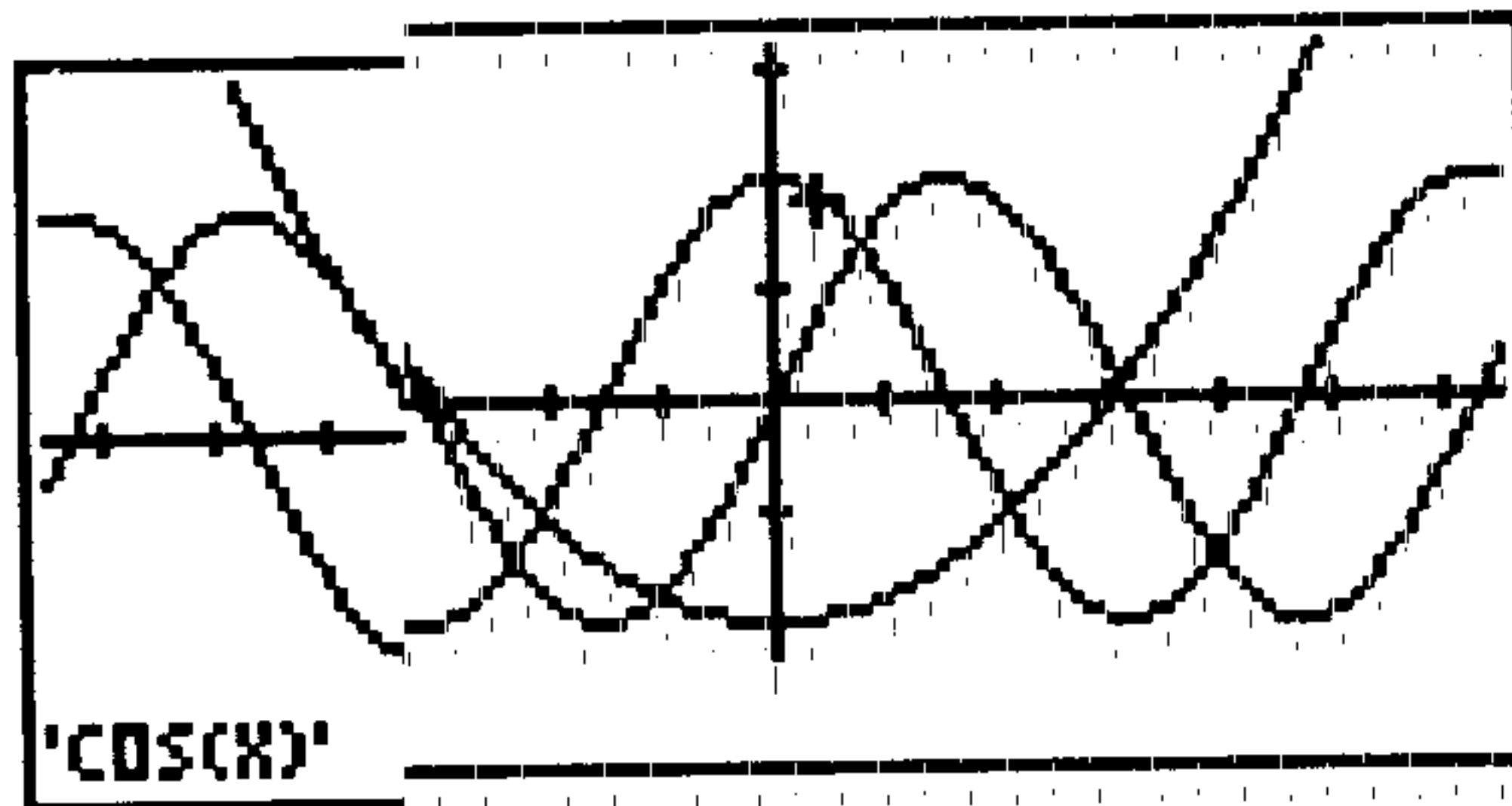
- press ← REVIEW to show the beginning of the equation list, which is usually enough to identify the first object; or
- (in the FCN menu) press NXT F(X). This displays the value of the selected object at the horizontal position of the cursor, and moves the cursor to the selected curve.

In the current example, moving the cursor a few pixels to the right, then pressing F(X) moves the cursor to the 'SIN(X)' curve:



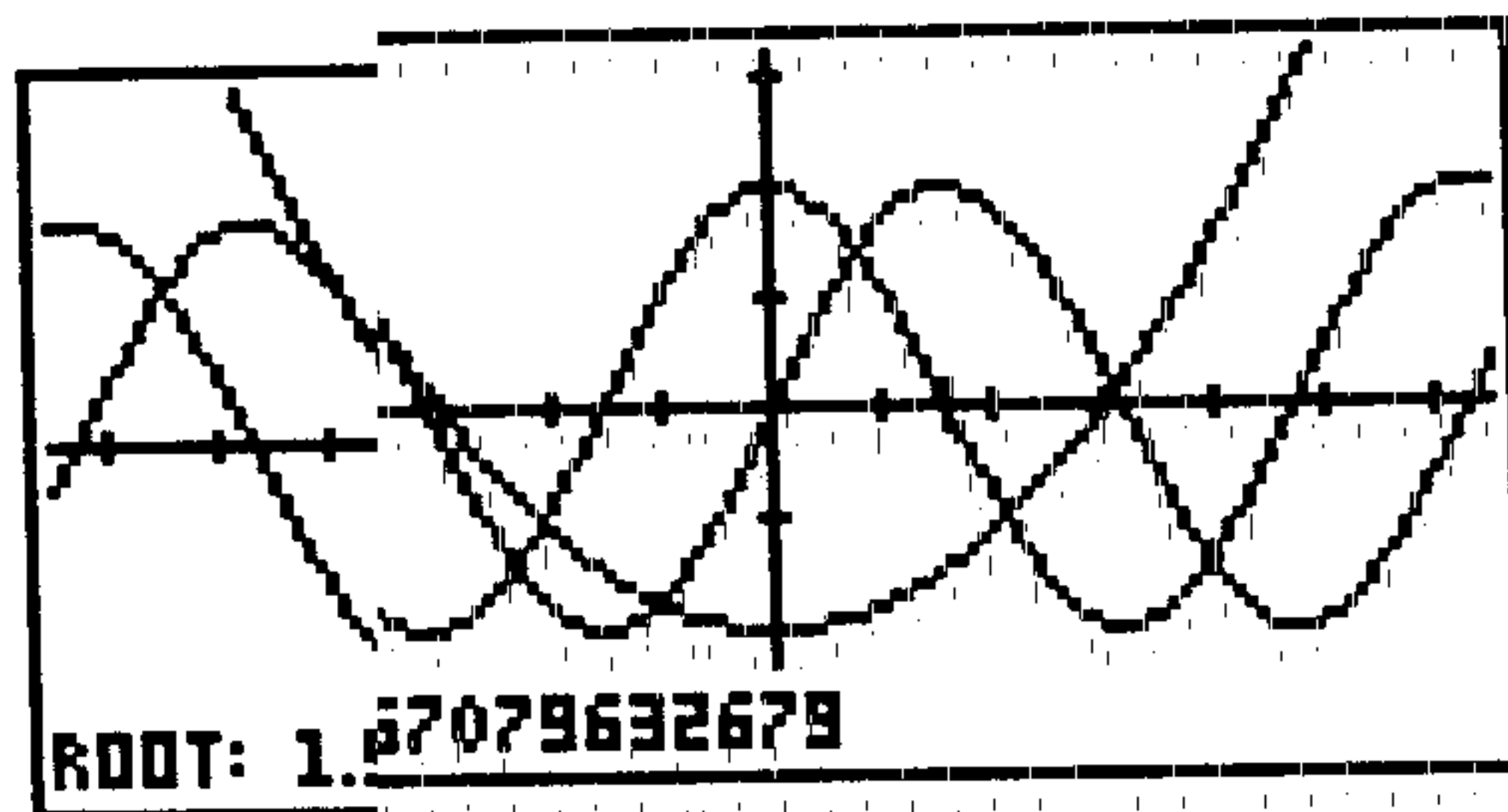
Plotting

At the end of the second menu page is NXEQ, which selects the next object in the current equation list:



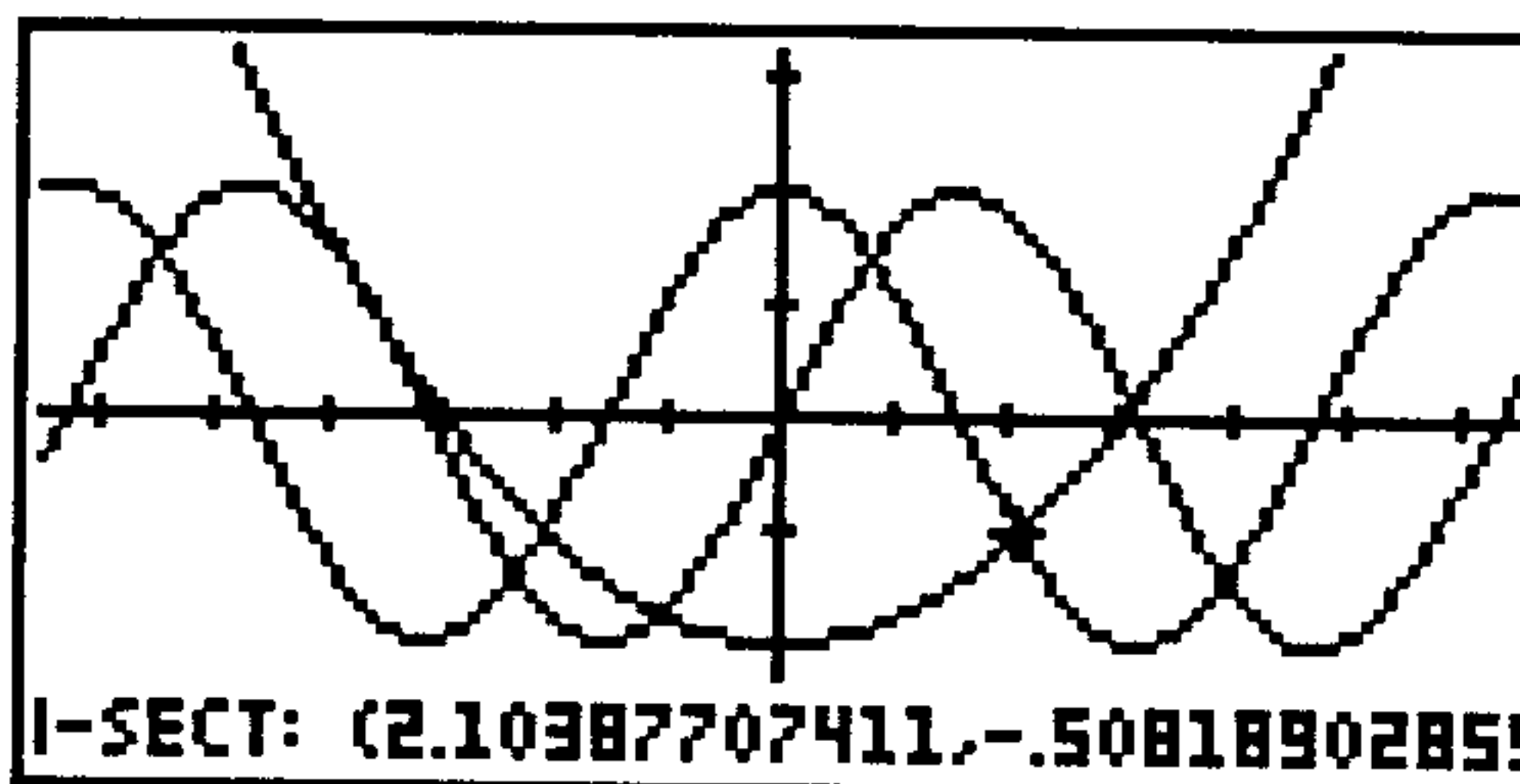
Notice that NXEQ also displays the newly selected object at the bottom of the screen. Using this operation, you can cycle through all of the objects in the current equation list. It works by actually moving the first object to the end of the list, so that the second object moves to the front.

Returning to the first page of the menu, pressing ROOT finds the zero of $\cos(x)$ at $\pi/2$:



ROOT invokes the root-finder on the selected object, using the horizontal cursor position as an initial guess. When it has finished, it displays the result at the bottom of the screen and move the cursor to the root.

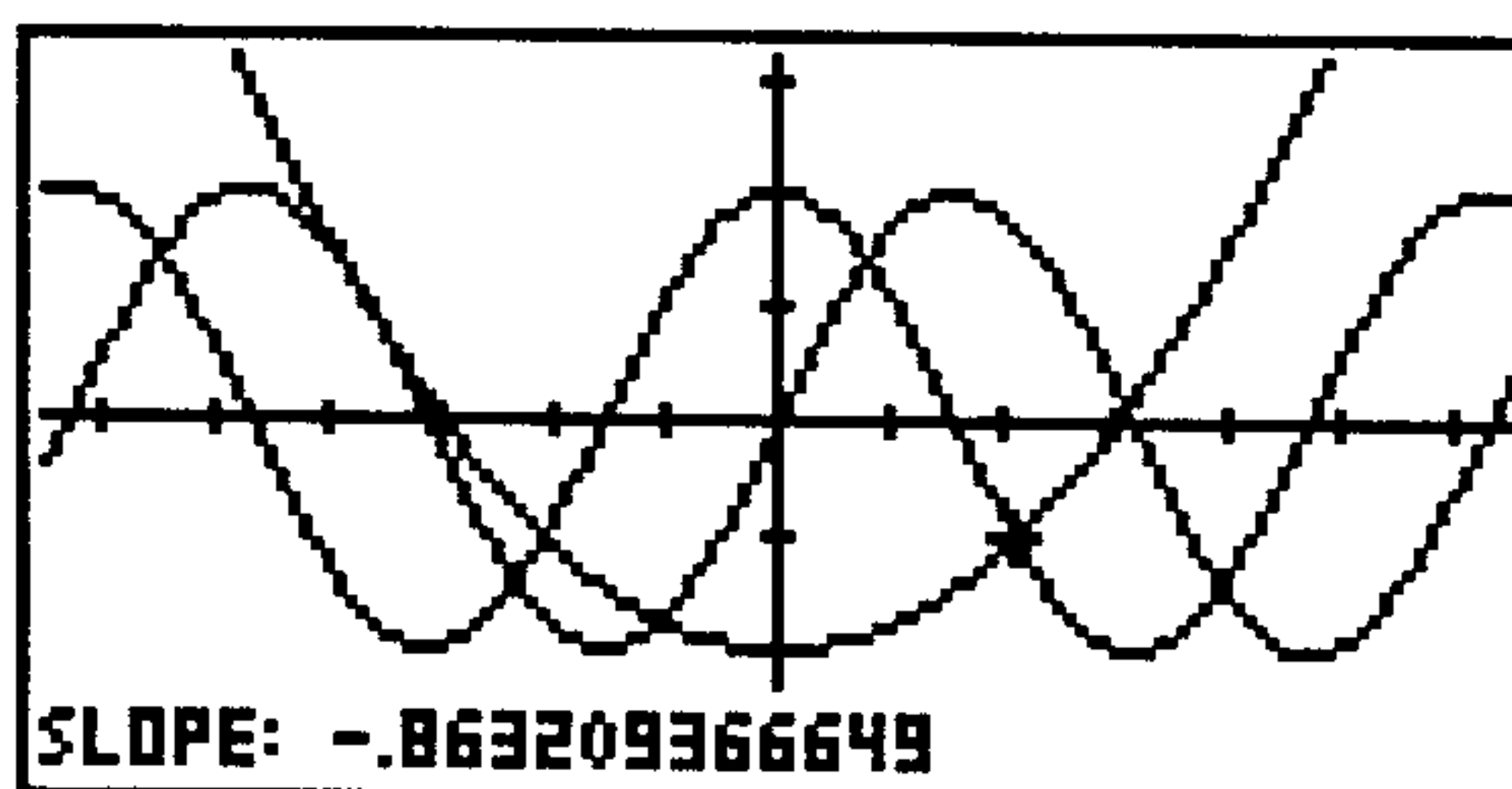
Now press any key to restore the menu, then ISECT:



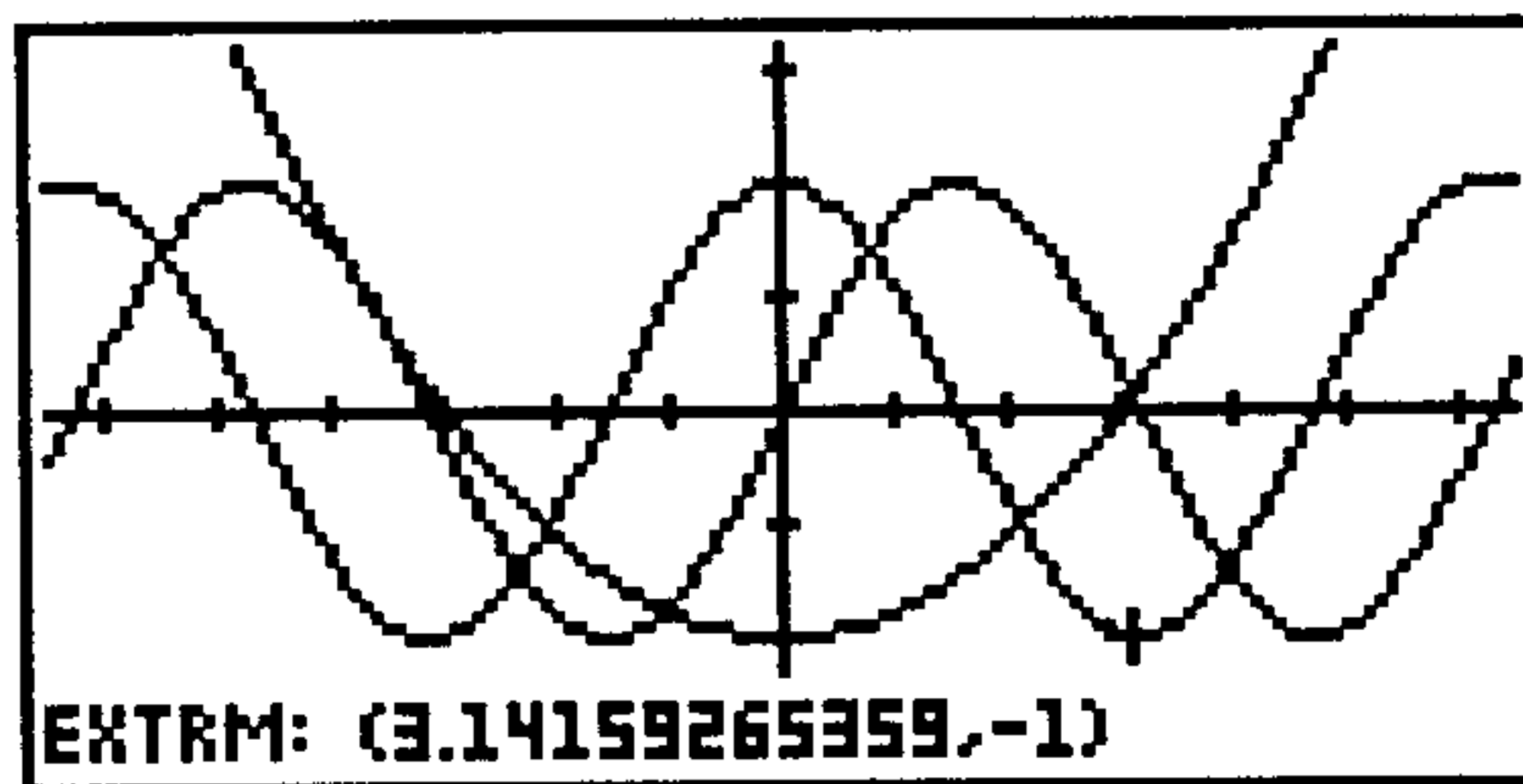
ISECT finds the intersection of the curves corresponding to the first two objects in the current equation list:

- When the first object in the list is an actual equation, ISECT finds the intersection of the two expressions that comprise the left and right sides of the equation; ISECT is the same as ROOT in this case.
- When the first two objects are both expressions, ISECT finds their intersection by combining them into an equation and invoking the root-finder.
- When the first object is an expression, and the second is an equation, ISECT substitutes the first expression for the left side of the equation, and solves the resulting equation.

The next entry in the function menu is SLOPE. This operation computes the slope of the selected curve at the x -position of the cursor:



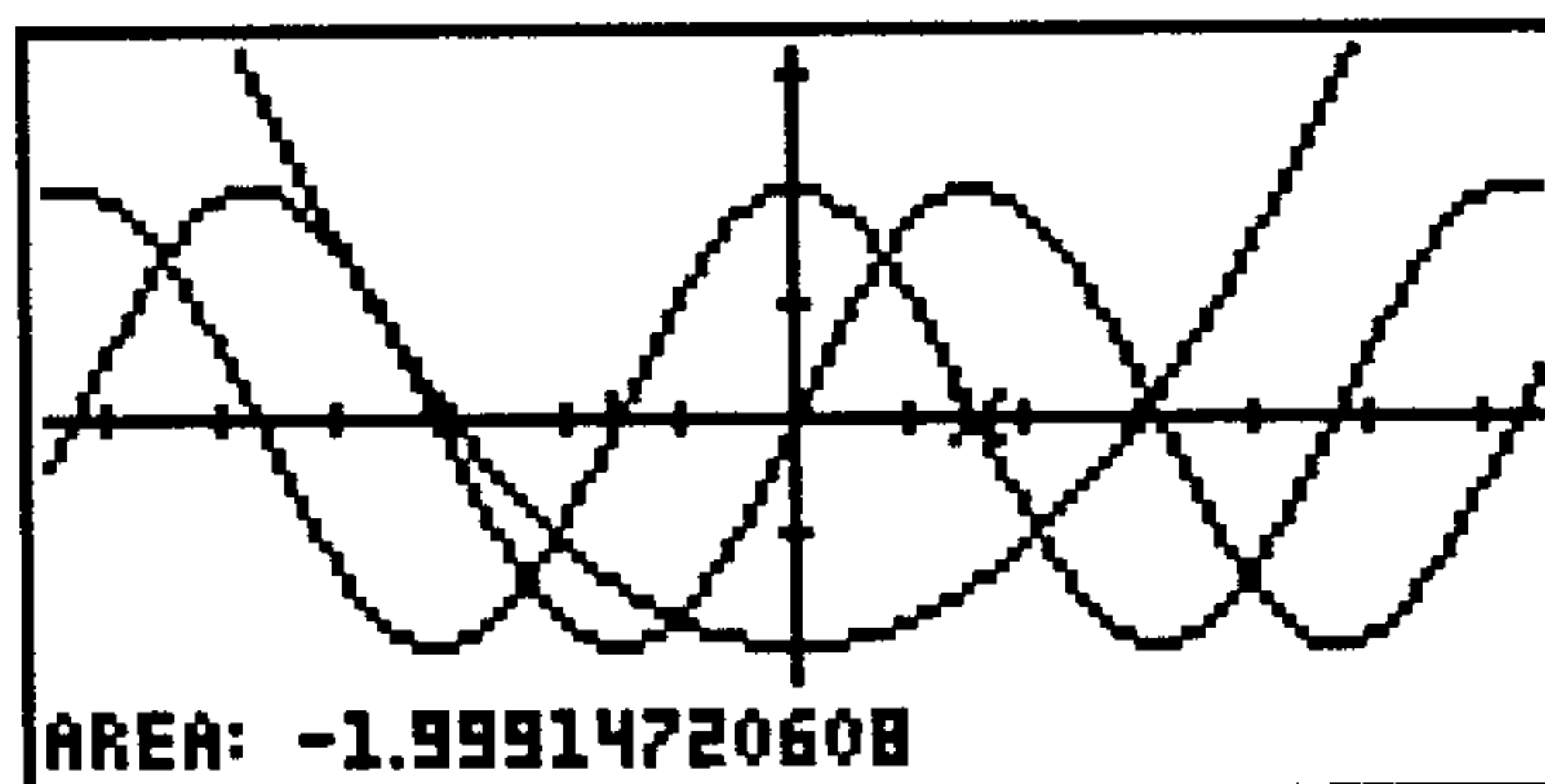
It does this by symbolically differentiating the selected object, then numerically evaluating the result. Differentiation is also used by EXTR, which finds the critical points of a curve. Pressing EXTR here finds the local minimum of the cosine curve at π :



EXTR differentiates the selected object, then applies the root-finder to the result to find a critical point where the derivative is zero.

AREA uses the HP 48's numerical integration facility (section 18.4.2) to find the area of a curve between two x -values selected by the cursor and the mark. For example, to find the area of the "hump" of the cosine curve between its zeros at $\pm\pi/2$:

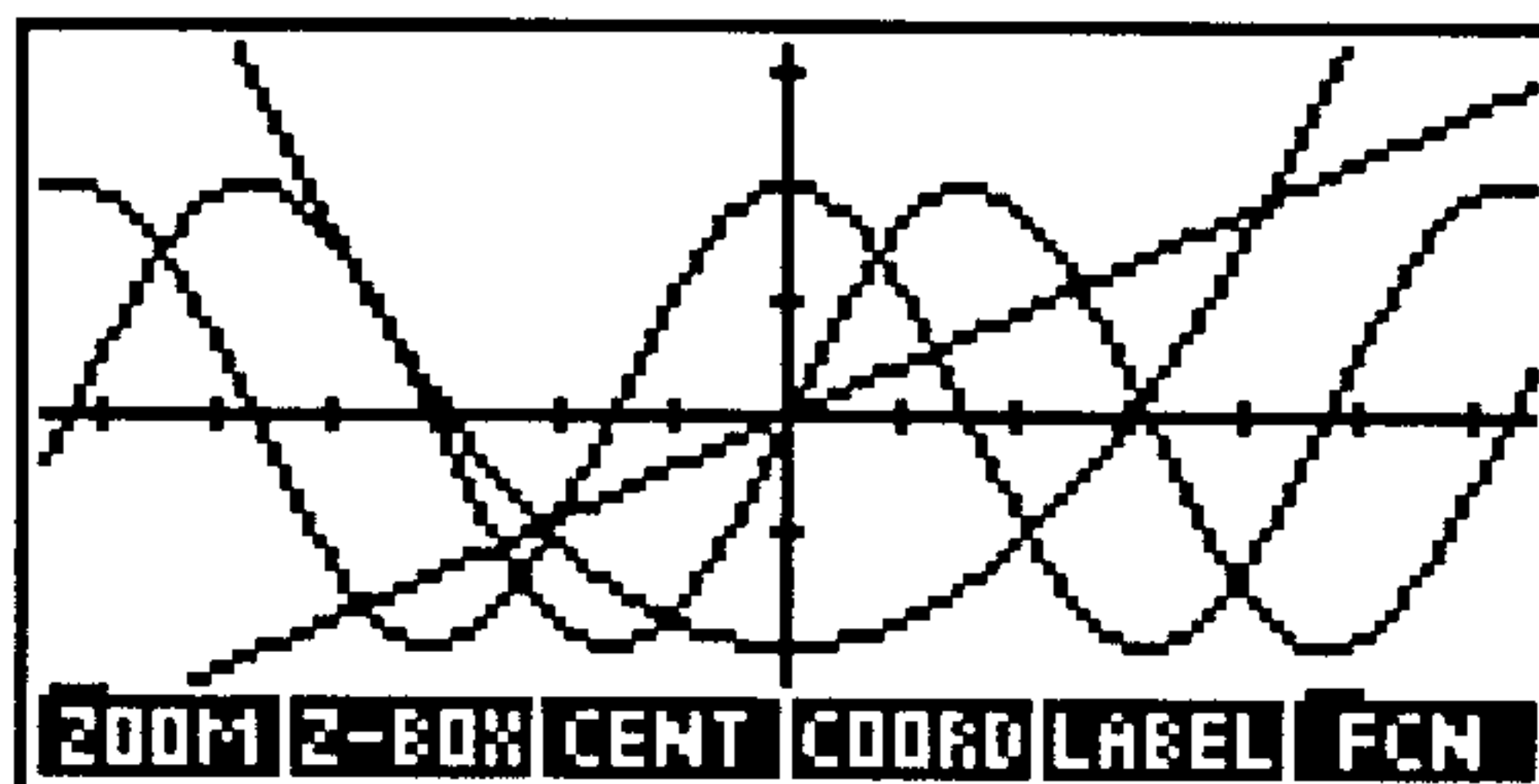
1. Move the cursor near the zero at $\pi/2$, and press ROOT ☐ ☐ (the first ☐ just restores the labels).
2. Move the cursor near the zero at $-\pi/2$, and press ROOT.
3. Press any key, then AREA:



The difference between this result and the ideal result -2 (negative because we integrated from right to left) arises primarily because the integration limits are the x -coordinates of the centers of the pixels near the zeros, rather than the computed coordinates. If you execute $\int$ using the values returned by ROOT (which are actually on the stack), you do obtain a result of 2.

The remaining FCN menu operation is F' (on the second page). This operation adds the derivative of the selected object to the head of the current equation list, then erases

the screen and executes DRAW again. For example, press $\equiv \text{NXEQ} \equiv$ to select the parabola, then $\equiv \text{F}' \equiv$:



The derivative of the parabola is a straight line with slope 0.5.

If you exit to the stack at this point ($\boxed{\text{ATTN}}$), you will see a display like this:

```

RAD
{ HOME }
3: Root: 1.57079632679
2: Root: -1.570796326...
1: Area:
   -1.99914720608
ERASE DRAW AUTO MARK YRNG INDEF

```

All of the function menu operations that compute results for display in the plot environment also return those results to the stack. Each result is tagged with the operation name to help identify it.

15.5 Conic Sections

HP48 function plots normally do not attempt to plot solutions of equations (see section 15.4.1), because the solving process is relatively slow (using a root-finder to compute each point), and because there may be multiple solutions. However, in the case of general quadratic equations, it is possible to solve each equation symbolically, and to find both branches of the solution. This is the basis of the *conic section* plot type, so named because the graphs of second-order equations are called conic sections.

The general procedure for making a conic section plot is the same as for a function plot,

where in this case you must execute CONIC, or use CONIC in the PTYPE menu, to select the conic plot type. The current equation must be an algebraic object, which should be an expression or equation in two variables, nominally second order in both. If you wish to use names other than X and Y as the independent and dependent variables, you can use INDEP and DEPND to specify alternate choices.

You can understand what the HP 48 does to make a conic plot by considering how you might do it yourself using function plots. Start with a general second-order equation:

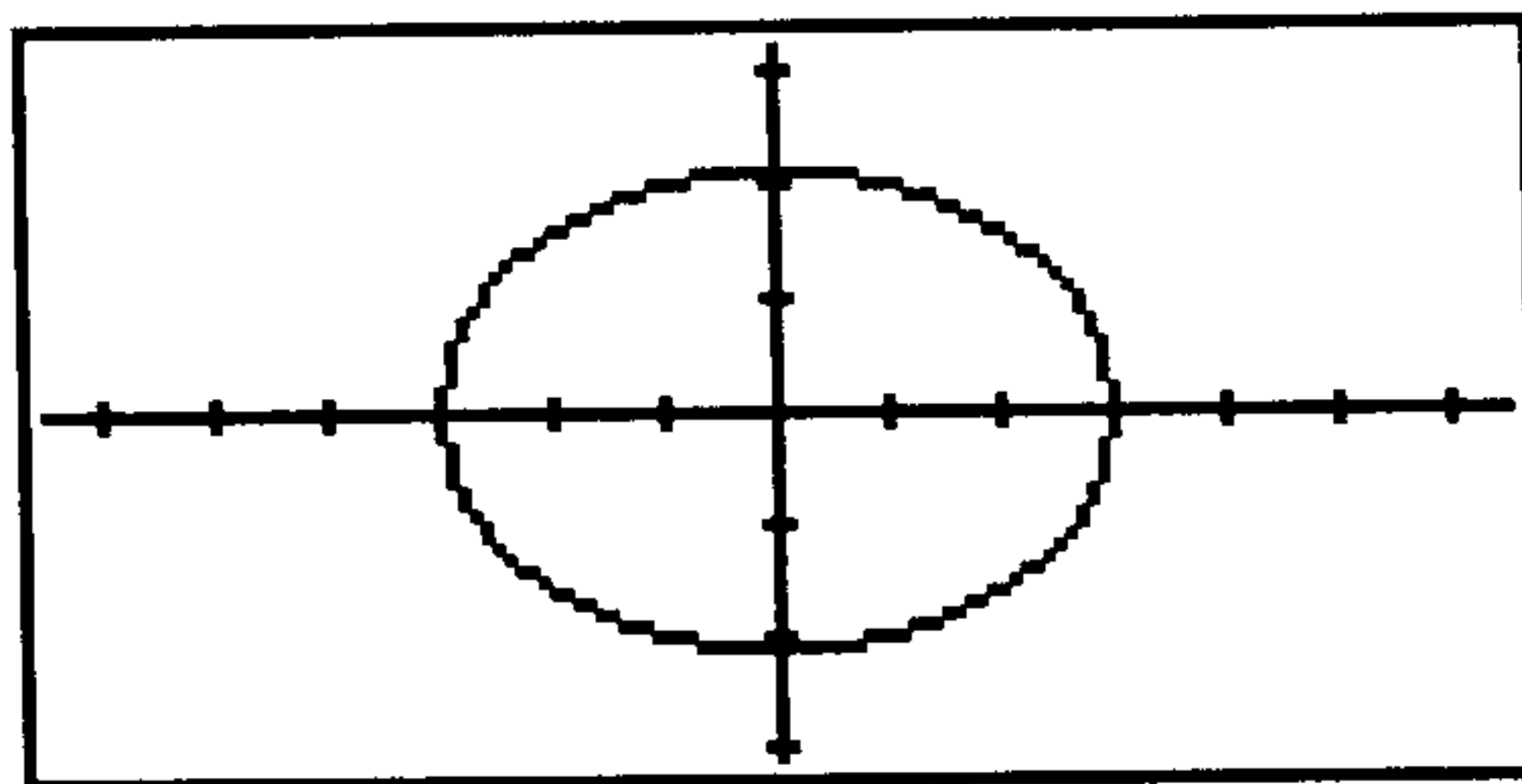
$$'A*X^2+B*Y^2+C*X+D*Y+E*X*Y+F=0'$$

Applying 'Y' QUAD (section 16.4.3) to this equation to solve for Y yields (after COLCT)

$$'Y=.5*(\sqrt{-(4*(A*X^2+C*X+F)*B)+(E*X+D)^2}*s1-E*X-D)/B'$$

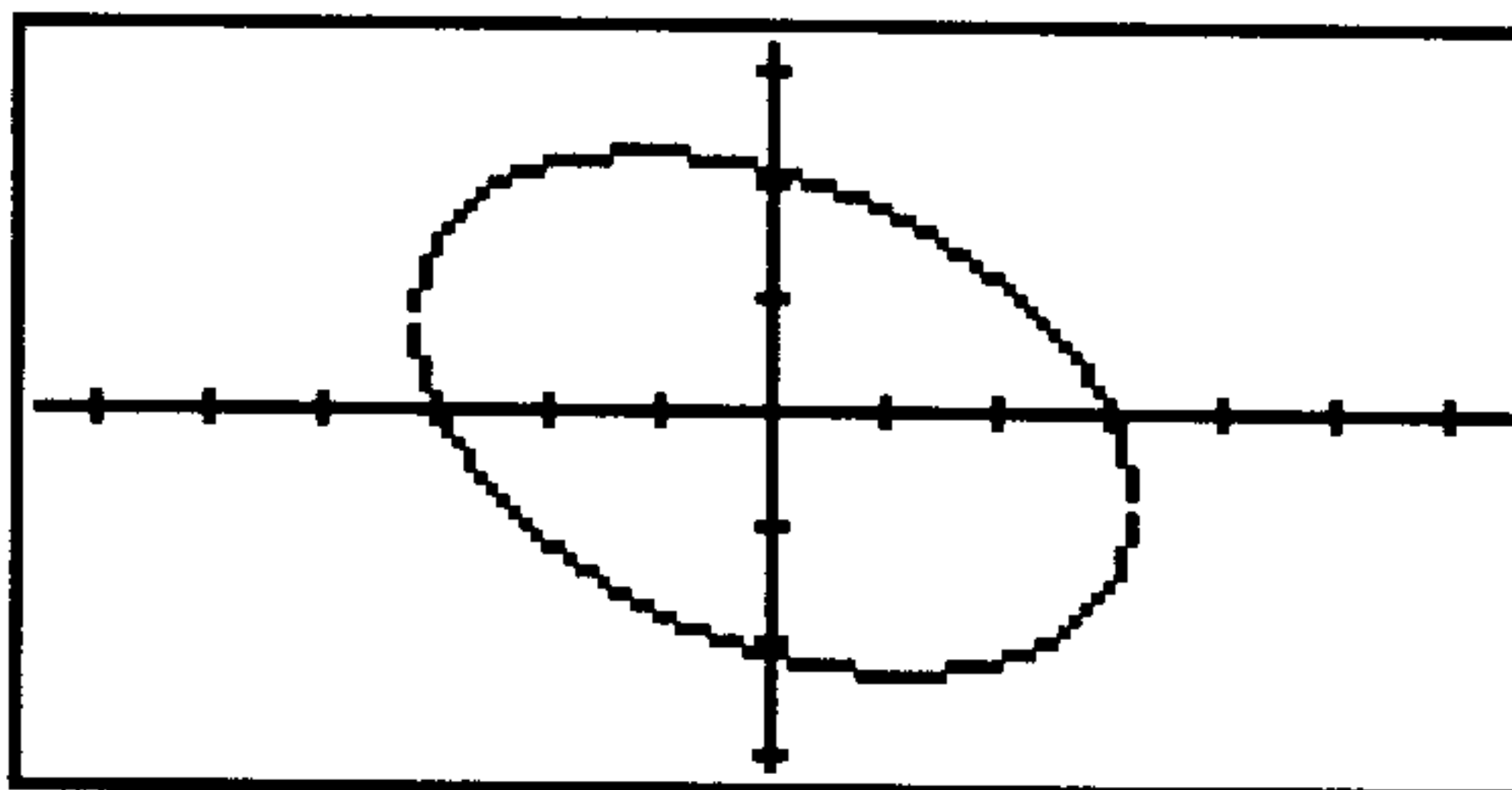
As you would expect for a quadratic equation, the solution is double-valued, as indicated by the presence of the s1 that represents ± 1 (section 16.5). If you store 1 in s1, you get an equation for one solution branch; storing -1 in s1 yields the other. To plot both branches of the solution simultaneously, you can use a current equation formed by equating the two right-sides of the separate branch equations, then execute an ordinary function plot using DRAW. This is essentially what the HP 48 does for conic plots, except that it makes a temporary version of the two-branch equation rather than disturbing the contents of EQ.

To try out conic plotting, store the general equation above in EQ and use the solver menu to store various values for the coefficients A-F, then execute DRAW. For example, with $A=1$, $B=2$, $F=-9$, and zero for the remaining coefficients, you obtain an ellipse (here we are using the default plot ranges):



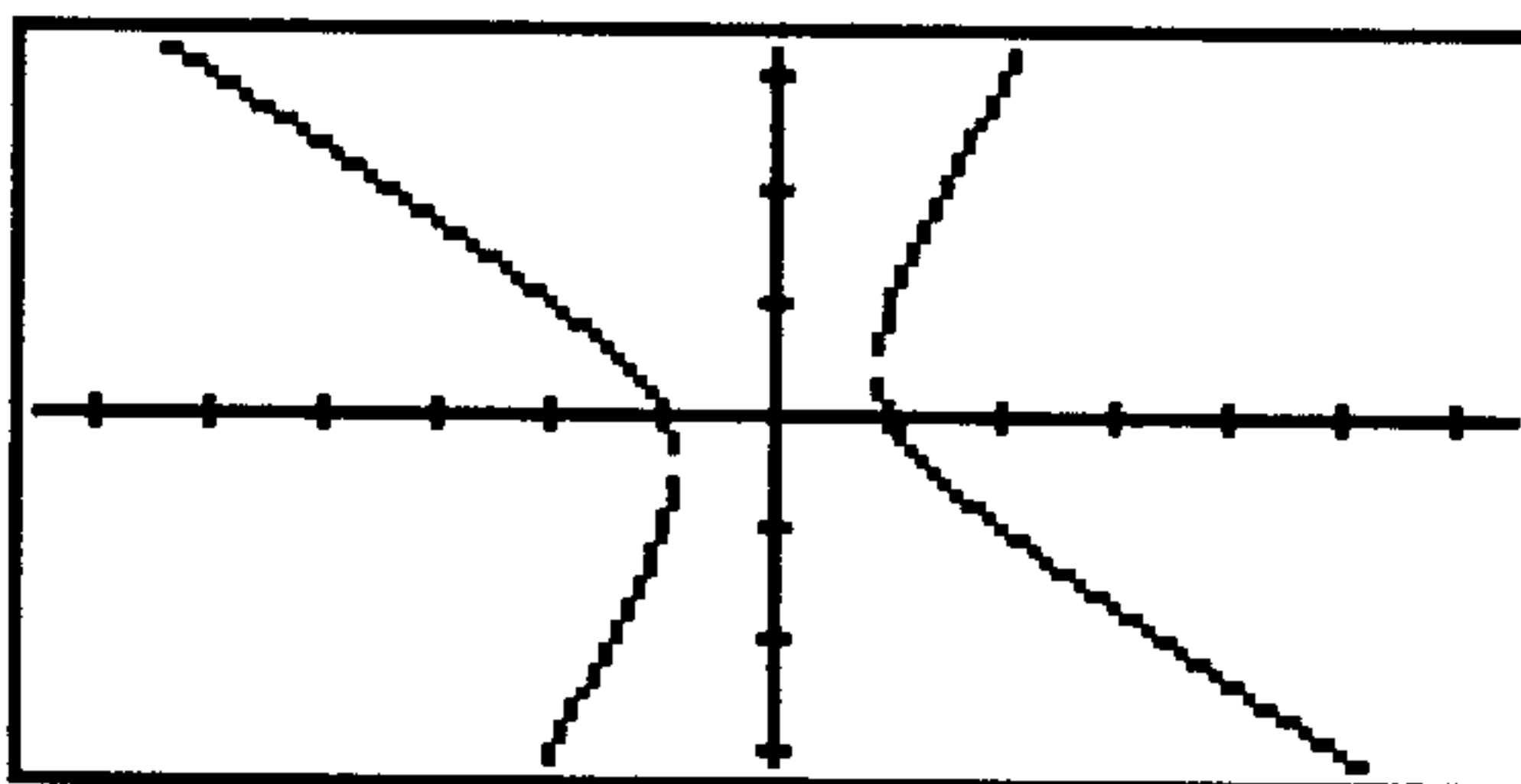
There is a delay before you see the ellipse drawn. This happens because values of X to the left and to the right of the ellipse give imaginary solutions for Y, for which no points are plotted. Making $E=1$ produces a rotated ellipse with the same intercepts as the

previous one:

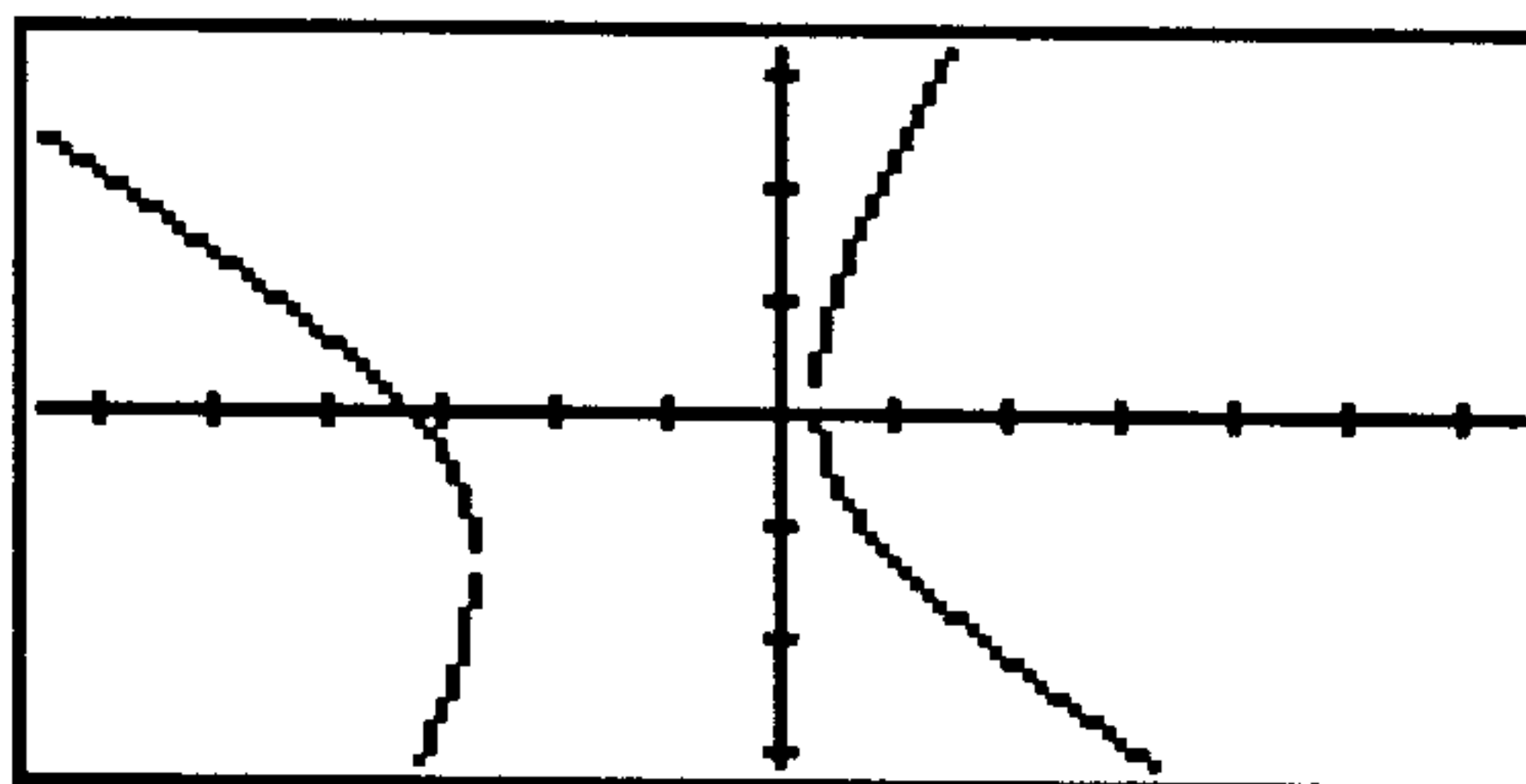


Both of the ellipses have “gaps” at their extreme X values. This is characteristic of most HP48 conic plots; it is caused by the separate plotting of the two branches.

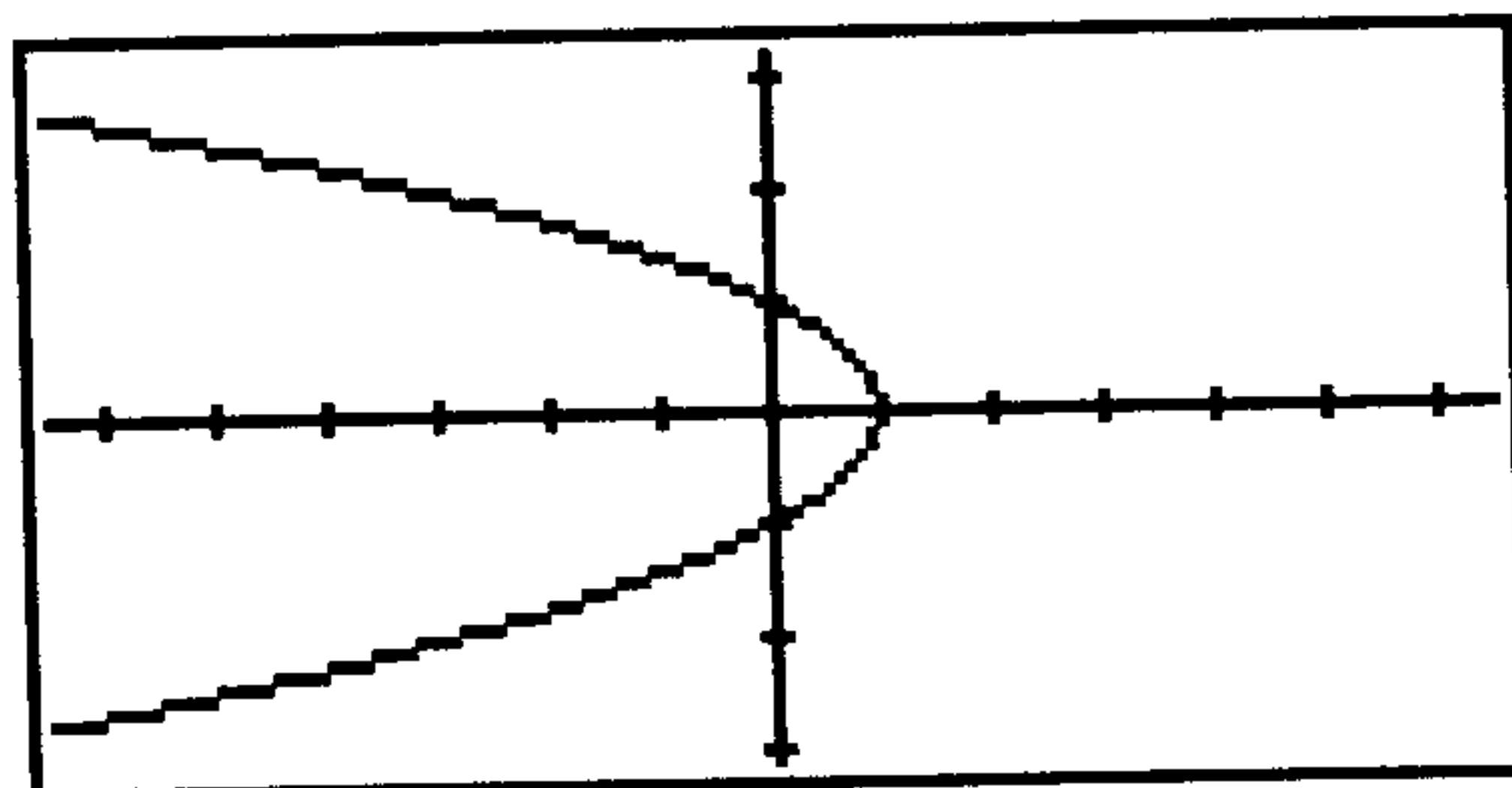
If A and B have opposite signs, the result is a hyperbola. Try $A=1$, $B=-1$, $E=1$, and $F=-1$:



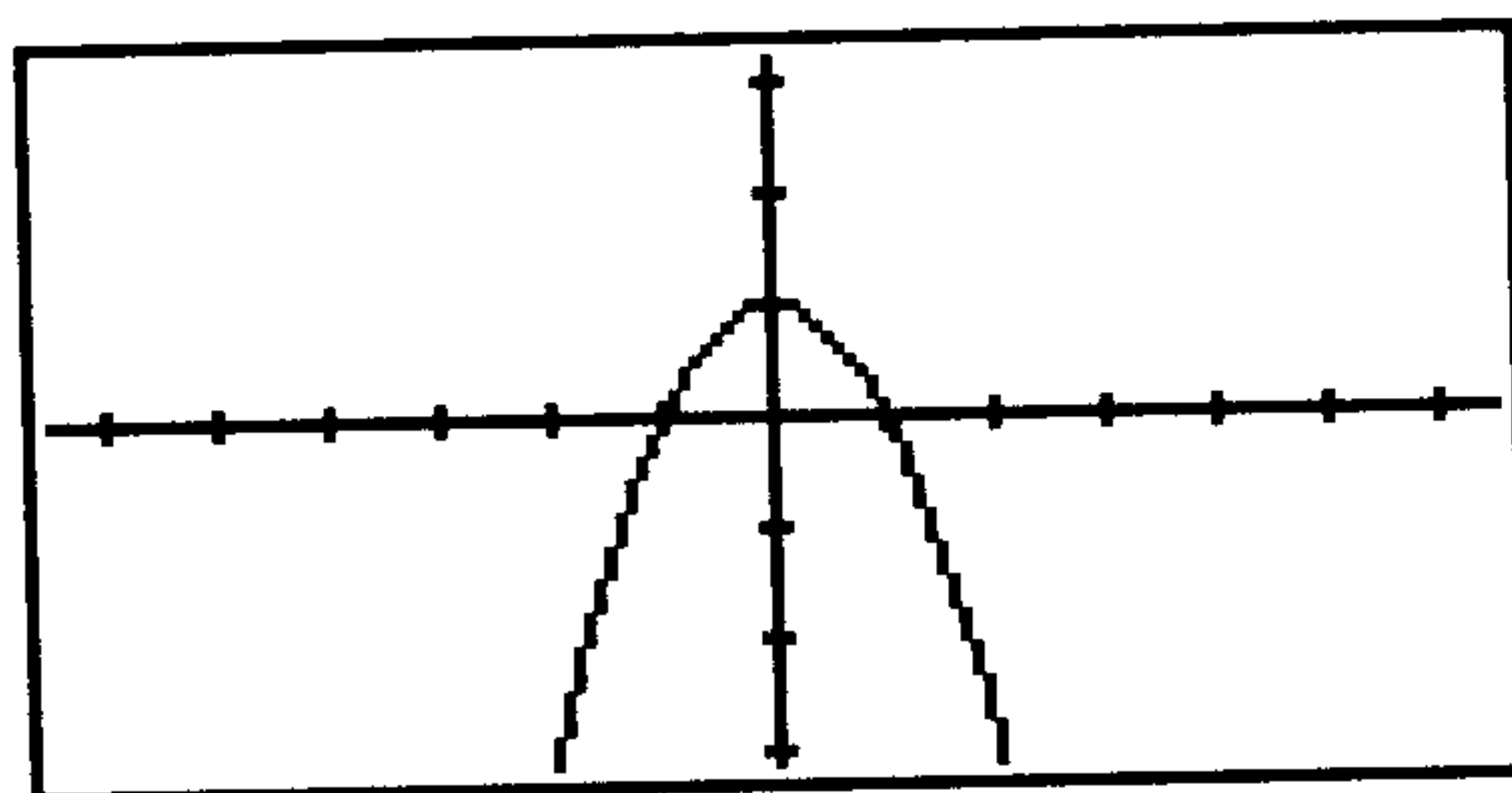
Adding a linear term displaces the conic section from the origin. For example, with $C=3$:



Removing the second-order term in X creates a parabola. Try $A=0$, $B=1$, $C=1$, $D=E=0$, and $F=-1$:



The parabola is turned on end if you interchange X and Y . $A=1$, $B=0$, $C=0$, $D=1$ and $F=-1$:



In this case, there is only one branch of the solution for Y .

The interpretation of the resolution parameter, and the effect of the curve-fill flag -31 , are the same for conic section plots as they are for function plots. Because x and y are treated on a more-or-less equal basis for conic plots, you should generally use the same plotting scale in both directions. **AUTO** will do this for you automatically; it adjusts the y -scale to be the same as the x -scale, while preserving the y -coordinate of the center of the screen.

15.6 Polar Plots

One way to plot an ellipse without the gaps produced by a conic plot is to re-express the defining equation in polar coordinates, then use the *polar* plot type (**POLAR**). For example, entering the equation for the first ellipse plotted in the preceding section, ' $X^2+2*Y^2-9=0$ ', make these substitutions:


```
{ X 'R*COS(θ)' } ↑MATCH DROP { Y 'R*SIN(θ)' } ↑MATCH DROP
```

which yields an equation in R and θ:

$$'(R*\text{COS}(\theta))^2 + 2*(R*\text{SIN}(\theta))^2 - 9 = 0'.$$

Solve for R:

```
'R' QUAD COLCT
```

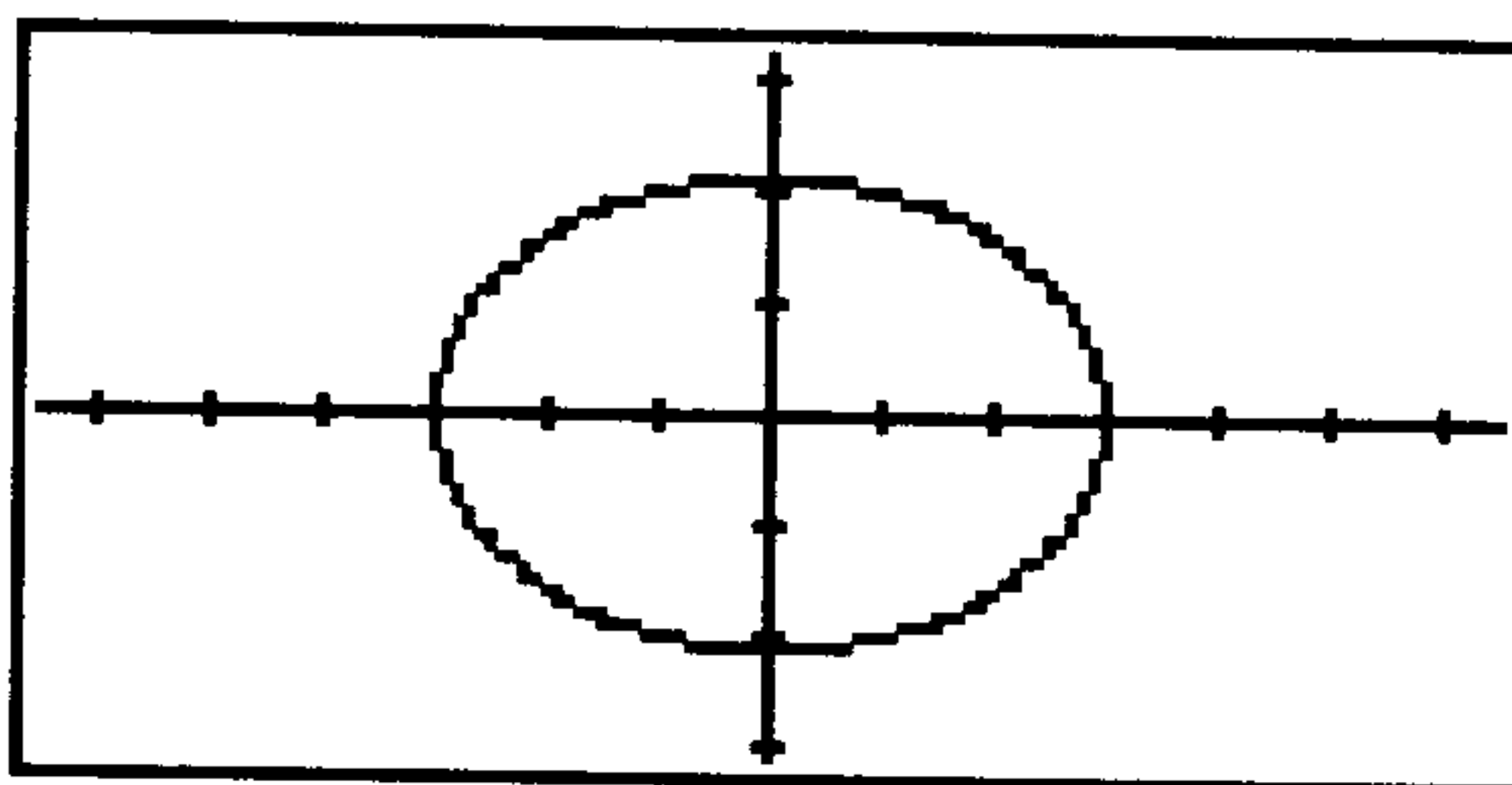
$$'R = \sqrt{(18*(2*\text{COS}(\theta)^2 + 4*\text{SIN}(\theta)^2))/(2*\text{COS}(\theta)^2 + 4*\text{SIN}(\theta)^2)*s1}$$

Choose the positive branch, and make it the current equation:

```
{ s1 1 } ↑MATCH DROP STEQ.
```

Now make a polar plot, with θ as the independent variable (enter “θ” with α $\rightarrow$ $\mathbf{F}$):

```
POLAR 'θ' INDEP ERASE DRAW
```



Polar plots follow much the same logic as function plots, except that the current equation is interpreted to represent the radial coordinate r as a function of the polar angle θ . The resolution parameter (section 15.2.6) determines the increment in θ between successive plotted points. The default resolution 0 yields an increment of 2° in degrees mode, 2 grad in grads mode, or $\pi/90 \text{ radians}$ in radians mode. The same rules (section 15.4) for interpreting expressions, programs, defining equations ($r = f(\theta)$), and general equations apply to polar and function plots, including the effect of flag -30. However, you do need to pay more attention to the independent variable for polar plots:

- Since the independent variable is the polar angle rather than the horizontal coordinate, there is no obvious default for the independent variable range. If you don't specify one using INDEP (section 15.2.4), DRAW will plot one cycle: 0 to 360°, 0 to 400 grads, or 0 to 2π , depending on the current angle mode.
- Selecting the polar plot type does not change the default independent variable name, which remains as X. This is convenient for function and conic plots, but less so for polar plots, where you are more likely to want to use a name like θ (α $\rightarrow$ F).
- According to the rules listed in section 15.2.7, LABEL will label the horizontal axis with the independent variable name, which is not a good choice for polar plots. Executing { "X" "Y" } AXES, for example, establishes X and Y as the labels, without affecting the current axes' positions.

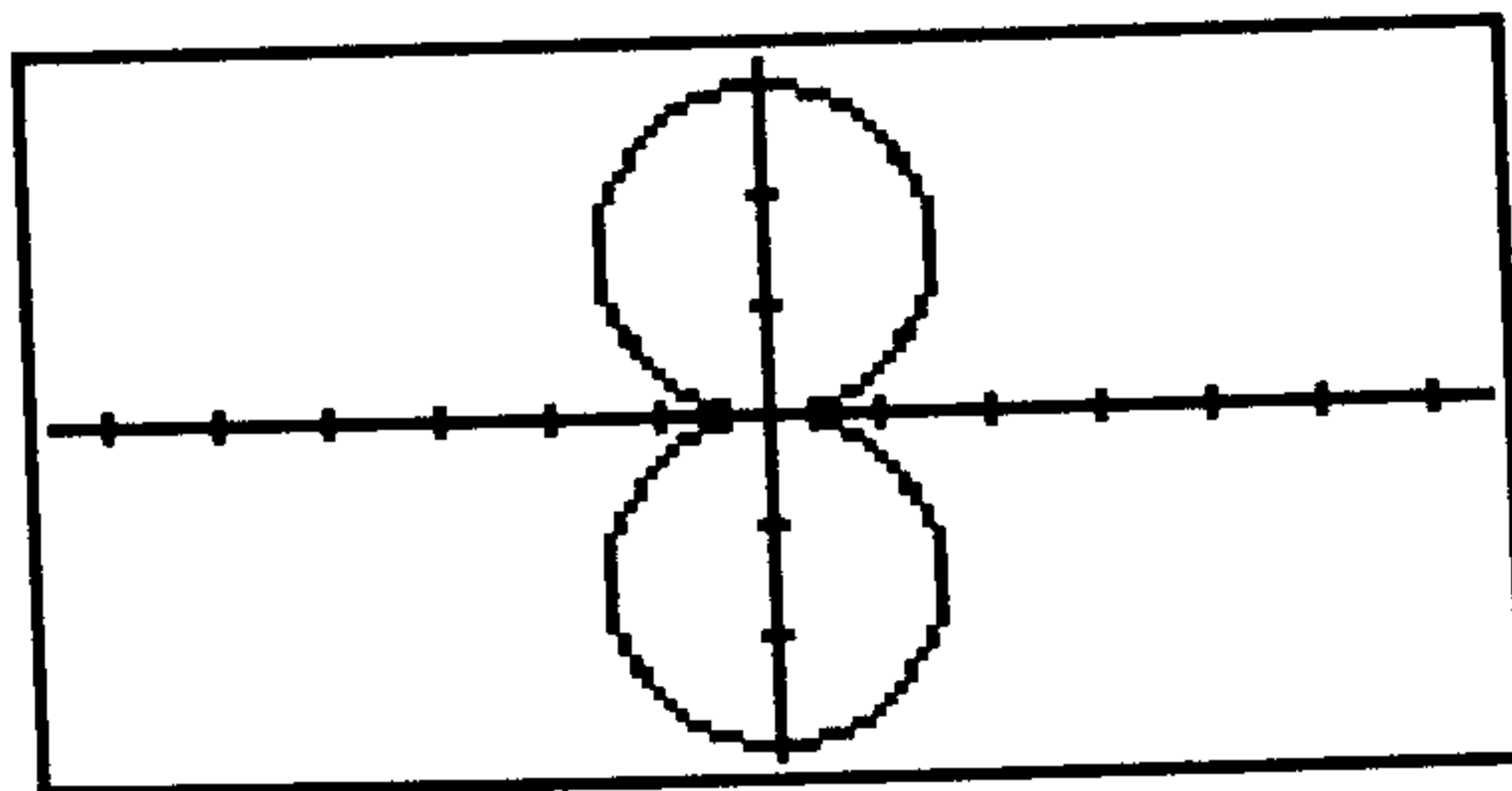
The polar plot analog of the special function plot case $x=\text{constant}$ is a current equation of the form $\theta=\text{constant}$, where θ is the independent variable. In that case, with flag -30 clear, DRAW plots a straight line through the origin, at an angle *constant* from the x-axis. If flag -30 is set, the two sides of the equation are plotted separately, producing a circle $r=\text{constant}$ and a spiral $r=\theta$.

15.6.1 Examples

The following examples are plotted in radians mode, with the plot ranges and current equations as indicated.

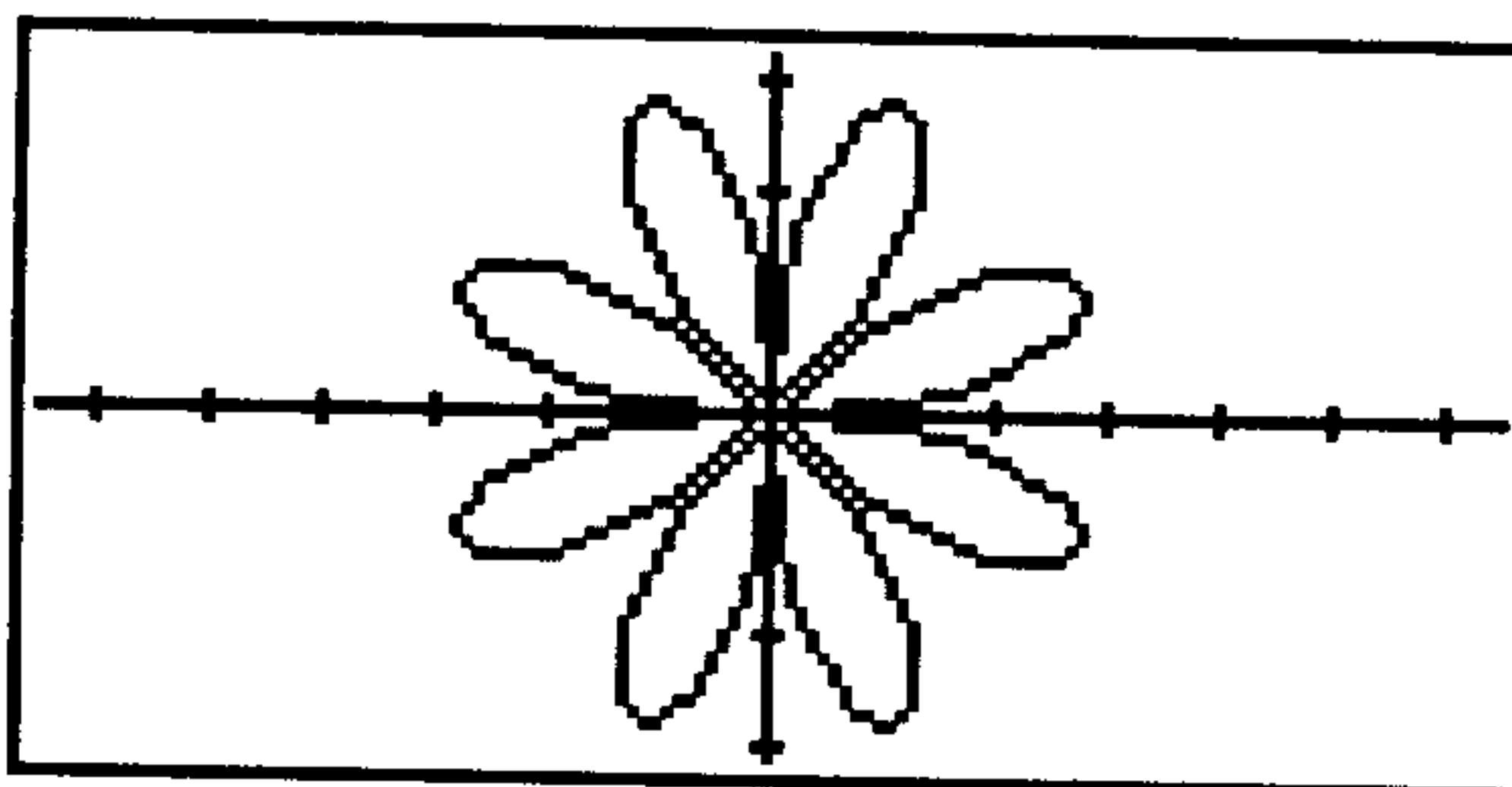
■ Figure 8

```
EQ:      R='ABS(3*SIN(θ))'
INDEP:   θ
XRNG:    -6.5  6.5
YRNG:    -3.1  3.2
```

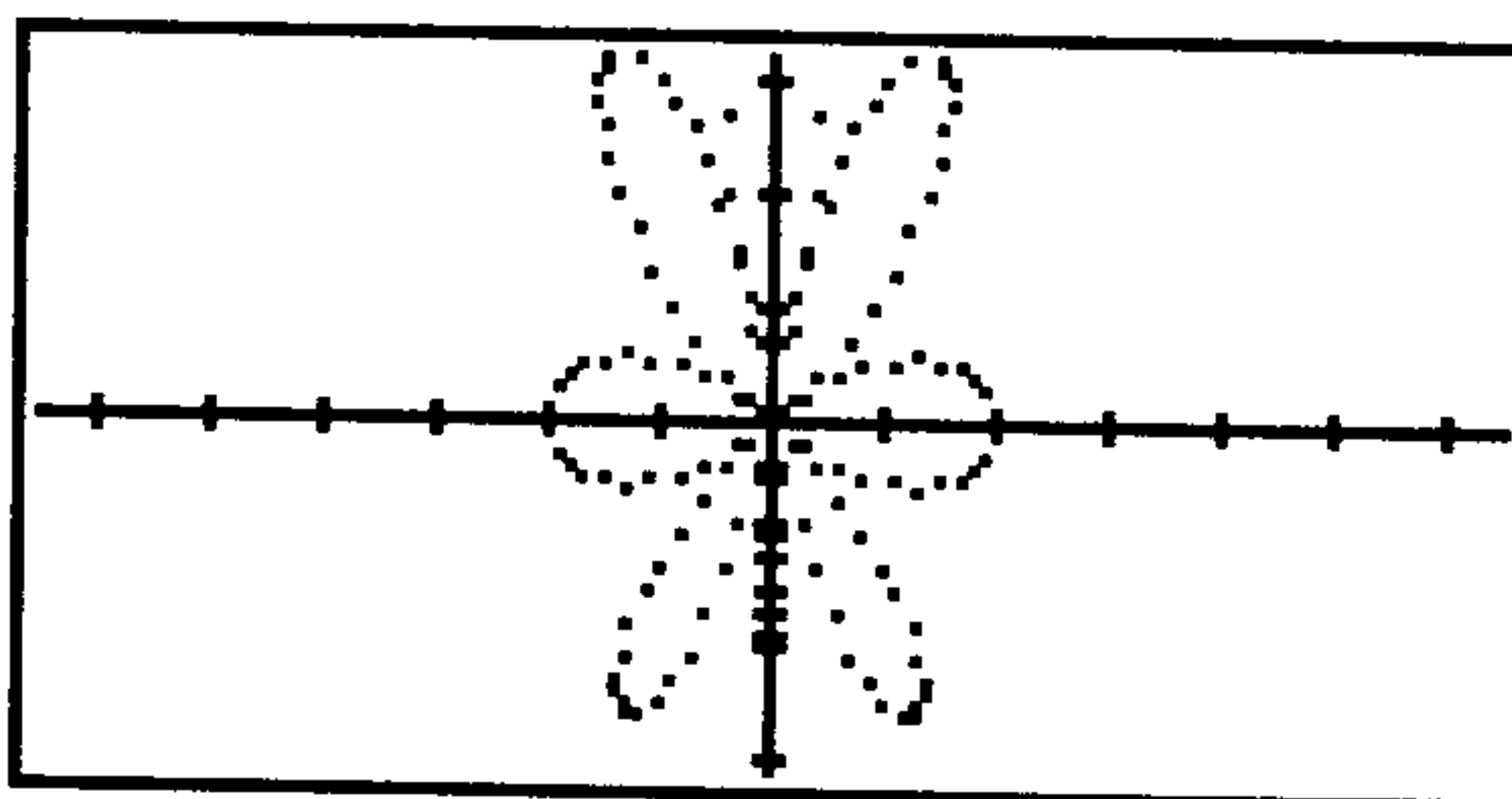


■ *Flower.*

EQ: $'R = \text{ABS}(3 * \text{SIN}(4 * \theta))'$
 INDEP: θ
 XRNG: -6.5 6.5
 YRNG: -3.1 3.2

■ *Teddy Bear*

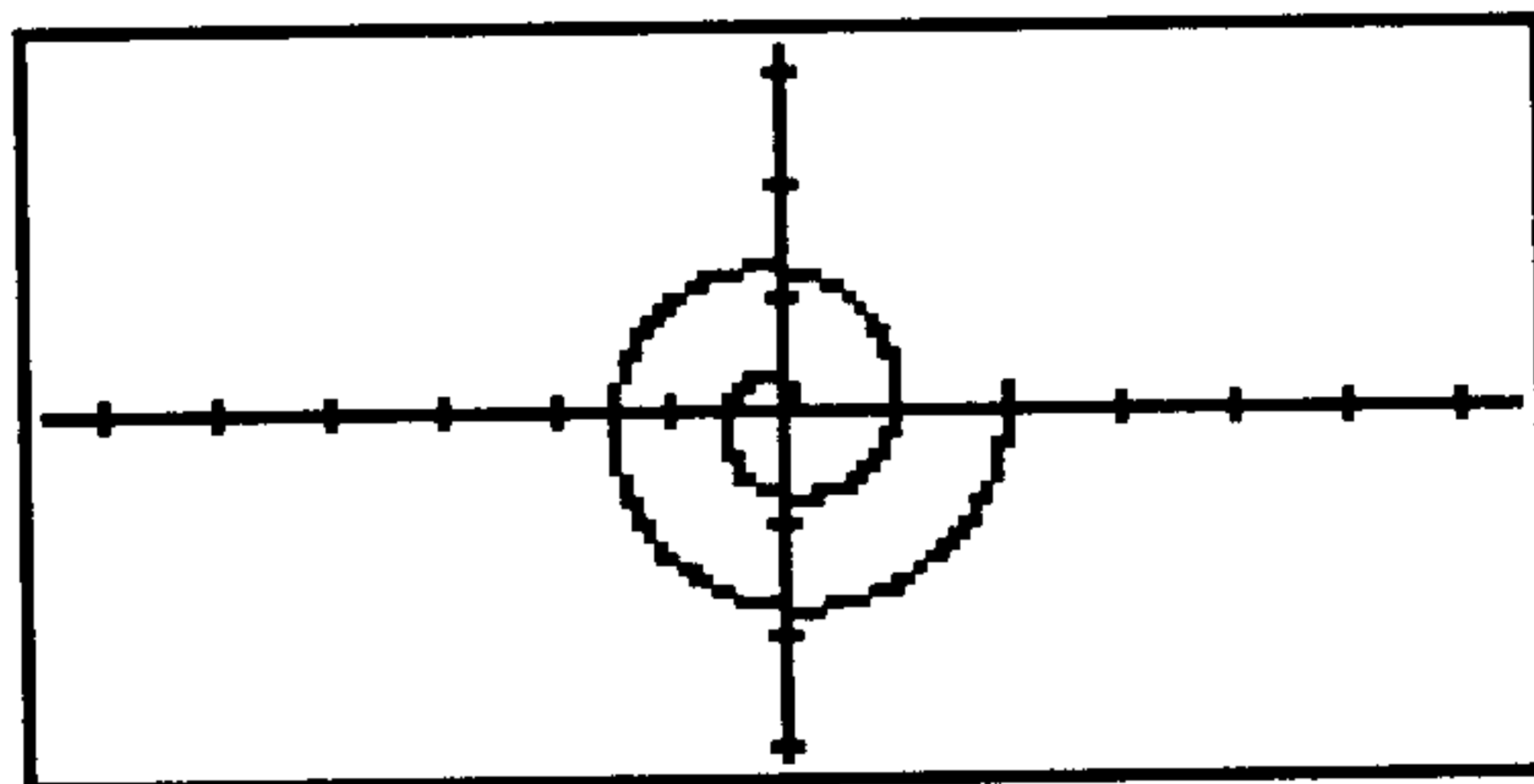
EQ: $'R = -2 * \text{COS}(4 * \theta) + \text{SIN}(\theta)^5'$
 INDEP: θ
 XRNG: -6.5 6.5
 YRNG: -1.55 1.6
 Flag -31: *set*



If you are sufficiently imaginative, you can see the figure of a teddy bear peering around the axes.

■ *Spiral.*

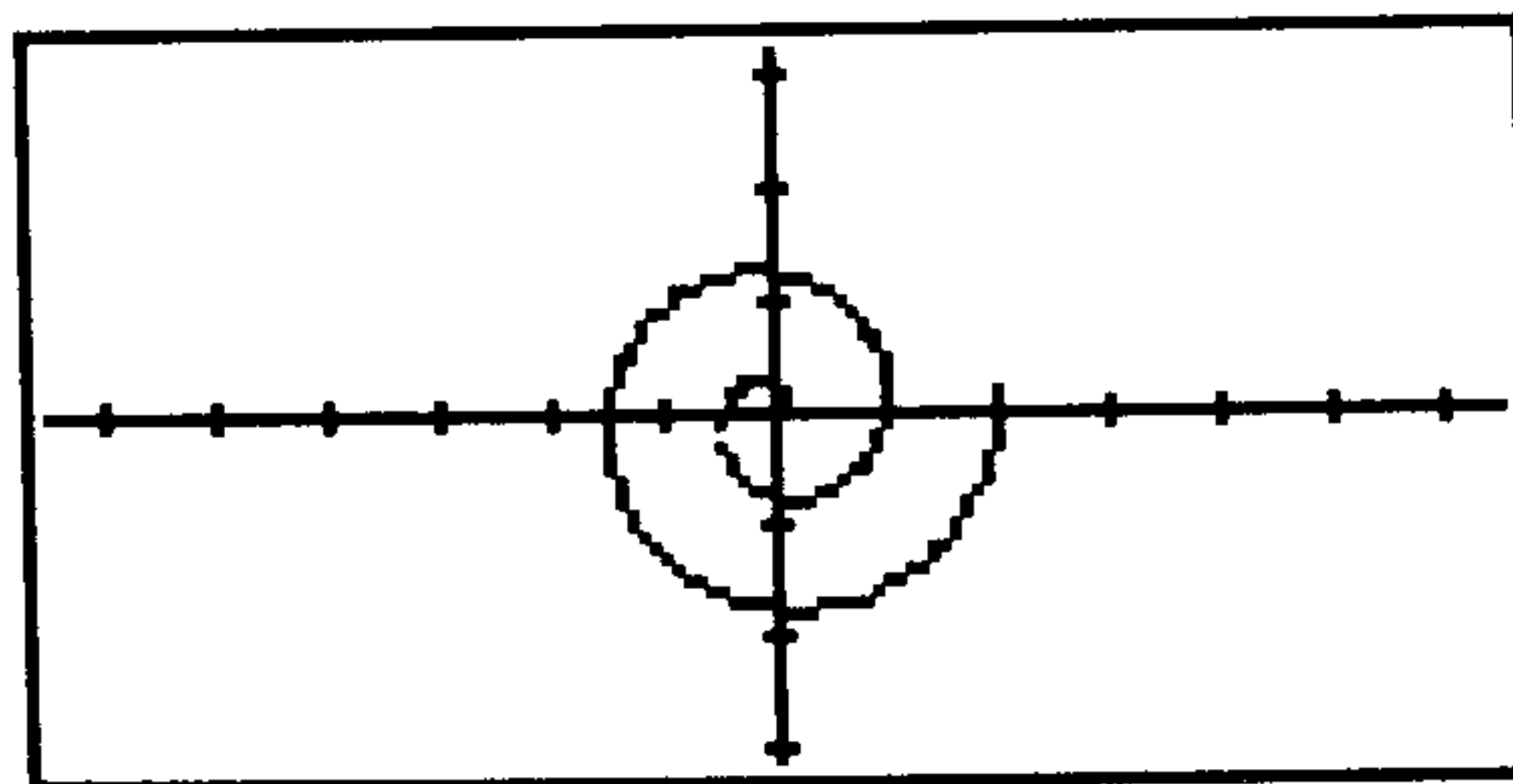
EQ: $'R=\theta/(2*\pi)'$
 INDEP: { θ 0 12.68 }
 XRNG: -6.5 6.5
 YRNG: -3.1 3.2
 Flag -31: *clear*



15.6.2 Varying the Angle Increment

If you plotted the last example, you may have noticed that the spiral appeared to be drawn slowly at first, then faster as the spiral grew. This is because when the radial coordinate is small, the default 2° increment is insufficient to cause consecutive points to fall within different pixels, so that some pixels get plotted more than once. Increasing the angle increment to 10° (.1745 *radians*), for example, speeds up the plotting dramatically, but at the expense of loss of resolution in the outer parts of the spiral.

The HP 48 does not provide for a variable plot resolution, but it is straightforward to make program replacements for DRAW that implement such a feature. The program VARPOL illustrates one approach. Using VARPOL instead of DRAW:



VARPOL	Variable θ Polar DRAW	0276
<pre> << RCLF → flags << { #0 #0 } PVIEW DRAX -3 CF -19 SF -16 SF RAD -22 SF 'EQ' RCL IF DUP TYPE 6 SAME THEN RCL END EQ→ IF DUP 0 SAME NOT THEN SWAP END DROP PPAR 3 GET IF DUP TYPE 5 == THEN OBJ→ DROP ELSE 0 '2*π' →NUM END ROT → θ << ROT θ SHOW θ 'θ' 2 →LIST tMATCH DROP 'θ' DUP STO DUP θ θ SQ OVER SQ + COLCT √ { #0 #0 } PX→C { #1 #0 } PX→C - RE ABS SWAP / .18 MIN → r dθ << FOR θ r →NUM θ →V2 PIXON dθ →NUM STEP >> flags STOF >> >> </pre>		
	Show the plotting. Draw axes. Symbolic execution mode. Complex →V2. Set polar mode, radians mode. Don't error for $r=0$. Get the current equation. Get the right side. Get independent variable. If it's a list, take it apart. Otherwise, plot one cycle. Save the independent var. name. Show independent variable. Replace with local name. Save the modified expression. Compute the radial coordinate. Plot the point. Compute the polar increment. Increment θ.	

VARPOL computes the derivative of the arc length as a function of θ , then increments θ by the amount necessary to increase the arc length by one pixel width. If a is the arc length measured along a curve, then the differential increment is related to the polar angle by

$$\delta a = \frac{\delta \theta}{(r^2 + r'^2)^{1/2}}$$

where $r=r(\theta)$ is the radial coordinate and $r' = dr/d\theta$. This program works only with current equations that are expressions or defining equations, which must have sensible derivatives; it also assumes that angular quantities are expressed in radians.

Notice that the plotting takes less time for small θ than when using DRAW. VARPOL generally takes longer than DRAW, because it has to evaluate the derivative of the current equation as well as the object itself, but it will often produce a more satisfactory plot.

15.7 Parametric Plots

Since HP48 pixels are addressed by rectangular coordinates, polar plots are actually made by converting the polar coordinates of computed point to rectangular coordinates:

$$\begin{aligned}x(\theta) &= r \cos \theta \\y(\theta) &= r \sin \theta\end{aligned}$$

x and y are thus functions of the independent coordinate θ . Polar plots are a special case of *parametric* plots. In the general case, the independent variable is not limited to a polar angle, but may be anything.

The current equation for a parametric plot may be an expression or a program that determines both coordinates at once by returning a complex number object. That is, an expression must have the general form ' $x(t)+i y(t)$ ', where t is the independent variable; a program must return an equivalent result. As for polar plots, you must ensure that the independent variable and its range are set appropriately with INDEP. If you do not specify any range, the HP48 will default to the current x-range, which is not likely to be a useful choice.

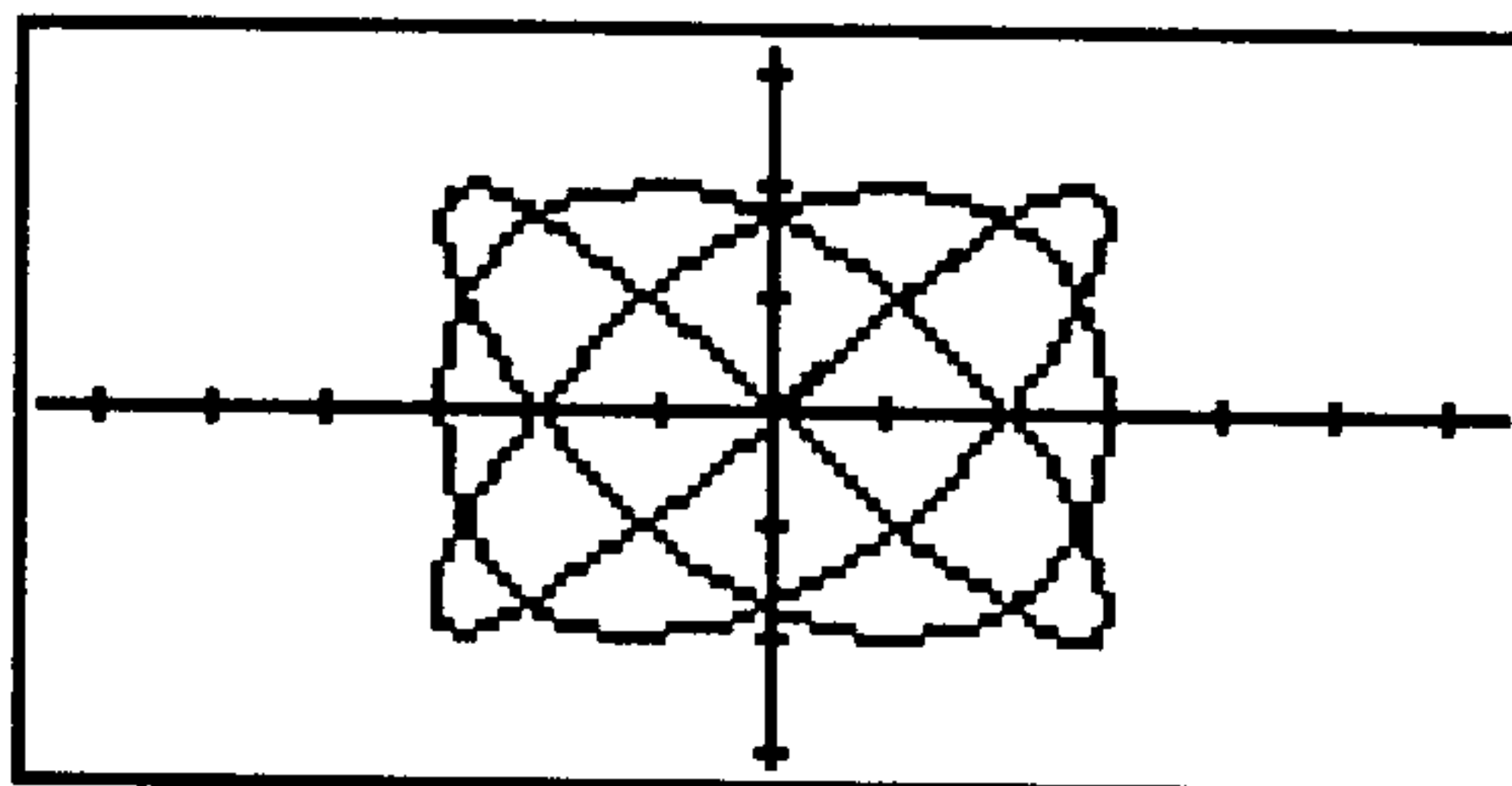
A nice example of a parametric plot is a *lissajous* figure, produced by driving the x - and y -coordinates with sinusoidal functions having different frequencies. Start by executing PARAMETRIC ($\leftarrow$ **PLOT** **PTYPE** **PARA**) to select the parametric plot type. Then store the following expression as the current equation:

$$'3*\text{SIN}(3*T)+2*\text{SIN}(4*T)*i'$$

Set the independent variable with

$$\{ T \ 0 \ 6.5 \} \text{ INDEP.}$$

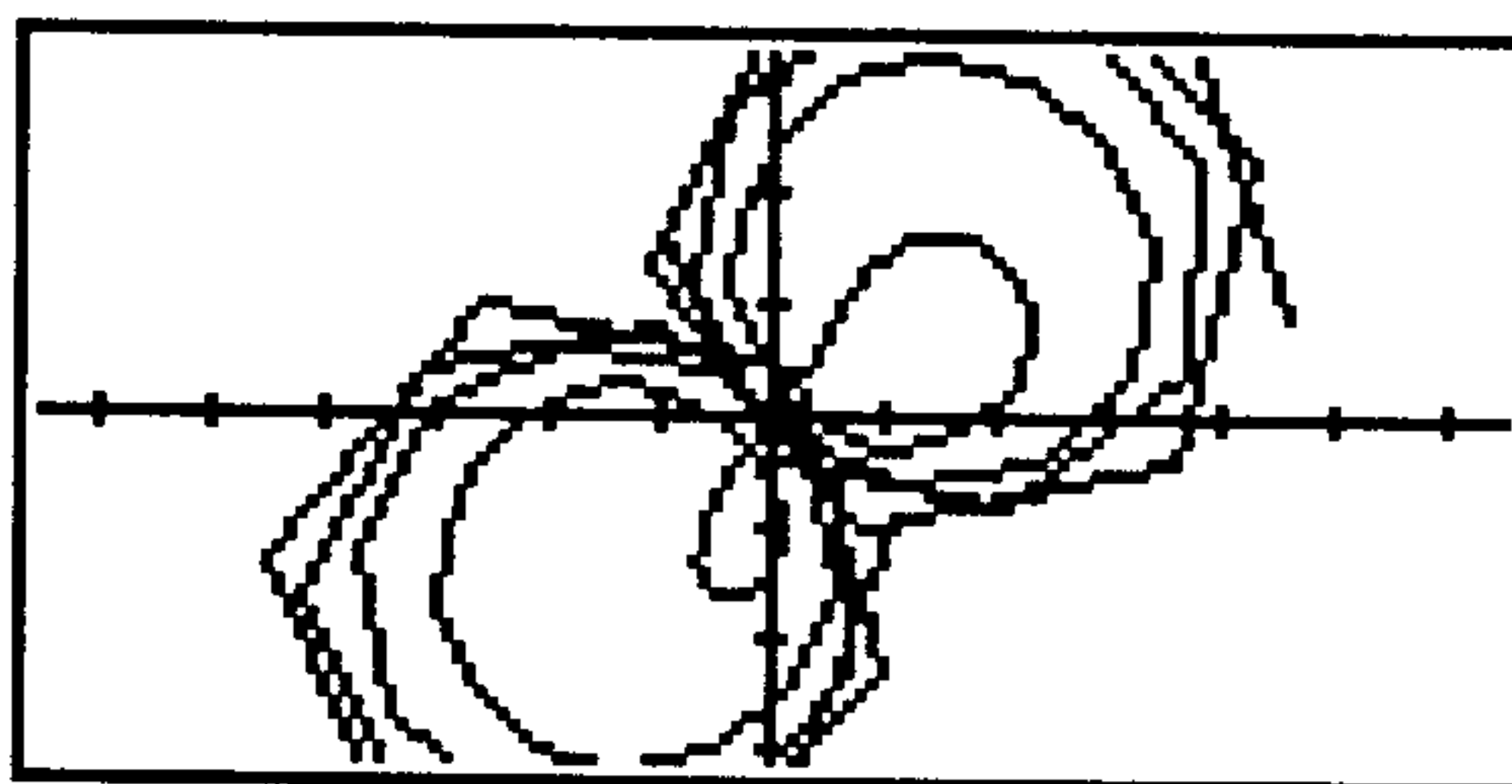
Then, using the default x- and y-ranges, DRAW produces this figure:



Choosing a suitable resolution for plotting presents the same difficulties for parametric curves as it does for polar curves. For example, consider the current equation

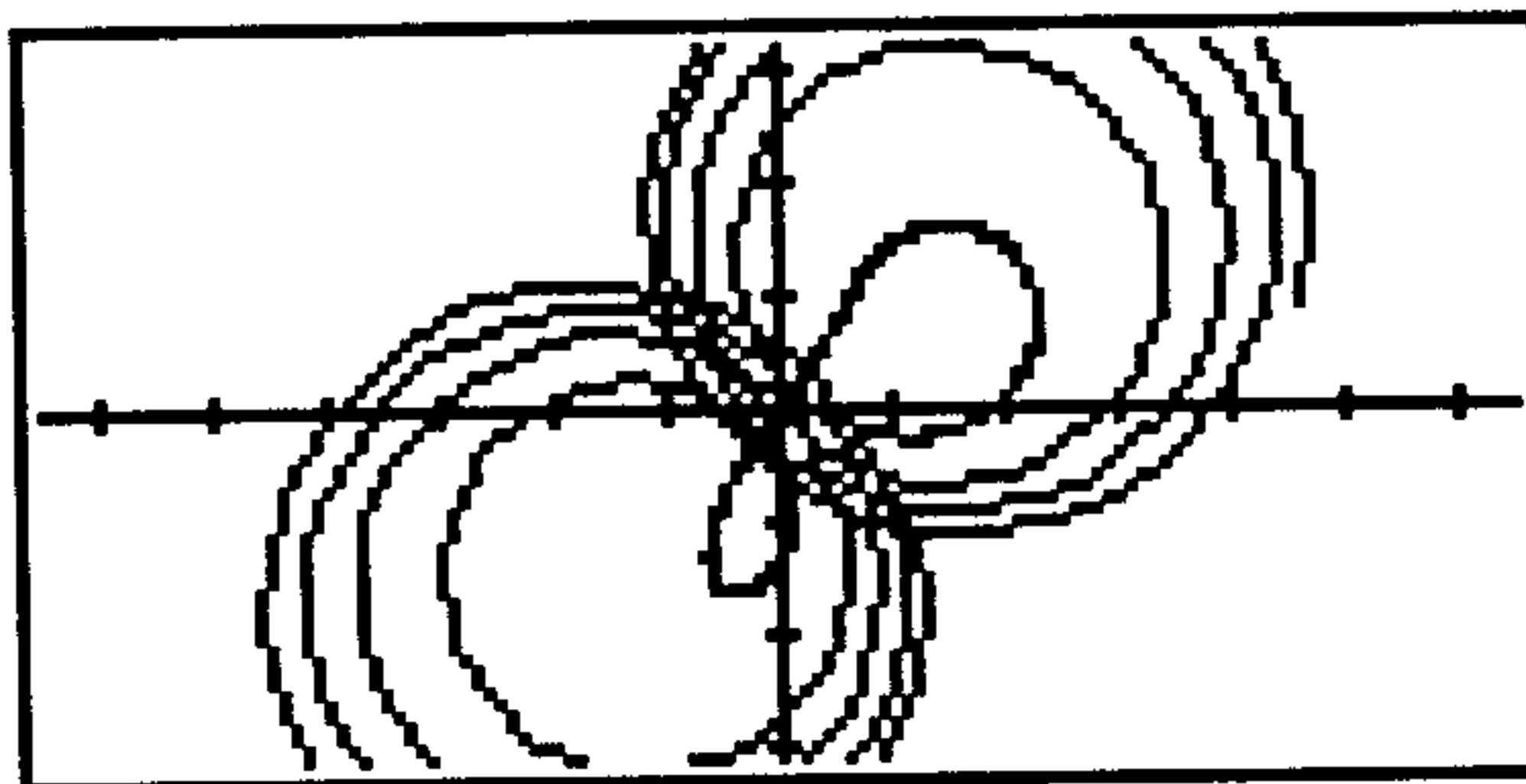
$$' \text{SIGN}(T)*\sqrt{\text{ABS}(T)}+T*\text{SIN}(T^3) + (\text{SIGN}(T)*\sqrt{\text{ABS}(T)}+T*\text{COS}(T^3)) * i'.$$

Plotting this expression over the range $-3 \leq T \leq 3$ ($\{ T \ -3 \ 3 \} \text{ INDEP}$), with the default resolution of 0, produces the following plot:



For parametric plots, resolution 0 is taken to mean 130 points equally spaced across the independent variable range. In this example, that resolution is too coarse, especially at the beginning and end of the curve.

Executing .01 RES for a finer resolution yields a smoother curve:



This is a more satisfactory picture, but an inordinate amount of time is spent plotting near $T=0$. This suggests the program VARPAR, a variation of DRAW for parametric plots analogous to VARPOL for polar plots. There are a few differences between VARPAR and VARPOL:

- VARPAR provides for variable resolution, rather than the one pixel spacing used by VARPOL. You set the resolution by using RES with a binary integer argument, which specifies the approximate pixel spacing between plotted points.
- Because of the variable resolution, VARPAR respects the curve fill flag -31, connecting successive points with straight-line segments if the flag is clear.
- VARPAR uses a maximum spacing of 5% of the independent variable range.

Like VARPOL, VARPAR requires that the current equation is differentiable. To use VARPOL with the current example therefore requires one addition: a derivative for SIGN, which does not have a built-in derivative (see section 18.1.1.1). The derivative of this function is not well-defined at zero, but for the current purpose we can just define the derivative to be zero everywhere:

```
'derSIGN(x,dx)=0' DEFINE
```

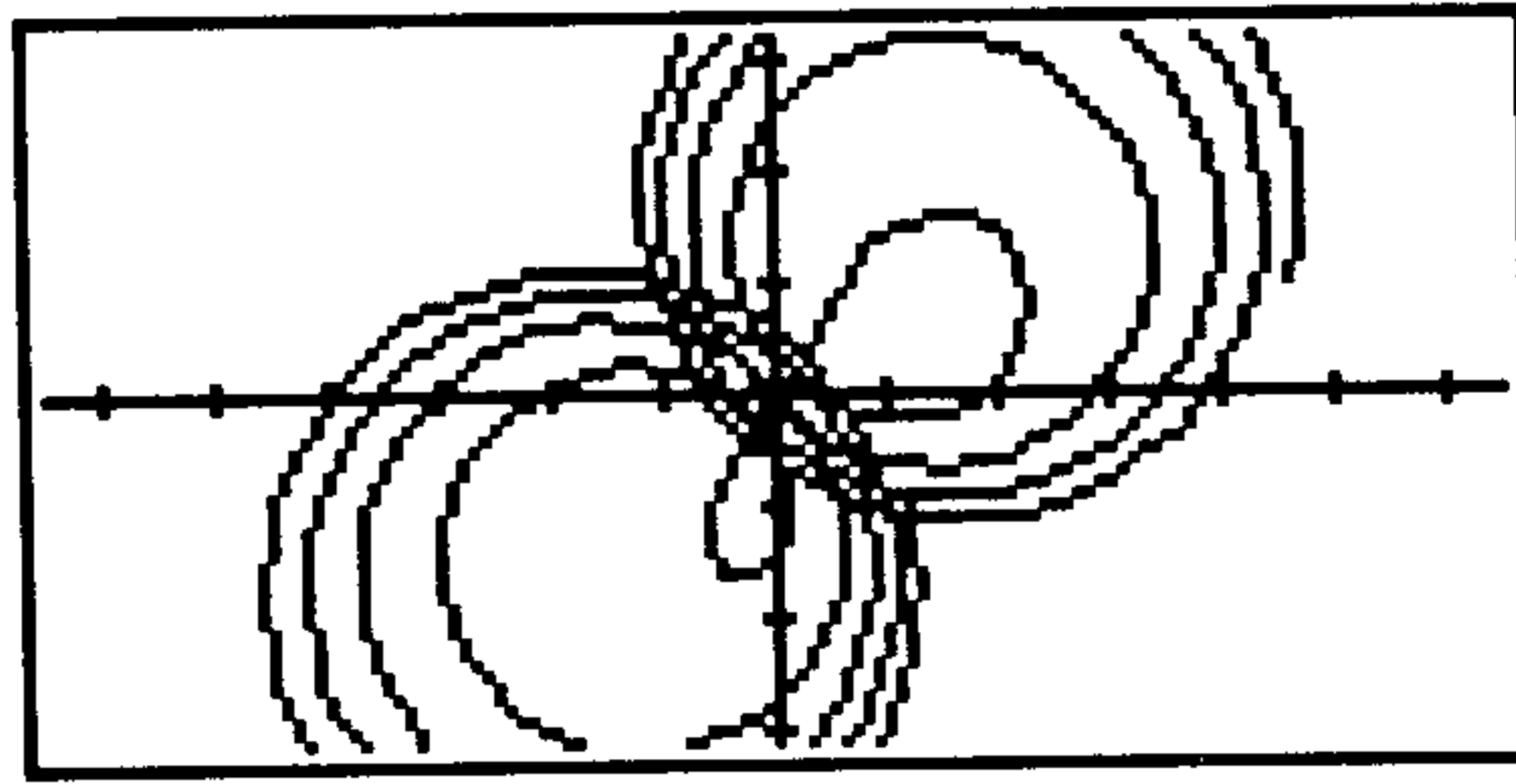
With this definition, executing

```
#3 RES DRAX VARPAR
```

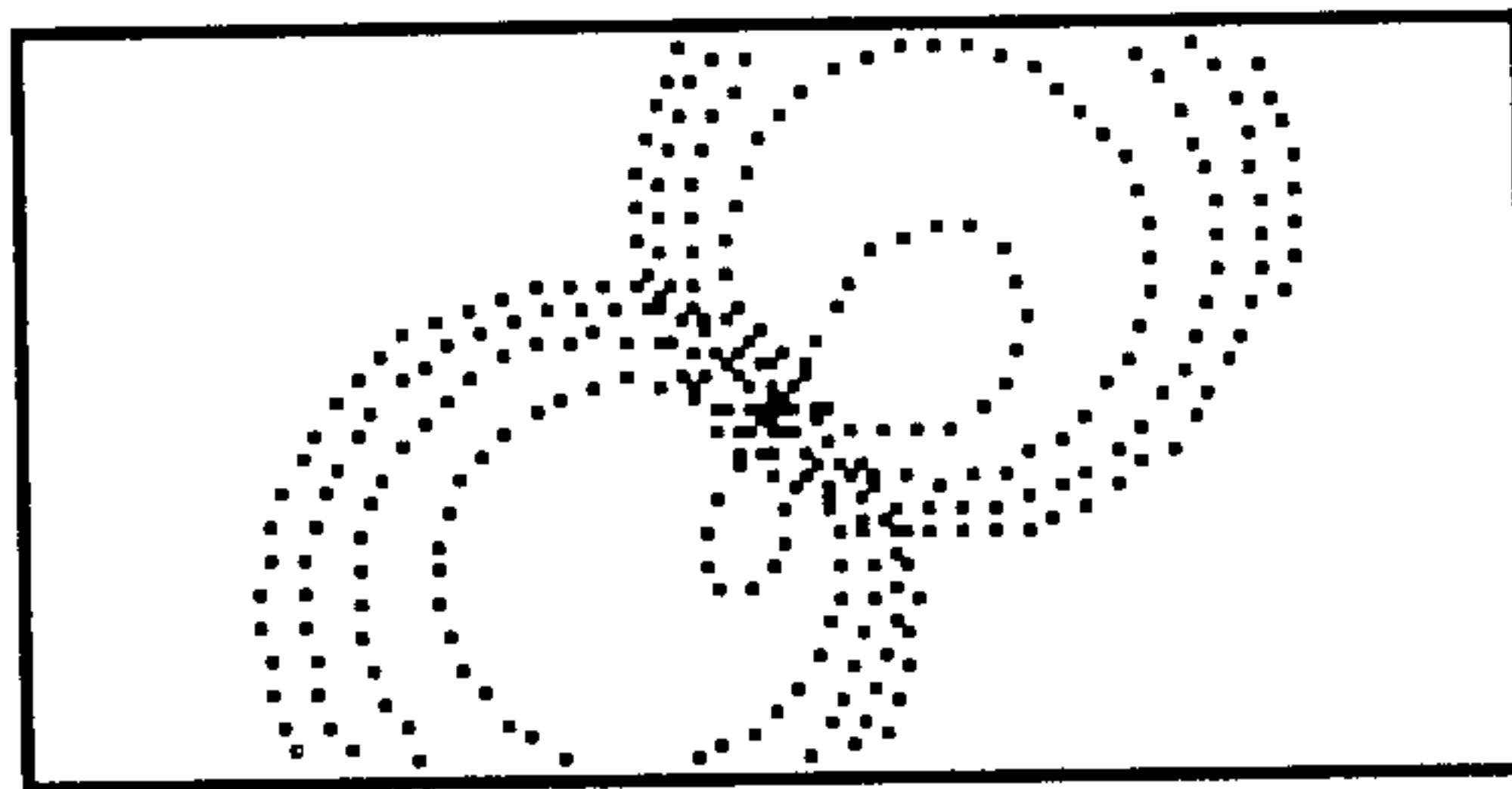
yields the picture shown next (following the program listing):

VARPAR	Variable δt Parametric DRAW	4723
<pre><< RCLF → flags << { #0 #0 } PVIEW -3 CF -16 CF -22 SF 'EQ' RCL IF DUP TYPE 6 SAME THEN RCL END PPAR 3 GET OBJ→ DROP ROT → t << ROT t SHOW t 't' 2 →LIST tMATCH DROP 't' DUP STO DUP t @ ABS { #0 #0 } PX→C { #1 #0 } PX→C - RE ABS PPAR 4 GET B→R * SWAP / 3 PICK 5 PICK - ABS .05 * MIN OVER 5 PICK 't' STO →NUM → f dt last << FOR t f →NUM IF -31 FS? THEN PIXON ELSE last OVER LINE 'last' STO END dt →NUM STEP >> flags STOF >> >></pre>	Save the flags. Show the plotting. Symbolic evaluation mode. Set rectangular mode. Don't error for $r=0$. Get the current equation. Get independent variable list. Save the independent variable name. Show of independent variable. Replace with local name. Width of a pixel. Multiply by resolution parameter. δt. Keep $\delta t < 5\%$ of t range. Expression, increment, and last point. If no curve fill... ...then just plot the next point. ...else connect to last point, and save. Compute the increment. Increment t	

VARPAR assumes that variables EQ and PPAR exist, that the resolution parameter is a binary integer, and that the independent variable entry is a list that includes a range.



The uniformity of the point spacing is more obvious if you plot with flag -31 set, and omit the axes:



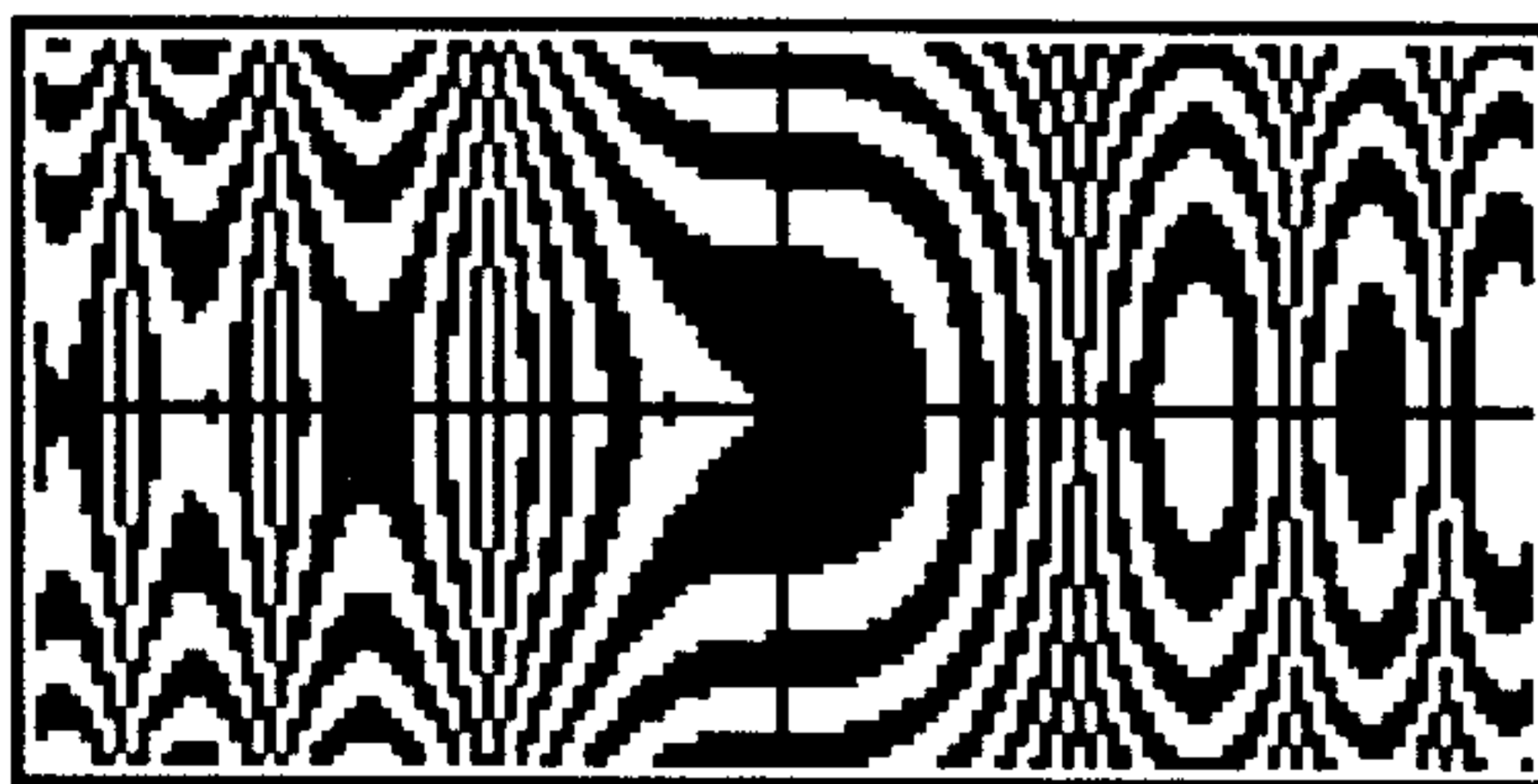
15.8 Truth Plots

A *truth-valued function* is one that evaluates to a logical value *true* or *false*. In HP 48 terms, this translates to an expression or a program that always returns 1 or 0, usually the result of a test command (section 9.3.1). A *truth plot* is then a map of a truth-valued function of two arguments. For each pair of coordinates (x,y) on the graph screen, the current equation object is evaluated; if it returns true, the corresponding pixel is turned on. The truth plot type is selected for DRAW by the command TRUTH (in the PTYPE menu).

For example, consider the function $x^3 + y^2$, mapped to truth values according to whether the function value *mod* 4 is less than 2. To plot the mapping, use

$$'(X^3+Y^2) \text{ MOD } 2 < 4'$$

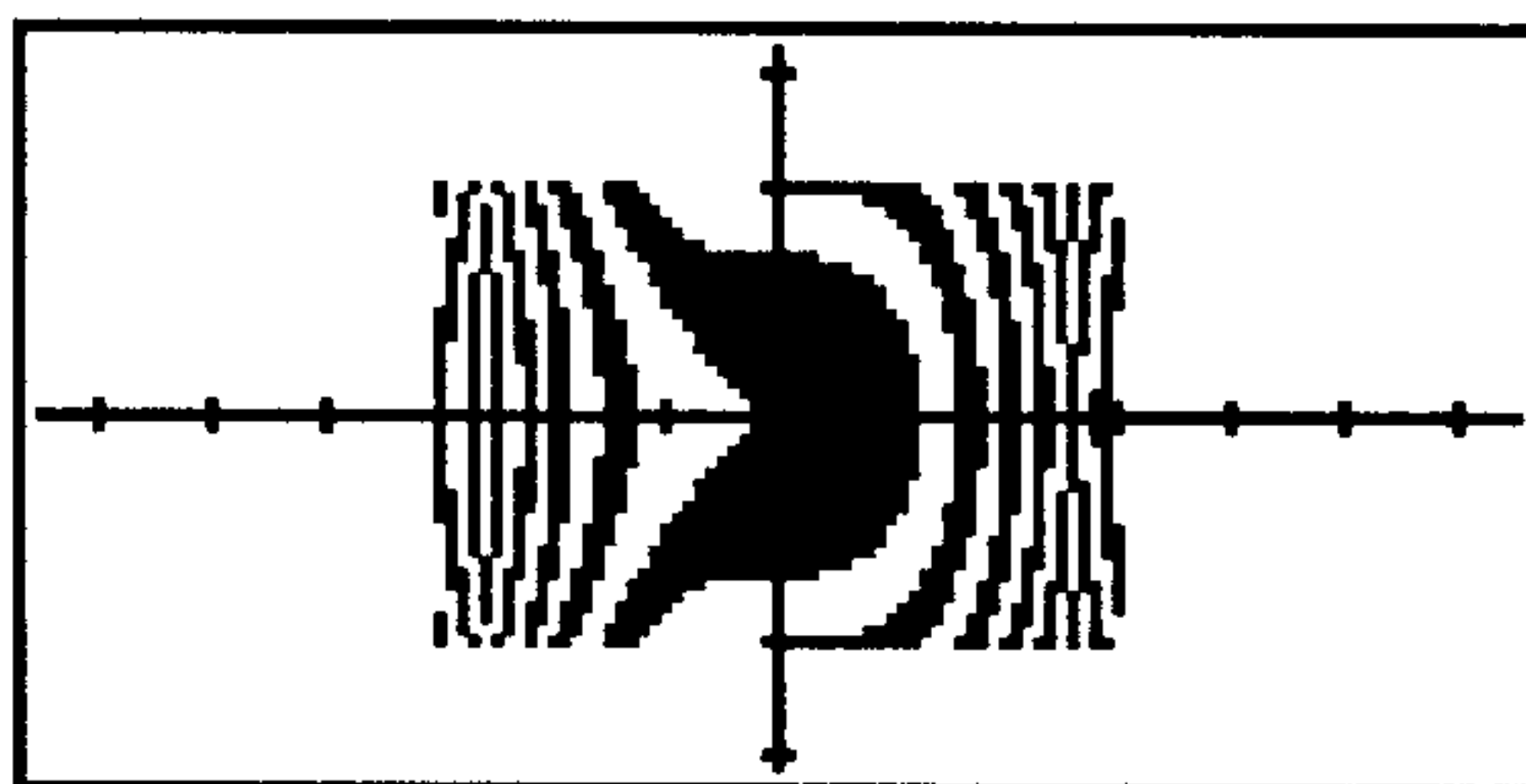
as the current equation, and make a truth plot using the default plot ranges:



Because truth plots involve two variables, you must specify names and ranges for the two variables. The default names are X and Y, and the default ranges are just the horizontal and vertical plot ranges. You can use INDEP (horizontal) and DEPND (vertical) to specify alternate names, and to restrict the plotting ranges to any rectangular region. For example, if you execute

```
-3 3 INDEP -2 2 DEPND,
```

then redraw the previous plot, you obtain:



Restricting the variables' ranges is often helpful because truth plots take a long time to generate. A typical function plot might require plotting 131 points, but a truth plot using the full screen ranges requires over 8000 evaluations of the current equation. You can use the resolution parameter to plot fewer columns, but it does not affect the y-increment--every row is always plotted.

Because truth plots take so long, it is convenient to have a means of interrupting the plotting such that you can resume it again later without having to start over. The program ITDRAW provides this capability. It creates a truth plot like DRAW, but if you press any key other than **ATTN** during its execution, the program halts. When you press **CONT**, ITDRAW resumes. While the program is suspended, you can perform any normal operations except those that might change the graph screen; if you want to do plotting, you should save the current picture, and restore it before continuing ITDRAW.

ITDRAW	Interruptible Truth DRAW	OBEF
<pre><< 'EQ' RCL IF DUP TYPE 6 == THEN RCL END (0,0) PIX? DROP PPAR OBJ→ DROP 4 ROLLD 3 DROPN ROT C→R 5 ROLL C→R → eq x y xmax ymax xmin ymin << PICT SIZE 1 - B→R SWAP 1 - B→R xmax xmin - SWAP / ymax ymin - ROT / → dx dy << 'x' x IF DUP TYPE 5 == THEN OBJ→ DROP 'xmax' STO 'xmin' STO END 'y' y IF DUP TYPE 5 == THEN OBJ→ DROP 'ymax' STO 'ymin' STO END STO STO { #0 #0 } PVIEW xmin xmax FOR x ymin ymax FOR y IF eq →NUM THEN '(x,y)' →NUM PIXON END IF KEY THEN DROP HALT { #0 #0 } PVIEW END dy STEP dx STEP >> { x y } PURGE >> >></pre>	Get current equation. Be sure PPAR and PICT exist. Create local variables. Compute number of steps. x and y increments. If x is a list, correct limits. Same treatment for y. Store local names in global variables. Show the plotting. x loop. y loop. If eq evaluates to <i>true</i>, then turn on the pixel. If a key has be pressed, then suspend execution. Restore the plot view. Purge the global variables.	

15.8.1 Julia Sets

The Mandelbrot set and other Julia sets provide an excellent illustration of truth plots. Julia sets are computed by the iterative process

$$z^2 + c \rightarrow z$$

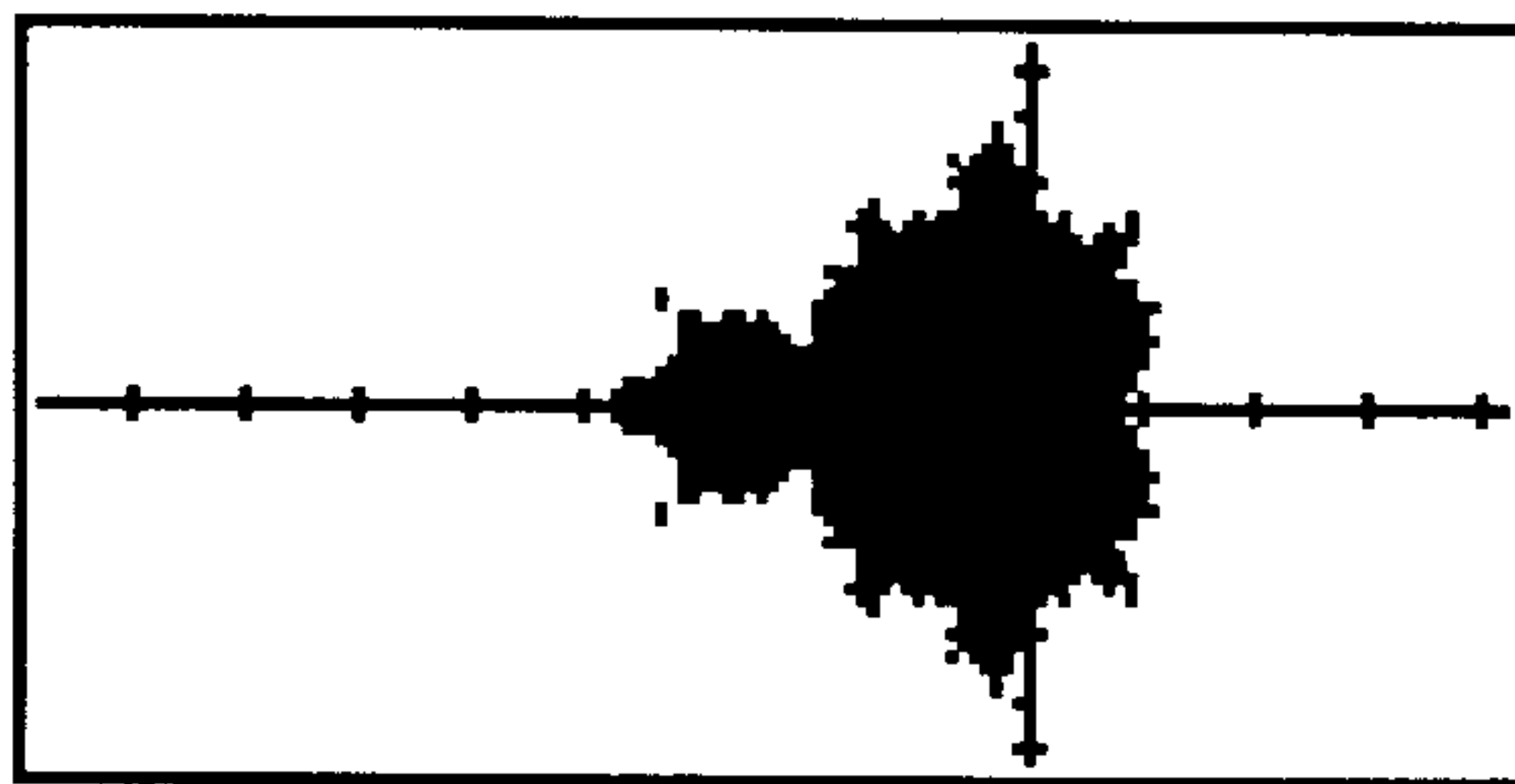
where z is a point in the complex plane, and c is a constant. A point z is in the Julia set for c if the iteration never diverges. The Mandelbrot set is a variation in which the iteration always starts at $z=0$, and the complex plane represents the constants c . To plot the Mandelbrot set, you can use the following program as the current equation:

MAND.EQ	Mandelbrot Current Equation	EAFB
<pre><< X Y R-C DUP 1 15 START SQ OVER + NEXT SWAP DROP ABS 3 < >></pre>	c and $z_0 + c$ Iteration loop. $z^2 + c.$ Return <i>true</i> if $z < 3$.	

With MAND.EQ selected as the current equation,

```
-21 CF -3.375 1.625 XRNG -1.192 1.231 YRNG ERASE DRAW
```

produces this picture:



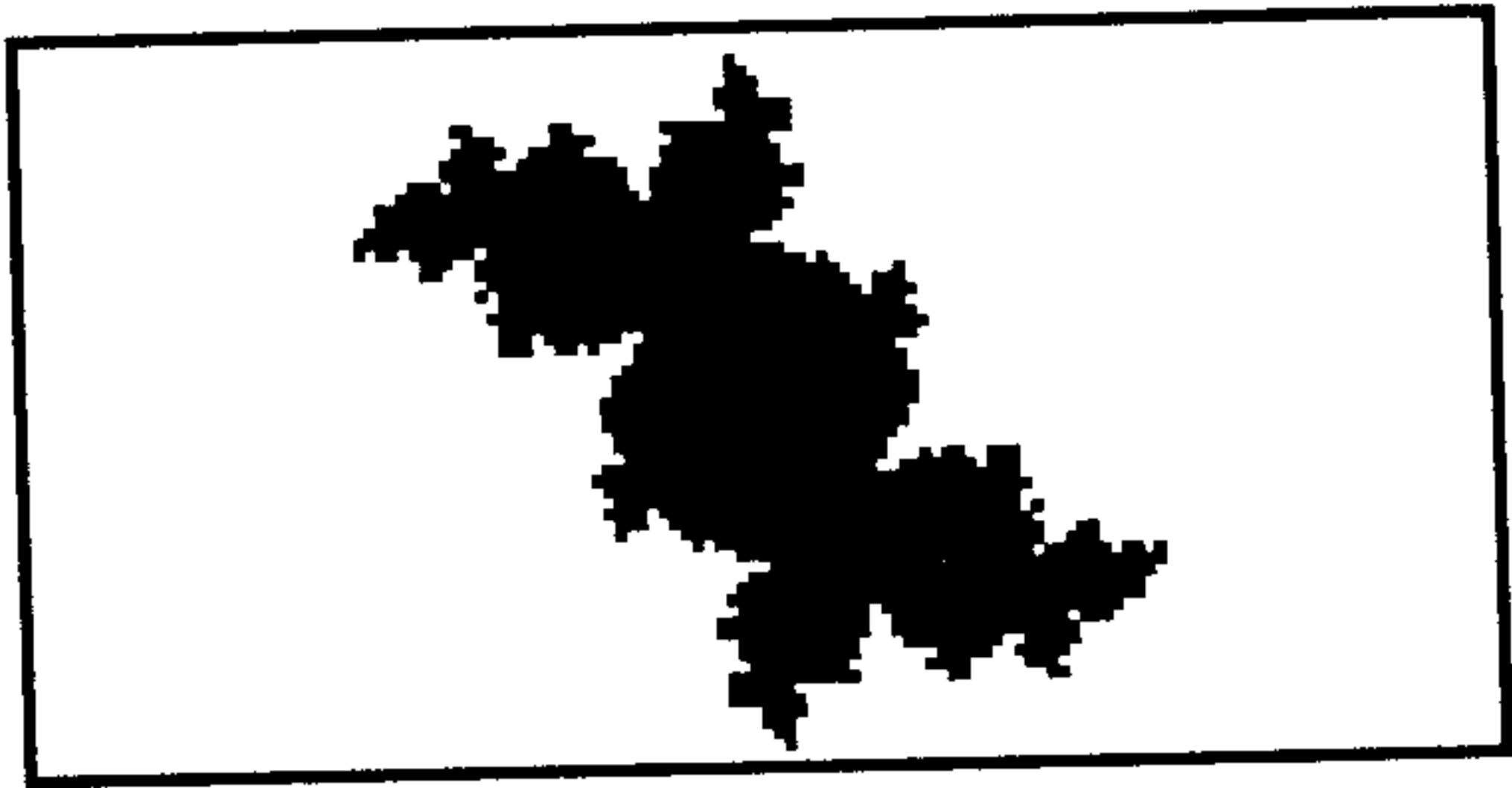
Fifteen iterations is a reasonable choice for the amount of detail possible with the plot ranges set to show the entire Mandelbrot set. Smaller scale pictures ("zooms") of regions near the boundary of the Mandelbrot set contain more interesting patterns, but

Plotting

more iterations per point are necessary to resolve the set to an acceptable level of detail. Choosing Julia set constants c from values at the Mandelbrot set boundary also produces attractive Julia set patterns. The following program JULIA is a generalization of MAND.EQ that can be used to plot either Julia sets or Mandelbrot zooms:

JULIA <i>Julia Plot Utility</i> 8566				
<i>level 3</i>	<i>level 2</i>	<i>level 1</i>		<i>level 1</i>
(c_x, c_y)	(x_0, y_0)	N		<i>flag</i>
<< 1 SWAP START SQ OVER + NEXT SWAP DROP 3 ABS < >>				Iteration loop. Return <i>true</i> if $ z < 3$.

JULIA takes as its arguments: (c_x, c_y) , the Julia set constant; (x_0, y_0) , the starting point of the iteration; and N , the number of iterations per point. For plots of the Mandelbrot set, (x_0, y_0) is always (0,0), and (c_x, c_y) are the points on the graph screen. For Julia sets, (c_x, c_y) is different for each plot, and (x_0, y_0) are the pixel coordinates. For example, this picture of the Julia set for $c = (-.11, .65)$



was produced by this sequence:

```
.34 DUP SCALE (0,0) CENTR { X -1.344 1.344 }  
INDEP { Y } DEPND ERASE DRAW (or ITDRAW)
```

with

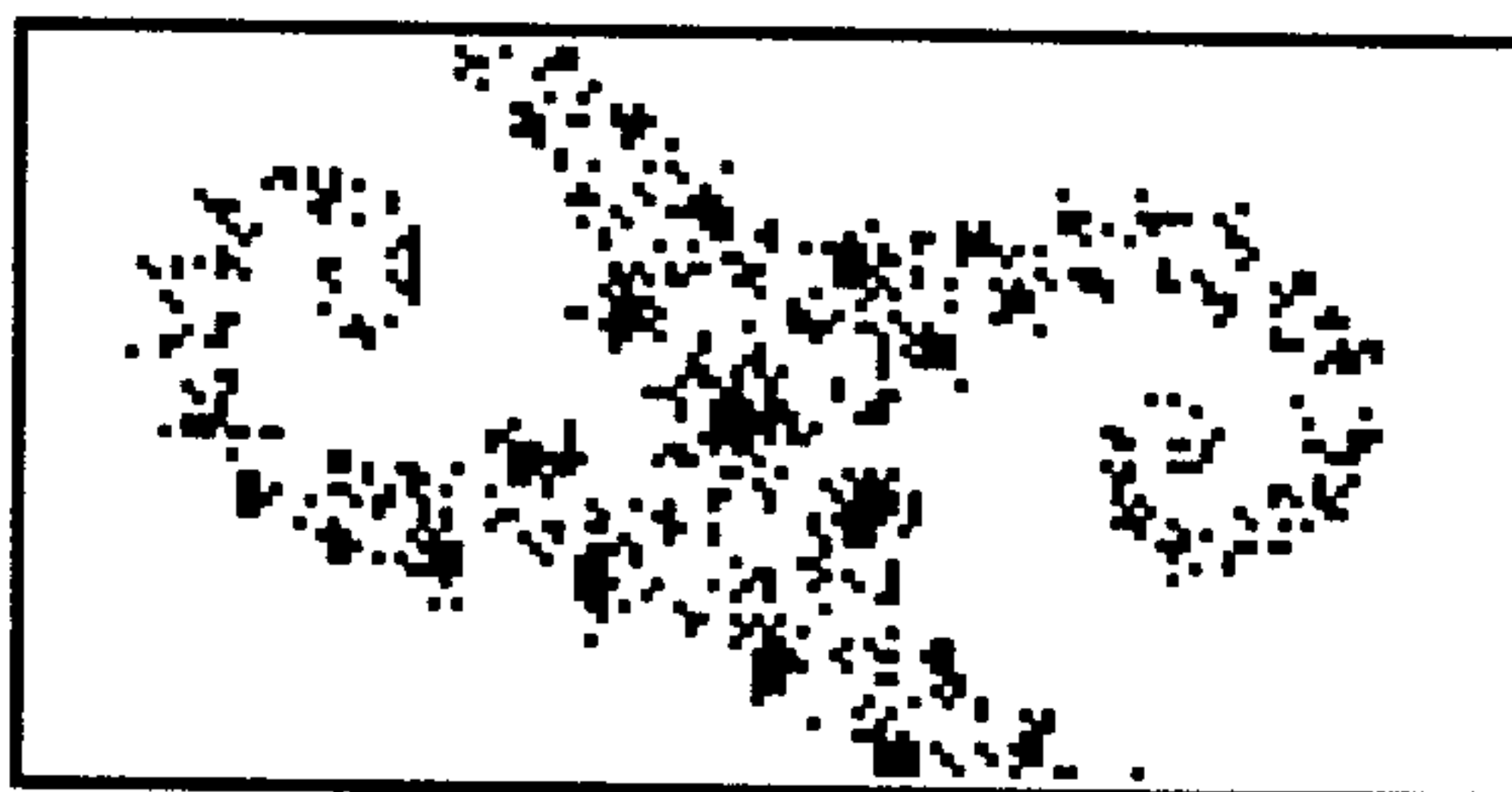

```
<< (-.11,.65) X Y R→C 25 JULIA >>
```

as the current equation.

For plots of regions that include a lot of points that are not in a Julia set, i.e. for which the iteration diverges, it may be faster to use a variation of JULIA that is slower per iteration but which terminates as soon as the iteration begins to diverge:

XJULIA <i>Julia Set Utility with Exit</i>					234D
<i>level 3</i>	<i>level 2</i>	<i>level 1</i>		<i>level 1</i>	
(c_x, c_y)	(x_0, y_0)	N	$\rightarrow$	<i>flag</i>	
<pre><< 1 SWAP IFERR START SQ OVER + IF DUP ABS 3 > THEN 0 DOERR END NEXT DROP2 1 THEN DROP2 0 END >></pre>					Loop parameters. Error trap for termination. Do the iteration. If $z > 3$, Terminate the iteration. Return <i>true</i>. Terminated, so return <i>false</i>.

Julia set pictures may take several hours to plot; unfortunately, some of the most interesting patterns may take 200 or more iterations per point, which is almost prohibitive in time and battery consumption. However, using the HP 48 compiler tools that Hewlett-Packard has published, it is possible to write an assembly language version of JULIA that is fifteen times as fast as the user-language version. This zoom of the Mandelbrot set near $(-.7454, .1130)$ was computed with 255 iterations per point:



15.9 Scatter Plots

A *scatter plot* is a plot of individual disconnected points that are derived from measured or computed data. The term *scatter plot* indicates that the plotted points appear to be scattered about the graph, in contrast to continuous plots, where the points usually follow each other in a regular progression that forms a smooth curve. DRAW makes scatter plots from data stored in the current statistics matrix, when the SCATTER plot type option is selected. Most of the concepts that we have studied in this chapter for continuous plots carry over directly for scatter plots--digitizing, plot scaling and rescaling, etc. Of the various plot parameters stored in PPAR, only the *resolution* is not relevant for scatter plots.

DRAW takes the data for a scatter plot from the statistics matrix variable ΣDAT (section 20.1), which plays a role for statistics analogous to EQ for function plotting and solving. Scatter plots are made from any two of the columns of the matrix (ΣDAT must have at least two columns for this purpose). By default, the first column of data corresponds to the horizontal coordinate, and the second column to the vertical. To make different choices, use i XCOL (section 20.3) to select the i th column as the horizontal dimension, and j YCOL to identify the j th column for the vertical. (i j COL Σ selects both columns at once.) The selected coordinates are not independent and dependent as defined for continuous plots, since the points are just coordinate pairs; but we'll call them x and y as before. The independent and dependent variable entries in PPAR are used only for axis labels, as explained in section 15.2.7.

For example, if your ΣDAT looks like this,

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix},$$

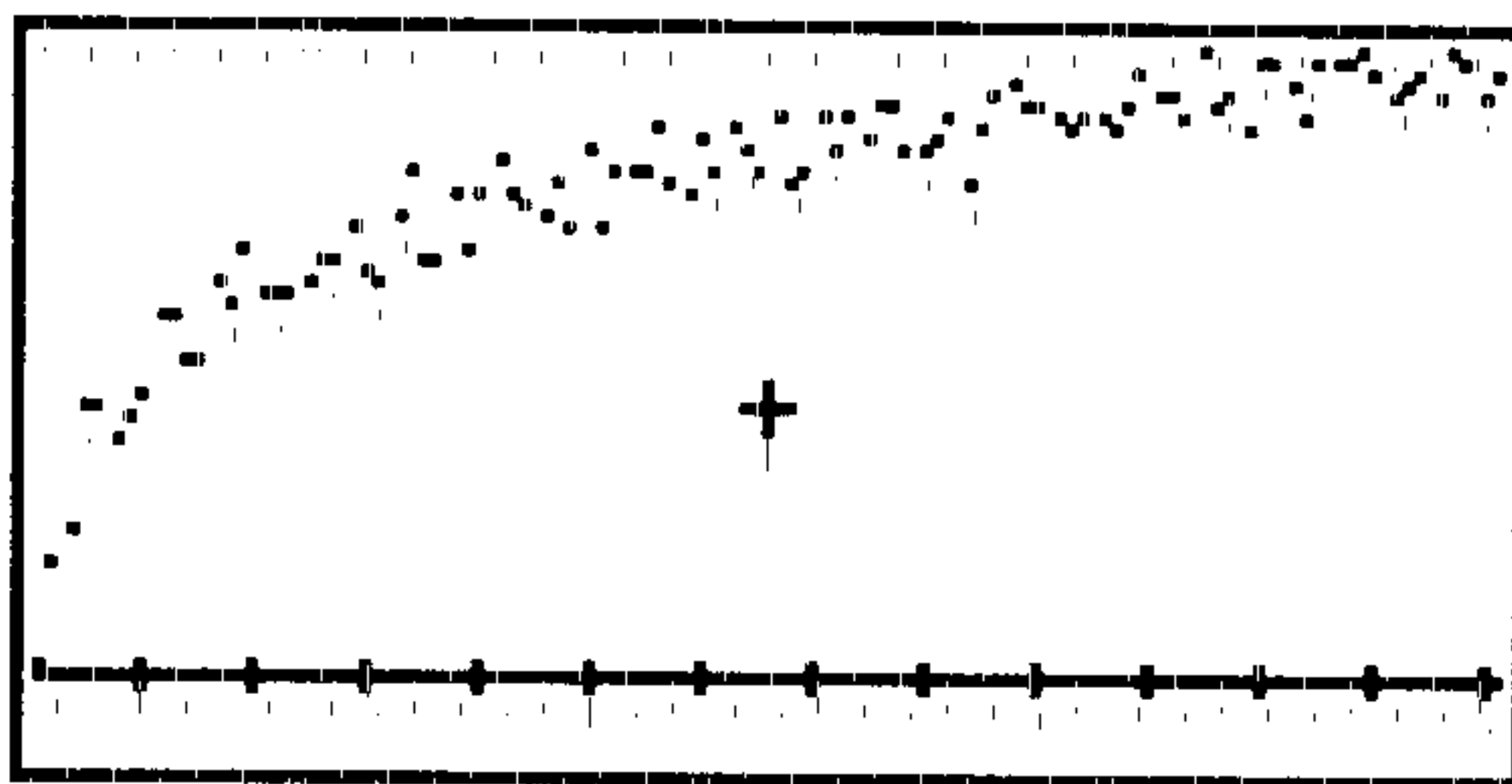
then with column 1 selected as the horizontal coordinate, and column 2 as the vertical coordinate, DRAW plots the points (11,12), (21,22), and (31, 32). 3 XCOL 2 YCOL DRAW plots the points (13,12), (23,22), and (33,32).

An example of a scatter plot is given in section 12.11.1.2 to demonstrate the use of a random number generator. Another example is given by plotting the "noisy" logarithmic curve

$$y_i = 5 \ln x_i + b_i,$$

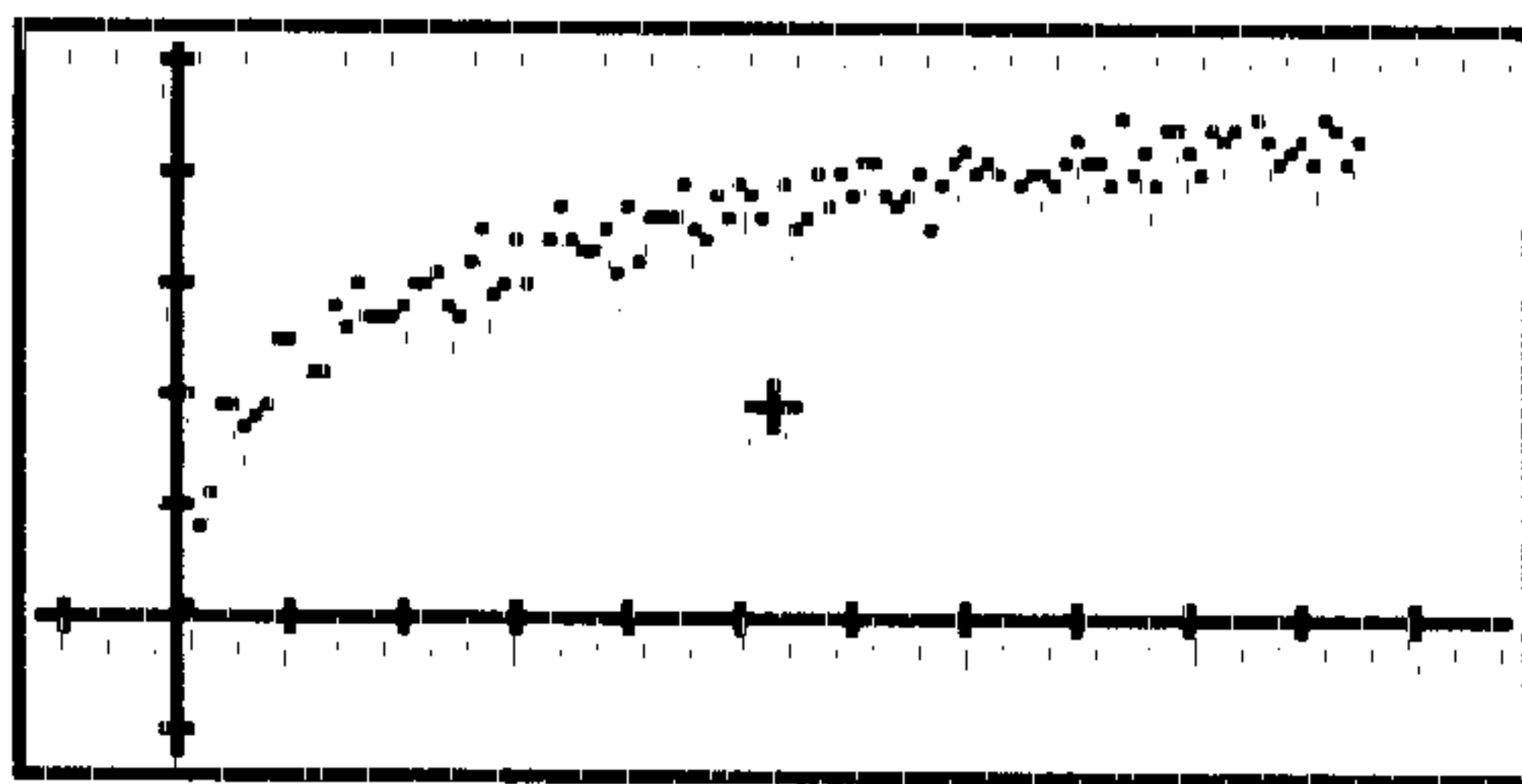
represented by the data generated in the example in section 20.3.2.1. After creating the

data, you can create a scatter plot using **SCATR** (in the third page of the **STAT** menu; also execute 1 XCOL 2 YCOL if necessary):



SCATRPLOT is a shortcut command that automatically selects the scatter plot type, then executes **AUTO** to produce a scatter plot. Autoscaling a scatter plot chooses x - and y -ranges according to the minimum and maximum x and y values in the data, so that there will be at least one point plotted at each edge of the screen, and none outside, with the usual addition 15% added at the bottom for the menu labels. For the current data, this view "hides" the y -axis; zooming out by a small factor frames the plot more nicely:

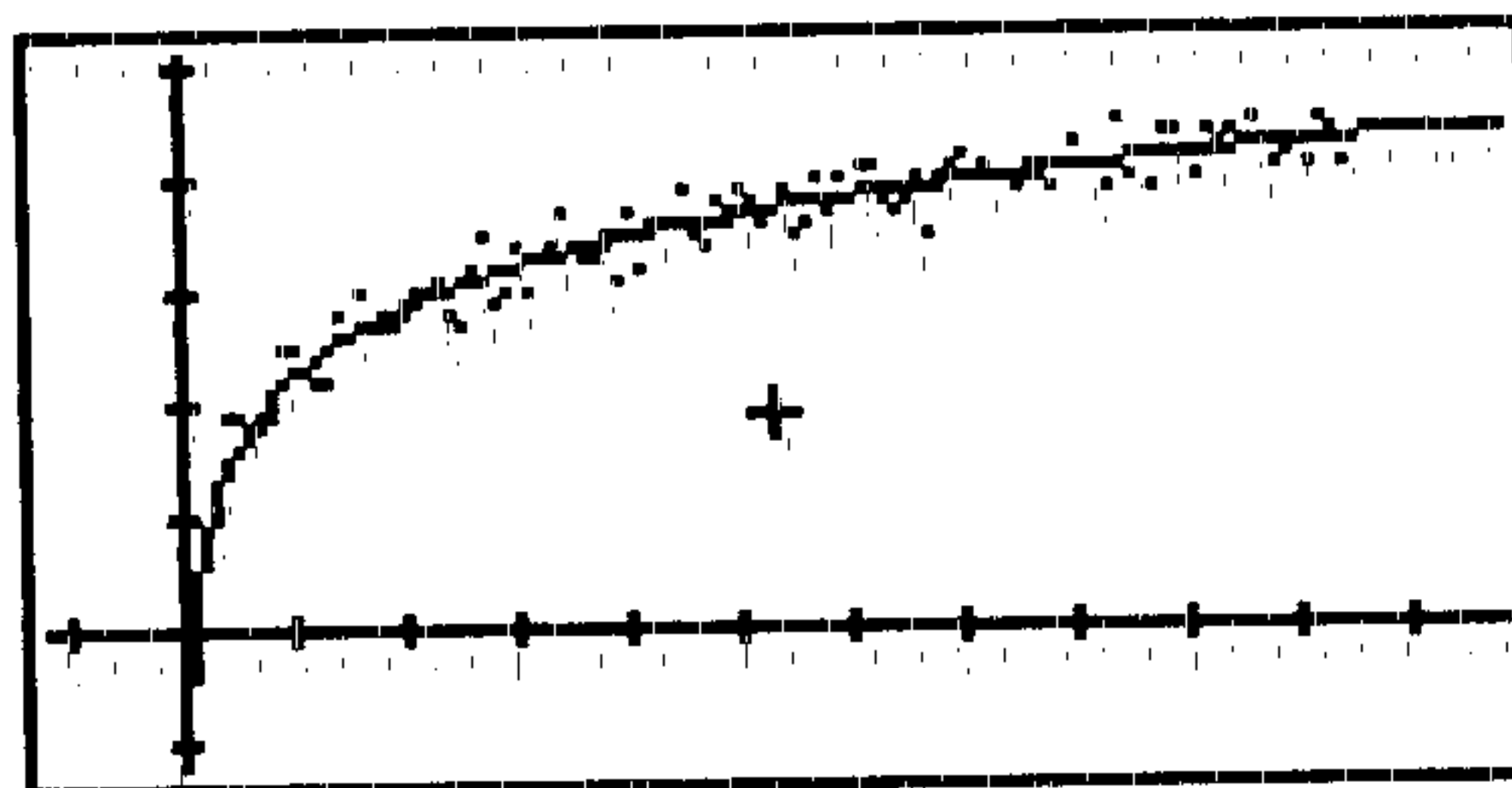
ZOOM **XY** 1.25 **ENTER** :



The command **SCLΣ** does the same autoscaling as **AUTO** for scatter plots, without doing the actual plotting. This command is not present in any menu; it is provided for HP28 compatibility.

15.9.1 Plotting Curve Fits

When you have data that corresponds to one of the linear or pseudo-linear curve models, you can determine the model coefficients and other properties of the least-squares fit curve directly from the plot environment. The **FCN** key (see section 15.4.2) computes and draws the curve that represents the least-squares fit of the current model to the scatter plot data:



(You may need to reset the resolution to 0 to reproduce this plot, if you have changed it while working some of the examples in previous sections.)

FCN changes the plot type to **FUNCTION**, and stores the expression for the fit curve in **EQ**, including the best-fit parameters. Pressing **FCN** again activates the function menu for further analysis. If you exit from the plot environment and recall the contents of **EQ**, you will find an algebraic object representing the fit curve; in the current example, the object is

$$'1.5848798893 + 4.90171452 * \text{LN}(X)'$$

The coefficients are the same as those computed by **LR** (section 20.3.2) for this data. The difference between the coefficients in this expression and the ideal values of 1 and 5 arises from the random noise included in the data.

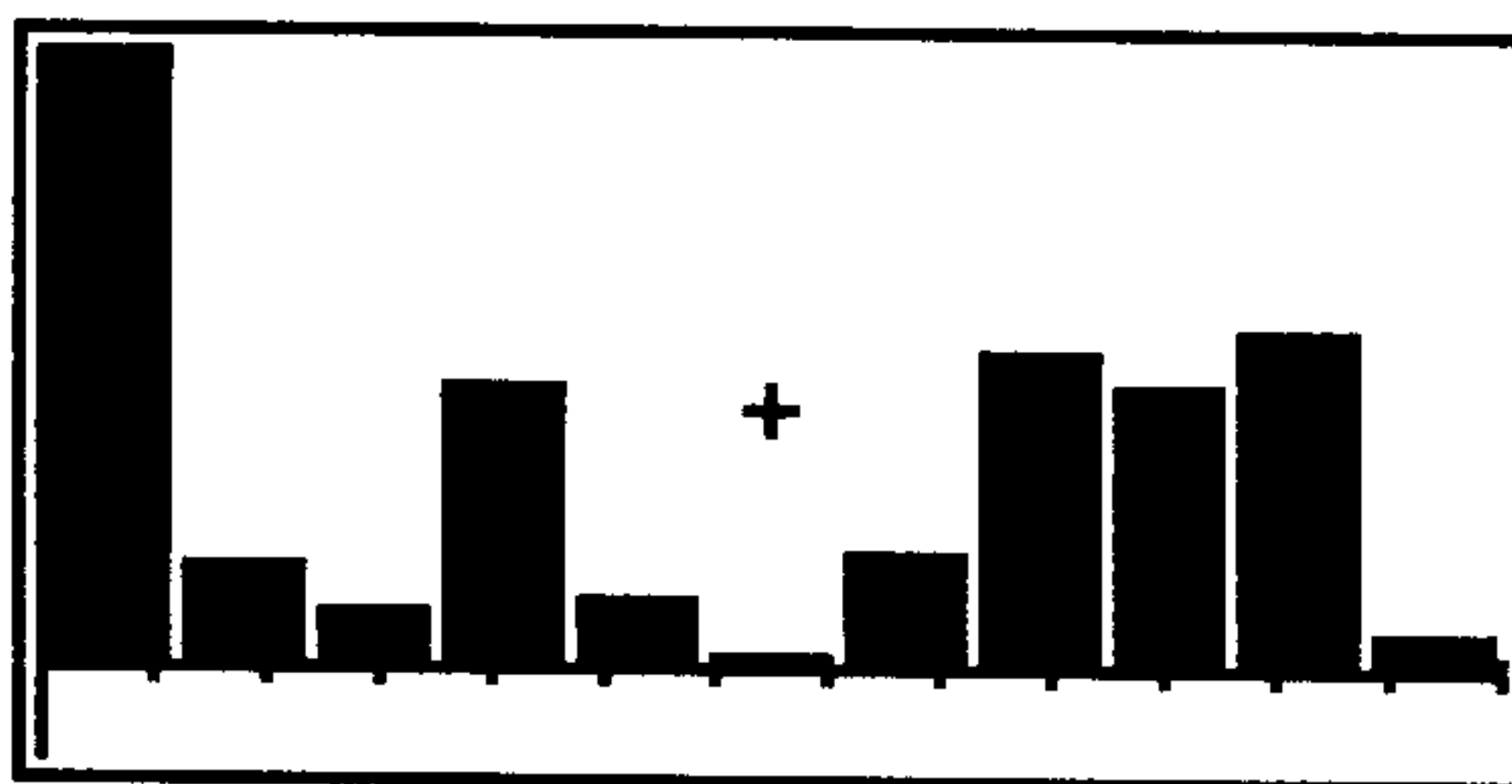
15.10 Bar Charts

A bar chart (plot type **BAR**) represents data values with vertical bars of lengths proportional to the values. This provides a nice visual presentation of data consisting of relatively few points, especially when there is no well-defined mathematical relationship between the points. **DRAW** plots a single bar for each data point in the current statistics matrix, from left to right in the order in which the points appear in the matrix. The points are taken from the column specified in **ΣPAR** by **XCOL**.

For example, the following are the monthly sales figures for one of Larken Publications' books for the period from August, 1990, through June, 1991:

<i>Month</i>	<i>Quantity</i>
August	485
September	74
October	40
November	216
December	55
January	7
February	85
March	246
April	221
May	266
June	22

This data would make a relatively sparse scatter plot, and is better represented as a bar chart:



This picture was created by entering each of the values from the table above, by entering each successive sales number with $\Sigma+$ (section 20.1), then pressing

1 XCOL BARPL

(in the third page of the STAT menu). The command BARPLOT provides a convenient means of creating a bar chart directly from the statistics menu. It automatically selects BAR as the plot type, then executes AUTO. The latter command scales bar charts as follows:

- The x-range is set from 0 to n , where n is the number of data points in Σ DAT.

- The y-range is set so that the highest bar (most positive data point) just reaches the top of the graph screen. $y_{\min}$ is chosen to be $y_{\max}$ less 115% of the distance from the top of the highest bar to the bottom of the lowest bar (most negative). On a default 64-pixel high screen, this just leaves room for the plot environment menu labels at the bottom of the screen.
- The resolution parameter is set to zero, so that the bars have a width of one logical unit. In combination with the x-range choice, this insures that the bars just fill the graph screen in the horizontal direction.

The bars are presented on the screen in the order in which the corresponding data points appear in ΣDAT , so there is only a weak relationship between the bars and the x-coordinate. With the plot parameters chosen by AUTO, the x-coordinate at the right edge of a bar is approximately equal to the bar number. In general, the bars are plotted, starting at the left edge of the screen, next to each other separated by a blank column, with widths determined by the resolution parameter:

Parameter	Bar Width
0 (default)	1 unit
Real number n	n units
Binary integer $\#n$	n pixels

15.11 Histograms

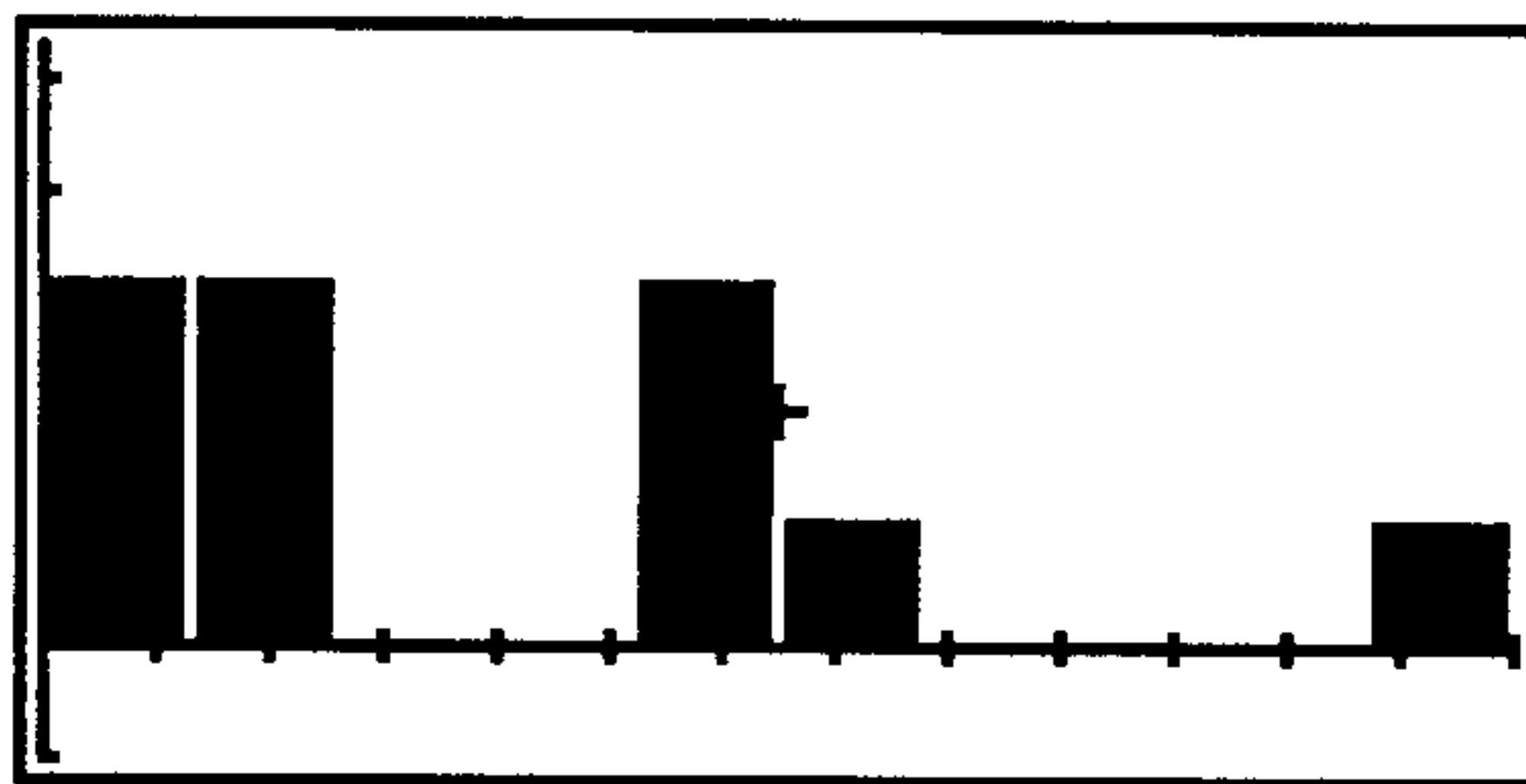
A *histogram* is a bar chart made from data after it has been sorted into numerical ranges, where each bar represents the number of data points that fall into a particular range. With HISTOGRAM as the selected plot type, DRAW counts the elements in one column (specified by XCOL) of ΣDAT that fall into ranges determined by the plot parameters in PPAR. In particular, the horizontal range is divided into N equal intervals, where

$$N = \text{CEIL} \left(\frac{x_{\max} - x_{\min}}{\text{resolution}} \right)$$

DRAW then makes a bar chart of the interval counts, with bars (approximately) as wide as the intervals they represent. As you can see from the formula above, the interval width is equal to the resolution parameter in PPAR. If the resolution does not divide evenly into the horizontal range, the rightmost bar will be narrower than the others. For a default resolution of 0, DRAW divides the horizontal range into 13 intervals, making the bars 10 pixels wide on a standard size graph screen.

For example, we can plot a histogram of the Larken book sales data from the previous section:

```
0 500 XRNG -1 5 YRNG 50 RES HISTOGRAM ERASE DRAW
```



Here you can see that the monthly orders tend to be of three sizes: fewer than 100, between 200 and 300, and more than 450.

The command HISTPLOT, which appears in the third page of the **STAT** menu, executes HISTOGRAM to select the histogram plot type, then AUTO to select plot ranges and draw the histogram. AUTO uses the minimum and maximum values in the selected Σ DAT column as the x -range, and uses the current resolution parameter to determine the bar widths. The y -range is set with $y_{\max}$ equal to the number of entries in Σ DAT, and $y_{\min} = -.15y_{\max}$, leaving room for menu labels. Usually, this makes the bars fairly short, but it does insure that the full vertical extent of the bars will fit on the screen regardless of the actual distribution of the data.

You can make full use of the y range for histograms by computing the histogram data separately, then using DRAW to plot a bar chart from the data. The histogram sorting is done by BINS, so-called because histogram intervals are often colloquially named "bins."

BINS (in the second page of the **STAT** menu) takes three arguments:

- $x_{\min}$ (level 3), which specifies the low (most negative) end of the data range that is to be counted;
- w (level 2), the bin width; and
- N (level 1), the number of bins.

BINS sorts the elements in one column (selected by XCOL) of Σ DAT into $N+2$ bins.

For each element x in ΣDAT , the n th bin count b_n is incremented, where

$$n = \text{IP} \left(\frac{x - x_{\min}}{w} \right) + 1 \quad \text{for } x_{\min} \leq x < x_{\max}$$

where $x_{\max} = x_{\min} + N \cdot w$. b_N is also incremented if $x = x_{\max}$. BINS returns the bin counts to level 2 as a one-column, N -element matrix:

$$[[b_1] \ [b_2] \ \dots \ [b_N]]$$

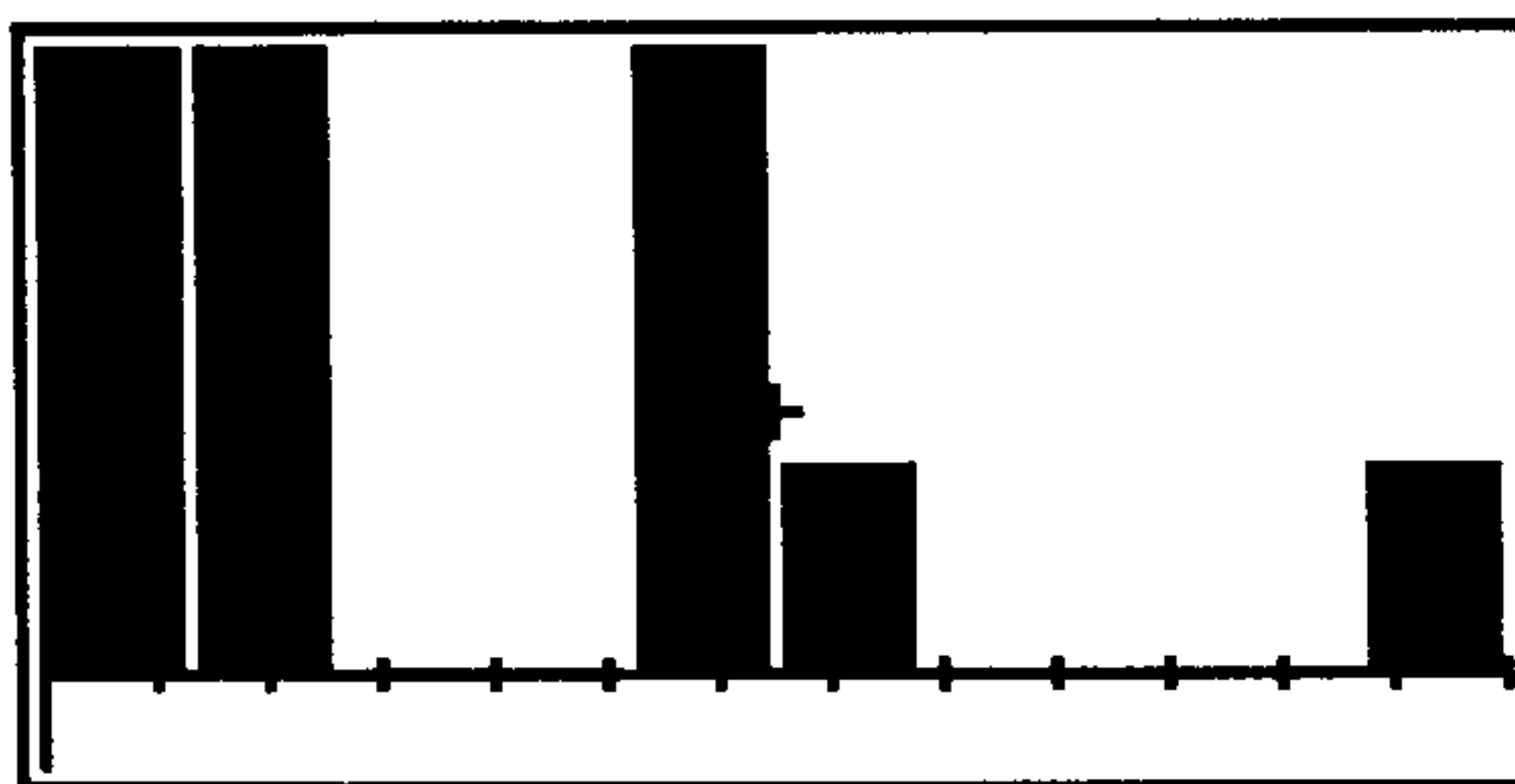
It also returns a two element vector $[b_< \ b_>]$ to level 1, where $b_<$ is the count of data points $x < x_{\min}$, and $b_>$ is the count of data points $x > x_{\max}$. For example, with the book sales data still in ΣDAT ,

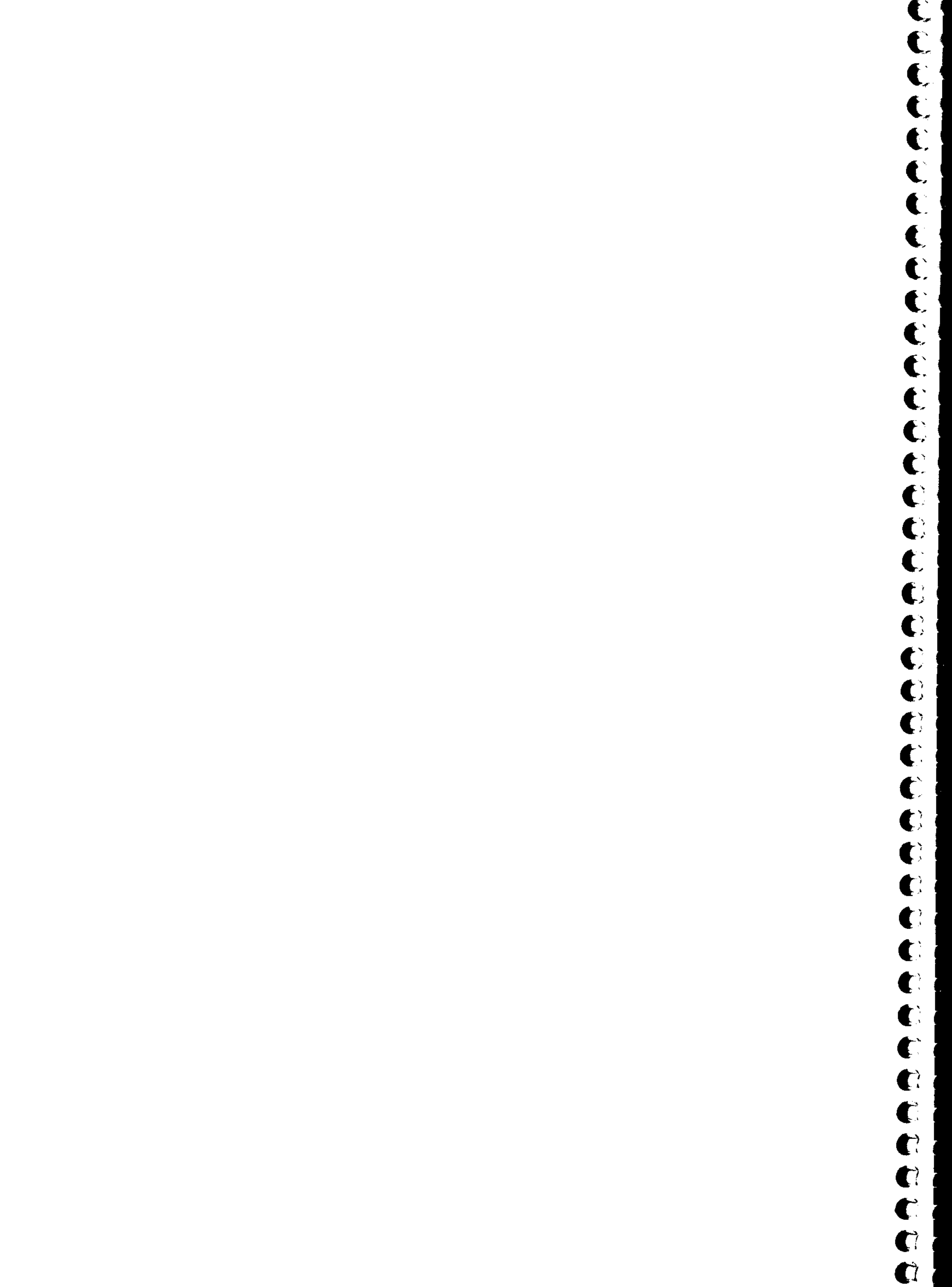
```
0 50 10 BINS  [[3][3][0][0][3][1][0][0][0][1]] [0 0]
```

You can observe that the maximum bin count is 3, so a y range from $-.45$ ($3 \times .15$) to $+3$ is appropriate. Thus, if you store the level 2 matrix in ΣDAT , and execute

```
0 500 XRNG  $-.45$  3 YRNG 50 RES BAR ERASE DRAW,
```

you obtain this picture:





16. Symbolic Objects and Solutions

The HP 48 and its predecessor the HP 28 are unique among calculators in their ability to apply mathematical operations to “symbolic” quantities--ones for which no numerical value has been assigned. In these calculators, the quantities are represented by objects, specifically names and algebraic objects.

In section 3.5, we discussed the procedure-class nature of algebraic objects, and their similarities and differences with program objects. The name *algebraic object* is not a particularly good one, since the objects may represent a wider variety of expressions than a strict use of the mathematical term *algebraic* might suggest. The object name originated in the early days of the HP28C design, when there was still considerable debate among calculator users over the relative merits of RPN calculators and so-called “algebraic” calculators. The HP 28 was intended to end that debate by combining the strengths of both calculator types into a unified interface; algebraic objects were so named to convey that the HP 28 could deal with objects formulated with a syntax familiar to users of algebraic-style calculators.

In subsequent discussions, we will favor the term *symbolic object*, which includes algebraic objects and global and local names. Name objects can often be considered as representing elementary expressions, and are usable in most contexts where algebraic objects are allowed.

16.1 Motivations

If you're a student learning algebra or calculus, or using their techniques in other mathematical or scientific studies, the HP 48's symbolic capabilities may be very exciting to you. However, if you're not directly interested in algebra for its own sake, you might wonder why symbolic capability in a calculator is important to you.

The concept of “multiple views” is popular among educators in mathematics, referring to different representations of a function. Each type of view provides different insights into the nature of a function. One view is numerical, describing a function in terms of a table of its values for various arguments. This is the only representation of a function that is possible with a simple numerical calculator. In recent years, graphing calculators have offered a second view--function plots that show functions' behavior over extended argument ranges. The HP 28 and the HP 48, however, remain the only calculators that offer the third view--the symbolic representation of a function as a mathematical expression--plus the ability to manipulate the expression itself.

Actually, if you use a programmable calculator at all for more than simple keyboard

arithmetic, you are performing a kind of symbolic operation. Any time you perform a calculation more than once, using varying data, you probably represent the calculation symbolically at some point. In particular, when you write a program to automate the calculation, that program is a symbolic operation. You write it to accept certain inputs, without specifying their values, and to compute an unknown result. This is no different in principle from writing a mathematical expression on paper. An expression also “works” with unspecified inputs (variables) and returns a previously unknown value when you evaluate it.

So, in the sense that any program is a symbolic calculation, any programmable calculator is a “symbolic” machine. The major contribution of the RPL-based calculators is that they allow you to apply mathematical operations to the programs themselves, and obtain new programs as results. For example, consider a program that recalls the value of a variable and doubles it. In a BASIC-language computer, for example, the program might look like this, where the “variable” is named X:

```
100 Y=2*X  
200 END
```

But suppose that after entering the program you realize that you are really interested in the sine of the result, $\sin(2x)$. In BASIC, you must modify the program by editing it: find the correct program line and enter the SIN in the right place, being sure to include parentheses.

On the HP 48, the original “program” consists of the symbolic object ‘2*X’. To change this into the new program ‘SIN(2*X)’, all you have to do is execute SIN when the original expression is in level 1. The parentheses are automatically inserted. In effect, the calculator writes a new program for you--all you have to do is use the same keystrokes on the symbolic “program” as you would use with a numerical quantity.

Another way to see the value of the HP 48 capabilities is to consider a general problem-solving process that consists of these steps:

1. Identify the problem.
2. Determine the known and unknown quantities.
3. Figure out the mathematical relationships between the quantities.
4. Solve the relationships for the unknowns in terms of the knowns.
5. For each set of known quantities, evaluate the solved relationships to obtain numerical values for the unknowns.

When you use a conventional calculator, the calculator can only enter the process at the final stage. Once you have equations for the unknowns, you can program those equations into the calculator, enter numerical values for the known variables, and run the programs to return the numerical values for the unknowns. The HP 48, on the other hand, can enter the process as early as step 2. You can use its symbolic capabilities to work out the relationships and solve for the unknowns--steps for which you would need pencil and paper when using another calculator. The symbolic solution that you find with the HP 48 is also the "program" you can use for repeated evaluation of the unknowns with different inputs. Even if the equations you derive can not be solved in closed form for the unknowns, you can still plot the equations or use HP Solve to obtain numerical results, without any further programming.

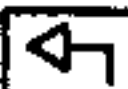
As an example of this process, consider the classic introductory calculus problem:

A farmer has 100 yards of fencing to enclose a rectangular field, which is bounded on one side by a river. What length (L) and width (W) of the field gives the maximum area?

■ *Solution:*

Steps	Keystrokes	Results
1. The length of the fence is 100 yards.	'L+2*W=100_yd' ENTER	'L+2*W=100_yd'
2. Solve for L.	'L' ◀ ALGEBRA ≡ISOL≡	'L=100_yd-2*W'
3. Assign this value to L.	◀ DEF	
4. The area of the field is L times W.	'L*W=AREA' ENTER	'L*W=AREA'
5. Substitute for L.	EVAL	'(100_yd-2*W)*W=AREA'
6. To find the maximum area, differentiate the expression.	'W' ENTER ▸ ∂	'-(2*W)+(100_yd-2*W)=0'
7. Collect terms.	≡COLCT≡	'100_yd-4*W=0'
8. Solve for W.	'W' ≡ISOL≡	'W=25_yd'

9. Assign this value to W
and evaluate L.

 **DEF** L **EVAL**

50_yd

Answer: The width of the field should be 25 yards, and the length 50 yards.

You can use the HP 48 to formulate and solve the entire problem. With a conventional calculator, all you can do is evaluate the final (dimensionless) answer after you have worked it out on paper, keeping track of the units yourself.

Another example is given by the program SIMEQ (section 12.11.3), which solves a system of simultaneous linear equations. Many other calculators provide this capability either through built-in commands or as program applications. However, without exception (including the HP 48's own built-in method using matrices and vectors), these require you to enter the coefficients and constants rather than the equations themselves. In other words, you must do the work yourself of inspecting the equations, collecting terms and rearranging if necessary, to determine the coefficients and constants. The SIMEQ program lets you enter the equations in any order, and without having to structure the individual equations in any particular way. It is the HP 48's ability to deal with expressions and equations as data to be manipulated--as symbolic objects--that makes it possible for you to write a program like SIMEQ in a straightforward, compact manner. In other calculator languages, writing a program like SIMEQ would require considerable ingenuity, and would likely end up being harder to use than the usual method of entering coefficients in order.

Some people express disappointment in the HP 48's limited symbolic *manipulation* facilities--rearranging and collecting terms, solving equations, integration and the like--especially when compared to the features of computer-based algebra systems. The HP 48's capabilities are nevertheless useful, as we will discuss in this and the next chapter, but symbolic manipulation is not the design center of the HP 48. The calculator is oriented towards practical engineering/scientific computation, with a underlying foundation of floating-point arithmetic. Its manipulation operations are intended as an adjunct to the numerical computation. The prime focus of the HP 48's symbolic processing is to provide more power and flexibility than are possible with a numeric-only calculator--the SIMEQ program is a prime example.

The RPL language itself is fully as capable as a basis for computer algebra as LISP or the other similar languages that are commonly used in computer algebra systems. The particular algebra feature set that is implemented in the HP 48 was chosen with regard to practicality on a device with modest processor speed and limited memory. At this writing, no other popular computer algebra system can execute from ROM, and none could possibly execute with only a few kilobytes of free RAM, as the HP 48 commonly does.

The HP 48 takes a generally *conservative* approach to the execution of symbolic operations. This means that it does not attempt to make decisions for you, but allows you to direct a symbolic calculation at each step. When you enter or compute an expression, the HP 48 does not force the expression into any particular form, but provides expression manipulation commands so that you can rearrange it if necessary. For example,

$$'A+B' \quad 'B' \quad + \quad \boxed{\rightarrow} \quad 'A+B+B'$$

The HP 48 does not automatically collect terms to return $'A+2*B'$; if you want that result, you can execute COLCT (section 17.1.1).

Another example of the conservative design is in the symbolic solutions returned by ISOL and QUAD (section 16.4). These commands return expressions representing *all* solutions to their arguments, not just one solution chosen for its “simplicity” or “familiarity.” The solutions are structured so that *you* can choose the solution or solutions that you want.

The HP 48 chooses a conservative approach for several reasons:

- The calculator can not know what you want. The factors that determine a choice of expression form, or of one solution from among many, are usually not contained in the expression itself but come from external considerations.
- There is no “standard” form for expressions.
- Solutions computed by the calculator should be general and should never obscure any possible solution.
- In a finite-precision, floating-point calculation, the *order* of operations is important. Two formally equivalent expressions, such as $'(A+B)+C'$ and $'(A+C)+B'$, may give quite different results when evaluated numerically (see the discussion of expression structure in section 3.5.2.1). When you set up an expression in a manner that takes this point into account, you do not want the calculator to rearrange the expression.
- Symbolic operations often require a large number of individual steps. If the calculator attempted to standardize the result of each step, it would slow down the overall process.

This approach to symbolic operations means that you will often obtain results that don’t “look” like you expect, or which you have to take extra steps to rearrange.

16.2 General Symbolic Problem Solving

In the preceding section, we outlined a five-step general problem solving process. Now we will review those steps, and see in more detail how they are realized on the HP 48. To illustrate the procedure, we will solve the following problem:

Dad is 40 years old, Son is 10. In how many years will Dad be twice as old as Son?

1. Identify the problem.

Sometimes it's helpful to restate a problem in a more general way, by using variables even for values that are already known. For example:

Dad is D years old, Son is S. In how many years T will Dad be N times as old as Son?

This allows you to solve the problem logically once, then enter various choices for D and S, and find a value of N for each set of choices.

2. Determine the known and unknown quantities.

The known quantities are the input parameters for the problem--these might be a single value, or a set of input data, or several sets. The corresponding variables are called *known* variables. *Unknown* variables represent the quantities you are going to calculate. In the example, the known variables are D, S, and N. The single unknown is represented by the variable T.

Keep in mind that the choice of which variables are known and which are unknown, is often arbitrary. In the example, you can specify T and solve for N, rather than the reverse as the problem was originally stated.

3. Figure out the mathematical relationships between the quantities.

This step consists primarily of converting the verbal or conceptual statement of the problem into one or more mathematical relationships. Working the example on the HP 48 proceeds as follows:

Verbal Statement	Keystrokes		Stack
(Purge existing variables.)	{ D T N } $\leftarrow$ PURGE		
Dad is D years old now.	D ENTER	1:	'D'
T years later, he'll be...	T +	1:	'D+T'

Similarly, Son is $S + T$
years old.

' $S + T$ ' **ENTER**

2: ' $D + T$ '
1: ' $S + T$ '

Dad is to be N times as
old:

N **[X]** **[←]** **=**

1: ' $D + T = (S + T) * N$ '

The equation in level 1 is the relationship you need.

4. Solve the relationships for the unknowns.

In this step, you apply standard rules of algebra to rearrange the equation such that the unknown variable is isolated as a single quantity on one side. The command **ISOL** (section 16.4.1) will do this for you automatically if the variable appears just once in the expression or equation. An automatic solution is also possible for general linear or quadratic equations, for which **QUAD** (section 16.4.3) will find one or two solutions regardless of the specific form of the equation. Since the current example is linear in T , **QUAD** can solve it:

' $D + T = (S + T) * N$ ' ' T ' **QUAD** **[↵]** ' $T = -((D - S * N) / (1 - N))$ '

It is also instructive to solve the equation using rearrangement methods, to illustrate the process when neither **ISOL** and **QUAD** is adequate. The steps on the HP 48 are essentially the same that you might use to solve the problem on paper:

Verbal Statement	Keystrokes	Stack
Enter the expression	' $D + T = (S + T) * N$ ' ENTER	1: ' $D + T = (S + T) * N$ '
Expand the product.	[←] ALGEBRA EXPA	1: ' $D + T = S * N + T * N$ '
Subtract T from both sides.	' T ' ENTER -	1: ' $D + T - T = S * N + T * N - T$ '
Simplify.	COLCT	1: ' $D = S * N + T * N - T$ '
Subtract $S * N$ from both sides.	' $S * N$ ' ENTER - COLCT	1: ' $-(S * N) + D = T * N - T$ '

At this point all terms containing T are on the right side of the equation, but since there are two such terms, you still can't use **ISOL**. First, you must use operations in the EquationWriter subexpression mode to merge the two terms. To activate this mode, press **[▽]** **[<]** (if you don't see a menu, press **ATTN**):

$$-(S \cdot N) + D = T \cdot N - U$$

RULES EDIT EXPR SUB REPL EXIT

We want to merge the arguments of the rightmost subtract function by factoring out the common T . To do this, the two arguments must have a similar form. With the object cursor on the last T , press **RULES** ***1** :

$$-(S \cdot N) + D = T \cdot N - T \cdot 1$$

$\div T$ $T \div$ $\div M$ $M \div$ $\div D$ $D \div$

Now select the subtract object by pressing **<** twice. Then press **RULES** to activate the operations menu for subtraction:

$$-(S \cdot N) + D = T \cdot N - T \cdot 1$$

$\div T$ $T \div$ $\div M$ $M \div$ **MF** $\div \div$

The common factor T is on the left in both of the products, so press **-M** to *merge left*:

$$-(S \cdot N) + D = T \cdot (N - 1)$$

÷T T÷ ÷M M÷ ÷D D÷

Now T occurs only once, so the equation can be solved by ISOL. Exit from the EquationWriter environment by pressing **ENTER**, then

$$'T' \text{ ISOL } \rightarrow 'T = (- (S * N) + D) / (N - 1)'$$

This is formally equivalent to the result obtained previously with QUAD, differing only in the order of some of the terms.

There are, of course, many problems for which it is impossible in principle to obtain a closed-form symbolic solution for a variable. $\sin x + x = y$, for example, can not be solved for x . This is where HP Solve is invaluable. If you are willing to forgo a symbolic solution, you can skip the current step in the problem-solving entirely, and use the solver menu to obtain numerical solutions for any equation, no matter how many times the unknown variable occurs.

5. *For each choice of known quantities, evaluate the solved relationships to obtain numerical values for the unknowns.*

Once you have an expression or equation that represents a solution to a problem, the only remaining step is to assign specific numerical values to the independent variables, and evaluate the solution object to obtain the corresponding values for the unknown. You can do this in two general ways on the HP 48:

- Assign values to the independent variables using STO, and evaluate the algebraic object that represents the solution. This causes substitution of the numerical values for the variable names, yielding a numerical value for the unknown. In the current example, you can assign values to D , S , and N from the original problem:

$$40 \text{ 'D' } \boxed{\text{STO}} \quad 10 \text{ 'S' } \boxed{\text{STO}} \quad 2 \text{ 'N' } \boxed{\text{STO}}$$

Then, with the solution $'T = (- (S * N) + D) / (-1 + N)'$ still in level 1,

EVAL $\rightarrow$ 'T=20'

The result shows that in 20 years, when Dad is $40+20=60$, and Son is $10+20=30$, Dad will be 2 times as old as Son. You can easily change values to obtain another result--for example, if you give the value 3 to N, and reevaluate the expression, you find that in 5 years, Dad (45) will be 3 times as old as Son (15). Note that if you want to evaluate the expression several times, you need to make copies of it, or to store it in a variable.

- Use HP Solve (Chapter 14). Make your solution object the current equation by pressing $\leftarrow$ **SOLVE** **ST** **EQ**, and press **SOLVR** to activate the Solver menu. Then,

40 **D** 10 **S** 2 **N** $\leftarrow$ **T** $\rightarrow$ T: 20.

The solver menu makes it easy to solve again using different parameters. For N=3:

3 **N** $\leftarrow$ **T** $\rightarrow$ T: 5.

16.3 Symbolic vs. Numerical Solutions

The following are reasons why a symbolic solution is desirable for almost any problem, and preferable to the purely numerical answers that are provided by conventional calculators:

- A symbolic solution is a “global” solution. You can study the behavior of a problem over a range of inputs, just by looking at the mathematical form of the solution.
- A symbolic solution acts as a “program” that allows you to determine numerical results at any time. Once you have the symbolic solution, you can assign values to the variables and evaluate the symbolic object to obtain specific numerical results.
- Even if you’re using HP Solve for purely numerical answers, it is faster when you want a series of results to rearrange the current equation symbolically so that you can use **EXPR=** rather than solving numerically each time.
- A symbolic expression tells you something about the calculation “history” and parameters that have contributed to an answer. Once you convert an expression to a number, you wipe out the logical trail that led to the number.

To solve a problem symbolically means to take the equations that represent the problem and rearrange them using the rules of algebra until you manage to isolate the unknown variable’s name. (In this discussion, we will use the term *equation* to refer either to an actual equation or to an expression $f(x)$ that is understood to represent the equation $f(x)=0$.) To “isolate” an unknown x means to obtain an equation of the form

$x = f(y, z, \dots)$, where x does not appear in the right side of the equation, and y, z , etc. are known quantities.

The HP 48 provides two types of tools to help you obtain symbolic solutions once you have entered the equation(s) for a problem. First, there are several commands for rearranging expressions, which approximate the steps you carry out in pencil-and-paper calculations. Included among these the EquationWriter RULES operations for detailed expression manipulation, EXPAN (expand), for distributing multiplication and powers, COLCT (collect), for combining like terms, and !MATCH and !MATCH for making a variety of substitutions. Second, there are two automatic expression/equation solvers, ISOL and QUAD, which can carry out several steps in the solving process at once.

ISOL and QUAD are certainly the easiest methods of solving for a variable. However, both have certain restrictions in their application: ISOL yields a true *solution* only if the unknown variable's name appears just once in the equation. QUAD permits multiple occurrences of the unknown's name, but a QUAD result is only a solution if the equation is second order (quadratic) or lower in the unknown. For equations that don't fit either of these criteria, the typical HP 48 symbolic solving process is a combination of the two types of solving tools. You use the expression manipulation commands to convert an equation into a form suitable for final solution by ISOL.

16.3.1 $\rightarrow Q$ and $\rightarrow Q\pi$

One of the problems associated with carrying out symbolic manipulations with floating-point numbers is the loss of information and precision associated with representing rational fractions with decimal numbers. The HP 48 provides two commands, $\rightarrow Q$ and $\rightarrow Q\pi$, that can help you reconstruct a fraction from a number that has been converted into decimal form.

$\rightarrow Q$, which stands for *to quotient*, attempts to find a rational fraction equivalent of a real number, expressed as an algebraic object. For example,

```
3  FIX .494347379239  →Q  ⏏  '43/87'
```

The number display setting is significant:

```
STD .494347379239  →Q  ⏏  '481/973'
```

$\rightarrow Q$ attempts to find a quotient of integers which matches its argument to the number of decimal places specified by the current number display mode. Standard (STD) mode indicates 11 decimal places, and fixed, scientific, and engineering modes specify the

number indicated by the original mode argument, e.g. nine places for 9 FIX, etc. In particular, $\rightarrow Q$ finds the fraction with the *smallest denominator* that matches the argument within the error limit set by the display format. For example,

STD 1.61803398875 $\rightarrow Q$ $\Rightarrow$ '514229/317811'.

Notice that the denominator does not have 11 digits. The fraction 139583862445/86267571272 is another representation of the original argument that evaluates to the same decimal value as 514229/317811; $\rightarrow Q$ chooses the latter because the denominator is smaller.

$\rightarrow Q\pi$ ($\leftarrow$ ALGEBRA menu, second page) is a variation of $\rightarrow Q$ that includes a factor of π in the fraction computation. That is, $\rightarrow Q\pi$ finds a quotient for its argument and another for the argument divided by π . It returns the quotient with the smaller denominator, multiplied by π if appropriate. Since π itself is irrational, and not precisely represented by a floating point number, $\rightarrow Q\pi$ will not necessarily return a result containing π even if you started with a product of (floating-point) π :

$\pi^2 / \rightarrow \text{NUM } 5 \text{ FIX } \rightarrow Q\pi \Rightarrow '1/2*\pi'$

$\pi^2 / \rightarrow \text{NUM STD} \Rightarrow '573204/364913'$

If you want to guarantee that a particular argument converts to a factor of π , you should divide the argument by π , execute $\rightarrow Q$, then multiply by π again:

10 FIX .7853981634 $\pi / \rightarrow \text{NUM } \rightarrow Q \pi * \Rightarrow '1/4*\pi'$

16.4 Automated Symbolic Solutions: QUAD and ISOL

The commands QUAD and ISOL automatically carry out several steps toward the symbolic solution of an expression or equation for an unknown. Each operates on an algebraic object in level 2, and a global name specifying the unknown variable in level 1. Each returns an equation representing the solution, with the name of the unknown variable on the left and an expression for the solution on the right. For example, using ISOL to solve an equation for X,

'((A-B)*X)=(- (A*Y)+C)' 'X' ISOL $\Rightarrow$ 'X=(- (A*Y)+C)/(A-B)'.

X appears only on the left side of the result. This does not change the current value of X; if you want to assign the right-side expression as the new value for X, you can execute DEFINE (section 8.5). Or, if you want to extract the right-side expression itself for

some further calculation, you can take the equation apart using EQ→ (in the first page of the **PRG** **OBJ** menu).

16.4.1 ISOL

ISOL solves an equation for an unknown much the same way you would with pencil and paper. It finds the (first) term containing the unknown, and moves it to the left side of the equation, moving all other terms to the right side. Then, if the unknown is contained in the argument of a function, the inverse of the function is applied to both sides of the equation. If the result does not have the unknown by itself on the left, then the whole process is repeated until only the unknown remains on the left side. For example,

$$'2*X+8=0' \quad 'X' \quad \text{ISOL} \quad \Rightarrow \quad 'X=-4',$$

and

$$'A+B*X/C=D' \quad 'X' \quad \text{ISOL} \quad \Rightarrow \quad 'X=(D-A)*C/B'$$

ISOL can find a solution for an equation if three conditions are met:

1. The unknown variable name appears just once in the equation.
2. The unknown appears only in the arguments of HP 48 *analytic* functions.
3. No variables in the equation contain algebraic objects or programs that have the unknown in their definitions.

If either of the first two conditions is not satisfied, ISOL returns the Unable to Isolate error. The second condition is really part of a circular definition--ISOL will only work with analytic functions, but part of the definition of an HP 48 analytic function is that the HP 48 “knows” its inverse and hence can isolate its argument. The object description *analytic* is loosely derived from the mathematical definition of an analytic function as one that is continuous and differentiable. If you inspect the HP 48 function set, it is usually easy to figure out why a particular function is analytic or not in the HP 48 sense if you keep the mathematical definition in mind. Functions like IP, FP, MOD, or MANT are not continuous, and hence are not classified as analytic. ABS is an example of a function that is continuous but not differentiable--its slope changes abruptly at 0.

16.4.2 SHOW

If the third condition listed for ISOL in the preceding section is not met, no error is reported, but the result may not be meaningful. ISOL only works at the immediately visible level of an expression. That is, it does not execute any of the variable names to substitute their values. For example, if you isolate B in 'A+B=C', the result is 'B=C-A'. But if A has the value 'B-C', then the result ISOL returns does not

represent a proper solution. To prevent such occurrences, you can evaluate the original equation repeatedly until all implicit uses of the unknown are revealed. This may have the unwanted side effect of making other substitutions that you do not want, including any current value of the unknown itself. The correct approach is to use **SHOW**, with the same arguments as you would use for **ISOL**. This command makes any substitutions necessary to make references to the specified name explicit in the level two object:

`'A+B=C' 'B' SHOW → 'B-C+B=C'.`

You can then proceed to solve the result equation.

SHOW has a second behavior that is complementary to that just described. If instead of entering the second (level 1) argument as a name, you enclose that name in a list, **SHOW** evaluates all names in the level 2 object *except* the specified name. The list may actually contain more than one name, in which case none of the listed names are evaluated by **SHOW**. Any functions with numerical arguments are executed. The primary use of **SHOW** in this manner is for cases where an algebraic object is to be executed repeatedly; **SHOW** is used to remove as many functions and symbolic references as possible from an expression to minimize its evaluation time. An example of this application is given in section 14.3.1. Note that symbolic execution mode (flag -3 clear) must be active when **SHOW** is evaluated with a list argument, to prevent the execution of functions that use the listed names as arguments.

16.4.3 QUAD

QUAD is designed for solving quadratic equations $ax^2 + bx + c = 0$, where x is the unknown variable, and a , b , and c are constants with respect to x . **QUAD** does not require the equation to have this form. It takes an arbitrary expression or equation and converts it to a second-order polynomial in the specified variable by computing a second-degree MacLaurin polynomial (like **TAYLR**). This representation is exact if the original expression is first or second order in the variable. **QUAD** then applies the quadratic formula to the coefficients in the polynomial to obtain its solution.

As part of the process of determining the coefficients, the original symbolic argument is evaluated. Therefore, if you want to prevent substitution for the names in the object, you must purge the corresponding variables before executing **QUAD**.

In keeping with the HP 48's generally conservative approach to rearranging expressions, **QUAD** does not attempt to constrain its result into a standard form, so you may have to do some manipulation of the result to make it look like a "textbook" solution. For example, if you solve the standard quadratic equation:

'A*X^2+B*X+C=0' 'X' QUAD

☞ 'X=(-B+s1*√(B^2-4*(A*2/2)*C))/(2*(A*2/2))',

the result is clearly not as compact as it might be. You can improve the appearance of the result with COLCT:

COLCT ☞ '.5*(√(-(4*A*C)+B^2)*s1-B)/A',

which is closer to but still not quite the same as the textbook result

$$\frac{-B \pm \sqrt{B^2 - 4AC}}{2A};$$

however, the two forms are equivalent when evaluated. Note that the $\pm$ is represented in the QUAD result by the variable s1. This concept is explained in the next section.

When its argument is only first order in the unknown, QUAD returns a single solution:

'A*X+B=0' 'X' QUAD ☞ 'X=-(B/A)'.

It even attempts to solve the zero-order case:

'A' 'X' QUAD ☞ 'X=-(A/0)'.

QUAD will not work, however, when the unknown variable must contain a unit object for dimensional consistency with the rest of the expression (TAYLR, which shares computing logic with QUAD, has the same problem.) If $f(x)$ is the argument expression, then QUAD computes

$$f(0) + x \cdot f'(0) + x^2 f''(0);$$

that is, f and its derivatives are evaluated at $x=0$, not at $x=0\_units$, so that the successive terms in the quadratic have inconsistent units. See section 21.3.8.

16.5 Multiple Roots

A fundamental result of algebra is that many expressions or equations, even some quite simple looking, have more than one solution (root). This principle is recognized in the behavior of QUAD and ISOL.

Quadratic equations always have *two* solutions, which are commonly combined into a single expression with the use of a $\pm$ sign. QUAD achieves this combination by returning both solutions as a single expression containing the (global) name `s1`, which represents a $\pm$. We will call a name of this type an *arbitrary sign*. The use of a global name in this manner gives you a means of choosing one sign or the other. To choose the positive root, you store `+1` in `s1` and evaluate the expression. For the negative root, you similarly use `-1`. You can also leave `s1` without a value as long as you like, so that you can perform additional calculations on both roots together.

ISOL also returns a single expression representing all possible solutions for its symbolic object argument. Such solutions may contain one or more arbitrary signs, so ISOL uses the names `s1`, `s2`, ... to represent each successive $\pm$ required by the solution. ISOL may also include the global names `n1`, `n2`, ..., as needed, which represent *arbitrary integers*. You may assign *any* integer value `0`, ± 1 , ± 2 , ..., to each of these names; each combination represents a different solution to the original expression. For example,

```
RAD 'SIN(X^2)=Y' 'X' ISOL ⌞ 'X=s1*√(ASIN(Y)*(-1)^n1+π*n1'
```

Here you can observe one arbitrary sign `s1` and one arbitrary integer `n1`. `n1` appears twice in the expression, meaning that the same choice of integer must appear in both places.

The appearance of arbitrary signs and integers may be confusing if you expect to find *a* solution to *a* problem. However, it is not ISOL or QUAD that is introducing complexity into your problem; they are just showing you the mathematically complete result, and not trying to choose one particular root as “better” than any other. As a matter of fact, there is no automatic criterion that the commands could use to choose one root over another; that is a choice that only you can make by considering factors of the problem that are separate from the equation being solved.

For example, consider the equation $x^2 = y^2$. For any y , there are two values of x that satisfy the equation, $x = y$ and $x = -y$. Mathematically, there is no distinction between the two; either could be the correct choice for a particular physical problem. You might prefer the positive root because it “looks nicer,” but such value judgments are not practical for an automated procedure like ISOL. Besides, $-y$ might be the preferred choice on other grounds.

This problem is obscured somewhat by HP Solve, which only returns one answer at a time. In cases with multiple roots, HP Solve usually (but not always) returns the root that happens to be closest to the value stored in the unknown variable (the “initial guess”) when the solving starts. By supplying an initial guess (section 14.5), you are choosing a particular root in advance. If you don’t supply an initial guess, you must take

your chances with whatever value was left in the unknown variable by a previous calculation. When the variable doesn't exist, the root-finder uses zero as an initial guess--which may or may not be a good choice for the problem at hand.

The HP 48 does provide a flag-controlled mode called *principal value mode*, in which a default choice for all arbitrary signs and integers is supplied automatically. When this mode is active (flag -1 clear), arbitrary signs are always chosen to be positive, and arbitrary integers are set to zero. The purpose of this mode is to provide *an* answer to a problem, perhaps to give you a general idea of the appearance of the answer, without the distraction of the arbitrary constants. However, the results returned by ISOL and QUAD in this mode may not be appropriate at all for a real problem.

For example, consider the equation $x^3 = -1$. You can see by inspection that $x = -1$ is one root; imagine that -1 is the correct choice for a particular problem. Solve this equation for x in principal value mode:

```
-1 SF 'X^3=-1' 'X' ISOL ⏎ 'X=(.500000000001,.866025403784)'
```

The complex result is in fact one of the three cube roots of -1 , but it is not the one we have specified. Now try solving again, with principal value mode off:

```
-1 CF 'X^3=-1' 'X' ISOL
```

```
⏎ 'X=EXP(2*π*i*n1/3)*(.500000000001,.866025403784)'
```

Translated to common notation, this result is

$$x = e^{2\pi i n_1/3} \left(\frac{1}{2}, \frac{\sqrt{3}}{2} \right),$$

where we have replaced the approximate decimal values with fractions. n_1 is an arbitrary integer, which means that you can choose any integer value $0, \pm 1, \pm 2, \dots$, for n_1 to obtain a cube root of -1 . There are only three distinct roots for a cubic equation, which you can obtain with any three consecutive values of n_1 . Other values of n_1 just reproduce the same roots. The following table lists the values returned by the HP 48 along with the exact roots, for $n_1 = 0, 1$, and 2 (the errors in the last decimal place arise from the inaccuracy of the floating-point representation of $1/3$):

n_1	HP 48 Result	Exact Value
0	(.5000000000001,.866025403784)	$(\frac{1}{2}, \frac{\sqrt{3}}{2})$
1	(-1,4.465E-12)	-1
2	(.5000000000001,-.866025403784)	$(-\frac{1}{2}, \frac{\sqrt{3}}{2})$

Unless you force it by setting principal value mode, the HP 48 does not attempt to choose one possible root over any other when you use ISOL. There's really no mathematical grounds on which the calculator could make such a choice. As you can see from this example, the "obvious" choice of $n_1 = 0$ does not return the "obvious" answer to the problem, $x = -1$.

You might wonder why ISOL and QUAD use arbitrary integer and sign names in their results, rather than perhaps returning one or more expressions, each of which represents a different root. There are three primary reasons:

1. In general, a problem may have any number of roots, even an infinite number. It is obviously impossible to return an infinite number of objects, and the HP 48 has no way to tell that there is a finite set of different roots among the infinite possibilities represented by one or more arbitrary integers.
2. By returning a single expression to represent a general result, that expression is immediately suitable for use as an argument for further operations, symbolic or numerical. Dealing with even a finite set of multiple results would be very difficult in a program.
3. The use of ordinary names to represent the arbitrary constants allows you to use the normal methods available for variables (STO, the VAR menu, HP Solve, etc.) to select values for the arbitrary signs and integers.

16.5.1 Using the Solver Menu to Select Roots

The solver menu (section 14.2) provides a convenient method for selection of individual roots from a multiple root solution provided by ISOL or QUAD. By storing an expression returned by one of these commands as the current equation, you obtain a solver menu containing all of the arbitrary signs and integers, any other variable names in the expression, and the $\equiv \text{EXPR} = \equiv$ key. Then you can use the menu keys to select values for the variables, and press $\equiv \text{EXPR} = \equiv$ to obtain the evaluated expression.

To illustrate, return to the example $x^3 = -1$. First, if necessary, press $\leftarrow$ **MODES** $\equiv \text{SYM} \equiv$ to activate symbolic execution mode (make sure the white square appears in the $\equiv \text{SYM} \equiv$ key label). Then:

Keystrokes:

$\boxed{\rightarrow}$ **MODES** **NEXT** -1 $\boxed{\equiv}$ **CF**
 $\boxed{\leftarrow}$ **MODES** 3 $\boxed{\equiv}$ **FIX**

'X³ = -1' **ENTER**
 'X' **ALGEBRA** $\boxed{\equiv}$ **ISOL**

PRG $\boxed{\equiv}$ **OBJ** $\boxed{\equiv}$ **EQ**

$\boxed{\leftarrow}$ **SOLVE** $\boxed{\equiv}$ **SEQ** $\boxed{\leftarrow}$
 $\boxed{\equiv}$ **SOLVR**

Results:

Principal value mode off.
 3 decimal places.

1: 'X = EXP(2 * π * i * n1 / 3) *
 (0.500, 0.866)'

2: 'X' Split the equation.
 1: EXP(2 * π * i * n1 / 3) *
 (0.500, 0.866)'

Make the result the
 current equation, and
 activate the Solver menu.

n1 is the only variable in the current equation. To compute all three roots, use n1 = 0, 1, and 2:

Keystrokes:

0 $\boxed{\equiv}$ **N1** $\boxed{\equiv}$ **EXPR** $\boxed{=}$

1 $\boxed{\equiv}$ **N1** $\boxed{\equiv}$ **EXPR** $\boxed{=}$

$\boxed{\rightarrow}$ **NUM**

2 $\boxed{\equiv}$ **N1** $\boxed{\equiv}$ **EXPR** $\boxed{=}$ $\boxed{\rightarrow}$ **NUM**

Results:

1: EXPR: (.500,.866)

2: EXPR: (.500,.866)
 1: 'EXPR: EXP(2 * π * i * /3)
 *(0.500,0.866)'

2: EXPR: (.500,.866)
 1: (-1.000,4.465E-12)

3: EXPR: (0.500,0.866)
 2: (-1.000,4.465E-12)
 1: (0.500,-0.866)

The stack now contains all three cube roots of -1.

16.6 Algebraic Objects as Programs

In preceding sections we have described expressions or equations as “programs.” In most other calculators and computer languages, a program is not a mathematical object—it might contain mathematical expressions, but the program itself is usually a series of numbered lines, each containing one or more instructions. But take away the line numbers, and what you have is just a series of data and instructions that are meant to be executed sequentially and automatically. It is easy to recognize that an HP 48 program works like this, since by definition a program contains a progression of any HP 48 objects that are executed when the program is executed. But it is not so obvious for algebraic objects, since they look like mathematical expressions or equations, which are not commonly thought of as programs (see also section 3.5).

Using the object decomposition tools discussed in Chapter 3 and elsewhere, it is a straightforward matter to convert algebraic objects into RPN sequences, and vice-versa. The program $\rightarrow$ RPN listed below automates this process, converting any algebraic object into a list, in which the objects that originally constituted the algebraic object’s definition are presented in the equivalent RPN order.

$\rightarrow$ RPN	Convert to RPN	7447
level 1		level 1
'expression'	$\rightarrow$	{ objects }
<pre> << OBJ→ IF OVER THEN → n f << 1 n FOR i IF DUP TYPE 9 SAME THEN RPN END n ROLL NEXT IF DUP TYPE 5 ≠ THEN 1 →LIST END IF n 1 > THEN 2 n START + NEXT END f + >> ELSE 1 →LIST SWAP DROP END >> </pre>		Take the expression apart. If the argument count is non-zero, then store the count and the function. For each argument: If it's an algebraic, then convert it to a list. Make sure the first object is a list. If there is more than one list, combine all of the lists. Append the function to the end of the list. Zero-argument function case.

An example of using $\rightarrow$ RPN:

'A+2*SIN(B/C)' $\rightarrow$ RPN $\rightarrow$ { A 2 B C / SIN * + }

16.7 Refining User-Defined Functions

An important constituent of HP 48 symbolic operations is the set of functions (section 3.1) that you can use within symbolic objects' definitions. The HP 48 provides an extensive set of real- and complex-valued functions; by installing libraries (section 3.4.11), you can add even more. In section 8.5, we describe the construction of so-called *user-defined functions*, which you can use to add to the built-in and library function set. User-defined functions on the HP 28 were subject to certain limitations compared with built-in functions; the HP 48 has added three commands to remove these limitations.

16.7.1 Preventing Evaluation: QUOTE

In the course of the ordinary evaluation of an algebraic object, each function's arguments are evaluated before the function itself is executed. There are certain functions, however, that have the ability to prevent the evaluation of one or more of their arguments. ∂ is a good example; neither the name representing the variable of differentiation nor the expression to be differentiated should be executed prior to ∂ itself. When you enter an expression such as ' $\partial X(\text{SIN}(X))$ ', the HP 48 automatically (and invisibly) enters the X and the $\text{SIN}(X)$ as full algebraic objects themselves. That is, the overall expression is equivalent to the RPN sequence

'X' 'SIN(X)' ∂ .

The first two objects are "quoted" (section 3.8) as algebraic objects so that execution leaves them unchanged as the sequence is evaluated. For most functions, the internal quoting is not done, e.g. ' $\text{MAX}(X, \text{SIN}(X))$ ' translates to the sequence

X X SIN MAX.

User-defined functions do not intrinsically have this ability to quote their arguments. If you evaluate ' $f(x,y,z)$ ', where f is the name of a user-defined function, x , y , and z are evaluated before f . To correct this deficiency, the HP 48 provides the function QUOTE, the execution of which simply returns its single argument unevaluated. Thus the evaluation of ' $f(\text{QUOTE}(x),y,z)$ ' executes y and z as usual before f , but leaves x unchanged for f .

As an example, suppose you want to make the polynomializer program POLY (section 17.4) into a user-defined function. Rather than rewriting POLY, it can be incorporated

into another program POLYF:

POLYF	Polynomializing Function				CEC9
	level 2	level 1		level 1	
	'f(x)'	'x'	→	'P _N (x)'	
	<< → f x << f x PC^{LY} >> >>				

Evaluating 'POLYF((X+1)^3,X)', for example, will expand the cubic polynomial, but the result will not contain X if that variable has a value. To keep X in the result regardless of any value X might have, you can quote the arguments:

'POLYF(QUOTE((X+1)^3),QUOTE(X))' EVAL → 'X^3+3*X^2+3*X+1'.

16.7.2 Applying Functions without Evaluation: APPLY

Another capability of certain built-in functions is to test their arguments for special cases for which the functions are to be returned unexecuted. For example, when SIN is applied to a symbolic argument, it just returns itself unexecuted, unless the argument is π , in which case it returns zero. The function APPLY extends this capability to user-defined functions.

APPLY is used in expressions with the syntax

'APPLY(name, expression, ... , expression)'.

When this is evaluated, each of the argument expressions is evaluated, and then the unevaluated name is appended to the result as if the name signified a user-defined function. For example,

'APPLY(F,1+1,X)' EVAL → 'F(2,X)'

To understand the use of APPLY, consider the execution of a typical user-defined function with a symbolic argument. Create a user-defined function for the cosecant:

'CSC(X)=INV(SIN(X))' DEFINE

Then, if Z is undefined,

`'CSC(Z)' EVAL`  `'INV(SIN(Z))'`.

Compare this with the result of evaluating `'SIN(Z)'`, which just returns the same expression unchanged. The following version of CSC uses APPLY for symbolic arguments:

CSC		Cosecant Function		5D29
level 1			level 1	
x			$\csc(x)$	
'symb'			'CSC(symb)'	
<pre> << → ↑↑ << IF { 6 7 9 } ↑↑ TYPE POS THEN ↑↑ 1 →LIST 'CSC' APPLY '↑↑' SHOW ELSE ↑↑ SIN INV END >> >> </pre>				Symbolic case. Defect work-around. Numeric case.

CSC executes APPLY as an RPN command. Used this way, APPLY expects two arguments: a global or local name in level 1 (the function name) and a list containing the arguments to which the function is to be applied.

$\{ expr_1 \dots expr_n \}$ 'name' APPLY  `'name(expr1, ..., exprn)'`.

The apparently pointless sequence `'↑↑' SHOW` in CSC has a definite purpose. Unfortunately, all versions of the HP48 (A-E) at the time of this writing have a defect in APPLY such that all of the expression arguments of its results are quoted in the sense described in section 16.7.1. This means that such results can not be evaluated further, since the internal quoting prevents the argument expressions from being evaluated. Without the SHOW you would find, for example,

`DEG 30 'Z' STO 'Z' CSC EVAL`  `2,`

but then

`'Z+Z' CSC EVAL`  `'CSC(Z+Z)'`

The last result would remain unchanged even after repeated uses of EVAL. During the execution of SHOW, the result is taken apart and then reassembled correctly, so that

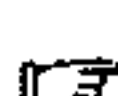
'Z+Z' CSC EVAL  1.15470053838

The SHOW slows execution a little, but the result is more generally useful. You can substitute any other name for $\updownarrow$, although it is best to use an uncommon one that you are not likely to use for a variable (section 18.1.2). Future versions of the HP 48 may correct this problem, in which case the SHOW would be superfluous.

The definition of CSC above uses a program its defining procedure (section 8.5), so that a derivative is not automatically supplied. But it is straightforward to define a derivative for CSC (see section 18.1.1.1)

'derCSC(x,dx)=CSC(x)/TAN(x)*dx' DEFINE

Then

'CSC(Y^2)' 'Y' ∂  'CSC(Y^2)/TAN(Y^2)*(2*Y)'.

16.7.3 Preserving Local Variables' Values: |

If a user-defined function contains any functions that delay evaluation of their arguments, such as ∂ or QUOTE, the local names associated with the user-defined function may be present in the function's result. Since the corresponding local variables are discarded after execution of the function, further evaluation of the result could result in the Undefined Local Name error. To forestall this problem, user-defined functions defined with expressions automatically inspect their results for the presence of any of the function's local names. If any are found, the result is returned as an argument of the *where* function |.

For example, consider the following user-defined function:

'F(x)= ∂ x(SIN(x))' DEFINE

Since ∂ in an expression executes in a step-wise manner (section 18.1), it and the local name x will be present in any result of evaluating F, e.g.

'F(Z)' EVAL  'COS(x)* ∂ x(x) | (x=Z)'

This result is more legible in the EquationWriter:

$$\cos(x) \cdot \frac{d}{dx}(x) \Big|_{x=Z}$$

You can read the expression starting at the vertical bar as “where x equals Z .” That is, the subexpression to the left of the bar is to be interpreted normally, with the understanding that ultimately the value Z is to be substituted for the symbol x . This is just what `|` does when it is executed; it evaluates its first argument, repeatedly if necessary while substituting for the names as indicated in the remaining arguments, until the names no longer appear. Thus, evaluating the previous result:

EVAL  'COS(Z)'

The syntax of the *where* construct provides for any number of variable assignments:

'expression ($x_1, \dots, x_n$) | ($x_1 = value_1, \dots, x_n = value_n$)'

Thus with `FXYZ` defined by

`'FXYZ(x,y,z)=QUOTE((x+y+z)/(x*y*z))'` DEFINE,

then

`'FXYZ(1,2,3)' EVAL  '(x+y+z)/(x*y*z) | (z=3,y=2,x=1).'`

Notice that there are no parentheses around the subexpression to the left of the `|` (assuming that flag `-53` is clear), even though that subexpression is an argument of `|`. `|` has the lowest precedence of any function. In an EquationWriter picture, the extent of its argument is shown by the bar's vertical size:

$$\frac{(x+y+z)}{x \cdot y \cdot z} \Big| z=3, y=2, x=1$$

The bar extends upwards to the top of the left argument, and downwards to the bottom of the argument, plus enough additional to span the variable assignments on its right.

17. Expression Manipulations

The symbolic solution commands ISOL and QUAD are easy to use, but fairly limited in the range of problems they can address. The HP 48 therefore provides a number of tools for performing symbolic manipulations on expressions and equations. These tools may be used to rearrange a symbolic object into a form suitable for ISOL or QUAD, or for any of a number of other purposes.

17.1 Extensive Manipulations

The most common operations you might perform in the course of finding symbolic solutions are these:

- *Reordering terms*: changing the order of sums and factors. For example, you might move all of the terms containing an unknown variable together so that multiple occurrences of the unknown can be merged into a single occurrence. This is achieved in the HP 48 by means of the various RULES operations in the Equation-Writer subexpression mode, especially the term-moving operations $\leftarrow T$ and $T \rightarrow$, and the association operations $A \rightarrow$ and $\leftarrow A$. Also, to move one term from one side of an equation to the other, you can subtract the term from both sides using the ordinary $-$ command.
- *Expansion*: distribution of products or powers over sums. The conversion of $a \times (b + c)$ into $a \times b + a \times c$ is an example of the distribution of a product over a sum. An example of the distribution of a power is the expansion of $(a + b)^2$ into $a^2 + 2ab + b^2$. Expansion in the HP 48 is represented by EXPAN, and by the RULES operations $D \rightarrow$ and $\leftarrow D$.
- *Merging terms*. Once you have all of the terms containing an unknown gathered together, the next step is usually to combine as many of these terms as you can, to minimize the number of occurrences of the unknown--to a single occurrence, if possible. The principal tools for this purpose are COLCT and the RULES merge operations $M \rightarrow$ and $\leftarrow M$.

As an example of HP 48 symbolic manipulation, consider solving the equation $a(x + y) = bx + c$ for x .

1. Press $\boxed{\leftarrow} \boxed{\text{EQUATION}}$, and enter the equation:

4. Move $B \cdot X$ to the left:

nine times (select the =)

$$A \cdot X - B \cdot X = C - Y \cdot A$$

5. Now merge the terms that contain X:

$$(A - B) X = C - Y \cdot A$$

6. Commute the factors of the first term:

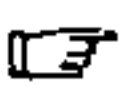
$$X(A - B) = C - Y \cdot A$$

7. Isolate X:  four times (select the =) **RULES**  **T**

$$X = \frac{(C - Y \cdot A)}{-B + A}$$

8. **ENTER** now returns the object ' $X = (C - Y * A) / (-B + A)$ ', in which X has been isolated on the left side of the equation.

This result can also be obtained by using programmable commands:

<i>Distribute:</i>	'A*(X+Y)=B*X+C' EXPAN	 'A*X+A*Y=B*X+C'
<i>Move term right:</i>	'A*Y' - COLCT	 'A*X=-(A*Y)+B*X+C'
<i>Move term left:</i>	'B*X' - COLCT	 'A*X-B*X=-(A*Y)+C'
<i>Merge terms:</i>	{ '&1*&2-&3*&2' '(&1-&3)*&2' } ↑MATCH	 '(A-B)*X=-(A*Y)+C' 1
<i>Isolate:</i>	DROP 'X' ISOL	 'X=(-(A*Y)+C)/(A-B)'

The final result is formally equivalent to the result obtained previously using RULES operations.

These examples illustrate the operation of EXPAN and COLCT compared with the use of RULES operations. Both approaches allow you to alter the form of an expression without changing its formal value. EXPAN and COLCT are "extensive" manipulation commands--they perform wholesale rearrangements potentially involving many terms in the same expression. Both have the shortcoming of trying to do many operations at once without providing you with any control of the precise form of their results. In many cases, you may be surprised or dissatisfied with the results because they don't match some particular form that you desire. For example, the expansion of x^3 could return any of the formally equivalent expressions xxx , x^2x , and xx^2 . There is no

“correct” choice; EXPAN happens to return xx^2 :

`'X^3' EXPAN → 'X*X^2'.`

You can obtain the other choices by using RULES to commute the arguments of the `*` (switching `'X*X^2'` to `'X^2*X'`), or by using EXPAN again to obtain `'X*(X*X)'`.

The RULES approach, on the other hand, suffers from being *too* specific. That is, it allows you to rearrange expressions into a wide variety of equivalent forms, but you must execute one careful step at a time. The path of individual operations you need to follow to change an expression from one form to another requires some thought. RULES also suffers from the general EquationWriter problem of display speed--for large expressions, it takes a long time to show the result of each successive operation.

The best approach for general use of these three commands is to use COLCT and/or EXPAN one or more times on an expression to get it roughly into the form you want. Then use RULES to rearrange parts of the expression, until you obtain the desired final version. To speed up the use of RULES on complicated expressions, you can extract subexpressions using SUB in the subexpression menu, rearrange the subexpressions using RULES, then substitute the subexpressions back into the main expression using REPL.

In the next two sections we will describe the operations of COLCT and EXPAN. In most cases you will not need to follow their workings to anywhere near the detail we present. You may even want to skip these sections at a first reading. Typically, it is easier to use these commands and “see what you get” than to try to predict the outcomes exactly. Keep in mind that the net value of an expression is the same before and after you apply either of the commands; you use them only to rearrange the expression, either as a preliminary to ISOL, or to change the order of calculations, or just to recast it into a more familiar form.

17.1.1 COLCT

The purpose of COLCT is to simplify an expression by combining sum and difference terms that differ only in their numerical coefficients. COLCT tries to reconstruct each term of an expression into the form $c * \text{subexpression}$, where c is a real or complex number, and then combines all terms with the same *subexpression* by adding their coefficients c . Any terms that consist only of numbers are combined into a single term. In addition to this simple reconstruction, COLCT tries to improve the identification of like terms by applying some standardization to the terms:

- Functions whose arguments are numbers are executed. In the expression ' $X+5*\text{SIN}(30)$ ', the arguments of SIN and $*$ are numerical. COLCT therefore returns ' $X+2.5$ ' (in degrees mode).
- *Factors* (arguments of $*$) in a term are put into a standard order, and combined into powers where appropriate:

$$'X*Y*X' \text{ COLCT } \Rightarrow 'X^2*Y'.$$

- Factors of similar quantities raised to powers are combined by adding their exponents:

$$'(X+Y)^Z*(X+Y)^T' \text{ COLCT } \Rightarrow '(X+Y)^{(Z+T)}'.$$

Even these rearrangements aren't enough to ensure that all terms which may appear suitable are actually combined by COLCT. For example, consider ' $2*(X+Y)+X$ '. You might expect COLCT to modify this expression to ' $3*X+2*Y$ '. However, COLCT leaves this expression unchanged, because COLCT does not distribute multiplication (in this case, it does not expand ' $2*(X+Y)$ ' into ' $2*X+2*Y$ '). Without distribution, the terms ' $2*(X+Y)$ ' and ' X ' do not contain common non-numerical factors, so they are not combined.

The basic operation performed by COLCT is *association*--the reordering of the arguments and functions in multiple sums and differences, and the reordering of factors in multiple products (and quotients). The associative property of addition (and subtraction) means that the order of addition doesn't matter: $a+(b+c)$ has the same value as $(a+b)+c$. Similarly, $(a*b)*c$ has the same value as $a(b*c)$; this is the associative property of multiplication. COLCT applies these rules systematically throughout an expression. An easy way to understand COLCT is to view an expression in its RPN form; for example, $(a+b)+(c+d+e)$ is

$$a \ b \ + \ c \ d \ + \ e \ + \ +.$$

Here the *summands* (arguments of $+$) a , b , c , d , and e represent any subexpressions that do not consist of $+$ (or $-$) and its two arguments, such as $2x$ or $\sin(3x+4)$. COLCT rearranges this expression by

1. Moving all of the summands a , b , c , d , and e to the left, and the $+$ operators to the right:

$$a \ b \ c \ d \ e \ + \ + \ + \ +.$$

2. Sorting the summands into a standard order.

3. Combining consecutive summands that are the same except for a numerical coefficient by adding the coefficients.

This process is applied recursively to the individual summands, so that collection of terms takes place at several levels at once.

To illustrate the process, consider the expression ' $Y+2*(X+Y+X)+(X+Y+3*X)$ '. In RPN form, this is

$$Y \ (2 \ X \ Y \ + \ X \ + \ *) \ + \ X \ Y \ + \ (3 \ X \ *) \ + \ +$$

Here we have inserted parentheses to help mark the sequences in which independent collection can take place. Now apply COLCT:

1. Move the summands to the left:

$$Y \ (2 \ X \ Y \ X \ + \ + \ *) \ X \ Y \ (3 \ X \ *) \ + \ + \ + \ +$$

2. Sort the summands:

$$(2 \ X \ X \ Y \ + \ + \ *) \ X \ (3 \ X \ *) \ Y \ Y \ + \ + \ + \ +$$

3. Combine consecutive common terms:

$$(2 \ (2 \ X \ *) \ Y \ + \ *) \ (4 \ X \ *) \ (2 \ Y \ *) \ + \ +$$

Converting back to algebraic form, this is

$$'2*(2*X+Y)+4*X+2*Y'$$

Step 2, sorting the summands, may appear a bit mysterious. The sorting is necessary for the combination of like terms, but often the final order produced by the sorting does not correspond to any obvious rules. For example, ' $A+B+Q+R$ ' COLCT $\rightarrow$ ' $A+Q+B+R$ ', so the sorting is not a simple alphabetization. Actually, the ordering algorithm used by COLCT is designed for optimum speed, and depends on the idiosyncrasies of the HP 48 CPU and the way it stores bits of information in memory. The design of COLCT emphasizes speed rather than some form of standard ordering also because any choice of "standard" ordering would be arbitrary, and generally as likely to be "right" or "wrong" as any other, including the one actually used.

17.1.2 EXPAN

Although the usual effect of COLCT is to make an expression "smaller" by combining terms, and of EXPAN is to make it "bigger" by expanding products of sums, you should not consider them as inverses of each other. Whereas COLCT is based on the associative properties of addition and multiplication, EXPAN is derived from the *distribution* of products, quotients, and powers of sums. The simple distribution rules are

- *multiplication*: $a(b+c) = ab+ac$.
- *division*: $(b+c)/a = b/a + c/a$.
- *involution* (powers): $a^{b+c} = a^b a^c$.

Each of these rules has a straightforward representation in the actions of EXPAN:

$'A*(B+C)'$ EXPAN $\Rightarrow$ $'A*B+A*C'$

$'(B+C)*A'$ EXPAN $\Rightarrow$ $'B*A+C*A'$

$'(B+C)/A'$ EXPAN $\Rightarrow$ $'B/A+C/A'$

$'A^{(B+C)}'$ EXPAN $\Rightarrow$ $'A^B*A^C'$

(You can also substitute $-$ for $+$ in the above examples.)

When both arguments of a product are sums, only the second sum is distributed:

$'(A+B)*(C+D)'$ EXPAN $\Rightarrow$ $'(A+B)*C+(A+B)*D'$

There are two additional special cases of the distribution of powers:

- Expressions of the form a^n expand to $a \cdot a^{n-1}$, where n is a positive integer real number. For example,

$'A^5'$ EXPAN $\Rightarrow$ $'A*A^4'$.

- Squares of sums are expanded from $(a+b)^2$ to $a^2 + 2ab + b^2$:

$'(A+B)^2'$ EXPAN $\Rightarrow$ $'A^2+2*A*B+B^2'$

$'SQ(A+B)'$ EXPAN $\Rightarrow$ $'A^2+2*A*B+B^2'$

It is also possible to distribute the logarithm of a product into a sum or difference of logarithms:

$'LN(A*B)'$ EXPAN $\Rightarrow$ $'LN(A)+LN(B)'$

$'LN(A/B)'$ EXPAN $\Rightarrow$ $'LN(A)-LN(B)'$

EXPAN distributes the antilogs of sums and differences as follows:

'EXP(A+B)' EXPAN $\Rightarrow$ 'EXP(A)*EXP(B)'

'EXP(A-B)' EXPAN $\Rightarrow$ 'EXP(A)/EXP(B)'

Similar expansions hold for the base 10 versions of these functions (LOG and ALOG).

These cases cover all of the potential rearrangements performed by EXPAN, if you generalize them by letting A, B, and C stand for any subexpressions. However, EXPAN does not necessarily make all possible expansions in an expression; specifically, EXPAN does not expand any subexpressions that are part of a distribution. For example,

'A*(B*(C+D)+E*(F+G))' EXPAN $\Rightarrow$ 'A*(B*(C+D))+A*(E*(F+G)).'

To understand this example, it's useful to write the expression in Polish notation (section 2.1):

$*(A, +(*(B, +(C,D)), *(E, +(F,G))))$

EXPAN works into the expression, looking for products for which at least one argument is a sum, i.e. patterns of the form $*(+(a,b),c)$ or $*(a, +(b,c))$. When it finds one in any subexpression, it distributes the multiplication, then does not attempt any further operations on the arguments of the sum. In the current example, the outermost subexpression is such a product, so the multiplication is distributed. The arguments of the sum, $(B*(C+D))$ and $(E*(F+G))$, are not expanded, even though they themselves are products of sums.

If you expand the example expression twice with EXPAN, the "inner" products are expanded. The result of the first expansion looks like this in Polish form:

$+(*(A, *(B, +(C,D))) , *(A, *(E, +(F,G))))$

Now the outermost subexpression is a sum, which is not a candidate for expansion. Therefore, in the second use of EXPAN, each of the arguments of the outer sum is considered in parallel. Both arguments are products, but neither is a product of a sum, so the analysis branches again, into four subexpressions--the two arguments of each of the "outer" products. Of these, two-- $*(B, +(C,D))$ and $*(E, +(F,G))$ --are suitable for expansion, and are duly expanded, completing the operation in those branches. The other two branches--the two A's--are dead ends, so the expansion is complete, and the second EXPAN returns

'A*(B*C+B*D)+A*(E*F+E*G)'

17.2 The EquationWriter Subexpression Mode

The EquationWriter includes a powerful adjunct that we will call *subexpression mode*, from the capabilities it provides for operations on individual subexpressions (the Owner's Manual calls this the *selection environment*). These operations range from command-line editing to the application of various identity transformations. We introduced this mode in section 6.7.6, with a brief discussion of the associated *subexpression menu*; here we will elaborate on some of the topics, and focus on the operations collectively called RULES.

Subexpression mode is one of three modes in which you can operate the EquationWriter. You can always tell which of the three is active by considering the cursor and the menu:

- In *entry mode*, where you can extend the currently displayed expression with additional objects, the cursor is an open box at the right end of the expression:

$$\sum_{N=0}^I \frac{X^I}{N+1} + \sqrt{\frac{X^3}{2+X}} + \cos(Z)^2 \square$$

&MPE &MP5 &MP4 &MP2 &MP1

The menu can be any ordinary command menu.

- In *subexpression mode*, the cursor is an inverse highlight of one object or subexpression:

$$\sum_{N=0}^I \frac{X^I}{N+1} + \sqrt{\frac{X^3}{2+X}} + \cos(Z)^2$$

RULES EDIT EXPR SUB REPL EXIT

The menu is either the main *subexpression menu*, as shown here, or one of the various RULES menus (section 17.2.3).

- In *viewing mode*, no menu appears:

$$1 + 2 \cdot \sum_{N=0}^I \frac{X^I}{N+1} + \sqrt{\frac{X^3}{2+X}} + \text{COS}(2)$$

This mode lets you view an expression that is too large for the display. Pressing any arrow key scrolls the display window in the indicated direction over the expression. If you enter viewing mode from entry mode, the cursor disappears. If you enter from subexpression mode, the cursor remains, and is reactivated in place when you exit viewing mode.

You can activate viewing mode from either other mode by pressing $\leftarrow$ **GRAPH**. Pressing that key again, or **ATTN**, returns to the previous mode. To switch from entry mode to subexpression mode, press $\leftarrow$ (which suggests moving “back” into the expression). Initially, the rightmost object in the current expression is highlighted. To return to entry mode, press **EXIT**. In either of these two modes, **ENTER** exits from the EquationWriter, with the modified expression entered into level 1. **ATTN** exits as well, but discards the EquationWriter expression.

Note that you can not enter subexpression mode from entry mode unless the active expression is complete--there are no functions without all of their arguments. Pressing $\leftarrow$ in such a situation flashes the Incomplete Subexpression warning and retains entry mode.

The *subexpression menu* contains the following entries, which we will discuss in the next sections:

- **RULES** activates a context-sensitive menu of transformations that may be applied to the selected subexpression.
- **EDIT** copies the selected subexpression into the command line for editing as text.

- **≡EXPR≡** extends the selection cursor to highlight the entire subexpression defined by the selected object.
- **≡SUB≡** copies the selected subexpression to the stack.
- **≡REPL≡** replaces the selected subexpression with a number or a symbolic object from level 1 of the stack.
- **≡EXIT≡** returns to the EquationWriter entry mode, with the open box cursor at the right end of the expression.

17.2.1 Navigating the Expression

The subexpression highlight cursor serves to *select* one object at a time in the displayed expression. For example, enter the following expression:

$$\frac{A^2 + B^2 + C}{D + E^3} + F$$

PARTS PROB HYP MATH VECTOR BASE

Now press to activate subexpression mode:

$$\frac{A^2 + B^2 + C}{D + E^3} + \text{F}$$

RULES EDIT EXPR SUB REPL EXIT

The four cursor keys, plus their right-shifted () extensions, move the selection cursor from object to object. The involution function ^ is a special case, since it is not normally visible in the EquationWriter. But it does appear when you select it--with the F highlighted, press :

$$\frac{A^2 + B^2 + C}{D + E^3} + F$$

RULES EDIT EXPR SUB REPL EXIT

As you move the cursor around an expression, it highlights one object at a time. However, for the purposes of the various menu operations, the cursor actually selects the entire subexpression defined by the highlighted object and its arguments, if any. You can make the entire selection explicit by pressing EXPR or $\boxed{+/-}$ (the latter is always available, whereas EXPR is not accessible when any of the RULES menus are active). For example, with the cursor still on the ^, pressing EXPR highlights its subexpression:

$$\frac{A^2 + B^2 + C}{D + E^3} + F$$

RULES EDIT EXPR SUB REPL EXIT

The rightmost + is the top-level function in this expression, so moving the cursor there and pressing EXPR highlights the entire expression:

$$\frac{A^2 + B^2 + C}{D + E^3} + F$$

RULES EDIT EXPR SUB REPL EXIT

Notice that whenever you move the cursor, it always reverts to the single-object highlight; otherwise the constant size-changing of the cursor as you move it about an expression would be distracting and confusing. You can also shrink a subexpression highlight back to a single object by pressing $\equiv$ **EXPR** $\equiv$ or $\boxed{+/-}$ again.

Although the highlight cursor motion is always generally in the direction indicated by the key arrows, it is sometimes difficult to predict just which object is next as you move the cursor. For example, with the cursor like this

$$\frac{A^2+B^2+C}{D+E^3}+F$$

RULES EDIT EXPR SUB REPL EXIT

pressing $\triangle$ moves the cursor to the divide bar

$$\frac{A^2+B^2+C}{D+E^3}+F$$

RULES EDIT EXPR SUB REPL EXIT

rather than to the 3 exponent, apparently because the 3 is not directly above the E. There is, of course, a complicated set of rules that determines just how the cursor will move. But it is not worth the trouble to list or learn the detailed rules. A little trial and error in ambiguous situations is faster than trying to analyze an expression yourself to save a few keystrokes.

17.2.2 Editing Subexpressions

It is useful to distinguish between two types of modification that you can apply to expressions as they are represented in the HP 48. The first, which we will call *editing*, is

the alteration of the text form of an expression without regard to its original mathematical structure. The second, called *transformation*, is the application of various mathematical rules to subexpressions, preserving the structure of the overall expression and usually its formal value as well. Typically, you *edit* an expression when you have entered it incorrectly or when you are using it as a typing aid to create a near copy of the expression. You *transform* an expression when you are trying to solve it or re-express it in a different but equivalent form. In this section, we will discuss editing in subexpression mode; transformation is described in section 17.2.3.

≡EDIT≡ copies the selected subexpression to the command line, where you can alter it freely using any of the normal facilities of the command line. There are two reasons for using the command line from the EquationWriter:

- You are already entering an expression with the EquationWriter, and you need to correct a previously entered portion.
- You wish to edit a complicated expression, where it may be easier to use the EquationWriter to pick out a portion for editing than to decipher the maze of parentheses that appears in the linear format display of the expression.

To illustrate the process, consider an expression entered and selected like this:

The screenshot shows a rectangular window with a mathematical expression:
$$\frac{A^2 + B^2 + C}{D + E^3} + F$$
 The '+' sign between the fraction and 'F' is highlighted with a small black square. Below the expression is a horizontal menu bar with the following options: RULES, EDIT, EXPR, SUB, REPL, and EXIT. The 'EDIT' option is currently selected.

≡EDIT≡ copies the subexpression defined by the + into the command line:

The screenshot shows a rectangular window representing the command line. At the top, it has 'RAD' on the left and 'ALG PRG' on the right. Below this is a line with '{ HOME FORM }'. The main area of the window contains the text: 'A^2+B^2+C'. At the bottom is a horizontal menu bar with the following options: +SKIP, SKIP+, +DEL, DEL+, INS, and +STK.

When you have finished editing the subexpression, **ENTER** restores the EquationWriter environment, with the new subexpression replacing the old. If you press **ATTN** when the command line is active, the original EquationWriter expression is restored.. In the example, changing the C to C³/2, then pressing **ENTER** yields this picture:

The screenshot shows a rectangular window containing a mathematical expression and a command menu at the bottom. The expression is
$$\frac{A^2 + B^2 + \frac{C^3}{2}}{D + E^3} + F$$
 The command menu at the bottom consists of six buttons labeled: RULES, EDIT, EXPR, SUB, REPL, and EXIT.

There are a few restrictions on the contents of the command line that you can send back to the EquationWriter:

- Only one object can be entered.
- The object must be of a type suitable for embedding in an expression: a real or a complex number, a name, a unit object or an algebraic object.
- If the original selected subexpression was a name-only field, e.g. a variable of integration or differentiation, then the returned object must also be a name.

If the command line contains a syntax error, then the Invalid Syntax error is reported, and the command line remains active for your corrections. If one of the above rules is violated, then the calculator beeps but returns to the EquationWriter, with the original expression unchanged. However, the command line text is saved in the command stack, where you can retrieve it by pressing **EDIT** again, followed by **◀** **CMD**.

The **↑STK** operation is available during a subexpression edit, for copying stack objects into the subexpression. If the EquationWriter was activated by **▽** with a stack object, that object will not appear on the stack during the edit. Thus if you want to copy *object*₂ into *object*₁, enter *object*₂ into level 2, and *object*₁ into level 1. Then press **▽** to start the EquationWriter, activate subexpression mode, select a subexpression, and press **EDIT**. At this point, **↑STK** will show *object*₂ in level 1, where you can use **COPY** to copy it into the command line. **ENTER** then returns the modified object to the EquationWriter environment.

17.2.3 RULES

The **RULES** key in the subexpression menu activates a menu of manipulation operations. These operations, which we are collectively calling the *RULES operations*,

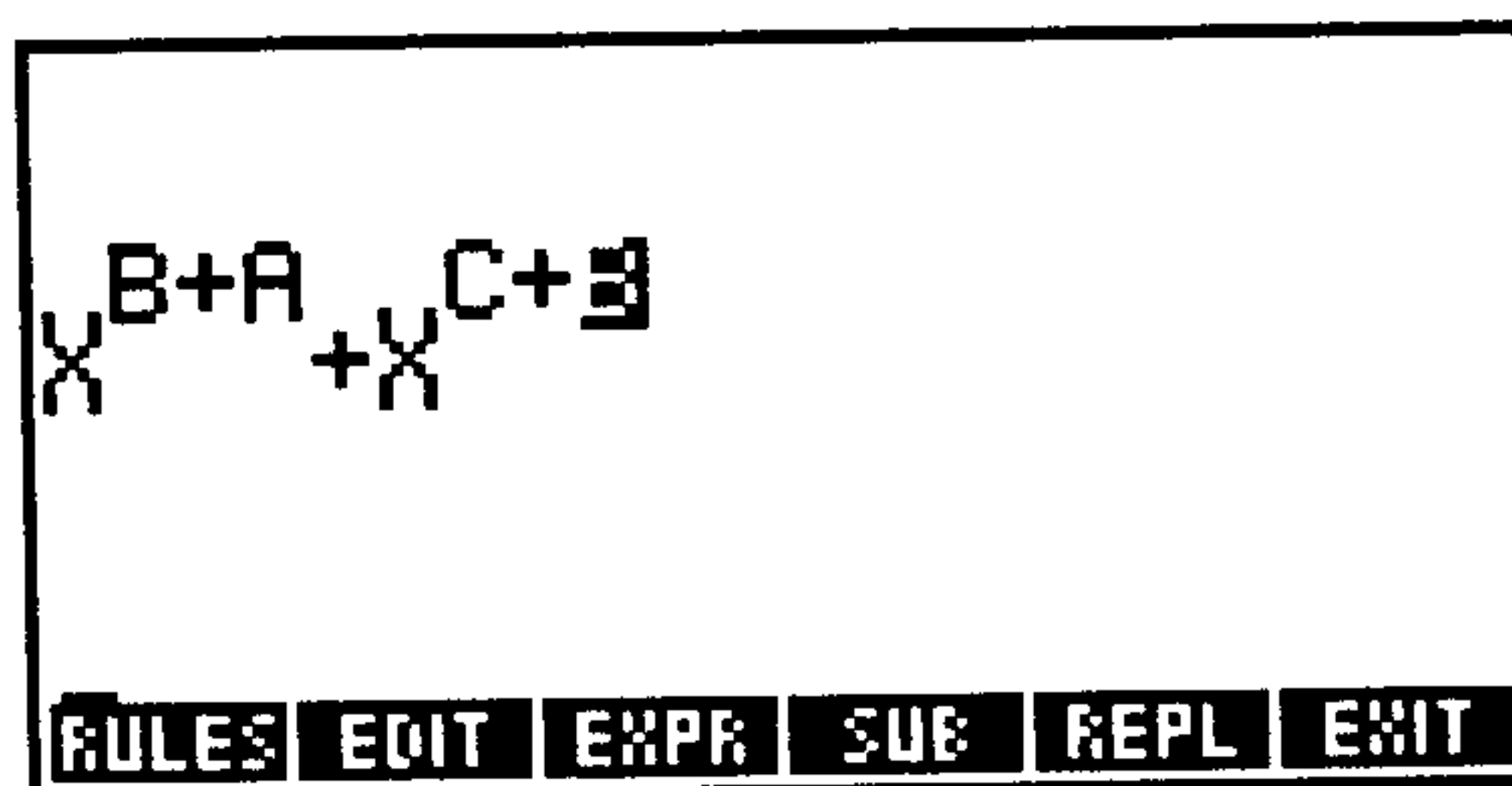
correspond to various mathematical rules for rearranging expressions (hence the name). The menu that you see when you press **≡RULES≡** varies according to the selected subexpression. At any time, the menu contains only those operations which may be applicable to the function that defines that subexpression. Some operations appear in all of the menus; others appear only for certain functions.

The selected function's arguments determine which of the available operations are usable in each individual case. For example, the RULES menu for * includes keys for moving factors, commuting, associating, distributing, merging, double negating, double inverting, and replacing the product of a logarithm with the logarithm of a power. In most cases, only a few of these can be applied. For $A*B$, the only options are commutation (to $B*A$), double negation (to $-(-A*B)$), and double inversion (to $INV(INV(A)/B)$).

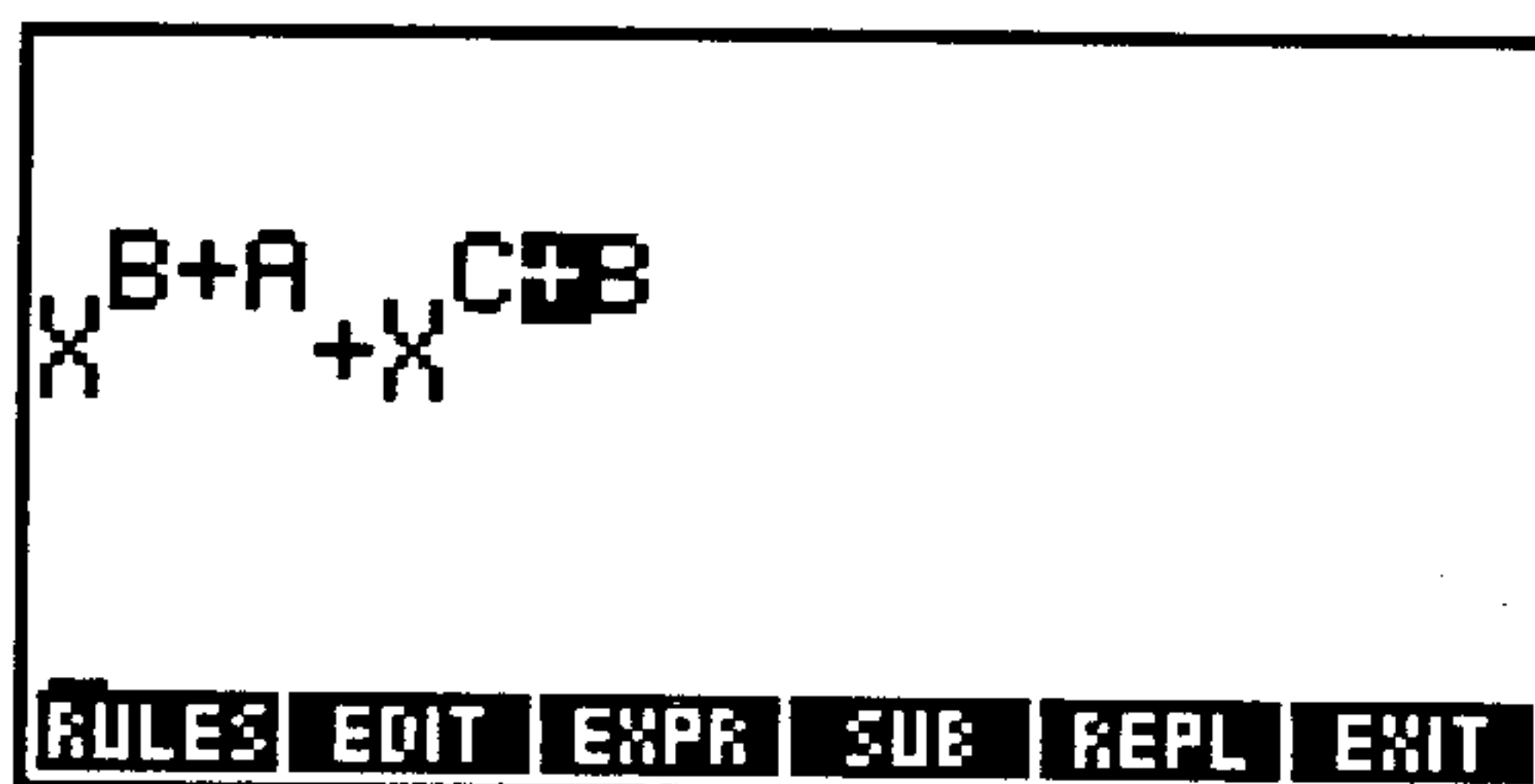
We used RULES manipulations as part of an overall equation solving process in the example in section. That example illustrates the basic use of RULES:

1. Activate the EquationWriter subexpression mode with an expression or equation.
2. Move the cursor to select an object; if necessary, use **≡EXPR≡** or **[+/-]** to verify that you have selected the correct subexpression.
3. Press **≡RULES≡** to activate the operations menu for the selected object.
4. Press an operation key.
5. Repeat steps 2-4 until the expression is rearranged as desired.
6. Press **[ENTER]** to place the revised expression on the stack, or **≡EXIT≡** to go to EquationWriter entry mode.

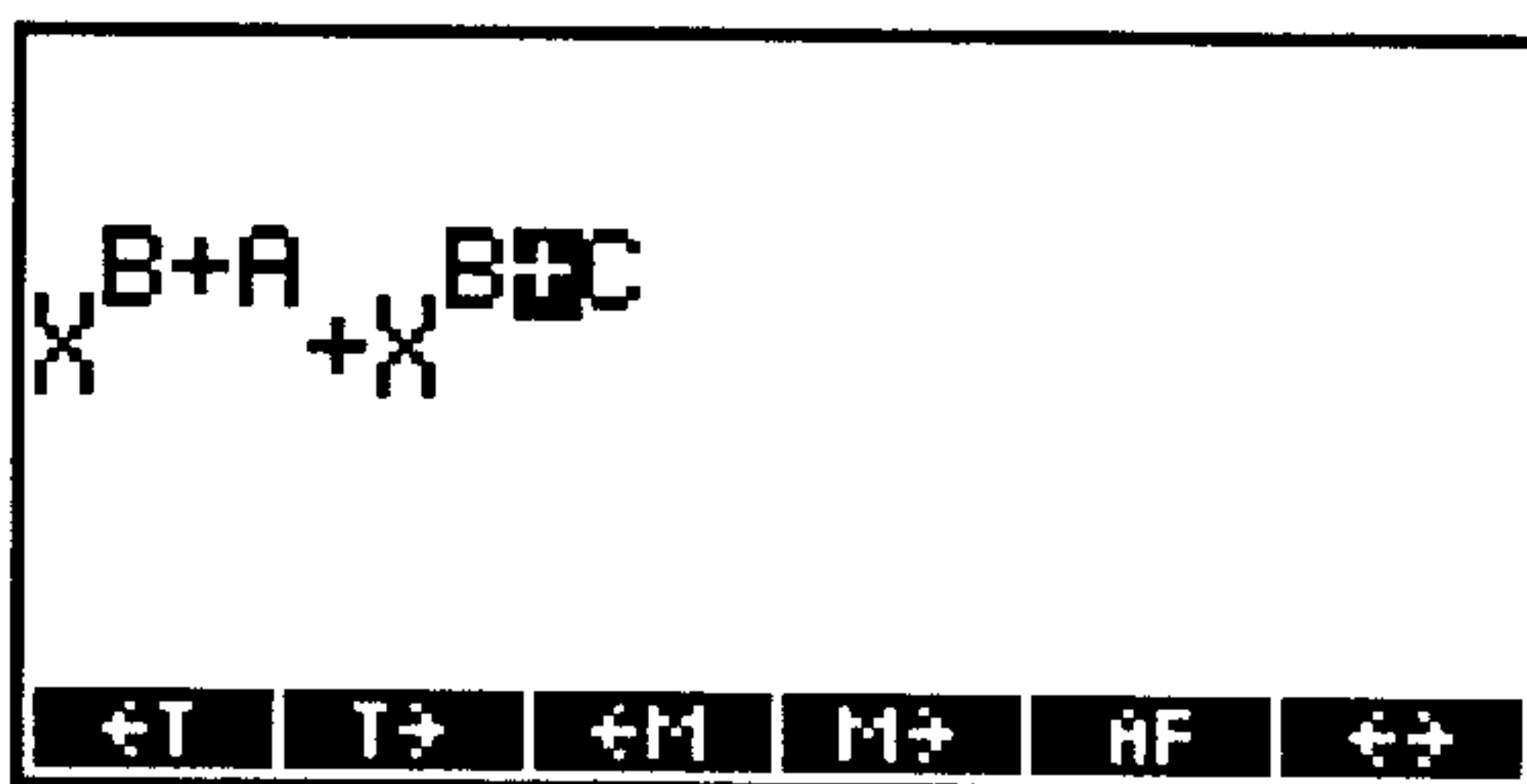
As another example, consider rearranging the expression ' $X^{(B+A)} + X^{(C+B)}$ ' to factor out the common factor ' X^B '. Starting with the expression in EquationWriter subexpression mode:



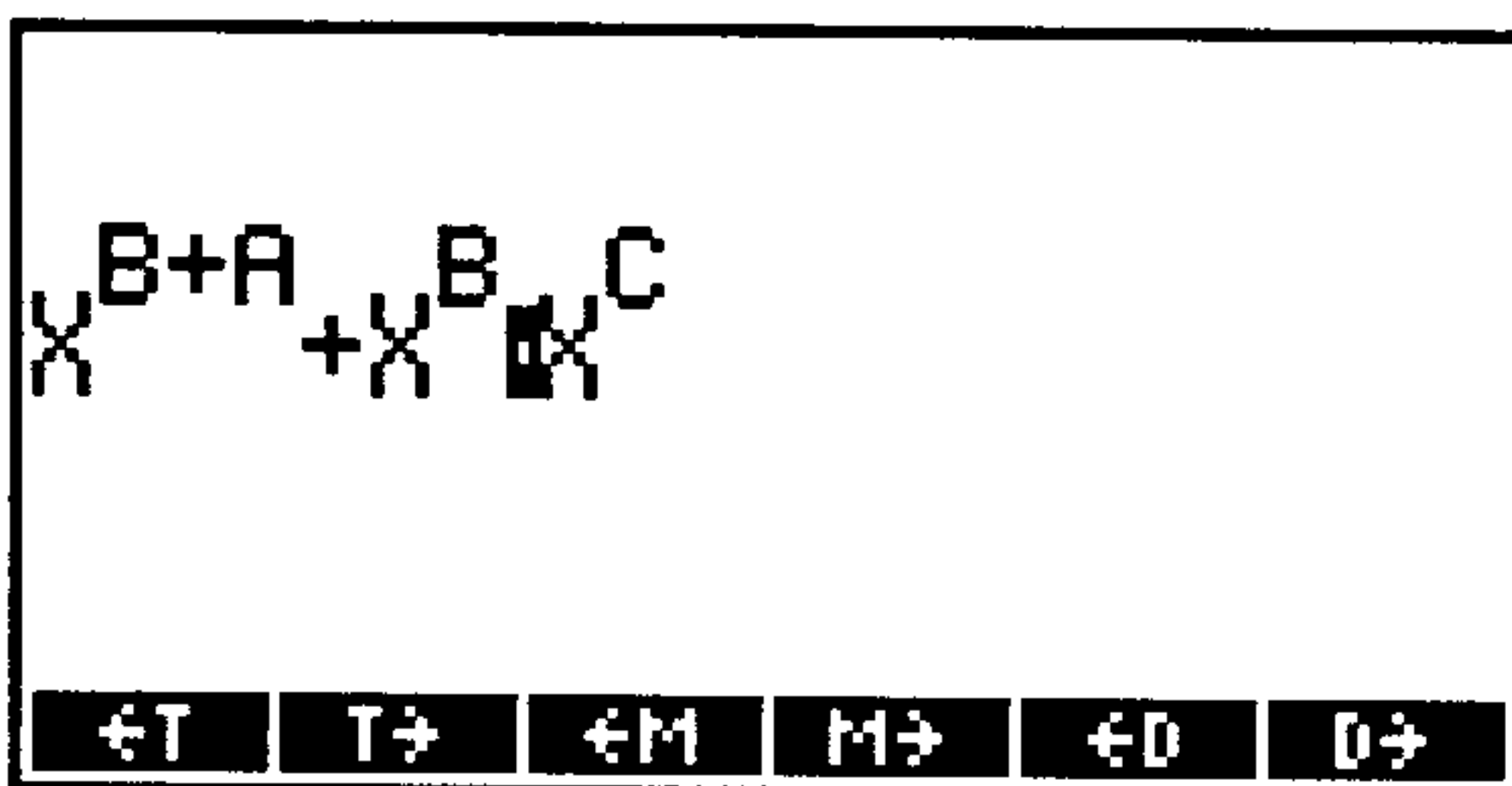
First, commute the second exponent's sum, so that the B has the same position in both exponents. Move the cursor to highlight the +, by pressing $\square$ once:



Now press $\equiv$ RULES $\equiv \rightarrow$:



Now distribute the exponent. Press $\square \square$ to highlight the ^, then $\equiv$ RULES $\equiv$ D $\rightarrow$:



Repeat the distribution for the X^{B+A} . Press $\square$ eight times to highlight the ^, then $\equiv$ RULES $\equiv$ D $\rightarrow$:

$$X^B \cdot X^A + X^B \cdot X^C$$

Finally, press $\triangleright$ four times to highlight the $+$, then $\equiv$ **RULES** $\equiv$ $\equiv$ **-M** $\equiv$:

$$X^B \cdot (X^A + X^C)$$

RULES operations share with EXPAN and COLCT the fundamental property of being *identity* operations, which never change the formal value of an expression. You can use them to rearrange an expression into a form more suitable for further calculations, with confidence you can't make mistakes that alter the symbolic value of the calculation that the expression represents. This differs significantly from the ordinary syntax checking that the command line editor performs.

The RULES operations are the ultimate "conservative" HP 48 symbolic manipulation system. They are primarily "single-step" operations that allow you to rearrange an expression into almost any form you want, making no changes that you don't explicitly specify. Of course, this means that a substantial rearrangement of an expression can require a long series of operations.

The expression manipulation facility called RULES here first appeared in the HP 28 under the name FORM. That implementation has two major problems:

- Subexpression selection is done with the linear format representation of algebraic objects, with all parentheses shown explicitly (such as with flag -53 set in the HP 48). With the line-wrapping necessitated by the 23-character display, this format makes it very difficult to identify subexpression structure.

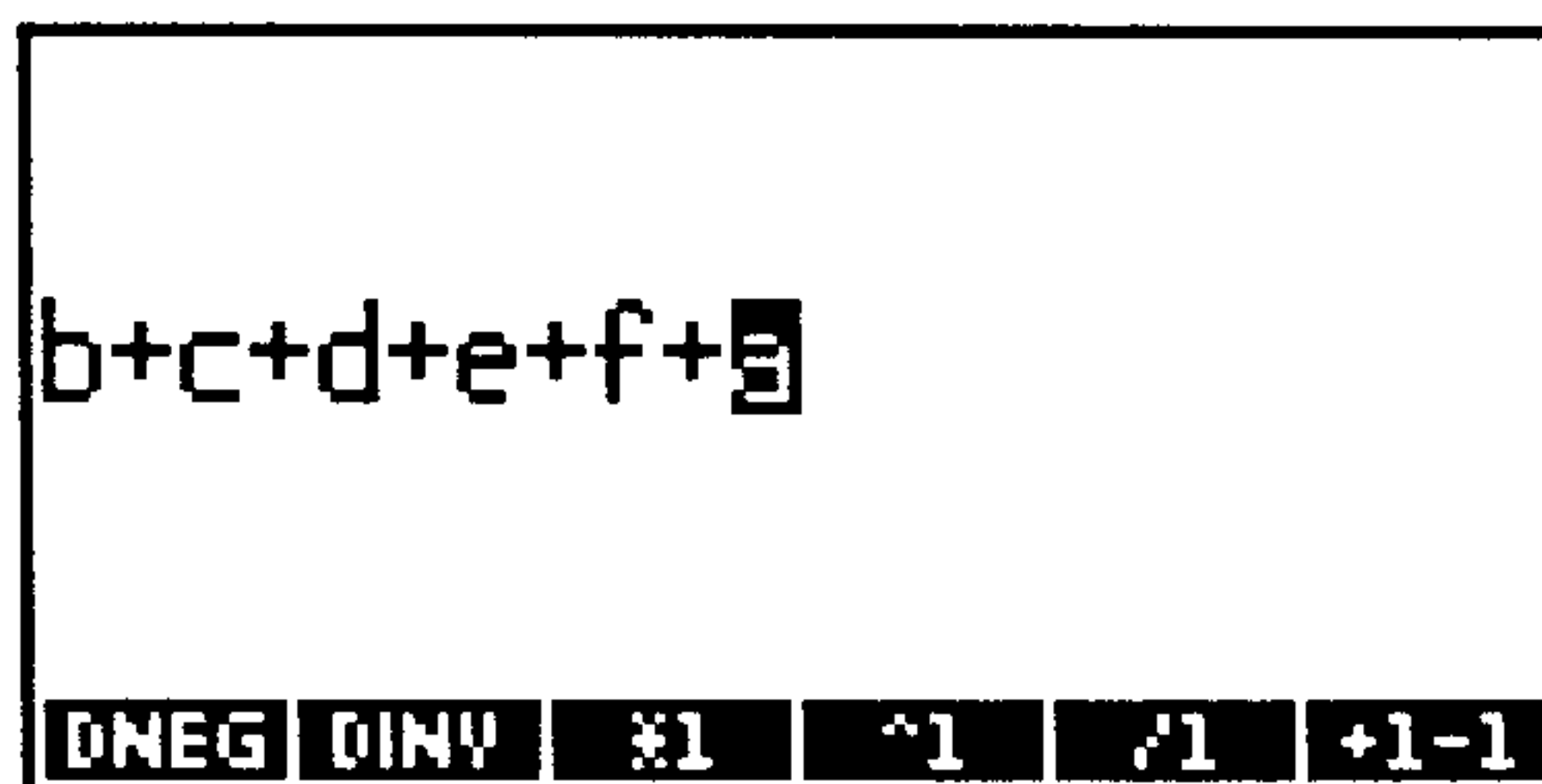
- All transformations are strictly oriented to the calculator's representation of expressions as a subexpression hierarchy. That is, the HP28 "sees" the expression $a + b + c + d + e + f$ as nested sums, i.e. $((((a + b) + c) + d) + e) + f$. Each sum is on a different level, and may contain one or more of the other sums as one of its arguments. We humans, on the other hand, tend to view the expression as a series of *terms*, each on the same level, more like $a + b + c + d + e + f$. One of the most common pencil-and-paper operations is the simple reordering of the terms, such as moving the a to the right end: $b + c + d + e + f + a$. This apparently simple rearrangement is a formidable task in HP28 FORM, involving many commutation and association operations.

These problems were addressed in the migration of FORM into HP48 RULES. First, FORM's linear format representation is replaced in the HP48 by the EquationWriter subexpression mode. The presentation of an expression in a two-dimensional format with familiar graphical constructs makes it quite easy to pick out functions and their arguments. The **≡EXPR≡** key lets you confirm visually that the function you select does span the subexpression you want.

Second, the HP48 adds new *term movement operations*, **←T** and **T→**, which allow you easily to move summands and factors. For example, the rearrangement described above becomes a one-step operation. With the left-most **+** selected:



≡RULES≡ **⌈→** **≡T→≡** moves the a all the way to the right:



This rearrangement is a very tedious exercise on the HP 28.

Of course, the (lack of) speed of the HP 48 EquationWriter remains as a deterrent to the use of RULES operations to replace pencil-and-paper calculations. Whether the latter is any faster than using RULES for a given problem is questionable, but at least when you do a problem by hand, you are kept busy constantly so that the time seems to pass more quickly.

17.2.3.1 Repeated Operations

HP 48 RULES adds one more feature that is not present in HP 28 FORM: *automatic repeated operations*. This means that in any situation where it makes sense to execute an operation more than once consecutively on the same object, you can use the right-shifted menu key as a shortcut. This applies to any of the following operations: $\overline{\overline{D}}$, $\overline{\overline{-D}}$, $\overline{\overline{A}}$, $\overline{\overline{-A}}$, $\overline{\overline{M}}$, $\overline{\overline{-M}}$, $\overline{\overline{)}}]$, $\overline{\overline{(}}$, $\overline{\overline{T}}$, and $\overline{\overline{-T}}$. Pressing $\overline{\overline{r}}\triangleright$ followed by any of these keys executes the associated operation repeatedly on one object until no further change is possible. For example, you can distribute this entire product

A*(B+C+D+E+F)

RULES EDIT EXPR SUB REPL EXIT

at one stroke by pressing $\overline{\overline{RULES}} \overline{\overline{r}}\triangleright \overline{\overline{D}}$:

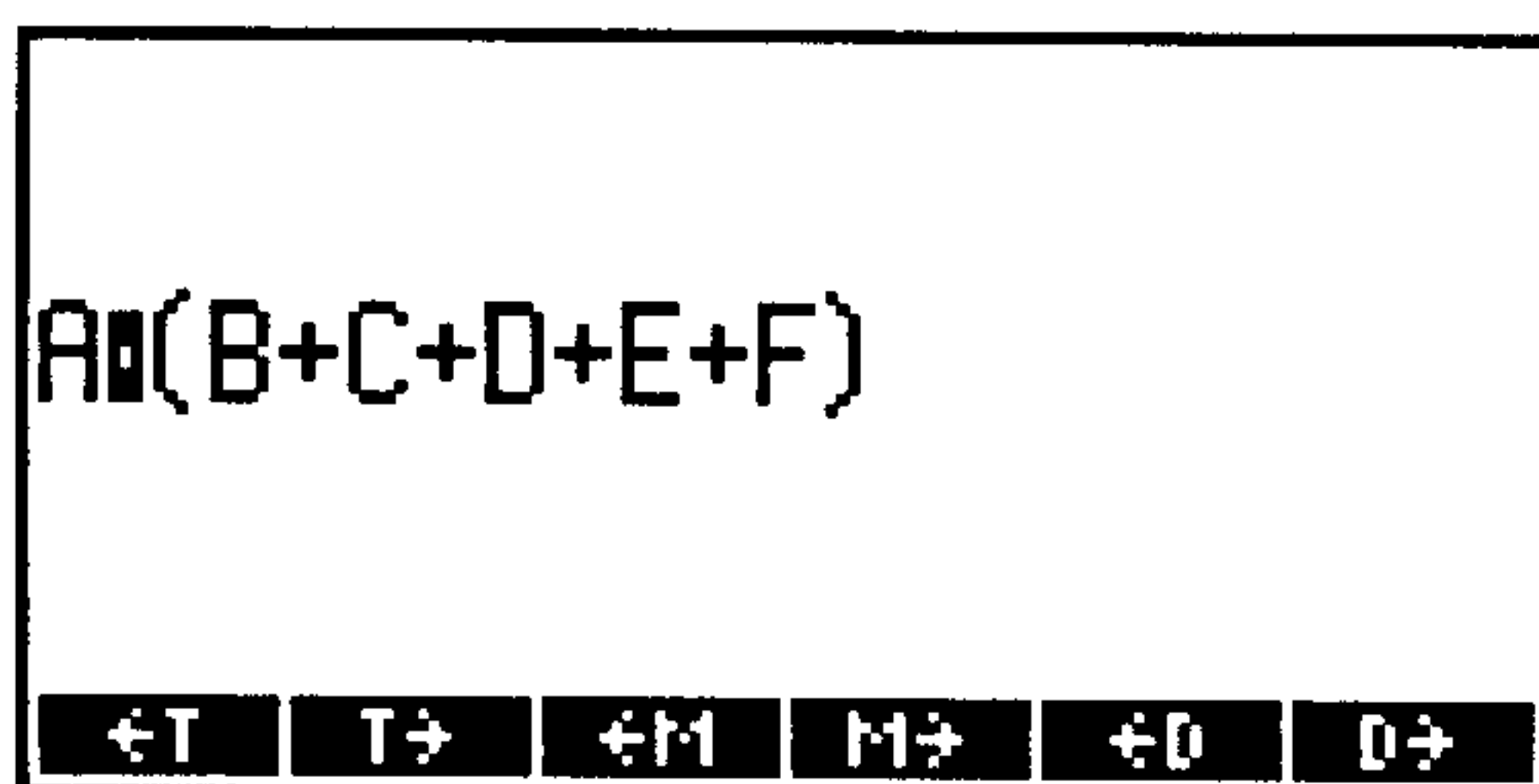
A*B+A*C+A*D+A*E+A*F

<T T> <M M> AF <>

Similarly, you can merge the entire sum of products in one operation. From here



pressing **RULES** **→** **M** yields



17.2.3.2 Condensed RULES Notation

Although the use of screen pictures is helpful in explaining the various RULES operations, it can also spread out a description so far that it is hard to see it all at once. Accordingly, in the following sections we will use a condensed notation to represent operations, where the “before” and “after” expressions are shown together on a single line, connected by the familiar **→** symbol. The selected object is indicated by a box around it, and the RULES operation is represented by its menu keybox. Thus the first step in the previous example would be shown as

$$A \cdot (B+C+D+E+F) \xrightarrow{\text{RULES} \rightarrow \text{D}} A \cdot B + A \cdot C + A \cdot D + A \cdot E + A \cdot F$$

17.2.3.3 Moving Terms

As mentioned previously, we generally view expressions as a sequence of *terms* added (or subtracted) together. Because of the associative and commutative properties of addition, the order of terms can be changed without affecting the formal value of an expression. In manual calculations, this is one of the most common operations in the process of rearranging or solving. For example, in beginning algebra classes we are

taught to move all of the terms containing an unknown to one side of an equation and the remaining terms to the other, or to put the terms of a polynomial in order of ascending or descending power. To provide this same ability for symbolic objects, the HP48 RULES system includes two operations, $\leftarrow T$ (*term left*) and $T \rightarrow$ (*term right*), for moving terms left and right in an expression. Furthermore, these operations incorporate *factors*--arguments of $*$ and $/$ --in their generalized definition of *term*, so that you can reorder products and quotients as well.

$\leftarrow T$ moves a term one position to the left:

$$A + B \boxed{+} C + D \quad \equiv \leftarrow T \quad \Rightarrow \quad A \boxed{+} C + B + D$$

There are several things to note in this example:

- The selected object for the operation is the $+$ to the left of the term that is moved, not the object C that is actually moved. This is because the term operations are associated with the functions $+$, $-$, $*$, $/$ and $=$, rather with than their arguments.
- The term moved left is to the *right* of the originally selected function. When a function is selected for a term operation, it defines the two terms on either side as the “candidates” for the operation. With that limited focus, it is only the right term that can be moved left, and the left term that can be moved right. For single movements, $\leftarrow T$ and $T \rightarrow$ could have been combined in a single “local commute” operation. The two separate operations are provided to aid in repeated movements of one term in one direction.
- After the operation, the selection cursor has moved to the next $+$ in the direction of the term motion, staying to the left of the moved object C . Thus to move that object further in the same direction, just push $\equiv \leftarrow T \equiv$ again:

$$A \boxed{+} C + B + D \quad \equiv \leftarrow T \quad \Rightarrow \quad \boxed{C} + A + B + D$$

After the second movement, there are no more $+$'s to the left, so the selected cursor ends up on the C .

In general, you can move a term continuously in one direction by pressing $\equiv \leftarrow T \equiv$ or $\equiv T \rightarrow \equiv$ repeatedly. To move a term all the way in either direction, use $\boxed{\leftarrow} \equiv \leftarrow T \equiv$ or $\boxed{\rightarrow} \equiv T \rightarrow \equiv$, which yield the same results as using the unshifted versions repeatedly until there is no further motion.

Term operations apply to factors as well:

$$A \cdot B \square C \cdot D \quad \equiv \overline{\overline{\overline{\leftarrow T}}} \quad \Rightarrow \quad A \square C \cdot B \cdot D$$

When sums and products are intermixed, the extent of a particular term and the range of its potential motion depends on the order of the functions. This is best illustrated by examples:

$$A \cdot B \cdot C + D \cdot E \cdot F + G \cdot H \square I \quad \equiv \overline{\overline{\overline{\leftarrow T}}} \quad \Rightarrow \quad A \cdot B \cdot C + D \cdot E \cdot F + G \square I \cdot H$$

$$A \cdot B \cdot C + D \cdot E \cdot F \square G \cdot H \cdot I \quad \equiv \overline{\overline{\overline{\leftarrow T}}} \quad \Rightarrow \quad A \cdot B \cdot C \square G \cdot H \cdot I + D \cdot E \cdot F$$

In effect, sequences of +’s (or –’s) and *’s (or /’s) form a kind of term hierarchy; a particular term can be moved around within its own sequence, but not beyond.

In most circumstances, = is treated the same as + or – for term movements, except for a sign change as a term moves across the equals sign:

$$A \cdot B + C \square D \quad \equiv \overline{\overline{\overline{\leftarrow T}}} \quad \Rightarrow \quad A \cdot B = -C \square D$$

However, an exception is made for the case where one side of an equation is a product or quotient. Instead of moving the entire product or quotient across the equals sign, $T \rightarrow$ or $\leftarrow T$ moves only the factor nearest the =, dividing or multiplying the expression on the opposite side:

$$A \cdot B \square -C + D \quad \equiv \overline{\overline{\overline{\leftarrow T}}} \quad \Rightarrow \quad A = \text{INV}(B) \square (-C + D)$$

$$\frac{A}{B} \square C \cdot D \quad \equiv \overline{\overline{\overline{\leftarrow T}}} \quad \Rightarrow \quad A = B \square (C \cdot D)$$

This exception is provided for the purpose of solving an equation for a variable, where you have isolated the variable on one side of the equation save for a multiplicative coefficient.

17.2.3.4 Commutation

Commutation is the exchange of the arguments of a two-argument function. The commutative laws of arithmetic can be summarized as

$$a + b = b + a \quad \text{Addition}$$

$$a - b = -b + a \quad \text{Subtraction}$$

$$a \cdot b = b \cdot a \quad \text{Multiplication}$$

$$\frac{a}{b} = \frac{1}{\frac{b}{a}} \quad \text{Division}$$

A typical use of commutation ($\overleftrightarrow{}$) is to reorder the terms of an expression so that terms with a common factor can be grouped together, as a preliminary to factoring out the common factor. For example, reordering $ax + by + cx$ to $by + ax + cx$ is done by commuting the arguments of the first $+$.

17.2.3.5 Association

Association is a change in the *precedence* of calculation, the order in which the operations are carried out. A common form of association is the conversion of $a + (b + c)$ into $(a + b) + c$. The fact that this represents a change in the order of calculation is easily apparent when you write the two expressions in RPN form, in which calculation proceeds from left to right:

$$a + (b + c) \quad \text{is} \quad a \quad b \quad c \quad + \quad +$$

$$(a + b) + c \quad \text{is} \quad a \quad b \quad + \quad c \quad +$$

There are two functions involved in any association (the two $+$'s in each example); for RULES, you must select the one that defines the entire subexpression that is associated. In $a + (b + c)$, the first $+$ should be selected; in $(a + b) + c$, the second. Furthermore, you need to specify a "direction" for the association. In an expression like $(a + b) + (c + d)$, the middle $+$ can be selected for association in combination with either of the other two $+$'s. $\overleftrightarrow{\text{A}}$ (*associate left*) works when the selected operator is to the left of the second operator, moving the parentheses to the left. For example,

$$(A+B) \boxed{+} (C+D) \quad \overleftrightarrow{\text{A}} \quad \Rightarrow \quad ((A+B)+C) \boxed{+} D.$$

(Here we have set flag -53 to show all of the implied parentheses.) Similarly, $\overrightarrow{\text{A}}$ works when the selected operator is on the right. The choice of "right" and "left" in the operations' names is rather arbitrary--equally good reasons could be offered for reversing the names. It's often easier to try one of the two choices and see if you get what you want, than to remember which is which. If you get the wrong effect, then use the opposite operation twice.

After execution of $A \rightarrow$ or $\leftarrow A$, the selected object is that function that defines the subexpression that contains all of the same objects as the original selected subexpression.

Additional operations provide for multiple associations in one step.

- $(())$ parenthesizes the sum-of-nearest-neighbors, or product-of-nearest-neighbors, ignoring intervening parentheses:

$$A + B \boxed{+} C + D \quad \equiv (()) \quad \Rightarrow \quad A \boxed{+} (B + C) + D$$

This operation has no apparent effect if the selected function is the first (or only) in the sequence, since these parentheses are already present, but hidden.

- $(\leftarrow$ moves the parenthesis on the left of the selected subexpression to include the next term to the left (which may result in a matched pair of parentheses disappearing):

$$A + B + (C \boxed{+} D) + E \quad \equiv (\leftarrow \equiv \Rightarrow \quad A + (B \boxed{+} C + D) + E$$

$$A + (B \boxed{+} C) + D \quad \equiv (\leftarrow \equiv \Rightarrow \quad A \boxed{+} B + C + D$$

- $(\rightarrow)$ moves the parenthesis on the right of the selected subexpression to include the next term to the right:

$$A + (B \boxed{+} C) + D \quad \equiv (\rightarrow \equiv \Rightarrow \quad A + (B + C \boxed{+} D)$$

17.2.3.6 Distribution

Distribution operations allow you to perform EXPAN-like expansions on individual functions and their associated subexpressions. Like association, distribution involves two functions, and you must select the one that defines the subexpression containing both. For example, to distribute the multiplication in the expression $A \cdot (B + C)$, you select the $\cdot$, then press $\equiv \leftarrow D \equiv$. This returns $A \cdot B + A \cdot C$, in which the $+$ is highlighted because it is the defining object for the new subexpression.

Two distribution operations are necessary because of the ambiguity of expressions like $(a + b) \cdot (c + d)$. $D \rightarrow$ distributes the sum on the right:

$$(A + B) \boxed{\cdot} (C + D) \quad \equiv D \rightarrow \equiv \Rightarrow \quad (A + B) \cdot C \boxed{+} (A + B) \cdot D.$$

$\leftarrow D$ distributes the sum on the left:

$$(A + B) \cdot (C + D) \quad \equiv \overline{\overline{-D}} \quad \Rightarrow \quad A \cdot (C + D) \cdot B \cdot (C + D).$$

For logarithms and antilogarithms, which are functions of one argument, there is no ambiguity, and only $D \rightarrow$ is allowed. Thus,

$$\boxed{\text{LN}} (A \cdot B) \quad \equiv \overline{\overline{D \rightarrow}} \quad \Rightarrow \quad \text{LN}(A) \cdot \text{LN}(B)$$

17.2.3.7 Merging

Merging is the inverse of distribution. That is, where distribution expands $a \cdot (b + c)$ into $a \cdot b + a \cdot c$, merging reverses the process, factoring $a \cdot b + a \cdot c$ into $a \cdot (b + c)$. Two forms of merge are necessary to handle ambiguous cases:

$$A \cdot B \cdot A \cdot B \quad \equiv \overline{\overline{M \rightarrow}} \quad \Rightarrow \quad (A + A) \cdot B,$$

and

$$A \cdot B \cdot A \cdot B \quad \equiv \overline{\overline{\leftarrow M}} \quad \Rightarrow \quad A \cdot (B + B).$$

Note that $\leftarrow M$ and $D \rightarrow$ are inverses of each other, as are $M \rightarrow$ and $\leftarrow D$. $\leftarrow M$ also handles the same logarithm and antilogarithm cases (in the opposite sense) as $D \rightarrow$; for example

$$\text{EXP}(A) \cdot \text{EXP}(B) \quad \equiv \overline{\overline{\leftarrow M}} \quad \Rightarrow \quad \boxed{\text{EXP}} (A + B).$$

17.2.3.8 Prefix Operations

This group of operations is organized around the “prefix operators” $-$ and INV , which also happen to be their own inverses: $a = -(-a)$ and $a = \text{INV}(\text{INV}(a))$. The basic operation is *distribute-prefix-operator*, $\rightarrow()$, which “pushes” one of these operators “into” its parentheses by altering the argument. For example,

$$\boxed{-} (A + B) \quad \equiv \overline{\overline{\rightarrow()}} \quad \Rightarrow \quad -A \cdot B$$

The inverse of $\rightarrow()$ depends on the operator that is distributed.

- For negation, the effect of $\rightarrow()$ is reversed by $-()$. The latter is called *double negate and distribute*, since it is equivalent to double negation (see below) followed by a distribution of a $-$ prefix operator. Examples:

$$\boxed{-} (A \cdot B) \quad \equiv \overline{\overline{\rightarrow()}} \quad \Rightarrow \quad -A \cdot B.$$

$$-A \cdot B \quad \equiv \overline{\overline{\rightarrow()}} \quad \Rightarrow \quad \boxed{-} (A \cdot B).$$

- *Inverse of power or inverse of inverse-product:*

$$\boxed{\text{INV}} (A^B) \equiv \boxed{\rightarrow()} \rightarrow A \boxed{\wedge}^{-B}$$

$$\boxed{\text{INV}} \left(\frac{\text{INV}(A)}{B} \right) \equiv \boxed{\rightarrow()} \rightarrow A \boxed{\cdot} B.$$

To return either of these subexpressions to its original form, you must use $\boxed{\rightarrow()} \rightarrow \boxed{1/\rightarrow()}$, which inverts a subexpression by inverting its arguments. Another example of $1/()$ is

$$\boxed{\text{EXP}} (A) \equiv \boxed{1/\rightarrow()} \rightarrow \boxed{\text{INV}} \text{EXP}(- (A)).$$

- *Double inverse:*

$$\boxed{\text{INV}} \text{INV}(A) \equiv \boxed{\rightarrow()} \rightarrow \boxed{A}.$$

The reverse operation is DINV (double-inversion) which takes any subexpression A and changes it into INV(INV(A)).

- *Double negative:*

$$\boxed{-} -A \equiv \boxed{\rightarrow()} \rightarrow \boxed{A}.$$

The reverse operation is DNEG (double-negation), which takes any subexpression A and changes it into $-(- (A))$.

- *Complex Operators.* Distribution of negation and multiplicative inverse is extended to the common complex operations of conjugate (CONJ), real part (RE), and imaginary part (IM). Using $\rightarrow()$, you can “commute” the operator through certain other real-linear functions: CONJ with +, -, *, /, NEG, INV, and SQ. and RE and IM with +, -, *, and NEG.

$$\begin{array}{lll} \boxed{\text{CONJ}} (\text{SQ}(A)) & \equiv \boxed{\rightarrow()} \rightarrow & \boxed{\text{SQ}} (\text{CONJ}(A)) \\ \boxed{\text{RE}} (A+B) & \equiv \boxed{\rightarrow()} \rightarrow & \text{RE}(A) \boxed{+} \text{RE}(B) \\ \boxed{\text{IM}} (A \cdot B) & \equiv \boxed{\rightarrow()} \rightarrow & \text{RE}(A) \cdot \text{IM}(B) \boxed{+} \text{IM}(A) \cdot \text{RE}(B) \end{array}$$

17.2.3.9 Unit Identities

The four simple identity operations *1, /1, ^1, and +1-1, can be used with any subexpression. They are used as preliminaries for merging, in cases where some symmetry is lacking that prevents the merge from working. For example, if you want to factor $A \cdot B + A$ into $A \cdot (B + 1)$, you can't use $M \rightarrow$ or $\leftarrow M$ because the two arguments of the + are

not both products. You can achieve the factoring like this:

$$A \cdot B + \boxed{A} \quad \equiv \boxed{*1} \quad \Rightarrow \quad A \cdot B + A \boxed{-} 1$$

$$A \cdot B \boxed{+} A \cdot 1 \quad \equiv \boxed{+M} \quad \Rightarrow \quad A \cdot (B + 1).$$

Another example:

$$A^B \cdot \boxed{A} \quad \equiv \boxed{\wedge 1} \quad \Rightarrow \quad A^B \cdot A \boxed{\wedge} 1$$

$$A^B \boxed{-} A^1 \quad \equiv \boxed{-M} \quad \Rightarrow \quad A \boxed{\wedge}^{B+1}.$$

/1 is the division analog of *1. Note that you can use *1 and /1 effectively as inverses of each other:

$$A \boxed{-} 1 \quad \equiv \boxed{/1} \quad \Rightarrow \quad A$$

$$\frac{A}{1} \quad \equiv \boxed{*1} \quad \Rightarrow \quad A$$

The last in this class of operations is +1-1, which you can use to split off terms as follows:

$$\boxed{3} \cdot A \quad \equiv \boxed{+1-1} \quad \Rightarrow \quad (3 + 1 \boxed{-} 1) \cdot A$$

$$(3 + 1 \boxed{-} 1) \cdot A \quad \equiv \boxed{-T} \equiv \boxed{COLCT} \quad \Rightarrow \quad (\boxed{2} + 1) \cdot A$$

$$(2 + 1) \cdot A \quad \equiv \boxed{+D} \quad \Rightarrow \quad 2 \cdot A \boxed{+} 1 \cdot A$$

$$2 \cdot A + 1 \cdot A \quad \equiv \boxed{\leftrightarrow} \equiv \boxed{/1} \quad \Rightarrow \quad 2 \cdot A + \boxed{A}$$

17.2.3.10 Adding Fractions

The AF operation allows you to combine a sum or difference of two subexpressions, one or both of which are ratios, into a single numerator over a common denominator. The most general form of this operation is

$$\frac{A}{B} \boxed{+} \frac{C}{D} \quad \equiv \boxed{AF} \quad \Rightarrow \quad \frac{A \cdot D + B \cdot C}{B \cdot D}$$

The + in this example can be replaced by -. Two additional points:

- AF only requires one of the two original subexpressions to be a ratio--you can combine $A + B/C$ into $(A \cdot C + B)/C$, for example.

- If the two subexpressions have the same denominator, use $M \rightarrow$ rather than AF , so that the final denominator is the same as the original.

17.2.3.11 Logarithms

L^* and $L()$ are a pair of operations based on the equivalence $\ln a^b = (\ln a) \cdot b$. L^* transforms the log of a power $\text{LN}(A^B)$ into the product $\text{LN}(A) \cdot B$; $L()$ reverses the transformation. Either works with natural logs (LN) or common logs (LOG). $L()$ expects the log to be the first argument of the product; if you have the form $B \cdot \text{LN}(A)$, you will have to commute the arguments with $\leftrightarrow$ before applying $L()$.

17.2.3.12 Exponentials

$E^$ and $E()$ are based on the equivalences $e^{ab} = (e^a)^b$ and $e^{a/b} = (e^a)^{1/b}$. $E^$ converts left-to-right in these equations, changing $\text{EXP}(A \cdot B)$ into $\text{EXP}(A)^B$, for example. $E()$ is the right-to-left operation. Either works with division as well as multiplication, and with ALOG instead of EXP .

17.2.3.13 Definition Expansions

The **RULES** menus for the trigonometric and hyperbolic functions include the $\rightarrow \text{DEF}$ operation, which replaces these functions with their definitions in terms of EXP and LN . The particular forms of the expansions are chosen from various possibilities to exhibit the same branch-cut behavior as the corresponding HP 48 numeric functions. Table 17.1 shows an example of each operation, where the results are obtained in radians mode, and X represents any subexpression.

The menu for EXP contains the $\rightarrow \text{TRG}$ operation, which translates an exponential function into a sum of trigonometric functions:

$$\boxed{\text{EXP}}(X) \quad \equiv \quad \boxed{\rightarrow \text{TRG}} \quad \rightarrow \quad \cos\left(\frac{X}{i}\right) \boxed{+} \sin\left(\frac{X}{i}\right) \cdot i$$

(The ALOG menu contains the same entries as the EXP menu, but $\rightarrow \text{TRG}$ does not apply to ALOG .)

17.2.3.14 Addition Angle Formulae

The addition angle formulae for trigonometric and hyperbolic functions are provided as transformations by the TRG^* operation (named by analogy with the multiply-argument operation for exponential, $\text{EXP}^$). The specific transformations are listed in Table 17.2.

Table 17.1. $\rightarrow$ DEF Expansions

Function	$\rightarrow$ DEF Expansion	Function	$\rightarrow$ DEF Expansion
SIN(X)	$\frac{(\text{EXP}(X \cdot i) - \text{EXP}(-(X \cdot i)))}{2 \cdot i}$	ASIN(X)	$-i \cdot \text{LN}(\sqrt{1-X^2} + i \cdot X)$
COS(X)	$\frac{(\text{EXP}(X \cdot i) + \text{EXP}(-(X \cdot i)))}{2}$	ACOS(X)	$\frac{\pi}{2} + i \cdot \text{LN}(\sqrt{1-X^2} + i \cdot X)$
TAN(X)	$\frac{(\text{EXP}(X \cdot i \cdot 2) - 1)}{(\text{EXP}(X \cdot i \cdot 2) + 1) \cdot i}$	ATAN(X)	$-i \cdot \text{LN}\left(\frac{(1+i \cdot X)}{\sqrt{1+X^2}}\right)$
SINH(X)	$-(\text{SIN}(X \cdot i) \cdot i)$	ASINH(X)	$-\text{LN}(\sqrt{1+X^2} - X)$
COSH(X)	$\text{COS}(X \cdot i)$	ACOSH(X)	$\sqrt{-\left(\frac{\pi}{2} + i \cdot \text{LN}(\sqrt{1-X^2} + i \cdot X)\right)^2}$
TANH(X)	$\text{TAN}(X \cdot i) \cdot -i$	ATANH(X)	$-\text{LN}\left(\frac{(1-X)}{\sqrt{1-X^2}}\right)$

Table 17.2. TRG* Transformations

Function	Transformation
SIN(X+Y)	$\text{SIN}(X) \cdot \text{COS}(Y) + \text{COS}(X) \cdot \text{SIN}(Y)$
COS(X+Y)	$\text{COS}(X) \cdot \text{COS}(Y) - \text{SIN}(X) \cdot \text{SIN}(Y)$
TAN(X+Y)	$\frac{(\text{TAN}(X) + \text{TAN}(Y))}{1 - \text{TAN}(X) \cdot \text{TAN}(Y)}$
SINH(X+Y)	$\text{SINH}(X) \cdot \text{COSH}(Y) + \text{COSH}(X) \cdot \text{SINH}(Y)$
COSH(X+Y)	$\text{COSH}(X) \cdot \text{COSH}(Y) + \text{SINH}(X) \cdot \text{SINH}(Y)$
TANH(X+Y)	$\frac{(\text{TANH}(X) + \text{TANH}(Y))}{1 + \text{TANH}(X) \cdot \text{TANH}(Y)}$

17.2.3.15 Collecting Terms

The final entry in all of the RULES menus is COLCT. This operation is useful in subexpression mode for simplifying by combining like terms. For example, you can transform $(1+2) \cdot X$ into $3 \cdot X$ by applying COLCT to the $+$.

In HP 48 versions through E, the COLCT operation in the RULES menu is a restricted "one-pass" version of the COLCT command. It does collect like terms, but it does not execute functions of numerical arguments other than the selected function itself. That is, $2 \cdot X + 3 \cdot X$ collects only to $(2+3) \cdot X$; to simplify further to $5 \cdot X$, you must use COLCT again on the $2+3$. Executed as a command outside of the EquationWriter environment, COLCT reduces $'2 \cdot X + 3 \cdot X'$ to $'5 \cdot X'$ in a single execution.

17.3 Pattern Matching and Substitution

Ordinary algebraic object evaluation permits the substitution of expressions for names. That is, if X is defined as $'Y+1'$, then $'2 \cdot X'$ EVAL returns $'2 \cdot (Y+1)'$. However, this does not provide for the reverse substitution--replacing instances of $Y+1$ with X . More generally, there is an endless number of useful transformation rules that are not explicitly provided via EXPAN, COLCT, RULES, etc. To cover such cases, the commands $\uparrow$ MATCH and $\downarrow$ MATCH let you create and execute your own transformation rules. A simple example is the reverse substitution just mentioned:

$$'2 \cdot (Y+1)' \{ 'Y+1' \ X \} \uparrow \text{MATCH} \Rightarrow '2 \cdot X' \ 1$$

The 1 returned to level 1 is a *true* flag (section 9.3), which indicates that a successful match was made.

MATCH $\uparrow$ and MATCH $\downarrow$ are similar in operation, so we will refer to them collectively as MATCH except when the distinction is important. Both commands take as arguments an expression to rewrite, a pattern expression, and a replacement expression. The result is the original expression with the replacement expression substituted for every instance of the pattern expression. The pattern and the replacement are entered together in a list:

$$'old' \{ 'pattern' \ 'replacement' \} \text{MATCH} \Rightarrow 'new' \ \text{flag}.$$

The *flag* indicates whether any substitution was made. The flag is helpful to indicate when an iteration or recursion with MATCH is making no further changes in an expression.

MATCH allows the pattern and the replacement result expressions to contain "wild card" names which will match any subexpression. Any local or global name that starts

with the "&" character is interpreted as a wild card. MATCH replaces the wild card names in the replacement expression by the subexpressions each name matches in the original expression. If the same wild card variable is used more than once in the pattern expression, then it must match identical subexpressions. If a wild card occurs in the replacement expression without occurring in the pattern expression, the Undefined Name error results.

This powerful extension allows the straightforward application of rewrite rules. For example, the addition law for cosines can be applied to any expression by means of the following program:

COSSUM <i>Cosine of a Sum</i> E417			
<i>level 1</i>		<i>level 2</i>	<i>level 1</i>
'expression ₁ '		'expression ₂ '	flag
<< { 'COS(&A+&B)' 'COS(&A)*COS(&B) - SIN(&A)*SIN(&B))' } ↑MATCH >>			Pattern. Replacement.

COSSUM can work with simple expressions:

'COS(X+Y)' COSSUM → 'COS(X)*COS(Y) - SIN(X)*SIN(Y))' 1,

or with more complicated expressions:

'LN(COS(X^2+1/Y))' COSSUM

→ 'LN(COS(X^2)*COS(1/Y) - SIN(X^2)*SIN(1/Y))' 1.

You can just as easily define the inverse transformation TOCOSSUM:

'LN(COS(X^2)*COS(1/Y) - SIN(X^2)*SIN(1/Y))' TOCOSSUM

→ 'LN(COS(X^2+1/Y))' 1.

TOCOSSUM		To the Cosine of a Sum		F8A4
level 1			level 2	level 1
'expression ₁ '		☐	'expression ₂ '	flag
<< { 'COS(&A)*COS(&B) - SIN(&A)*SIN(&B))' 'COS(&A + &B)' } ↑MATCH >>			Pattern. Replacement.	

The pattern and replacement expressions are combined in a list in order to provide for an optional third argument, a conditional test expression. That is, the most general level 1 argument has the form

{ '*pattern*' '*replacement*' '*test*' }.

When MATCH is executed, the *replacement* is only substituted for the *pattern* if *test* evaluates to a *true* flag. Most commonly, a test is used to restrict the domain of a transformation. For example, simplification of the square root of a square is only valid when the argument of the square is positive:

```
'√(3^2)+√((-3)^2)' { '√(&1^2)' &1 ' &1 ≥ 0' }
↑MATCH ☐ '3+√((-3)^2)' 1
```

The last refinement in the use of MATCH is the choice of ↑MATCH or ↓MATCH, which determines whether a pattern search is made upwards or downwards through a subexpression hierarchy (section 3.5.2.1). The distinction is illustrated by the following examples:

```
'A+B+C' { '&1+&2' D } ↓MATCH ☐ 'D' 1
'A+B+C' { '&1+&2' D } ↑MATCH ☐ 'D+C' 1
```

↓MATCH looks for a pattern match starting at the highest level of the subexpression. In terms of the equivalent RPN sequence (section 2.1), this means that ↓MATCH works from right to left through the object sequence. In this case, the second + is the highest level function, which you can verify by setting flag -53 to show the original expression as '(A+B)+C', or by executing OBJ→:

'A+B+C' OBJ→ 'A+B' 'C' 2 +.

The pattern expression '&1+&2' matches any sum, so the entire expression is replaced by 'D'.

↑MATCH works "from the inside out," working from the lowest level subexpressions upwards (i.e., from left to right through the RPN object sequence). For this example, the subexpression A+B matches the pattern, so it is replaced by D. However, the subexpression defined by the second + in the original expression, which is now changed to 'A+D', is *not* considered for replacement. This rule is followed by both ↑MATCH and ↓MATCH: *A subexpression that has been matched once (and hence replaced), is not considered as a candidate for any further matches.* Consider the following examples:

'(A+B)+(C+D)' { '&1+&2' 'E+F' } ↓MATCH → 'E+F' 1

'(A+B)+(C+D)' { '&1+&2' 'E+F' } ↑MATCH → '(E+F)+(E+F)' 1

Again, ↓MATCH replaces the entire expression, because it matches the pattern at the top level (the middle +). However, it does not attempt to replace the new expression, even though it does match the pattern. The reason for the rule is quite apparent in this example: the replacement process would continue forever otherwise, as each successive replacement matches the pattern. Similarly, ↑MATCH replaces the A+B and the C+D, but not the expression defined by the middle +, because that sum's arguments have already been replaced.

17.4 Simplifying Polynomials

Because of the generality of the HP 48 manipulation commands, they can be difficult to use for rearranging particular types of expressions into standard forms. A very common case is ordinary *polynomials*, which are sums of powers of a variable

$$\sum_{n=0}^N a_n x^n.$$

To expand the expression '(X+A)^4', for example, into a sum of powers of X, you need to execute EXPAN and COLCT several times. It is quite difficult to predict how many executions of the commands are needed, and in what order they should be applied. Furthermore, to rearrange the result terms in order of ascending or descending powers requires the use of RULES. You can do better by taking advantage of the fact that a Taylor's series is a polynomial:

'(X+A)^4' 'X' 4 TAYLR COLCT

➡ '6*A^2*X^2+4*A^3*X+4*A*X^3+A^4+X^4'.

To use TAYLR, you must specify the name of the polynomial variable (level 2), and the order of the polynomial. If you don't know the order in advance, you can try any number that you are sure is larger than the actual order. You obtain the fastest execution if you specify a number equal to the polynomial order. This method is preferable to the blind use of EXPAN and COLCT because it is faster and more predictable.

The built-in commands are awkward to use in many cases because of their generality. You can do considerably better by making assumptions about the structure of certain expressions and taking advantage of the HP 48's ability to dissect objects. For polynomials, the assumption is that an expression is a sum of arbitrary coefficients times powers of a single variable. It can therefore be represented as a list of coefficients, with the corresponding powers indicated by the position of each coefficient in the list. Whether the lists are in ascending or descending order of powers is a matter of taste; we will use descending order in the programs presented below. For lack of a better term, we will use the term *polynomialize* to refer to the conversion of any expression into this standard form, either as an expression including the variable or as a list of coefficients. For example, $(X+A)^4$ polynomialized as a list becomes

{ 1 '4*A^3' '6*A^2' '4*A' 1 },

where the X is implied, or stored as a separate object. Using the list representation, common mathematical operations can be represented by programs that use and preserve the list structure. In this section, we will describe a series of programs designed for operations with polynomials. The programs are listed all together at the end (section 17.4.1).

The first program, PADD, performs polynomial addition by adding the corresponding terms in two coefficient lists:

{ A2 A1 A0 } { B1 B0 } PADD ➡ { A2 'B1+A1' 'B0+A0' }.

The programs for polynomial negation (PNEG), subtraction (PSUB), multiplication (PMULT), division (PDIVD), and involution (PPWR) use methods similar to those of PADD. PSUB could be written as a variation of PADD, but it is simpler to combine PNEG and PADD.

{ A2 A1 A0 } { B1 B0 } PSUB ➡ { A2 '-B1+A1' '-B0+A0' }.

Polynomial multiplication is achieved by multiplying each term in one list by each term in the other, and adding up the corresponding products.

$$\{A2 \ A1 \ A0\} \{B1 \ B0\} \text{ PMUL}$$

$$\Rightarrow \{ 'A2*B1' \ 'A2*B0+A1*B1' \ 'A1*B0+A0*B1' \ 'A0*B0' \}.$$

Polynomial division is more difficult than the preceding operations, because it does not in general return a polynomial. The division program PDIV returns three lists, representing the remainderless quotient, the numerator of the remainder, and the denominator, respectively.

$$\{1 \ 5 \ 7\} \{1 \ 2\} \text{ PDIVD} \Rightarrow \{1 \ 3\} \{1\} \{1 \ 2\}$$

represents the division of $x^2 + 5x + 7$ by $x + 2$, which simplifies to $x + 3$ with a remainder of $1/(x + 2)$.

The involution program PPWR considers only positive integer powers, so that the result is an ordinary polynomial. In this case, involution is equivalent to repeated multiplication of the base polynomial by itself. However, programming the involution that way is rather inefficient, since it involves computing the same products many times. On the other hand, using a full-blown multinomial expansion is an involved process requiring a substantial program just to compute the expansion coefficients. The approach used in PPWR is an intermediate one, in which any polynomial above first order is treated as a binomial:

$$P_N(x) = a_N x^N + P_{N-1}(x)$$

An integer power of the polynomial can be computed from the binomial expansion:

$$P_N^i(x) = \sum_{n=0}^i \left\{ \begin{matrix} i \\ n \end{matrix} \right\} (a_N x^N)^n P_{N-1}^{i-n}(x)$$

The powers of the lower-order polynomial P_{N-1} are computed recursively by the same method. This method works for any polynomial raised to any power, but it does require a lot of free memory for large polynomials and powers.

For example, using PPWR to expand $(x + 2)^5$:

$$\{1 \ 2\} \ 5 \ \text{PPWR} \Rightarrow \{1 \ 10 \ 40 \ 80 \ 80 \ 32\}$$

The programs illustrated so far become even more useful when combined with other programs that convert between the list and symbolic object representations of a polynomial. The conversion from list to symbolic object is quite straightforward, as you can see in the program PEXPR. The level 1 argument for PEXPR can be any object allowed in an algebraic object:

```
{ 1 2 3 } 'X' PEXPR → 'X^2+2*X+3'
{ 1 2 3 } 10  PEXPR → 123
```

The conversion from an expression to a list is more difficult. One approach might be to use `↑MATCH` or `↓MATCH` (section 17.3) to replace the operators `+`, `-`, `*`, `/` and `^` in an expression with user-defined function versions of `PADD`, `PSUB`, etc., and convert names, numbers, and non-polynomial functions to zeroth- or first-order coefficient lists. Evaluating the result would then return the coefficient list for the whole expression. However, that method is difficult to adapt for functions other than those for which explicit polynomial versions are provided. Instead, the program `PLIST` takes an expression apart one level at a time using `OBJ→`. Applying this command to an algebraic object returns the “top-level” object, its arguments if any, and a count of those arguments:

$$'f(arg_1, arg_2, \dots, arg_n) \text{ OBJ} \rightarrow arg_1 \ arg_2 \ \dots \ arg_n \ n \ f$$

`PLIST` operates on a symbolic object, or any object that can appear in an algebraic object, plus the name of the designated polynomial variable. Its action depends on the type of the level 2 argument:

- For a real or complex number, or a unit object, or any name other than the polynomial variable name, `PLIST` returns the object in a one-element list, representing a zero-order polynomial.
- If the argument is a name that matches the polynomial variable, `PLIST` returns the list `{ 1 0 }`.
- If the argument is an algebraic object, it is dissected with `OBJ→`. If the top-level object is not one of the polynomial operators, then `PLIST` just returns the original object in a one-element list. For the two-argument operators, `PLIST` applies itself recursively to each of the arguments, then executes the appropriate polynomial list program to combine the arguments.
- Division is included as one of the special polynomial operators even though it does not return a polynomial in all cases, primarily to permit simplification of expressions where a denominator is actually a factor of its numerator, and so may be canceled out. Any remainder is converted back to a fraction by `PLIST`, and treated as a zero-

order term.

- PLIST calls LPWR for involution, rather than PPWR, since the second argument of ^ may be a general expression rather than just a real number. LPWR is an extension of PPWR that does additional argument screening and then calls PPWR if appropriate.
- If PLIST's first argument is an equation, it still returns only one coefficient list, treating the = as a subtraction.

As an example of using PLIST, polynomialize $(x+1)^3 - 4(x-1) + 2$ into a list:

`'(X+1)^3-4*(X-1)+2' 'X' PLIST` $\Rightarrow$ `{ 1 3 -1 7 }`.

The goal set out at the beginning of this section is to devise a fast and automatic means of simplifying polynomials. The final step towards this goal is the program POLY, which polynomializes an expression into a new expression, "hiding" the intermediate list stages:

`'(X+1)^3-4*(X-1)+2' 'X' POLY` $\Rightarrow$ `'X^3+3*X^2-X+7'`.

or

`'((3*X^2-3)/(X+1))^2-(X+1)^2' 'X' POLY` $\Rightarrow$ `'8*X^2-20*X+8'`

Such reductions are essentially intractable using EXPAN, COLCT, and RULES. You can achieve similar results in some cases using TAYLR, but POLY is faster and does not evaluate any of the names in the argument expression.

POLY takes an arbitrary expression and a name as arguments, and returns a new algebraic object expressed as a polynomial in the named variable. POLY invokes PLIST to sort out the polynomial coefficients, then PEXPR to convert the coefficient list back into an expression.

When applied to a polynomial containing names other than the polynomial variable, POLY used as above returns the coefficients in unsimplified forms, e.g.

`'A*X+A*X' 'X' POLY` $\Rightarrow$ `'(A+A)*X'`

Some coefficients may be simplified by applying COLCT to the result, but this may also defeat the polynomial ordering. To provide better additional simplification, POLY accepts a list of names for its argument. The first argument is polynomialized with respect to the first name in the list; each coefficient is polynomialized with respect to

the second name; and so forth for each name in the list. Thus

'(X+A)^3+(X+A^2)^2' 'X' POLY

☞ 'X^3+(1+3*A)*X^2+(2*A^2+3*A^2)*X+(A^2^2+A^3)'

'(X+A)^3+(X+A^2)^2' { X A } POLY

☞ 'X^3+(3*A+1)*X^2+5*A^2*X+(A^4+A^3)'

This strategy may not be helpful if the result contains a fraction, since it will be polynomialized with respect to the second name, which counteracts the polynomialization with respect to the first.

17.4.1 Polynomial Programs

PADD	Polynomial Add	0E9E
level 2	level 1	level 1
{ P _N }	{ P' _M }	{ P'' _{max(M,N)} }
<pre><< DUP2 SIZE SWAP SIZE IF < THEN SWAP END OBJ→ DUP 2 + ROLL OBJ→ DUP 2 + ROLL → nshort nlong << 1 nshort FOR n nshort 1 + ROLL + nshort ROLLD NEXT nlong →LIST >> >></pre>		Determine the lists' sizes. Put longer list in level 1. Dump lists on stack. Save list sizes. Add each pair of terms: Recombine into a list.

PSUB	Polynomial Subtract	9282
level 2	level 1	level 1
{ P _N }	{ P' _M }	{ P'' _{max(M,N)} }
<pre><< PNEG PADD >></pre>		Negate and add.

PNEG	<i>Polynomial Negate</i>		351E
	<i>level 1</i>	<i>level 1</i>	
	$\{P_N\}$	$\rightarrow$	$\{-P_N\}$
<pre> << OBJ→ → n << 1 n START NEG n ROLL NEXT n →LIST >> >> </pre>			Explode list, save count. Negate each term. Recombine list.

PMUL	<i>Polynomial Multiplication</i>		BF6D
	<i>level 2</i>	<i>level 1</i> <i>level 1</i>	
	$\{P_N\}$	$\{P'_M\}$ $\rightarrow$	$\{P''_{N+M}\}$
<pre> << DUP SIZE ROT DUP SIZE IF DUP 4 PICK > THEN 4 ROLL 4 ROLL END → list1 n1 list2 n2 << {} 1 n2 FOR m 0 + list2 m GET → mt << list1 OBJ→ 1 SWAP START mt * n1 ROLL NEXT >> n1 →LIST PADD NEXT >> >> </pre>			Get list sizes. Put longer list in level 4. Start empty result list. For each element of list2: “Multiply” the result list by x^1. Get the mth element of list2. Multiply by each list1 element. Add products to the result list.

PDIVD		Polynomial Divide			4D4C
level 2	level 1		level 3	level 2	level 1
$\{P_N\}$	$\{P'_M\}$	$\rightarrow$	$\{P_{quotient}\}$	$\{P_{remainder}\}$	$\{P'_M\}$
<pre> << DUP2 SIZE SWAP SIZE IF OVER - DUP 0 < THEN DROP2 {0} 3 ROLLD ELSE SWAP ROT DUP 1 GET → n d t << {} 3 ROLLD 0 SWAP START DUP 1 GET t / ROT OVER + 3 ROLLD 1 n FOR m OVER m GET d m GET 3 PICK * - ROT m ROT PUT SWAP NEXT DROP 2 OVER SIZE MIN 1E499 SUB NEXT d >> END >> </pre>			Get list sizes. If the net order is negative, then return 0 as quotient. divisor count, list, and first term. Initial empty quotient list. Repeat $m - n + 1$ times ($m = \text{dividend size}$): Divide leading term by t. Append result to quotient list. Subtract the result * divisor from the dividend ... Discard the first term. Repeat.		

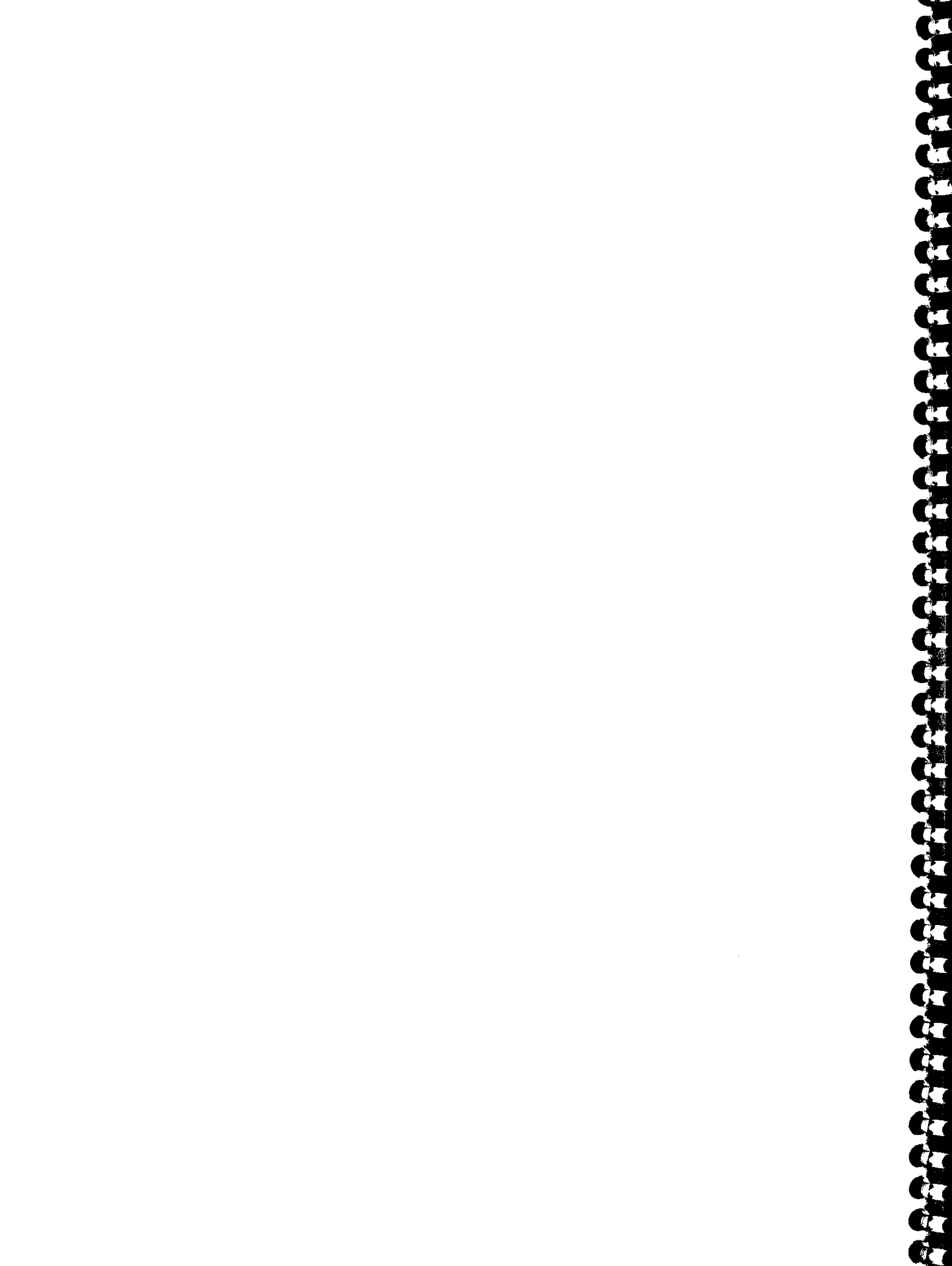
PEXPR		Polynomial Expression			F240
level 2	level 1		level 1		
$\{P_N\}$	x	$\rightarrow$	$P_N(x)$		
<pre> << → x << OBJ→ 0 SWAP 1 FOR n n 1 + ROLL x n 1 - ^ * + -1 STEP >> >> </pre>			Save value. Explode the coefficient list. Start with zero sum. Get the nth coefficient. Add $a_n x$. Iterate.		

PPWR	Polynomial Power		8E38
	level 2	level 1	level 1
	$\{P_N\}$	n	$\{P_N^n\}$
<pre> << CASE DUP 0 == THEN DROP2 {1} END DUP 1 == THEN DROP END OVER SIZE 1 == THEN SWAP 1 GET SWAP ^ 1 ->LIST END OVER SIZE 1 - ROT DUP 1 GET SWAP 2 1E499 SUB 4 ROLLD -> n N aN <<{0} SWAP 0 n FOR i DUP i PPWR n i COMB aN n i - ^ * 1 ->LIST PMUL ROT IF 'i≠0' THEN 1 N START 0 + NEXT END PADD SWAP NEXT DROP >> END >> </pre>			Check for simple cases: $n=0$. $n=1$. $constant^{constant}$ case. Get first coefficient a_N. "Rest" of list $\{P_{N-1}\}$. Leave $\{P_{N-1}\}$ on stack. Initialize a result polynomial. For 0 to n: Compute P_{N-1}^i. ${}_nC_i$. ${}_nC_i a_N^{n-i}$. ${}_nC_i a_N^{n-i} P_{N-1}^i$. Raise power of old sum by N. Add new polynomial. Return P_N^n.

PLIST	Polynomial to List			BFED
	level 2	level 1		level 1
	'f(x)'	'x'	☐	{P _N }
<pre> << -> x << CASE { 6 7 9 0 1 13 } OVER TYPE POS DUP { 6 7 } x TYPE POS AND NOT THEN DROP x 514 DOERR END DUP 3 > THEN DROP 1 ->LIST END 2 < THEN IF x OVER SAME THEN DROP { 1 0 } ELSE 1 ->LIST END END DUP OBJ-> { + - * ^ = NEG / } SWAP POS DUP THEN ROT x PLIST IF ROT 2 SAME THEN ROT x PLIST SWAP ROT ELSE SWAP END IF DUP 7 < THEN { PADD PSUB PMUL LPWR PSUB PNEG } SWAP GET EVAL ELSE DROP PDIVD x PEXPR SWAP x PEXPR SWAP / 1 ->LIST PADD END SWAP DROP END DROP DROPN 1 ->LIST END >> >> </pre>	Save polynomial variable name. Is $f(x)$ a legal object type? And is x a global or local name? If not, then Bad Argument Type. If f is a number or unit object, return as zero-order. If f is a name, then if it's the same as x, then return as first order; otherwise as zero-order. Get top-level function and arguments. If a polynomial function, then polynomialize one argument. If two arguments, then polynomialize the other. If it's not /, then execute the polynomial operation. For division, compute the fraction, and add the remainder. Discard copy of $f(x)$. Return $f(x)$ as zero order.			

LPWR	<i>List to Power</i>			00CA
	<i>level 2</i>	<i>level 1</i>	<i>level 1</i>	
	$\{P_N\}$	$\{n\}$	$\rightarrow$ $\{P_N^n\}$	
<pre> << CASE DUP SIZE 1 ≠ THEN DROP SWAP END 1 GET DUP TYPE THEN DROP SWAP END DUP 0 < THEN DROP SWAP END DUP FP THEN DROP SWAP END PPWR END >> </pre>				Power not zero-order. Power not a real number. Negative power. Fractional power. Do the involution.

POLY	<i>Polynomialize Expression</i>			99FB
	<i>level 2</i>	<i>level 1</i>	<i>level 1</i>	
	$'f(x)'$	$'x'$	$\rightarrow$ $'P_N(x)'$	
	$'f(x)'$	$\{names\}$	$\rightarrow$ $'P_N(x)'$	
<pre> << IF DUP TYPE 5 SAME THEN OBJ→ 1 - ELSE 0 END →LIST → x rest << x PLIST IF rest SIZE THEN OBJ→ → n << 1 n START rest POLY n ROLL NEXT n →LIST >> END x PEXPR >> >> </pre>				If the 2nd argument is a list, then explode it. Otherwise, "rest" is null. Save first name, and rest. Polynomialize with respect to x. If rest is non-null: Then explode the coefficient list, and for each term, polynomialize against rest list. Recombine coefficient list. Convert list to expression.



18. Calculus

The HP 48's mathematical capabilities extend into the realm of the calculus, including differentiation, integration, and polynomial approximations. The derivative function ∂ can symbolically differentiate expressions containing almost any combination of HP 48 functions for which sensible derivatives exist (see section 3.1). By applying the derivative repeatedly, the TAYLR command generates Maclaurin series for arbitrary expressions, which can be generalized into Taylor's polynomials at any point. Computing antiderivatives is a more difficult problem, but the HP 48 can symbolically integrate a fixed set of common integrand patterns. For integrals that do not match the built-in patterns, the HP 48 can still compute accurate numerical integrals.

Although summation might not strictly be considered as part of the calculus, we will discuss summations as a preliminary to the treatment of integration. Besides the close mathematical association between summation and integration, the HP 48 functions Σ and $\int$ have a number of similar properties.

18.1 Differentiation

The derivative function ∂ is quite straightforward to use, except that you must choose whether to carry out a chain-rule derivative in steps or all at once. In either case, you have to identify

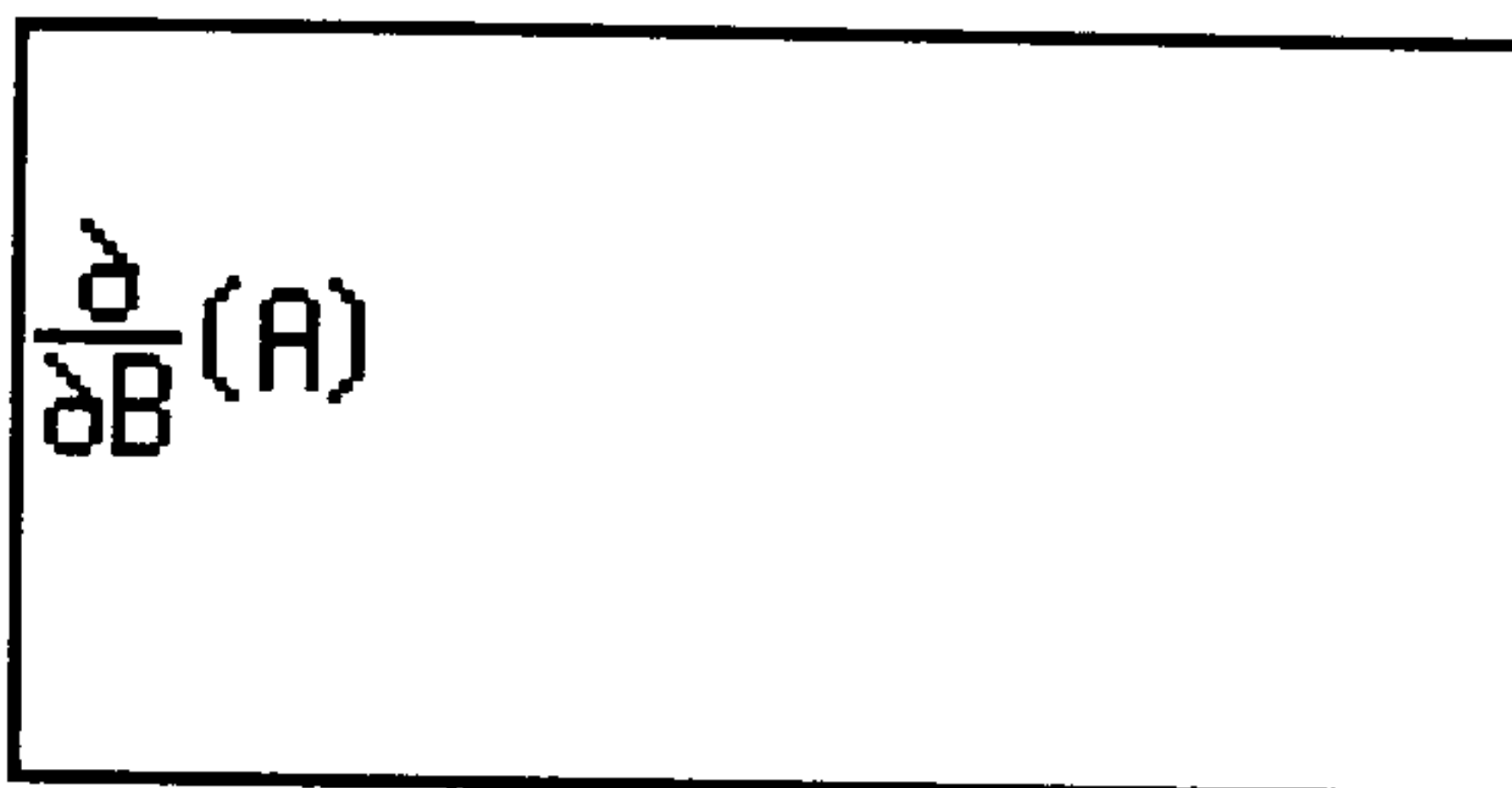
- a. the expression to be differentiated, and
- b. the variable of differentiation.

Since ∂ is a *function*, you can specify these items as two RPN stack arguments, or as arguments for ∂ within an expression. This choice of RPN or expression format also determines whether the differentiation is performed in a single operation (RPN) or one step at a time (expression).

To use ∂ as a stack command, you enter the expression to be differentiated into level 2, and the name of the differentiation variable into level 1, then execute ∂ . The result is an expression representing the derivative of the original expression. It is fully differentiated--the ∂ function does not appear in the result.

The expression syntax for a derivative does not follow the normal HP 48 convention of parenthesizing stack arguments, where $A B \partial$ should become ' $\partial(A,B)$ '. Instead, the form of ∂ within expressions is modeled after the standard written form $D_B A$ (derivative of A with respect to B): this derivative is expressed in HP 48 syntax as ' $\partial B(A)$ '. The parentheses are necessary to separate the expression A from the name B , and from the

remainder of the expression, if any. The more common ratio form is realized in the EquationWriter presentation of derivatives (section 6.7.3.3), where the differentiation variable can be shown as a denominator with the prefix ∂ :



$$\frac{\partial}{\partial B}(A)$$

The HP 48 uses the symbol ∂ rather than d or D to leave the ordinary letters available for names. dA/dB is perhaps a more common written form than $D_B A$, but given the general HP 48 rules for naming variables, 'dA/dB' could also mean dA divided by dB, so this form is not used in the linear format.

∂ also differs from other HP 48 functions in that it executes differently as a stack command than it does when it is part of an expression. In the RPN case, the derivative is repeatedly executed until the ∂ function is no longer present. However, when an expression containing ∂ is evaluated, the derivative is only carried out as far as a single application of the chain rule of differentiation, which is:

$$\frac{d}{dx} f(g(x)) = \frac{df}{dg} \frac{dg}{dx}.$$

For example, to compute $\frac{d}{dx} \sin(\cos(x))$, multiply

$$\frac{df}{dg} = \frac{d}{d(\cos(x))} \sin(\cos(x)) = \cos(\cos(x))$$

by

$$\frac{dg}{dx} = -\sin(x),$$

to obtain the result

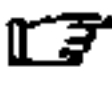
$$-\cos(\cos(x))\sin(x).$$

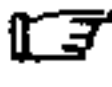
On the HP 48, you can watch this calculation unfold by entering

RAD '∂X(SIN(COS(X)))',

then

EVAL  'COS(COS(X))*∂X(COS(X))'

EVAL  'COS(COS(X))*(-SIN(X)*∂X(X))'

EVAL  'COS(COS(X))*(-SIN(X))'.

Each EVAL applies the chain rule through one level. If you are not interested in the intermediate result, you can obtain the final result in one step:

'SIN(COS(X))' 'X' ∂  'COS(COS(X))*(-SIN(X))'.

The one-step derivative obtained by including ∂ within an expression is consistent with the general flavor of HP 48 expression evaluation (section 3.5.2), which substitutes values for each variable in an expression, but does not recursively evaluate the substituted subexpressions. This style of differentiation is quite useful as a teaching tool, whereby you can observe results of successive applications of the chain rule.

18.1.1 Calculus with Trigonometric Functions

If you differentiate expressions containing trigonometric functions while the HP 48 is in *degrees* or *grads* mode, factors of $\pi/180$ or $\pi/200$ will appear in the result:

DEG 'SIN(X)' 'X' ∂  'COS(X)*(π/180)'

This is correct mathematically, but it may surprise you when you take a derivative without thinking about the angle mode. In effect, any trigonometric function is a different function in each angle mode. For example, in radians mode, SIN corresponds directly to the usual *sine* function, where the argument is expressed in radians. However, in degrees mode, SIN(x) is actually a representation of $\sin(180x/\pi)$. Differentiating the latter expression returns $(180/\pi) \cos(180x/\pi)$.

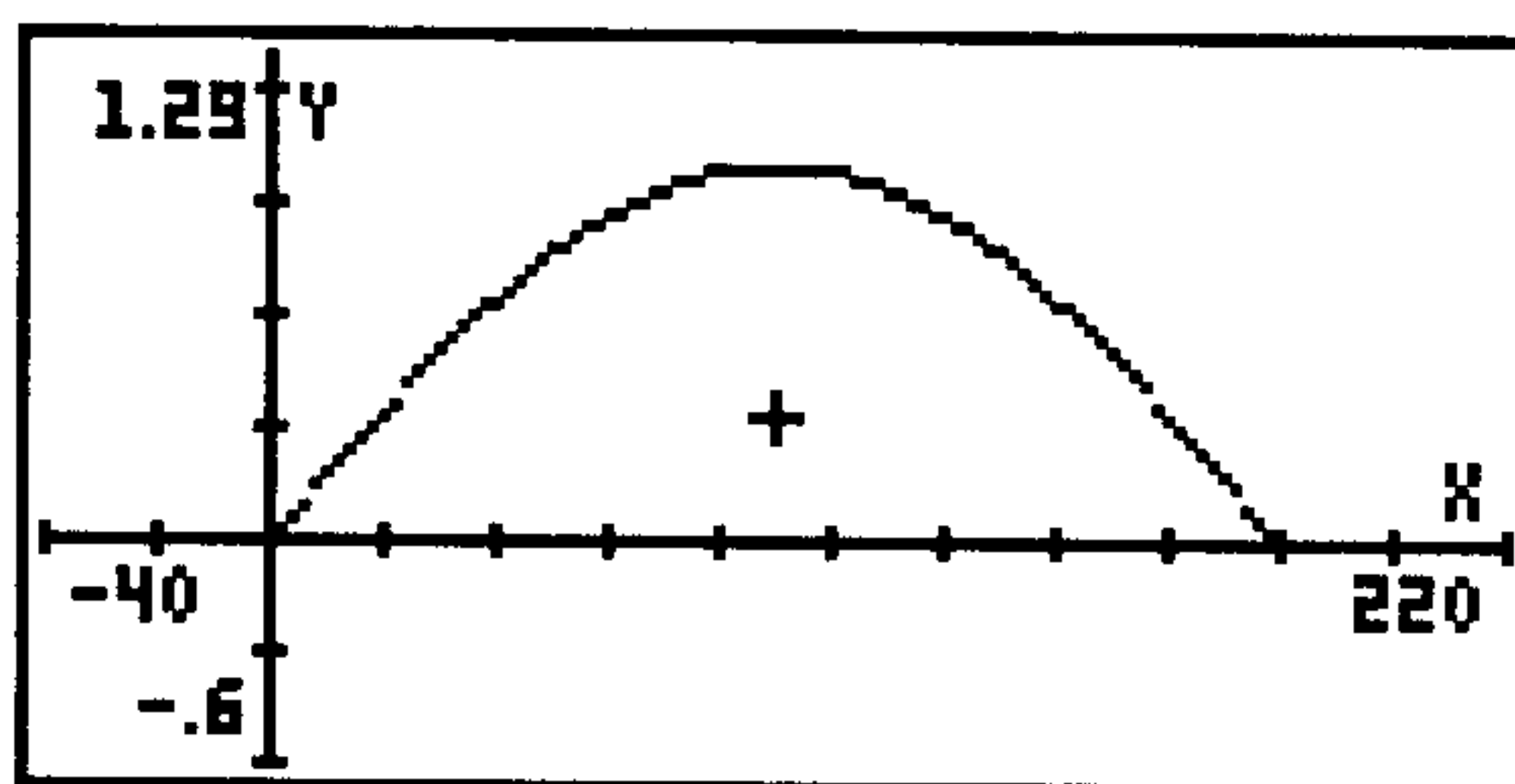
The same reasoning applies to integrals. For example, if you compute (see section 18.4.2)

$$\int_0^{\pi} \sin x \, dx$$

in radians mode, you obtain 2, as you might expect. But in degrees mode:

$$\int(0,180,\text{SIN}(X),X) \rightarrow \text{NUM} \quad \boxed{114.591559026}$$

The difference in the two results is the factor of $180/\pi$ that arises in the integral of SIN in degrees mode. The result is easy to understand if you plot SIN between 0 and 180:



Even a quick glance at the picture shows that the area under the curve is certainly closer to 115 than to 2.

18.1.1.1 User-Defined Derivatives

Almost all HP48 functions that represent continuous and differentiable functions have derivatives included in their built-in definitions. The percent functions %, %CH, and %T are exceptions, having no built-in derivatives. Attempting to differentiate an expression containing any of these functions yields a strange-looking result:

$$' \%T(X,Y)' \quad 'X' \quad \partial \quad \boxed{'\text{der}\%T(X,Y,1,0)'}$$

This is an example of the HP 48's provision for *user-defined derivatives*, which are derivatives that you can define to substitute for missing derivatives. When the function ∂ encounters a function ϕ for which no derivative is available, it does not error. Instead, it evaluates the global name $\text{der}\phi$, created by appending the function name ϕ to the letters "der." If the corresponding variable exists, then the stored object, which should be a user-defined function (section 8.5), is executed. Otherwise, the unevaluated symbolic form of the function is returned, as in the example above. This strategy applies to:

- Built-in or library functions with no associated derivatives, such as the percent functions, or discontinuous functions like IP and MOD.
- User-defined functions for which no definition has been supplied.

- User-defined functions with programs as their defining objects (section 8.5.3).

The value of this approach is that it allows you to continue with calculations that might otherwise be stalled, by supplying the missing derivatives. If you define the necessary $\text{der}\phi$'s before differentiating, you will not see them in the results. But you can also wait until one appears in a result, then store a definition for it and evaluate the result to compute the derivative. For example, the following creates a user-defined derivative for $\%T$:

```
'der%T(x,y,dx,dy)=(dy/x-y/(x^2)*dx)*100'  DEFINE
```

With this definition, you can evaluate the previous result:

```
'der%T(X,Y,1,0)'  EVAL  ⇨  '-(Y/X^2*100)'.
```

(where neither X and Y has a current value).

In general, the user-defined derivative of a function of n arguments requires $2n$ arguments. The first n of these are the arguments of the original function, and the second n are the derivatives of the first n arguments with respect to the differentiation variable. These latter arguments are necessary for chain-rule differentiation. You can see the argument structure in this symbolic example:

```
'∂xϕ(y1,y2,⋯,yn)'  EVAL  ⇨  'derϕ(y1,y2,⋯,yn,∂x(y1),∂x(y2),⋯,∂x(yn))'
```

$T\%$ is an example of a two-argument function. It is defined as

$$T\%(x,y)=100\frac{y}{x},$$

and its derivative with respect to z is

$$\frac{dT\%}{dx}(x,y)\frac{dx}{dz} + \frac{dT\%}{dy}(x,y)\frac{dy}{dz}.$$

$dT\%/dx$ is $-100y/x^2$, and $dT\%/dy$ is $100/x$. The presence of the factors dx/dz and dy/dz necessitates the extra two arguments for $\text{der}T\%$. The easiest way to write $\text{der}T\%$ as a user-defined function is to name the four arguments x , y , dx , and dy , and then write the defining expression like the total differential of $T\%$, i.e.

$$\frac{dT\%}{dx}dx + \frac{dT\%}{dy}dy = 100\left(-\frac{y}{x^2}dx + \frac{dy}{x}\right)$$

Simplifying this a little, and converting it to HP48 syntax leads to the definition for der%T given above.

User-defined derivatives are not necessary for user-defined functions that are defined with algebraic objects. Differentiating such functions automatically carries the differentiation through their internal definitions. For example, if you use the following to create a user-defined function for the secant:

'SEC(x)=INV(COS(x)) DEFINE,

then you can differentiate expressions containing SEC:

RAD 'SEC(θ)' 'θ' ∂ ⌞ 'SIN(θ)/COS(θ)^2'

The derivative of $\sec(\theta)$ can also be expressed as $\tan \theta / \cos \theta$. You can write a derivative for SEC that reflects this form:

'derSEC(x,dx)=TAN(x)/COS(x)*dx*(3.14159265359/ACOS(-1))' DEFINE.

(The factor of $\pi/\cos^{-1}(-1)$ provides for differentiation in any angle mode.) However, ∂ only uses user-defined derivatives when no default is available, so it will not use derSEC in differentiating SEC. If you want to force the use of derSEC, you must rewrite SEC with a program as its defining object:

<< → x << x COS INV >> >> 'SEC' STO

For example, with these definitions, and 'θ^2' stored in the variable X, then

'SEC(X)' 'θ' ∂ ⌞ 'TAN(θ^2)/COS(θ^2)*(2*θ)'.

18.1.2 Formal Derivatives

Executing ∂ as an RPN command may cause repeated evaluation of the differentiated expression. If the differentiation variable has a value (anywhere in the current path), the variable will disappear from the result. For example, with 0 stored in X,

'SIN(COS(X))' 'X' ∂ ⌞ 0,

which is not too helpful if you are looking for a symbolic result.

One simple way to ensure a symbolic result is to purge the variable of differentiation

before executing ∂ . However, this is undesirable when you need to keep its stored value for further calculations. You might save the value, purge the variable, differentiate, and then restore the variable, but there may be one or more variables with the same name in parent directories, so that you would have to find and treat all of them the same way.

The program FDER listed below illustrates a neat solution to this problem. It uses $\uparrow$ MATCH (section 17.3) to substitute a local variable for the differentiation variable, then differentiates with respect to that local variable. The local variable is given its own name as its value, so that repeated evaluation of that name during differentiation never does anything except return the same name (with a global variable, this trick would lead to endless execution--see section 3.6.1). After differentiation, the original global name is substituted back for the local name. Using this program in place of ∂ always produces a result expressed in terms of the differentiation variable:

'SIN(COS(X))' 'X' FDER → 'COS(COS(X))*-SIN(X)'

FDER	Formal Derivative			3B64
	level 2	level 1		level 1
	'f(x)'	'x'	→	'df/dx'
<pre><< DUP → x ↑↑ << x SHOW x { ↑↑ } + ↓MATCH DROP '↑↑' DUP STO ↑↑ ∂ x '↑↑' STO EVAL >> >></pre>				Create dummy variable ↑↑. Make references to x explicit. Replace original variable with ↑↑. ↑↑ will return ↑↑. Differentiate. Restore original variable.

The only flaw in FDER is that you can obtain an incorrect result if the differentiated expression already contains the local name used by FDER. To minimize this possibility, FDER and other programs listed later in this chapter use the unusual local name $\uparrow\downarrow$. This name has no mnemonic value, is obviously not used in common mathematical exercises, and gives a poor appearance to expressions containing it. If by some chance you do like to use this (local) name in your calculations, you can always rewrite FDER using any other name you prefer.

18.2 Taylor's Polynomials

The N th degree Taylor's polynomial for a function $f(x)$ at the point $x = x_0$ is defined by:

$$f(x) = \sum_{n=0}^N \frac{(x-x_0)^n}{n!} \frac{d^n}{dx^n} f(x) \big|_{x=x_0}$$

The special case of $x_0 = 0$ is called MacLaurin's formula:

$$f(x) = \sum_{n=0}^N \frac{x^n}{n!} \frac{d^n}{dx^n} f(x) \big|_{x=0}$$

These definitions are valid for functions for which all derivatives of f exist up to degree n . For $N = \infty$, the polynomial is equal to the function f . For finite N , the polynomial constitutes an approximation to the function; the higher the degree, the better the approximation.

The TAYLR command computes the N th degree Taylor's polynomial for a function at the origin (MacLaurin's formula). To use TAYLR, you enter an expression for the function in level 3, the polynomial variable name in level 2, and the polynomial degree in level 1.

■ *Example.* Compute the fifth-order Taylor's polynomial for $\sin x$ at $x = 0$.

```
RAD 'SIN(X)' 'X' 5 TAYLR ⏏ 'X-1/3!*X^3+1/5!*X^5'
```

In this result, the even-degree terms are absent, because they are all proportional to $\sin^n 0 = 0$.

To produce a Taylor's polynomial at a point x_0 other than the origin, it is only necessary to make a translation of the coordinate system such that $x = x' + x_0$, use TAYLR with the variable x' , then translate the system back by substituting $x' = x - x_0$. The program TAYLRX0 performs these operations. TAYLRX0 uses the same input arguments as TAYLR, with an additional argument to specify the point x_0 .

■ *Example.* Compute the 3rd degree Taylor's polynomial for $\sin x$ at the point $x = \pi/2$.

```
RAD 'SIN(X)' 'X' 'π/2' 3 TAYLRX0 ⏏ '1-.5*(X-π/2)^2'.
```

Note that the result is the same as the 3rd degree Taylor's polynomial for $\cos(x - \pi/2)$, which follows from the identity $\sin x = \cos(x - \pi/2)$.

TAYLRX0					Taylor's Polynomial at x_0	47E3
level 4	level 3	level 2	level 1		level 1	
'expression'	'name'	x_0 †	degree	☐	'polynomial'	
<pre><< RCLF → x x0 n f << -3 CF x SHOW x 'x+x0' EVAL 2 →LIST †MATCH DROP x n TAYLR x 'x-x0' EVAL 2 →LIST †MATCH DROP f STOF >> >></pre>					Save the name, x_0 , degree, flags. Symbolic evaluation. Make all instances of x explicit. Substitution patterns. Substitute $x + x_0$ for x . Make the expansion. Reverse substitution pattern. Substitute $x - x_0$ for x . Restore flags.	

† x_0 may be a name, expression, or number.

You can only apply TAYLR meaningfully to functions for which the function itself and its derivatives up to the N th-order are defined at $x = 0$. For example, you can not compute a polynomial for $\sqrt{x}$, since its first derivative is proportional to $1/\sqrt{x}$, which is infinite at $x = 0$.

18.3 Summations

The summation function Σ provides an automated means for executing definite sums

$$\sum_{\text{index} = \text{start}}^{\text{stop}} \text{summand}$$

Used as an RPN command, Σ uses four arguments:

$$\text{index} \ \text{start} \ \text{stop} \ \text{summand} \ \Sigma \ \text{☐} \ \text{sum.}$$

Index should be a global or local name, and *start*, *stop*, and *summand* can be algebraic objects or other objects allowed in expressions. Normally, *summand* contains at least one use of the *index* name.

When Σ itself is embedded in an expression, the syntax is

$$\Sigma(\text{index}=\text{start},\text{stop},\text{summand}).$$

When Σ is evaluated, it returns one of the following:

- The original sum, unchanged, if either of the limits are symbolic;
- A symbolic sum of terms, if both limits evaluate to numbers and the summand contains symbolic arguments other than the index; thus

$$' \Sigma(l=1,3,l+A)' \text{ EVAL } \Rightarrow '1+A+(2+A)+(3+A)'$$

- A numeric sum, if the limits and the summand all evaluate to numbers; thus

$$' \Sigma(l=1,100,l^2)' \text{ EVAL } \Rightarrow 338350$$

If you differentiate a sum, the derivative is just pushed inside of the summation:

$$' \Sigma(l=A,B,F(X,l))' \text{ 'X' } \partial \Rightarrow ' \Sigma(l=A,B,\partial X(F(X,l)))'$$

The derivative ignores the limits, which are presumed to be integer functions with no continuous derivative. Differentiating with respect to the *index* always returns zero.

Similar considerations apply to the integration of a sum. When the integral is evaluated, if the summand is an integrable pattern, the result is the (unevaluated) sum with the summand replaced by its definite integral:

$$' \int(A,B,\Sigma(l=1,M,\text{SIN}(N*X)),X) \text{ EVAL EVAL } \\ \Rightarrow ' \Sigma(l=1,M, -(\text{COS}(N*B)/N)) - \Sigma(l=1,M, -(\text{COS}(N*A)/N))'$$

This is more legible in EquationWriter form as

$$\int_{A=1}^B \sum_{l=1}^M \text{SIN}(N*X) \text{ EVAL EVAL } \Rightarrow \sum_{l=1}^M - \left(\frac{\text{COS}(N*B)}{N} \right) - \sum_{l=1}^M - \left(\frac{\text{COS}(N*A)}{N} \right)$$

If the summand is not integrable, the integral is not pushed inside the sum; the result retains the sum as the integrand.

18.3.1 Summation Patterns

The HP 48 pattern-matching capabilities (section 17.3) may be used to develop a summation processor similar to the integral command $\int$ (section 18.4), which matches its integrands against a table of pre-defined patterns. For example, a simple sum of integers evaluates to

$$\sum_{i=1}^n i = n(n+1)/2.$$

In HP 48 pattern-matching terms, any sum of the form $\Sigma(\&1=1,\&2,\&1)$ can be replaced by $\&2*(\&2+1)/2$, assuming $\&2$ is an integer.

The program XSUM below takes any algebraic object and tests its sums against a small database of patterns stored in the variable Σ PATTERNS. If a match is found, the corresponding sum is replaced with a formula from the database. The Σ PATTERNS listed below contains replacements for the sum of integers, squared integers, and cubed integers. You can add additional pattern/replacement pairs to Σ PATTERNS as you please.

$'\Sigma(i=1,N,i)'$ XSUM $\rightarrow$ $'N*(N+1)/2'$

XSUM	Extended Sum	8BFA
	level 1 level 1	
	expression ₁ $\rightarrow$ expression ₂	
<pre><< ΣPATTERNS DUP SIZE $\rightarrow$ pats n << DO pats n DUP 1 - SWAP SUB UNTIL !MATCH 'n' DUP 1 STO- DECR 0 == OR END >> >></pre>		Save arguments. Next pattern. Quit if there is a match, or no more patterns.

Σ PATTERNS	Summation Patterns	78CB
	level 1	
	$\rightarrow$ { list }	
<pre>{ '$\Sigma(&1=1,&2,&1)'$ '$&2*(&2+1)/2'$ '$\Sigma(&1=1,&2,&1^2)'$ '$&2*(&2+1)*(2*&2+1)/6'$ '$\Sigma(&1=1,&2,&1^3)'$ '$&2^2*(&2+1)^2/4'$ }</pre>		Σi Σi^2 Σi^3

The program INTMATCH in section 18.4.1.3 is similar in spirit to XSUM, but it uses a more careful approach that permits more flexible matching than XSUM. The methods used by XINT can easily be adapted to summation matching.

18.4 Integration

Unfortunately, there is no analog of the chain rule of differentiation that allows you to compute the integral of an arbitrary expression directly from the antiderivatives of the functions in the expression. General-purpose symbolic integration algorithms require considerable memory resources, so the HP 48 limits itself to integration of a certain set of integrands, using its pattern-matching ability to generalize the set. For these and any other integrals, you also can invoke a sophisticated numerical integrator.

$\int$ is an HP 48 *function* (section 3.1), so that it can be included in expressions or executed as an RPN command. The RPN usage for $\int$ is as follows:

lower-limit upper-limit integrand name $\int$ $\Rightarrow$ result,

where *name* is a name object representing the variable of integration, and the other three arguments can be arbitrary expressions. The arguments appear in the same order in which they appear in the expression form of an integral:

' $\int$ (lower-limit, upper-limit, integrand, name)'

The single function $\int$ combines both symbolic and numerical integration, with the choice of method determined implicitly by the current function execution mode (section 3.5.5.2). With flag -3 clear (symbolic execution), $\int$ uses the pattern-matching system to return a symbolic result. With flag -3 set (numerical execution), $\int$ uses a numerical algorithm to return a real number result. You can also obtain a numerical result regardless of the flag setting by using $\rightarrow$ NUM to evaluate $\int$.

18.4.1 Symbolic Integration

Evaluation of $\int$ in symbolic execution mode (flag -3 clear) searches the integrand for functions for which the HP 48 knows antiderivatives, and which are linear in the variable of integration.

- A function linear in a variable x has an argument of the form $\pm a \cdot x \pm b$, where a and b are constants (relative to x). The HP 48 can integrate $\text{SIN}(X)$, so therefore it can also integrate $\text{SIN}(2 \cdot X - 6)$ or $\text{SIN}(A \cdot X + Z)$.
- Functions may be multiplied or divided by a constant, e.g. $C \cdot \text{SIN}(X)$ or $\text{SIN}(X)/(A+B)$.

Each such function is integrated and removed from the integrand. The final result is a sum of integrated terms plus an integral containing any remaining terms.

Note that symbolic integration (like ISOL--see section 16.4.1) does not attempt to find

indirect references in the integrand to the variable of integration. When the integrand contains variables that in turn contain expressions in the variable of integration (at any level), you should execute **SHOW** as a preliminary to $\int$, to make explicit all possible references to the integration variable.

Symbolic integration is performed in two steps. The first step, which is performed by $\int$, uses the *where* function $|$ to show a formal result prior to evaluation at the limits. For example,

```
RAD '∫(A,B,COS(X),X)' EVAL
      'SIN(X)/∂X(X) | (X=B) - (SIN(X)/∂X(X) | (X=A))'.
```

This result is more legible in the EquationWriter environment:

$$\left. \frac{\sin(X)}{\frac{\partial}{\partial X}(X)} \right|_{X=B} - \left. \frac{\sin(X)}{\frac{\partial}{\partial X}(X)} \right|_{X=A}$$

This is an expanded form of the common “double-where” notation used with integral results:

$$\left. \frac{\sin x}{\frac{\partial}{\partial x}(x)} \right|_{x=A}^{x=B}$$

which represents the difference between the expression on the left evaluated at the upper limit and at the lower limit.

The derivative that appears in the denominator of this result computes the linear coefficient of the integration variable as it appears in the argument of the function that is matched. That is, if the current integrand were $A \cdot X + B$, the result would have a denominator of $\partial X(A \cdot X + B)$, which evaluates to A . Using the derivative in this manner

is the most effective way to determine the linear coefficient.

The second step in symbolic integration is evaluation at the limits. Applying EVAL again to the last result returns

$$'SIN(B) - SIN(A)',$$

in which the integration variable has been replaced by its values at the limits.

You may find that most commonly you execute EVAL twice without even bothering to view the intermediate result. The HP 48 preserves the two-step procedure for pedagogical purposes, and to permit manipulations before the final evaluation for cases where automatic evaluation might lead to errors (see section 21.3.9).

Note that $\int$ always computes a definite integral, requiring explicit limits of integration. You can obtain an indefinite integral by choosing formal (undefined name) limits, then discarding the term arising from the lower limit. The user-defined function INDEF automates this process:

INDEF		Indefinite Integral		4CEE
level 2		level 1		level 1
'integrand'		'name'	☐	'integral'
<pre> << → f x << x f x SHOW 'x' DUP DUP2 STO ROT 4 PICK ∫ IF { '∫(&1,&2,&3,&4)' x } 1MATCH THEN DROP f ELSE OBJ→ 3 DROPN EVAL x ROT 2 →LIST 1MATCH DROP END >> >> </pre>		Create local variables. Show integration variable. Prevent evaluation of x. Integrate with dummy formal limits. If <i>true</i>, integration failed. Return original argument. Discard lower limit. Evaluate at the upper limit. Substitute original variable.		

INDEF takes two arguments, an integrand expression and the name of the variable of integration.

$$'X+1/(1+X^2)' \quad 'X' \quad \text{INDEF} \quad ☐ \quad 'ATAN(X)+X^2/2'$$

Because of its user-defined function structure, INDEF may also be evaluated within an expression. However, you should use QUOTE (section 16.7.1) to prevent evaluation of the arguments, e.g. 'INDEF(QUOTE(F(X)),QUOTE(X))'.

18.4.1.1 Integration Patterns

The set of integrand terms that the HP 48 can actually integrate symbolically is listed in Table 18.1 below. This set was chosen to include the following:

- All of the built-in functions which have closed-form anti-derivatives expressible in terms of other built-in functions, except LNP1. Thus ATAN is included, but not ! (factorial/gamma function)
- Derivatives of all built-in functions. For example, since the derivative of ATAN(X) with respect to X is given by the HP 48 as INV(1+X^2), the latter pattern is included, along with some minor variants.
- Additional "common" patterns, such as 1/TANH(X) and 1/(SIN(X)*COS(X)).
- Summations using Σ (section 18.3).
- Derivatives.
- Linear combinations of the above patterns, where the coefficients are constants.

In a number of cases, the built-in integrator will fail to match a pattern that you may think is obvious, especially if it is formally equivalent to a pattern that can be integrated. For example, INV(1+X^2) integrates successfully to ATAN(X), but INV(1+SQ(X)) does not. The integral pattern matching has a certain amount of flexibility, but not enough to cover all permutations of a pattern that you can obtain with a chain of identity transformations. In many cases, you should try expanding, collecting, and otherwise reorganizing terms so that the integrator can find a good match. If you apply EXPAN to INV(1+X^2), you obtain INV(1+X*X), which still does not integrate. However, next using COLCT returns INV'(1+X^2), which is one of the recognized patterns. POLY (section 17.4) is also a good way to reduce a general polynomial to a form that can be integrated.

Table 18.1 lists the integrand patterns recognized by the HP 48, and their corresponding antiderivatives. In the table, ϕ represents a linear function of the variable of integration; the listed antiderivatives should be divided by the first order coefficient in ϕ . Patterns marked with a $\dagger$ work also when 1/(...) is replaced by INV(...).

Table 18.1. HP 48 Symbolic Integral Patterns

Pattern	Antiderivative	Pattern	Antiderivative
ACOS(ϕ)	$\phi * \text{ACOS}(\phi) - \sqrt{1 - \phi^2}$	TAN(ϕ)	$-\text{LN}(\text{COS}(\phi))$
ALOG(ϕ)	$.434294481904 * \text{ALOG}(\phi)$	TAN(ϕ)/COS(ϕ)	INV(COS(ϕ))
ASIN(ϕ)	$\phi * \text{ASIN}(\phi) + \sqrt{1 - \phi^2}$	1/TAN(ϕ)†	LN(SIN(ϕ))
ATAN(ϕ)	$\phi * \text{ATAN}(\phi) - \text{LN}(1 + \phi^2)/2$	1/(TAN(ϕ)*SIN(ϕ))†	$-\text{INV}(\text{SIN}(\phi))$
COS(ϕ)	SIN(ϕ)	TANH(ϕ)	LN(COSH(ϕ))
1/(COS(ϕ)*SIN(ϕ))†	LN(TAN(ϕ))	TANH(ϕ)/COSH(ϕ)	INV(COSH(ϕ))
COSH(ϕ)	SINH(ϕ)	1/TANH(ϕ)†	LN(SINH(ϕ))
1/(COSH(ϕ)*SINH(ϕ))†	LN(TANH(ϕ))	1/(TANH(ϕ)*SINH(ϕ))†	$-\text{INV}(\text{SINH}(\phi))$
1/(COSH(ϕ)^2)†	TANH(ϕ)	$\sqrt{\phi}$	$2 * \phi^{1.5}/3$
EXP(ϕ)	EXP(ϕ)	1/ $\sqrt{\phi}$ †	$2 * \sqrt{\phi}$
EXPM(ϕ)	EXP(ϕ) - ϕ	1/(2* $\sqrt{\phi}$)†	$2 * \sqrt{\phi} * .5$
LN(ϕ)	$\phi * \text{LN}(\phi) - \phi$	ϕ^Z (Z symbolic)	IFTE(Z = -1, LN(ϕ), $\phi^{(Z+1)/(Z+1)}$)
LOG(ϕ)	$.434294481904 * \phi * \text{LN}(\phi) - \phi$	ϕ^z (z real, $\neq 0, -1$)	$\phi^{(z+1)/(z+1)}$
SIGN(ϕ)	ABS(ϕ)	ϕ^0	ϕ
SIN(ϕ)	$-\text{COS}(\phi)$	ϕ^{-1}	LN(ϕ)
1/(SIN(ϕ)*COS(ϕ))†	LN(TAN(ϕ))	1/ ϕ †	LN(ϕ)
1/(SIN(ϕ)*TAN(ϕ))†	$-\text{INV}(\text{SIN}(\phi))$	1/(1 - ϕ^2)†	ATANH(ϕ)
1/(SIN(ϕ)*TAN(ϕ))†	$-\text{INV}(\text{SIN}(\phi))$	1/(1 + ϕ^2)†	ATAN(ϕ)
1/(SIN(ϕ)^2)†	$-\text{INV}(\text{TAN}(\phi))$	1/(($\phi^2 + 1$)†	ATAN(ϕ)
SINH(ϕ)	COSH(ϕ)	1/($\sqrt{\phi - 1} * \sqrt{\phi + 1}$)†	ACOSH(ϕ)
1/(SINH(ϕ)^2)†	$-\text{INV}(\text{TANH}(\phi))$	1/ $\sqrt{1 - \phi^2}$ †	ASIN(ϕ)
1/(SINH(ϕ)*COSH(ϕ))†	LN(TANH(ϕ))	1/ $\sqrt{1 + \phi^2}$ †	ASINH(ϕ)
1/(SINH(ϕ)*TANH(ϕ))†	$-\text{INV}(\text{SINH}(\phi))$	1/ $\sqrt{\phi^2 + 1}$ †	ASINH(ϕ)
SQ(ϕ)	$\phi^3/3$		
TAN(ϕ)^2	TAN(ϕ) - ϕ		

18.4.1.2 Derivative and Integral

The derivative of an integral, in the most general case where the limits and the integrand are functions of the differentiation variable, is given by:

$$\frac{\partial}{\partial x} \int_{l(x)}^{u(x)} f(x, t) dt = f(x, u(x)) \frac{\partial u}{\partial x} - f(x, l(x)) \frac{\partial l}{\partial x} + \int_{l(x)}^{u(x)} \frac{\partial}{\partial y} f(x, t) dt$$

In HP 48 terms, this definition translates to:

$$\begin{aligned} & \text{'}\int(L(X),U(X),F(X,T),T)\text{' } 'X' \partial \\ & \text{'}F(X,U(X))*\text{der}U(X,1)-F(X,L(X))*\text{der}L(X,1)+\int(L(X),U(X),\text{der}F(X,T,\partial X(X),\partial X(T)),T)\text{'} \end{aligned}$$

Here L , U , and F may be built-in, library, or user-defined functions, and $\text{der}L$, etc., represent their derivatives. We have used user-defined functions here to illustrate the most general case; built-in functions yield simpler results because their derivatives ($\text{der}F$, etc.) are expressed in terms of other built-in functions.

For example,

$$\begin{aligned} & \text{'}\int(0,X^2,(T*X)^2,T)\text{' } 'X' \partial \text{ COLCT} \\ & \text{'}\int(0,X^2,2*(\partial X(T)*X+\partial X(X)*T)*(T*X),T)+2*X^3^2*X\text{'}. \end{aligned}$$

Although the integral in this result appears to be tractable, the presence of the ∂ 's prevents it from being matched by any of the built-in patterns. Evaluating an integral expression does not evaluate the integrand (as you would expect for the arguments of most functions), so any further evaluation of this result does not remove the derivatives. To finish the evaluation, you have to extract the integrand, evaluate it (keeping T undefined), and then reapply the integration. Manually, you can extract the integrand by selecting it in the EquationWriter subexpression mode and using SUB to copy it to the stack, and REPL to replace it after it is simplified. Alternatively, you can apply the program INTEVAL, which evaluates an expression while ensuring that integrands within the expression get evaluated before the integrals.

INTEVAL		Integrand Evaluation	C09E
		level 1	level 1
		'expression ₁ '	'expression ₂ '
<pre><< DUP → fn fx << << → f1 f2 f3 f4 << f1 f2 f3 f4 ∫ >> >> 'fx' STO fn { '∫(&1,&2,&3,&4)' 'fx(&1,&2,&3,QUOTE(&4))' } MATCH DROP EVAL >> >></pre>		Create local variables. User-defined function definition. Store in fx. Replacement pattern. Replace the integral. Evaluate the expression.	

If you execute INTEVAL on the result above, you obtain

$$' \int (0, X^2, 2 * T * T * X, T) + 2 * X^3^2 * X '.$$

The derivatives are gone, but the integrand still does not match an allowed integral pattern. However, COLCT helps here, by transforming the $T * T$ into T^2 :

$$\text{COLCT EVAL EVAL} \Rightarrow '2 * (X^2^3 / 3) * X + 2 * X^3^2 * X'$$

This result can be simplified further by using POLY (section 17.4.1):

$$'X' \text{ POLY} \Rightarrow '2.66666666667 * X^7'.$$

The case of integration of a derivative is more straightforward than the opposite, so the HP 48 realization is also correspondingly simpler. The definition

$$\int_l^u \frac{\partial}{\partial t} F(t) dt = F(u) - F(l)$$

translates on the HP 48 to

$$' \int (L, U, \partial T(F(T)), T) ' \text{ EVAL EVAL} \Rightarrow 'F(U) - F(L)'$$

18.4.1.3 Adding Integration Patterns

The list of integration patterns built into the HP 48 is fixed, and there is no provision for adding additional patterns for the $\int$ command itself. However, it is possible to use MATCH commands to program a substitute for $\int$ that tests integrals against an extensible list of patterns. The program XINT listed below demonstrates one such approach.

XINT uses a list of patterns stored in a variable IPATS (*Integral PATternS*). Each pattern is represented by a MATCH list, where the first object is an integrand pattern containing $\uparrow$ as the integration variable, and the second is the antiderivative. The third object is the optional MATCH test (section 17.3). For example, the MATCH patterns needed for

$$\int_a^b \frac{1}{x(x+1)} dx = \ln\left(\frac{x}{x+1}\right) \Bigg|_{x=a}^{x=b}$$

are represented by the list

$$\{ '1/(\uparrow\downarrow * (\uparrow\downarrow + 1))' 'LN((\uparrow\downarrow / (\uparrow\downarrow + 1))' \}.$$

With this list stored as one element of the IPATS list,

$$' \int (A, B, 1/(X*(X+1)), X)' \text{ XINT EVAL } \Rightarrow 'LN(B/(B+1)) - LN(A/(A+1))'$$

A powerful application of the optional MATCH test is to permit the generalization of some patterns in a manner similar to that of $\int$ itself. For example, the above pattern can easily be generalized to integrands of the form $c/(x(x+d))$, where c and d are constants. However, the IPAT entry should include a test to ensure that c and d are not functions of x :

$$\{ '&1/(\uparrow\downarrow * \uparrow\downarrow + \&2)' '&1 * LN((\uparrow\downarrow / (\uparrow\downarrow + \&2))' \\ '&NOT(FUNOF?(QUOTE(\&2), \uparrow\downarrow) OR (FUNOF?(QUOTE(\&1), \uparrow\downarrow)' \}$$

The test here uses a user-defined function FUNOF? (listed below), which uses MATCH to check its first argument for the presence of its second argument.

Another extension provides for cases where the integration variable appears uniformly as a linear form. The test for these cases is provided by the user-defined function LINEAR?. It uses OBJ $\rightarrow$ to dissect its first argument (much in the manner of $\rightarrow$ RPN in section 16.6), looking for instances of a name specified by the second argument. It returns *true* if the name appears at least once, and only to first-order.

Using LINEAR?, the integration of $\cos x \cdot \sin x$ can be represented in IPATS by

$$\{ 'COS(\&1)*SIN(\&1)' \\ '((-COS(2*\&1))/(2*\partial\uparrow\downarrow(\&1)))*(ACOS(-1)/\pi)' \\ 'LINEAR?(QUOTE(\&1,QUOTE(\uparrow\downarrow))' \\ \}$$

Then

$$' \int (A, B, COS(C*X)*SIN(C*X), X)' \text{ XINT } -2 \text{ SF EVAL EVAL } \\ \Rightarrow ' - (COS(2*(C*B))/(2*C)) + COS(2*(C*A))/(2*C)'$$

LINEAR?	Linear Form Test			B895
	level 2	level 1	level 1	
	'expression ₁ '	'name'	✕	flag
<< - e x << 0 0 1 0 0 -> args lin go non occ << << 1 SWAP START lin EVAL NEXT >> 'args' STO << - sube << IF go THEN CASE sube TYPE 9 SAME THEN sube OBJ-> CASE OVER NOT THEN DROP2 END { NEG + - * / } SWAP POS DUP NOT THEN DROP 1 -> non << args EVAL >> END DUP 4 < THEN DROP args EVAL END SWAP DROP 5 SAME THEN 1 -> non << lin EVAL >> lin EVAL END occ SWAP lin EVAL occ DUP ROT - ROT lin EVAL occ ROT - IF AND THEN 0 'go' STO END END END sube x SAME THEN 1 'occ' STO+ IF non THEN 0 'go' STO END END END END END >> >> 'lin' STO e lin EVAL go occ AND >> >> >>	Initialize local variables. Subroutine for multiple arguments Main test routing Continue if go is true. If current object is an expression, Then take it apart. Do nothing for zero-arg functions If non-linear, Then make non true and check the arguments. Handle +, -, NEG. For /, treat denominator an non-linear, and numerator as linear. For *, check first argument, and second arguments. If x appears in both, then quit. Zero arguments. Is it x? Then increment the occurrence count. If inside a non-linear expression, Then quit. Save the subroutine. Test the original argument. Return true if x was found, with no non-linear occurrences.			

FUNOF?	Function Of?			OBEF
	level 2	level 1		level 1
	'expression'	'name'	☐	flag
<pre> << -> f x << f x DUP 2 ->LIST !MATCH SWAP DROP >> >> </pre>				Store arguments. Test if f contains x .

XINT	Extended Integration			EA6F
	level 1		level 1	
	'expression ₁ '	☐	'expression ₂ '	
<pre> << << -> low up int x << int x {↑↑} + !MATCH DROP IPATS DUP SIZE 0 -> pats n t << DO pats n GET UNTIL !MATCH 't' STO 'n' DECR NOT t OR END IF t THEN DUP up 2 ->LIST 'f' APPLY SWAP low 2 ->LIST 'f' APPLY - { 'f(&1,&2)' '&1 (↑↑=&2)' } !MATCH DROP ELSE DROP low up int x f END >> >> >> -> f << EVAL { 'f(&1,&2,&3,&4)' 'f(&1,&2,QUOTE(&3),QUOTE(&4))' } IF !MATCH THEN EVAL END >> >> </pre>				Start definition of f . Replace integration var. with $↑↑$. Set up for loop through patterns. Get the next pattern. Repeat until a match, or list is exhausted. If there was a match: Dummy function of upper limit. Dummy function of lower limit. Replace f with $ $ forms. No match; return integral. Store the function f Try evaluating the argument. If f 's remain, Replace f with f , and execute f .

18.4.2 Numerical Integration

The HP 48's symbolic integration capability is limited to a small number of integrand patterns, but even the list of all known closed-form indefinite integrals is fairly short. In many cases, especially in a practical problem-solving context, a numerical integral is as useful or even better than a symbolic integral. The HP 48, like its HP calculator predecessors back to the HP 15C uses a Romberg numerical integration method (see, for example, Press, Flannery, Teukosky, and Vetterling, *Numerical Recipes*, Cambridge University Press, 1986). The integrand is sampled at non-uniform intervals, which helps prevent errors due to periodicity in the function. The integrand is generally not sampled at the integration limits, which is helpful when the limits occur at a singularity or other difficult point.

The numerical evaluation of $\int$ produces a series of increasingly accurate estimates of the integral, derived from sampling intervals that are halved at each iteration. The process terminates when three successive estimates differ by amounts less than an error tolerance that you specify, or after a maximum of sixteen iterations have produced no apparent convergence (at this point the integrand has been evaluated 65535 times). The error tolerance is determined by the current real number display setting: n FIX (or SCI or ENG) specifies an error tolerance $\epsilon = 10^{-n}$. This in turn relates to the probable error in the numerical integral:

$$\text{error} \leq \epsilon \int |f(x)| dx.$$

The use of the display format to determine the integration accuracy eliminates the need for an another explicit argument for $\int$ in addition to those needed for symbolic integration (this use of the display format is similar to that of $\rightarrow Q$ --see section 16.3.1). The syntax for the two forms of integration is therefore the same, so that you don't have to choose one form or another in advance. For example, if executing $\int$ in symbolic execution mode returns an expression that still contains an integral, you can go ahead and obtain its numerical value by executing $\rightarrow \text{NUM}$ on the expression.

As a function, $\int$ can only return one result to the stack. However, it is also helpful to obtain an estimate of the error of the calculation as well as the integral value itself. $\int$ returns the error estimate as a real number stored in a variable **IERR**, which is created in the current directory if it does not already exist. A successful integration yields a positive value for **IERR**; if the integrand is essentially always positive or always negative, you should find that the value is approximately ϵ times the integral. For example, with a display setting of 5 FIX,

```
'∫(0,1,EXP(X),X)' →NUM 1.71828182875,
```

and then

IERR 1.71814488996E-5

(here we are showing the results in STD format). Dividing the latter result by the integral value yields $9.9992\text{E}-6$, which is comparable to the error tolerance 10^{-5} derived from the display setting. When the integrand is a reasonably well-behaved function, the actual error can be substantially less than the prediction represented by IERR. In this example, the result differs from the correct value ($e-1$) only in the tenth decimal place.

If the integration results have not converged after sixteen iterations, the number -1 is stored in IERR. The value returned to the stack is the last estimation of the integral.

The choice of error tolerance for any integral is a trade-off of numerical precision against execution time. Reducing the error tolerance increases the number of integrand samples that are necessary. Consider, for example, the following integral:

$$\int_0^{\pi} (5+x \cos 3x) dx = 5\pi - 2/9 \approx 15.4857410458$$

Evaluating the object ' $\int(0,\pi,5+X*\text{COS}(3*X)),X$ ' with different display formats, we find:

Display	Result	Error	IERR	Samples	Time(sec)
1 FIX	15.6	0.1	1.6	7	0.8
3 FIX	15.486	$5.5\text{E}-6$	0.015	31	2.8
5 FIX	15.48574	$8.3\text{E}-8$	$1.5\text{E}-4$	63	5.4
7 FIX	15.4857410	0	$1.5\text{E}-6$	127	10.7
9 FIX	15.48574104	$1\text{E}-10$	$1.5\text{E}-8$	127	10.8
STD	15.4857410457	$1\text{E}-10$	$1.5\text{E}-10$	255	21.3

Notice that reducing the error tolerance below $\epsilon=10^{-7}$ (7 FIX) doesn't appreciably improve the integration, which has reached the intrinsic accuracy limit of the calculator. This is because the integrand is a reasonably smooth curve in the region of interest, and can be accurately represented by a fitted curve based on the 127 sample points.

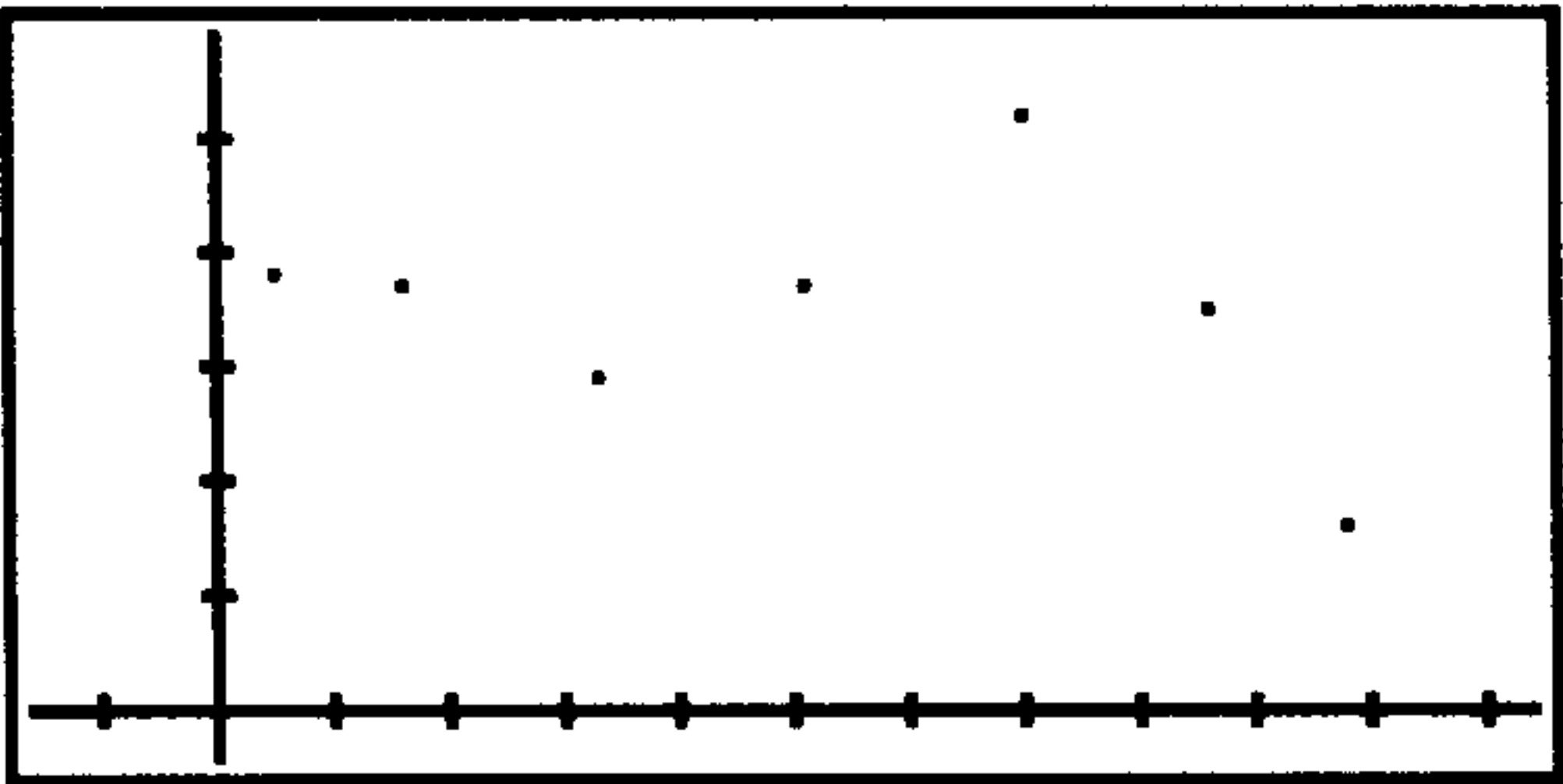
It is interesting to show where an integrand is actually sampled, which you can do with the following program. INTSAMP takes an expression containing an integral, and replaces the integrand with a function that plots each value that it returns as it is integrated.

INTSAMP	Integration Samples		77F1
	level 1		level 1
	'expression'	→	x
<pre><< 0 → int << DRAX { #0 #0 } PVIEW << → f x << f →NUM DUP x SWAP R→C PIXON >> >> 'int' STO { 'f(&1,&2,&3,&4)' 'f(&1,&2,int(&3,&4),&4)' } MATCH DROP →NUM >> >></pre>			Create a local variable int. Watch the sampling. Evaluate and plot a point. Make int a user-defined function. Match patterns. Replace integrand with int. Evaluate the integral.

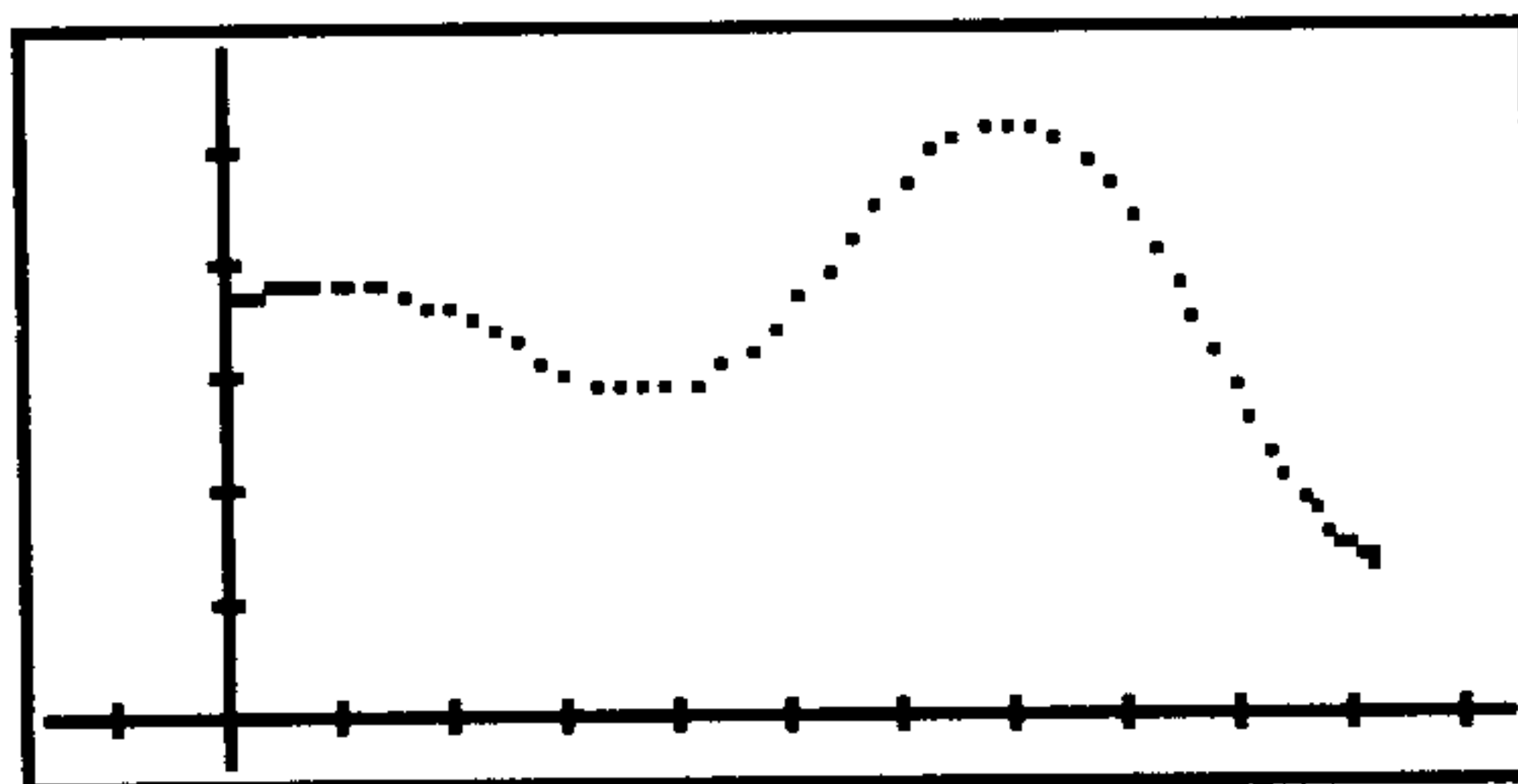
Using INTSAMP on the current example,

```
1 FIX -.5 3.5 XRNG -.5 8 YRNG 'f(0,π,5+X*COS(3*X),X)' INTSAMP
```

yields this picture:



The seven sample points make a fairly sparse representation of the function. Repeating the exercise after 5 FIX plots 63 sample points:



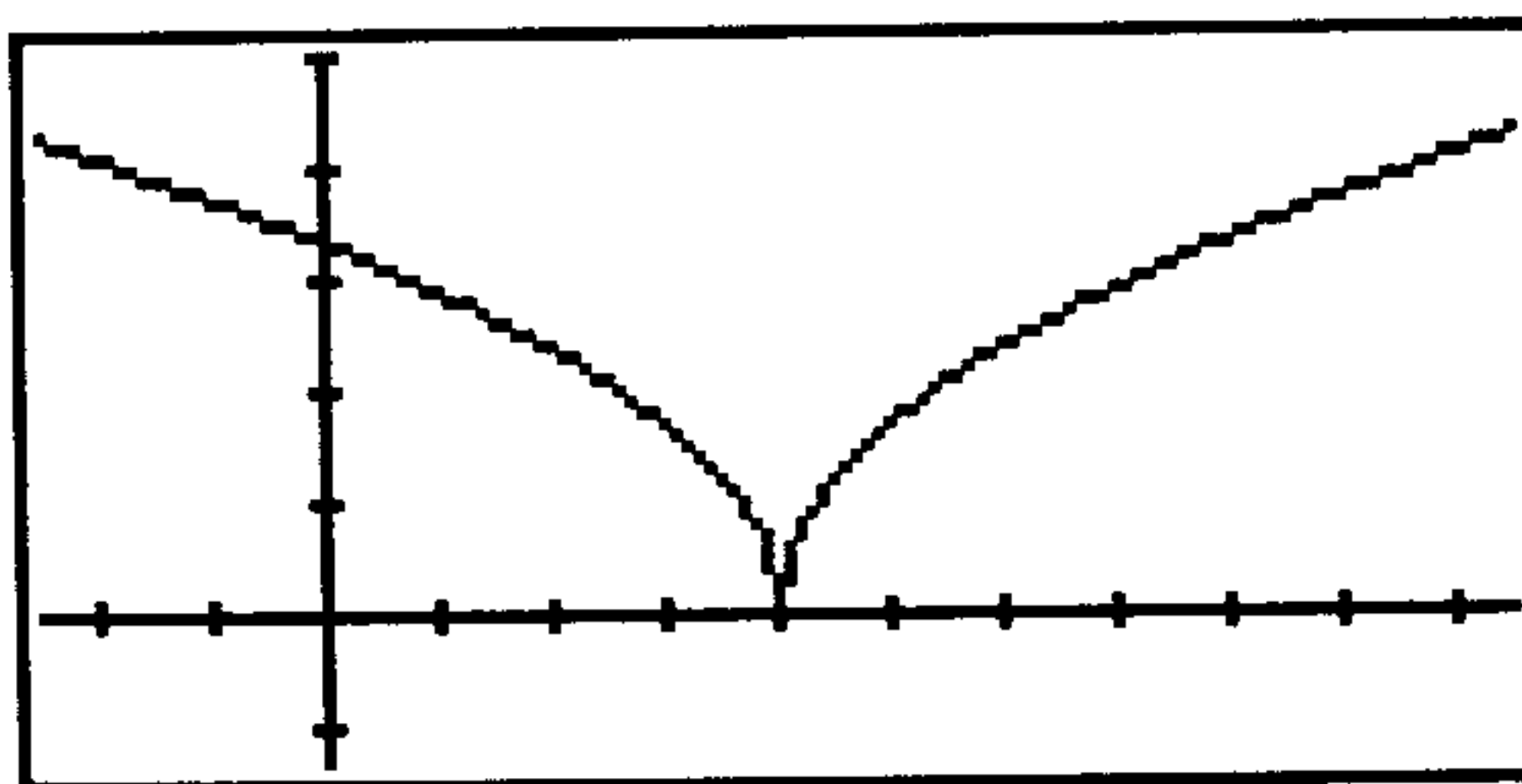
You can see that increasing the number of sample points in this example should not make a dramatic improvement in the integration accuracy.

18.4.3 Integration Strategies

Not all integrands are nice smooth functions like $x \cos x$, and so will not always numerically integrate so nicely. For example, consider the integral

$$\int_{-.5}^{2.5} \sqrt{|x-1|}$$

Plotting the integrand from $-.625$ to 2.625 (y-range -0.4 to 1.5) shows a cusp at $x=1$:



The cusp makes the approximation of the function for numerical integration difficult; with an error tolerance of 10^{-5} , executing

$$' \int (-.5, 2.5, \sqrt{\text{ABS}(X-1)}, X)' \rightarrow \text{NUM} \quad \boxed{2.44948455085}$$

takes nearly four minutes (4095 samples).

In this case, and in many other numerical integration problems, a simple change in the

setup of the problem can make a great deal of difference in the integration time and the accuracy of the result. One useful strategy is to break up the domain of the integration into two or more smaller regions. In the current example, the cusp at $x = 1$ is the obvious segmentation point:

$$\int(-.5,1,\sqrt{\text{ABS}(X-1),X}) + \int(1,2.5,\sqrt{\text{ABS}(X-1),X}) \rightarrow \text{NUM} \quad \boxed{2.44948983536}$$

takes only two seconds, and the result is accurate to two more decimal places than the previous result.

The program SEGINT automates the process of segmenting an integral. It takes as arguments an expression containing an integral, and a second object that specifies the segmentation point. It returns a new expression containing the original integral split into two parts:

$$\int(-.5,2.5,\sqrt{\text{ABS}(X-1),X}) \quad 1 \quad \text{SEGINT} \\ \boxed{\int(-.5,1,\sqrt{\text{ABS}(X-1),X}) + \int(1,2.5,\sqrt{\text{ABS}(X-1),X})}$$

SEGINT		Segment an Integral	2F81
		level 2	level 1
		$\int_1$	limit
			$\int_2$
<pre><< - limit << { 'f(&1,&2,&3,&4)' 'f(QUOTE(&1),limit,QUOTE(&3),&4) + f(limit,QUOTE(&2),QUOTE(&3),&4)' } !MATCH DROP EVAL { 'f(&1,&2,QUOTE(&3),&4)' 'f(&1,&2,&3,&4)' } !MATCH DROP >> >></pre>		Save intermediate limit as limit. Split the integral. Replace limit with its value. Unquote the integrand.	

SEGINT goes to some trouble to avoid any evaluation of the integrand or limits, so that you can check its result before proceeding with the integration. If you want to skip that step, you can use NSEGINT:

$$\int(-.5,2.5,\sqrt{\text{ABS}(X-1),X}) \quad 1 \quad \text{NSEGINT} \quad \boxed{2.44948983536}$$

NSEGINT	Numerical Segment Integral			8482
	level 2	level 1		level 1
	'integral'	limit	☞	value
<pre> << → ↑↑ << { '∫(&1,&2,&3,&4)' '∫(&1,↑↑,&3,&4)+∫(↑↑,&2,&3,&4)' } ↓MATCH DROP →NUM >> >> </pre>				Save intermediate limit as ↑↑. Split the integral. Evaluate the expression.

Not all cases have segmentation points as obvious as that in the previous example. For example, consider the integral

$$\int_1^{\infty} \frac{1}{x(x+1)} dx$$

If you attempt to evaluate this directly on the HP 48, you obtain a nonsense answer:

$$'∫(1,MAXR,1/(X*(X+1)),X)' \rightarrow \text{NUM} \quad \text{☞} \quad 0$$

This is a numerical precision problem: the integrand is effectively zero over most of the domain between the limits. You can do better by using NSEGINT to divide the domain into two parts:

$$5 \text{ FIX } ∫1,MAXR,1/(X*(X+1)),X' \quad 100 \quad \text{NSEGINT} \quad \text{☞} \quad 0.68320$$

It is hard to have confidence in this result, since there is nothing special about the segmentation point $X=100$. Indeed, using $X=1000$ instead returns 0.69215, suggesting that these approximations may only be accurate to one or two places.

A better strategy is transform the integral into one more suited for numerical integration, by an appropriate change of variables. That is, if we define a variable $y=g(x)$, then an integral

$$\int_{x_l}^{x_u} f(x) dx$$

can be re-expressed as

$$\int_{g(x_l)}^{g(x_u)} f(g^{-1}(y)) \cdot \frac{d}{dy}(g^{-1}(y)) dy$$

For integrals where one or both limits are infinity, a particularly useful transformation is $y = \text{atan}(x)$, which maps the entire real x -axis onto the interval $-\pi/2 \leq y \leq \pi/2$.

The program CHVAR listed next transforms an integral expression according to the rules given above, with the variable change specified by an equation of the form ' $y=g(x)$ '. For the current example, we will use the arc-tangent transformation:

```
'∫(1,MAXR,(1/(X*(X+1)),X)' 'Y=ATAN(X)' CHVAR
```

```
⌞ '∫(ATAN(1),ATAN(MAXR),1/(TAN(Y)*(TAN(Y)+1))*(1+TAN(Y)^2),Y)'
```

This symbolic result is not particularly illuminating, except that you can observe that the integration limits have become $\text{atan}(1) = \pi/4$ and $\text{atan}(\infty) = \pi/2$. The result easily integrates numerically:

```
STD →NUM ⌞ .693147180555
```

which compares well with the exact result ($\ln 2$), which is 0.693147180560 to twelve places.

CHVAR	Change of Variables	011D
level 2	level 1	level 1
'∫f(x)dx'	'y=g(x)'	'∫h(y)dy'
<pre><< DUP EQ→ 4 ROLL OBJ→ DROP2 RCLF → y g low up f x flags << -1 SF x ISOL EQ→ SWAP DROP → ginv << g x low 2 →LIST ↑MATCH DROP g x up 2 →LIST ↑MATCH DROP f x ginv 2 →LIST ↑MATCH DROP ginv y FDER * y 4 →LIST ↑↑ APPLY { '↑↑(&1,&2,&3,&4)' '∫(&1,&2,&3,&4)' } ↑MATCH DROP flags STOF >> >> >></pre>		Take apart the expressions. Create local variables. Principal value mode. Compute and save $g^{-1}(y)$. $g(x_l)$ $g(x_u)$ $f(g^{-1}(y))$ dg^{-1}/dy Create a dummy function. Replace ↑↑ with $\int$. Restore flags.

Note that unlike the other programs in this section, CHVAR expects its first argument to

be an algebraic object containing only an integral. CHVAR uses the program FDER (section 18.1.2) to compute unevaluated derivatives.

18.4.4 Programs as Integrands

In the HP28 and earlier calculators, it is possible to obtain a numerical integral by representing the integrand with a program (called *implicit* integration on the HP28). The program contains no explicit variable of integration, but instead is designed to take a value from the stack (the integration sample point), and return the integrand value at that point.

Although the HP 48 does not specifically provide for program integration, it is simple to convert a program to an integrable form. It is only necessary to convert it to a user-defined function, then integrate that function. That is, given a program $\ll \text{program} \gg$, you must recast it as

$$\ll \rightarrow x \ll x \text{ program} \gg \gg$$

and store it in a variable. Then you can integrate it, e.g.

$$' \int(A,B,F(X),X)' \rightarrow \text{NUM},$$

where F is taken here as the user-defined function name.

As usual, it is possible to automate this process by means of a program:

PRGINT <i>Program to Integral</i> 8E67				
level 3	level 2	level 1		level 1
$\ll \text{program} \gg$	lower	upper		integral
$\ll \rightarrow \text{low up prog}$ $\ll$ $\ll \rightarrow x$ $\ll x \text{ prog EVAL}$ $\gg$ $\gg \rightarrow f$ $\ll ' \int(\text{low},\text{up},f(x),x)' \rightarrow \text{NUM}$ $\gg$ $\gg$ $\gg$				Save arguments. user-defined function. Store as f. Evaluate the integral.

PRGINT takes three arguments: a lower limit, an upper limit, and a program for the integrand. The program should take one value from the stack, and return one value. PRGINT returns a numerical integral, computed to a precision determined by the number display setting. For example, you can compute the integral

$$\int_0^5 x^2 dx$$

with

```
3 FIX 0 5 << SQ >> PRGINT 41.667
```

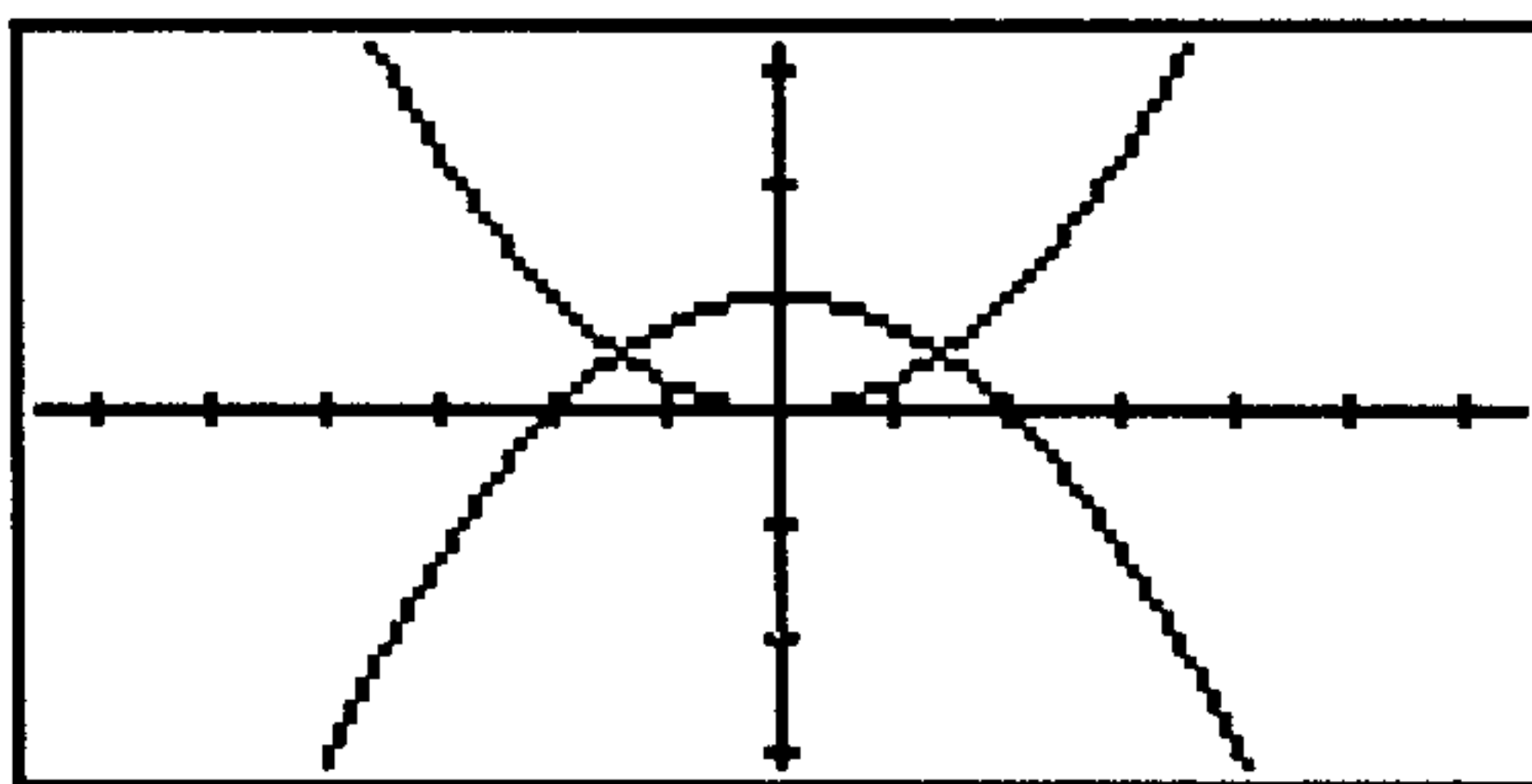
18.4.5 Multiple Integrals

Because $\int$ is an HP 48 function, it is straightforward to use it to compute a *double integral*, where the integrand itself contains an integral. Multiple integrals, where integrals are nested any number of times, present no additional difficulties.

■ *Example.* Find the area of a region bounded by the parabolas $y = x^2$ and $y = 4 - x^2$.

To help you visualize the problem, start by plotting the two curves:

```
'PPAR' PURGE -12.4 12.8 YRNG 'X^2=4-X^2' STEQ ERASE DRAW
```



To find the points where the curves intersect, execute

```
RCEQ 'X' QUAD COLCT 'X=1.41421356237*s1',
```

which shows that the intersections are at $x = \pm\sqrt{2}$. The area bounded by the two curves is therefore given by the integral

$$\int_{-\sqrt{2}}^{\sqrt{2}} dx \int_{x^2}^{4-x^2} dy$$

As an algebraic object, this integral is

$$' \int(-\sqrt{2},\sqrt{2},\int(X^2,4-X^2,1,Y),X)',$$

which looks like this in the EquationWriter:

$$\int_{-\sqrt{2}}^{\sqrt{2}} \int_{x^2}^{4-x^2} 1 dY dX$$

Executing EVAL once, you obtain:

$$' \int(-1.41421356237,1.4142135627,1*Y|(Y=4-X^2)-(1*Y|(Y=X^2)),X)'$$

Because the integrand is invisibly quoted (section 18.4.1.2), further EVAL's don't change this result. COLCT helps, by resolving the *where* forms:

$$\text{COLCT} \Rightarrow ' \int(-1.41421356237,1.4142135627,4-X^2-X^2,X)'$$

Then

$$\text{EVAL EVAL} \Rightarrow 7.54247233266$$

This same result can be obtained in a single operation by applying →NUM to the original integral, but the double numerical integration takes longer than the symbolic evaluation to obtain the same accuracy.

18.4.6 Polynomial Approximations

The HP 28 provides an approximate form of symbolic integration, in which an integrand is converted to a Taylor's polynomial which is then integrated. This primitive type of integration may not seem particularly useful, since an integral computed this way will

not much resemble the correct symbolic form of the integral unless the integrand happens to be a polynomial in the integration variable. But in many circumstances, you may be less interested in the precise mathematical form of an expression than in determining a reasonable view of its behavior over some region of interest. For this purpose, the HP28 approximate symbolic integral is an improvement over numerical integrals, and you may want to apply the method on the HP48 for integrands that do not match any of the built-in patterns. It is much faster to compute the value of a polynomial as you vary some parameter in the expression than to evaluate an integral numerically for the same values of the parameter.

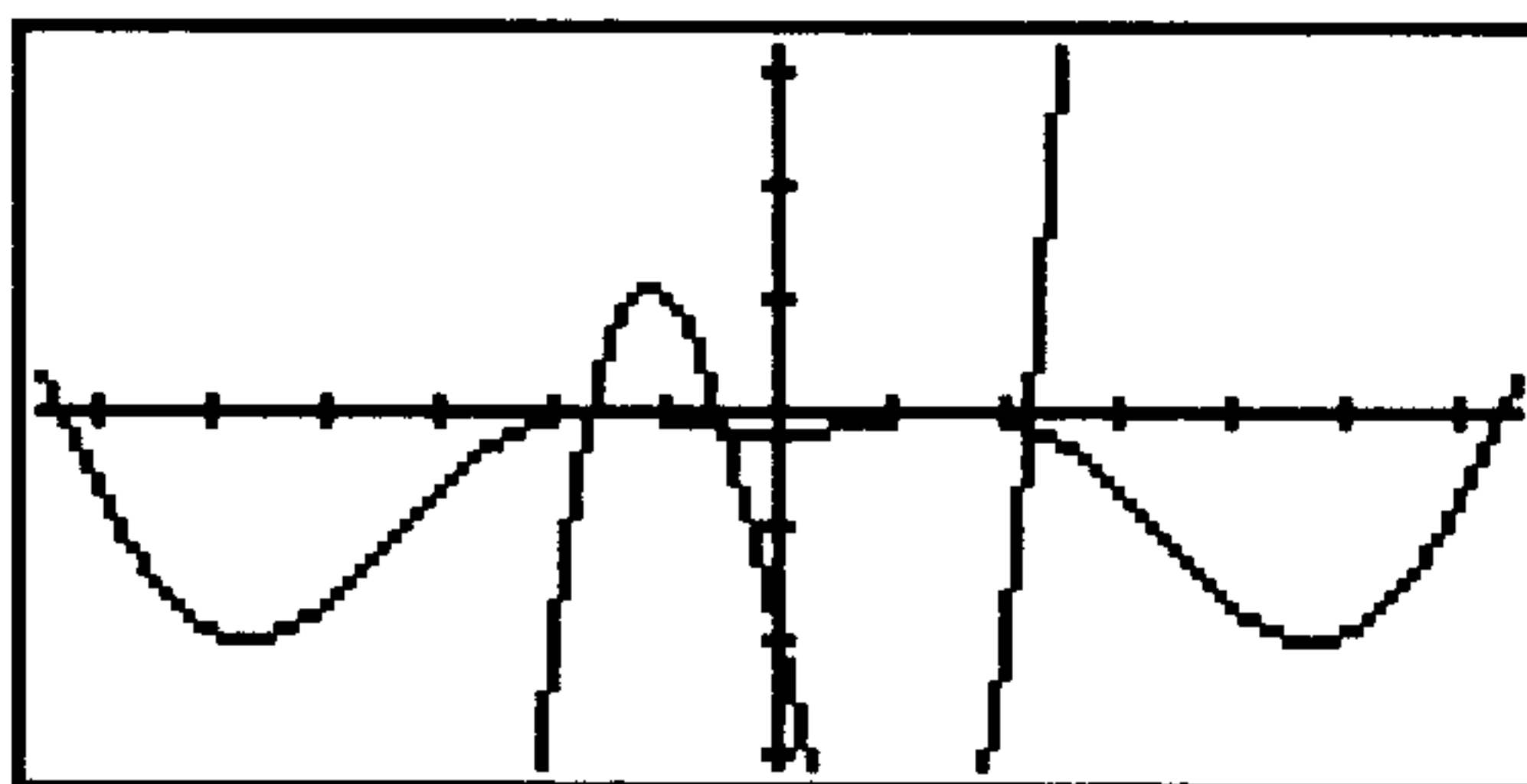
As an example of the appropriate use of such an approximation, consider the solution of the equation

$$\frac{1}{\pi} \int_{-\pi/2}^x t \cos^2(t) dt = x^3 - 4x - 2.$$

or, in HP48 syntax,

$$'1/\pi * \int(-\pi/2, X, T * \cos(T), T) = X^3 - 4 * X - 2'$$

Even executing 3 FIX to reduce the numerical integration times, it takes over six minutes to plot this equation (default plot parameters, radians mode):



Then, using FCN ISECT with the cursor right at the rightmost intersection of the curves, it takes an additional 30 seconds to find the root of the equation at $X=2.203$.

An alternate approach is to approximate the integrand on the left side of the equation with a polynomial:

$$'T * \cos(T)' \quad 'T' \quad 5 \quad \text{TAYLR} \quad \Rightarrow \quad 'T - 3/3! * T^3 + 5/5! * T^5'$$

Then, integrate this result from $-\pi/2$ to X :

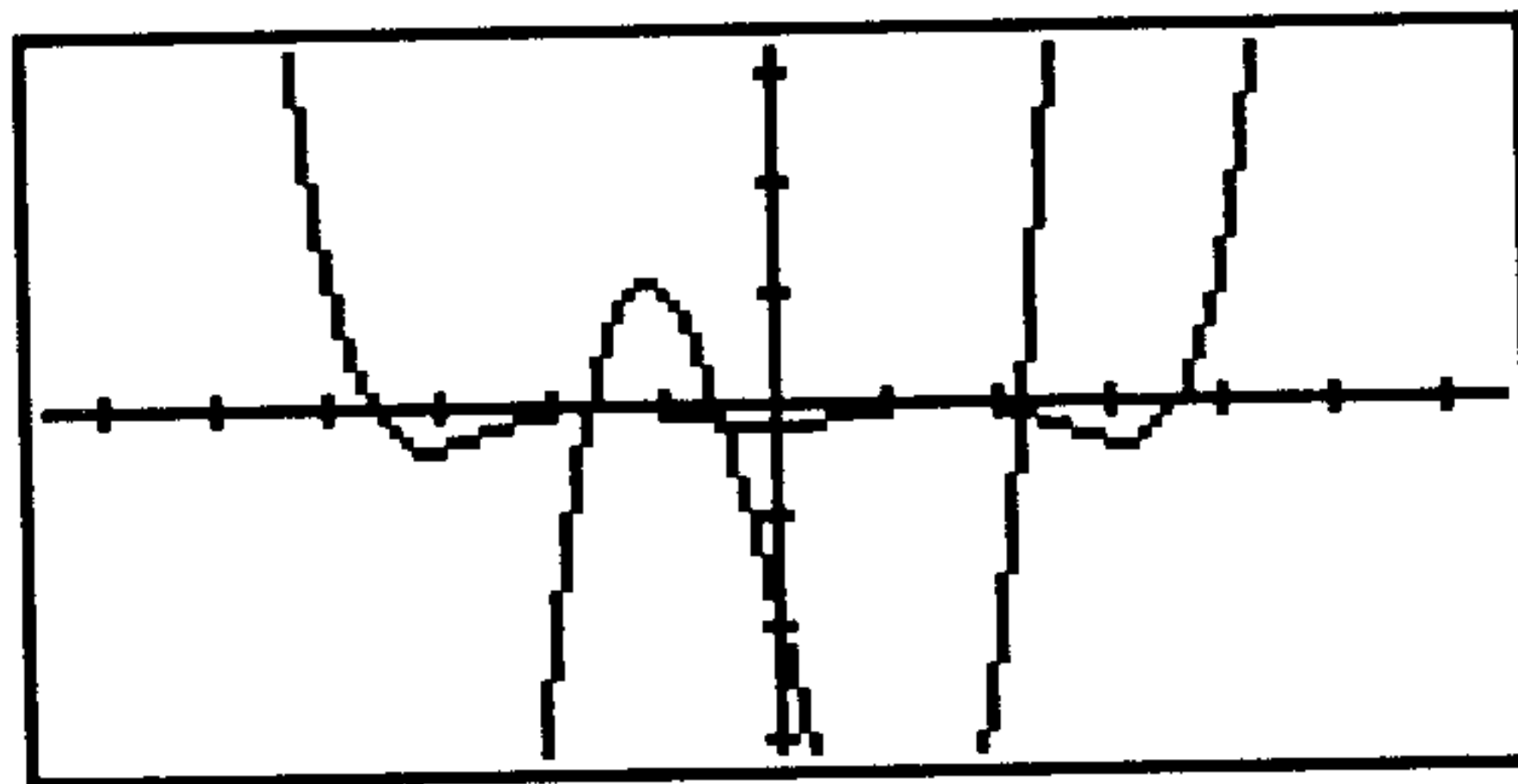
```
'X' PURGE -2 SF '-π/2' 'X' ROT 'T' ∫ EVAL COLCT
```

```
␣ '-.577009430249+.5*X^2-.125*X^4+6.944444444455E-3*X^6'
```

(Here we have set flag -2 to convert π to a number, only to keep the final result short enough to be reasonably legible.) To create the approximate equation representing the original problem, divide this result by π , and equate it to ' $X^3-4*X-2$ ':

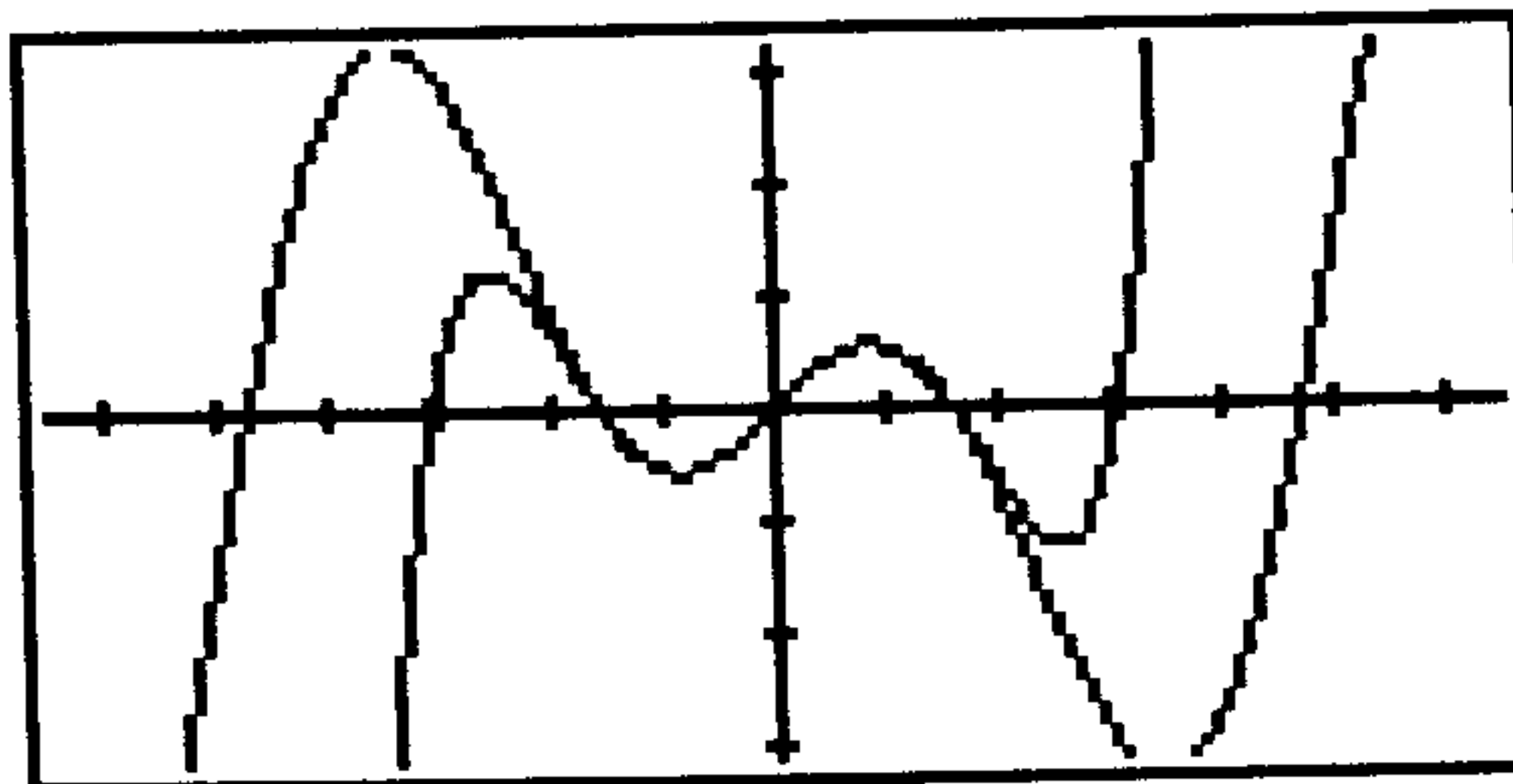
```
π / 'X^3-4*X-2' = STEQ
```

For this equation, DRAW takes only 37 seconds:



ISECT (5 seconds) returns 2.205, which differs from the previous result by only 0.1%. For more accuracy, you can increase the order of the Taylor's polynomial. You can get an idea of the validity of the approximation by plotting ' $X*\cos(X)$ ' and its polynomial approximation together:

```
{ 'X*COS(X)' 'X-3/3!*X^3+5/5!*X^5' } STEQ DRAW
```



The two curves are substantially the same in the range $|x| < 2$, and diverge elsewhere. Repeating the exercise with a higher order approximation yields a better match over the region of interest $-\pi/2 < x < 2.3$; for example, with a 9th order polynomial, the approximate root differs by 0.0003% from that obtained from the numerically evaluated integral.

19. The Time System

The HP 48 time system provides these general capabilities:

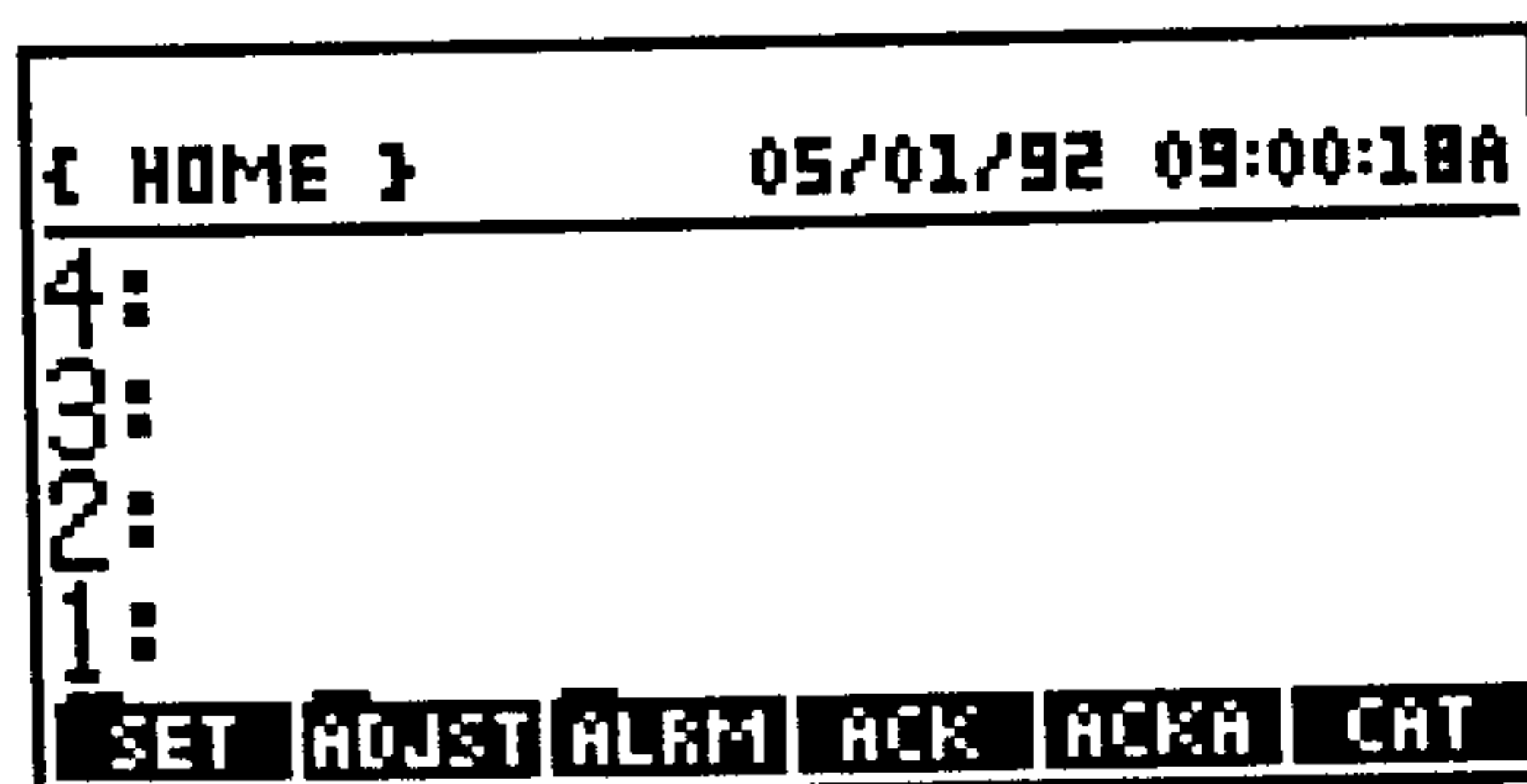
- An optional real-time “ticking” digital clock display, including the current date.
- Commands that return the current time and date.
- Time and date arithmetic.
- Appointment and object execution alarms.

Ordinary manual time operations--setting or adjusting the clock, setting alarms, and the alarm catalog--are straightforward and simple to use. This chapter will focus on aspects of the time system that are not emphasized in the *Owner's Manual*, especially the computational and programmable aspects of the time system.

19.1 The Clock

The HP 48 clock is based on a quartz crystal that nominally oscillates at 32,768 Hz. The crystal is comparable in accuracy to that found in common quartz watches, except that it does not usually enjoy the same temperature regulation as a wristwatch that spends the day next to your skin, and so may not achieve a time accuracy quite as high. There is no means on the HP 48 to adjust the clock rate. (There is a program available on the Hewlett-Packard calculator electronic bulletin board that can determine the actual clock rate experimentally, and set periodic alarms to adjust the time to maintain better time precision.)

You can view the current HP 48 time by activating the “ticking” clock display in the status area. This happens automatically whenever you activate the the alarm catalog ($\boxed{\rightarrow}$ **TIME**) or the time menu ($\boxed{\leftarrow}$ **TIME**)



“Ticking” refers to the second-by-second update of the time display. By default, the

time is displayed in 12-hour format, using A or P to indicate a.m. or p.m. You can select a 24-hour format manually by pressing 12/24 in the time-set menu ($\leftarrow$ **TIME** $\equiv$ **SET** $\equiv$), or in a program by setting flag -41. Similarly, you can use $\equiv$ **M/D** $\equiv$ or set flag -42 to change the default date display from *month/day/year* to *day/month/year*.

The current time is encoded in HP 48 memory by means of a 52-bit RAM register and a 32-bit electronic counter. The register stores the scheduled time of the “next event”—the next time-dependent action that the HP 48 must take, such as the next alarm to come due, the next one-second display clock update, or the ten-minute inactivity calculator turn-off. (There is a second time counter that is used for cursor blink, which is therefore independent of the rest of the time system.) The counter is a countdown timer that represents the time remaining until that next event. The current “real” time then is the difference between the values in the next event register and the counter. That time can be anywhere in the range January 1, 1990 to Dec 30, 2089.

The counter decrements by one for each four crystal oscillations, meaning that the fundamental clock “tick” is actually 1/8192 second. With 32 bits for the counter, the longest the HP 48 can go without a time event is 2^{31} ticks, slightly more than 3 days. That occurs when the HP 48 is turned off, and no alarms are scheduled within that time—then the calculator wakes itself up just long enough to reschedule the next event time. Usually, all of this processing happens so fast that it is not noticeable. The only exception is when you have a large number of alarms stored in the alarm catalog (section 19.2.5); the time required to scan through the list of alarms to find the next due alarm could make the digital clock display miss an occasional one-second display update.

The time encoded by the next event register and the timer is deliberately designed to be protected from accidental reset. Even system halts and memory resets (section 5.8) do not affect the time data, even if a reset is caused automatically by the detection of a memory fault. A special checksum is maintained for the next event register; only if that checksum indicates a problem with the time register content itself is the clock reset to the default of midnight, January 1, 1990.

19.1.1 Setting and Reading the Time

The HP 48 uses ordinary real numbers to represent the time and date in stack operations. A *time number* has the format *hh.mmssssssss*, where *hh* is one or two digits representing the hour 0-23, and *mm* are two digits representing the minute 0-59. The first two digits *ss* are a whole number 0-59 of seconds; the remaining digits encode the decimal fraction seconds. Thus 11.0519801 represents 11 hours, 5 minutes, and 19.801 seconds.

Date numbers are real numbers used to represent calendar dates. These numbers use

one of two forms *mm.ddyyyy*, or *dd.mmyyyy*, depending on whether the date format flag -42 is clear or set, respectively. Here the digits *dd* are the day number 1-31, *mm* is the month 1-12, and *yyyy* is the four-digit year. You can switch the format by setting or clearing flag -42, or by pressing **M/D**.

The time-set menu (**TIME SET**) provides menu keys for setting the clock manually:

- **>TIME** takes a time number from the stack and sets the clock to that time.
- **>DATE** uses a date number to set the system date.
- **A/PM** switches the time between a.m. and p.m. It is normally used after **>TIME** when a post-noon time is entered in a 12-hour form, i.e. 3 **>TIME** **A/PM** sets the time to 3:00 p.m.

You can adjust the time by hour, minute, or second increments or decrements by using the *time-adjust menu* (**TIME ADJUST**):

{ HOME }		05/01/92 09:00:27A			
4:					
3:					
2:					
1:					
HR+	HR-	MIN+	MIN-	SEC+	SEC-

Each menu key adds or subtracts one of the indicated time units from the system time. **HR+** and **HR-** are handy when you change time-zones; the remaining keys are used to correct the time to match an external clock. Note that none of these keys affects the clock rate.

For programming purposes, the command **TIME** (in the second page of the **TIME** menu) reads the HP 48 clock and returns the current time in time number format. The inverse operation is **→TIME** (first page of the **TIME SET** menu), which takes a time number from level 1, and sets the clock to match the specified time. Neither **TIME** nor **→TIME** is affected by flag -41--both work with time numbers expressed in 24-hour format.

For timing applications where speed and consistency is important for reading the clock, you can use **TICKS** (**TIME** menu, page 2). This command returns the time as a

binary integer, which represents the system clock time in *ticks*--units of 1/8192 seconds. The absolute magnitude of the number is not particularly useful; typically TICKS is useful for computing the time *difference* between two executions. An example is provided by the program TIMED, in section 12.9, which uses TICKS to mark the start and end of a timed execution. After the execution, when speed is not critical, the difference in the two times is converted to ordinary seconds.

A counterpart to TICKS is CLKADJ (*CLocK ADJust*), which adds a specified (real) number of ticks to the system time. CLKADJ, found in the second page of the **[<] TIME** **≡ADJUST≡** menu, is the programmable version of the clock-adjust operations in the first page. For example, to advance the clock by one hour, execute

8192 3600 * CLKADJ.

CLKADJ's argument can be number of ticks up to $\pm 25,834,291,200$, which is ± 365 days ($8192 \cdot 3600 \cdot 24 \cdot 365$). CLKADJ automatically corrects the date, if necessary. For this reason, using CLKADJ to shift the system time is preferable to using TIME and $\rightarrow$ TIME, e.g.

TIME 1 + $\rightarrow$ TIME.

The latter method is less accurate because it doesn't account for the time to execute TIME 1 +, and takes no account of possible date changes.

DATE (**[<] TIME** menu, second page) and $\rightarrow$ DATE (**[<] TIME** **≡SET≡** menu) read and set the date using date numbers, much like TIME and $\rightarrow$ TIME use time numbers. However, the date commands *are* sensitive to the date format flag; for example, if the date is July 1, 1992, then DATE returns 7.011992 when flag -42 is clear, or 1.071992 when flag -42 is set.

For program display purposes, you can obtain the combined time and date as a string object by using TSTR. This command takes a date number (level 2) and a time number (level 1), and returns a string like those used in the alarm set menu (**[<] TIME** **≡ALRM≡**):

-41 CF -42 CF 10.281991 18.12428 TSTR **[<]** "MON 10/28/91 06:12:42P"

TSTR respects flag -41 (12/24 hour mode) and -42 (date format). It truncates its time argument to integer seconds. A simple application of TSTR is shown by the following program, which takes a date number and returns a three-letter string for the day of the week to level 2, and a number 1-7 for the numerical day of the week (Sunday=1).

DOW	Day of Week			6DDA
	level 1		level 2	level 1
	date	☞	"day"	day-number
<< 0 TSTR 1 3 SUB "SUNMONTUEWEDTHUFRISAT" OVER POS 1 - 3 / 1 + >>				Get the DOW string. Find in master string. Convert position to day number.

19.1.2 Time Arithmetic

The third page of the  **TIME** menu contains four commands that assist with time arithmetic, by providing for conversion between time numbers and their equivalents in decimal hours, and for direct addition and subtraction of time numbers. A more thorough implementation of time (or date) arithmetic on the HP 48 would have provided separate object types for these quantities, so that ordinary + and - could be used. That approach would, however, have consumed considerable additional built-in ROM, at the expense of other HP 48 features. The use of time numbers in the *hh.mmssss* format at least has historical precedent in HP calculators, all the way back to the HP 45.

Direct addition and subtraction of time numbers is provided by the commands HMS+ and HMS-. Each takes a pair of time numbers, and returns the sum or difference as a time number. Thus, for example, to determine the time that is 3 hours and 45 minutes later than 8:25:37 AM:

8.2537 3.45 + HMS+  12.1037.

These commands do not constrain their results to the normal time number range 0-24 hours--their action is to convert the minutes/seconds parts of the time numbers to decimal hours, perform the addition or subtraction, then convert the decimal part of the result into .mmssss format. This means that you can apply the operations equally well to angles measured in degrees, minutes and seconds:

45.12 56.37 HMS-  -11.25

finds the angle that is 56°37' less than 45°12'.

→HMS converts an angle or time from ordinary decimal form into the time number format:

1 R→D →HMS  57.1744806247

shows that 1 radian is $57^{\circ}17'44''.806247$. Following the usual HP 48 naming convention, the inverse of $\rightarrow\text{HMS}$ is $\text{HMS}\rightarrow$.

19.1.3 Date Arithmetic

The date arithmetic analog of $\text{HMS}+$ is $\text{DATE}+$, which computes a new date by adding a number of days (level 1) to an initial date (level 2). For example, the one hundredth day of 1992 is found by

```
-42 SF 31.121991 100 DATE+  $\Rightarrow$  9.041992,
```

which is 9 April, 1992. In the example, we used the *dd.mmyyyy* format for the date numbers; as you might expect, $\text{DATE}+$ is sensitive to the date format flag -42 . The number of days argument can be positive or negative.

You can also determine the number of days between two dates by executing DDAYS with the two date numbers as its arguments. For example, to find the number of days between July 1, 1975 and February 17, 1978:

```
-42 CF 7.011975 2.171978 DDAYS  $\Rightarrow$  962
```

Notice that the earlier date is entered first to return a positive result.

The allowed range of dates for the date arithmetic commands is from October 15, 1582 to December 31, 9999, which spans a range of 3,074,323 days. The latter number is the maximum number-of-days argument permitted by $\text{DATE}+$. A date number argument outside of this range causes the Invalid Date error.

The next two programs provide a combined form of date and time arithmetic. The first, $\text{ADD\&T}$, adds a specified number of days and hours (expressed as a time number with minutes and seconds) to an initial date and time, returning a final date number and time number:

```
1.151991 12.30 32 4.15 ADD\&T  $\Rightarrow$  2.161991 16.45
```

(with flag -42 clear) adds 32 days, 4 hours and 15 minutes to January 15, 1991, 12:30 p.m., returning February 16, 1991 at 4:45 p.m.

The second program, $\text{DD\&T}$, finds the difference in days and hours between two events each specified by a date number and a time number:

```
2.161991 16.45 1.151991 12.30 DD\&T  $\Rightarrow$  -32 -4.15
```


shows that January 15, 1991 is 32 days, 4 hours and 15 minutes earlier than February 16, 1991.

ADD&T <i>Add Date and Time</i>						6063
<i>level 4</i>	<i>level 3</i>	<i>level 2</i>	<i>level 1</i>		<i>level 2</i>	<i>level 1</i>
<i>date₁</i>	<i>time₁</i>	Δ <i>days</i>	Δ <i>time</i>		<i>date₂</i>	<i>date₁</i>
<< SWAP 24 * HMS+ HMS+ DUP 24 / FLOOR ROT OVER DATE+ SWAP 24 * ROT SWAP HMS- >>					Δ time in hours. New time. No. of days in new time. Add to date. Subtract from time.	

DD&T <i>Delta Days and Time</i>						7688
<i>level 4</i>	<i>level 3</i>	<i>level 2</i>	<i>level 1</i>		<i>level 2</i>	<i>level 1</i>
<i>date₁</i>	<i>time₁</i>	<i>date₂</i>	<i>date₁</i>		Δ <i>days</i>	Δ <i>time</i>
<< 4 ROLL ROT DDAYS 24 * SWAP ROT HMS- HMS+ DUP 24 / IP SWAP OVER 24 * - >>					Initial Δ days in hours. Total time difference. Actual Δ days. Remaining Δ time.	

19.2 The Alarm System

An HP48 *alarm* is an event that is scheduled to occur at a specified date and time. There are two types of alarms, according to the type of action associated with the scheduled events:

- An *appointment alarm* sounds an audible signal and displays a message.
- A *control alarm* executes an object, usually a program or the name of a program.

The HP48 can store any number of both kinds of alarms (limited by available memory). The alarms are stored as a list in a global variable in a normally invisible subdirectory of the home directory (see section 5.3.2). The alarm list is not accessible through ordinary variable commands like STO or RCL because the data in the list is represented by uneditable system objects for fast access by the time system. You can view the contents

of the alarm list by activating the alarm catalog (section 19.2.5), or by recalling individual alarms with RCLALARM (section 19.2.4).

When the clock reaches a time that matches an alarm time, we say that the alarm *comes due*. Alarms that have not yet come due are called *pending alarms*; alarms set for times earlier than the current time are called *past alarms*. The calculator also keeps track of *past due* alarms--alarms that have come due but have not been acknowledged (see below).

An alarm is specified by four parameters:

- The *date* of the alarm.
- The *time*.
- The *repeat interval*, for alarms that automatically reschedule themselves at constant intervals. This parameter is zero for single-occurrence alarms.
- The *alarm execution object*. For appointment alarms, this object is a string object; for control alarms, it may be any other type of object, which is executed when the alarm comes due.

19.2.1 Setting Alarms Manually

Setting an alarm consists of storing the four alarm parameters into the alarm list. For manual entry, the best way to do this is to use the *alarm-set menu* ($\leftarrow$ TIME $\equiv$ ALRM $\equiv$):

```

{ HOME }      05/01/92 09:00:53A
Enter alarm, press SET
FRI 05/01/92 12:00:00A

>DATE>TIME A/PM EXEC RPT SET

```

When you first enter the menu, this display looks as above. As you enter alarm parameters, the display in the stack area changes to reflect the new entries (the initial default is the current date at 12 a.m.). For example, entering 9.3 $\equiv$ >TIME $\equiv$ stores 9:30 a.m. as the alarm time:


```

[ HOME ]          05/01/92 09:01:07A
Enter alarm, press SET
FRI 05/01/92 09:30:00A

[>DATE] [>TIME] [A/PM] [EXEC] [RPT] [SET]

```

>DATE, >TIME, and A/PM work the same way as they do in the *time-set* menu (section 19.1.1). RPT activates another menu for entering repeat intervals:

```

Select repeat interval

4:
3:
2:
1:

[WEEK] [DAY] [HOUR] [MIN] [SEC] [NONE]

```

Here you enter a number of time units, using any of the units shown in the menu labels, then press the corresponding menu key. For example, for a weekly meeting alarm you would enter 1 WEEK (7 DAY would work as well). If you change your mind and don't want to enter a repeat interval, or you want to clear a previously entered repeat interval, press NONE. After pressing any of the repeat menu keys, the initial alarm-set menu is restored.

EXEC stores the alarm execution object, which also determines whether the alarm is an appointment alarm or a control alarm. Entering a string object creates an appointment alarm; any other object creates a control alarm. There is no special provision here for entering a string--you must use [] and [α] as usual to enter a string. If you want to modify a string already entered (such as when you are editing an alarm using EDIT in the alarm catalog), [] EXEC recalls the current string to the stack, where you can use [▽] or [←] EDIT to change it.

Of the four alarm parameters, only the date and time are strictly required. If you do not enter a repeat interval, zero is taken as the default, which means a non-repeating alarm. If you don't supply an execution object with EXEC, the alarm will be an

appointment alarm that only displays the date and time when it comes due.

You can enter the alarm parameters in any order, and change the settings as many times as you want. The alarm is not actually stored in the alarm list until you press **≡SET≡**. When you do so, the alarm is set, and the menu is automatically changed to the main time menu.

Between the time when you first enter a parameter in the alarm-set menu and when you press **≡SET≡**, the pending alarm parameters are stored as a list in a global variable **ALRMDAT** created in the current directory. The list has the format used by **STOALARM** (section 19.2.4). Storing the pending alarm data this way during alarm entry allows you to maintain access to other calculator resources--in particular, you can change menus and then return to the alarm-set menu without losing the entry in progress. If the alarm-set menu is activated when **ALRMDAT** already exists, the display will show the parameters that are stored there. When you finally press **≡SET≡**, **ALRMDAT** is automatically purged. (Changing directories during alarm entry forces the start of a new alarm. The **ALRMDAT** in the original directory will remain in place until you either purge it, or return to the directory and complete entering that alarm.)

19.2.1.1 Cancelling Alarm Repeats

An alarm interval that is too small may cause an endless loop when it comes due, because it executes and reschedules too fast for you to get a chance to delete it. Even a system halt (section 5.8) won't help, because that (by design) does not affect the rescheduling of alarms. You can escape from this predicament by pressing **ON** - **4** (together). This disables rescheduling of the next alarm, which allows you to activate the alarm catalog and change the repeat interval for the alarm, or to delete it altogether. Turning the calculator off, then on, re-enables rescheduling in case you used **ON** - **4** unnecessarily.

19.2.2 Appointment Alarms

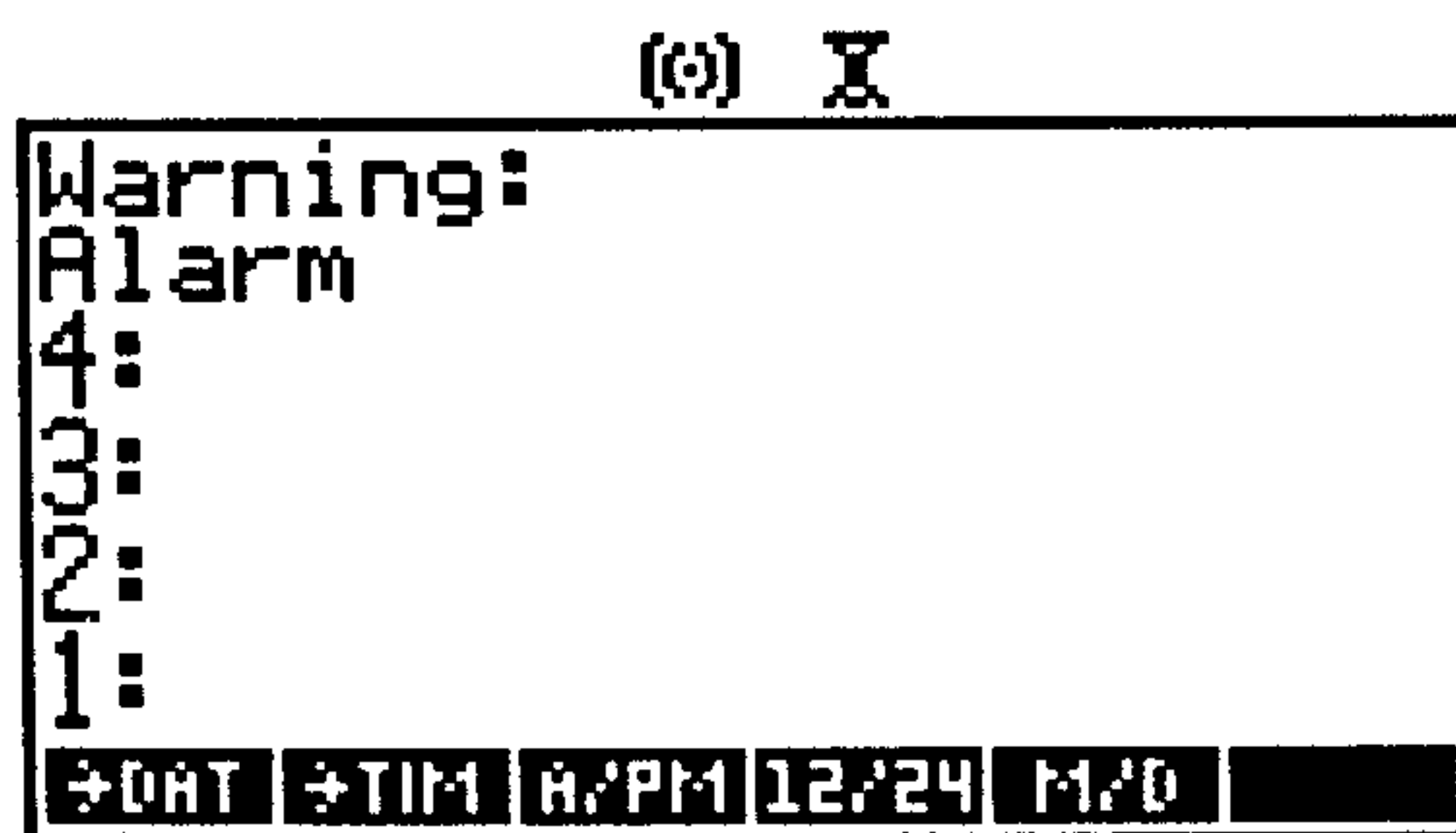
An appointment alarm is an alarm defined with a string stored as its execution object. This type of alarm is intended for use as an appointment reminder: when it comes due, it is *announced*--the HP48 beeps, turns on the busy and the alert annunciators, and displays the day, date, time, and a one-line message. The "beep" in this case is a two-stage audible alarm: first a brief preliminary "chirp," then, after about four seconds, a repeated two-tone "cuckoo" sound. The chirp gives you a chance to shut off the alarm (by pressing any key) before the more annoying cuckoo starts, which is helpful in a meeting or a classroom when you want to minimize the disturbance to other people. You can also prevent the audible alarm entirely by setting flag -57. (This does not affect other kinds of beeps--nor does setting flag -56, which suppresses error beeps and BEEP sounds, affect alarms). Of course, if you disable the audible alarm, you must be

watching the display to see an appointment alarm announced.

When the alarm chirps, its message is displayed in the medium font in the topmost display row. The alarm's day, date, and time are displayed in the second row:



The cuckoo alarm sound repeats up to ten times, lasting until about fifteen seconds after the original chirp. During that time, you may *acknowledge* the alarm by pressing any key. This stops the alarm sound, turns off the busy and alert annunciators, and restores the previous display contents. If you fail to acknowledge the alarm with a key press, the alarm display and sound vanish, but the alert annunciator remains on. There are other causes that activate this annunciator; when you see it, the best thing to do is turn the HP 48 off, then on, whereupon it will display a warning message that explains the alert. With an unacknowledged alarm, you will see:



This warns you to press **TIME** :

(G)

```

{ HOME }          05/01/92 09:30:46A
-----
Past due alarm:
FRI 05/01/92 09:30:00A
Important Meeting

[SET] [ADJUST] [ALARM] [ACK] [ACKA] [CAT]

```

This display shows the oldest unacknowledged alarm. You can then go ahead and acknowledge the alarm by pressing ACK. If there are additional unacknowledged alarms, the next oldest is displayed, which you can then acknowledge, and so on. You can also execute ACKALL (ACKA) to acknowledge all past due alarms at once.

An appointment alarm announcement can occur at any time when the HP48 is ready for key presses, i.e. whenever the busy annunciator is off. This means that an announcement can occur when any built-in environment is active, including the plot environment, the MatrixWriter, and the EquationWriter, but not when a command or a user program is executing. If an alarm comes due when the busy annunciator is on, the alert annunciator is turned on, but the alarm is not otherwise announced audibly or visibly until the current execution is complete.

The alarm-set menu always requires you to enter the date of an alarm as a date number. Sometimes, however, it is more convenient to enter days in terms of days of the week or other intervals, such as "one week from tomorrow" or "the Tuesday after next." The program NEXTDAY listed below provides a means for doing this. You can use it to enter an alarm date to begin an alarm entry; it always exits to the alarm-set menu, where you can complete the entry. You can also execute NEXTDAY after starting an alarm, but you should take care not to change directories and thereby lose track of the data already stored in ALRMDAT. When you execute NEXTDAY, it displays this menu:

```

              HALT
{ HOME }
-----
4:
3:
2:
1:
[NEXT] [END] [TMRW] [TM+1] [WEEK] [MINTH]

```


The first four menu keys add a predetermined number of days to the current alarm date:

- **TMRW** sets the alarm date to the next day--tomorrow.
- **TM+1** sets the alarm date to two days later--the day after tomorrow.
- **WEEK** increments the alarm date by one week.
- **MNTH** increments the alarm date by four weeks.
- **QUIT** (in the second page) exits to the alarm-set menu without changing the current date.

NEXT sets up a new menu:

HALT .						
{ HOME }						
4:						
3:						
2:						
1:						
SUN	MON	TUE	WED	THU	FRI	

Here you press one of the seven menu keys (**SAT** is in the second page) to indicate the day of the week. Thus, to set an alarm for next Wednesday, you execute **NEXTDAY**, then press **NEXT** followed by **WED**.

2ND works the same way as **NEXT** except that it sets the alarm date to the specified day one week later than the next occurrence of that day. That is, to set an alarm for the Thursday after next, execute **NEXTDAY**, then press **2ND** **THU**.

When **NEXTDAY** exits to the alarm-set menu, the current alarm data is not automatically displayed (that only happens when you activate the menu with **ALRM**, and when you press any of the menu keys). The data is displayed when you enter the next alarm parameter, or you can display it immediately by pressing **REVIEW**.

NEXTDAY	Next ...Day Alarm Utility	33B8
<pre> << IF 'ALRMDAT' VTYPE -1 == THEN DATE 0 "" 0 4 ->LIST 'ALRMDAT' STO END 'ALRMDAT' 1 GET -> now << IF { { "NEXT" << 0 1 CONT >> } { "2ND" << 7 1 CONT >> } { "TMRW" << 1 0 CONT >> } { "TM+1" << 2 0 CONT >> } { "WEEK" << 7 0 CONT >> } { "MNTH" << 28 0 CONT >> } { "QUIT" << 0 0 CONT >> } } TMENU HALT THEN -> inc << { { "SUN" << 1 CONT >> } { "MON" << 2 CONT >> } { "TUE" << 3 CONT >> } { "WED" << 4 CONT >> } { "THU" << 5 CONT >> } { "FRI" << 6 CONT >> } { "SAT" << 7 CONT >> } } TMENU HALT now DOW SWAP DROP - IF DUP 1 < THEN 7 + END inc + >> END now SWAP DATE+ 'ALRMDAT' 1 ROT PUT 37.01 TMENU >> >> </pre>	If ALRMDAT doesn't exist, then create default. Get current alarm date. Start of TMENU list. Next ...day. Second ...day. Tomorrow. Day after tomorrow. 1 week from now. 4 weeks from now. No change. Display the menu. Sunday. Monday. Tuesday. Wednesday. Thursday. Friday. Saturday. Show menu for ...day cases. Days from now to alarm. Add one week if necessary. Add 7 for 2nd ...day case. New date. Store the new date. Return to the alarm-set menu.	

19.2.2.1 Unacknowledged Repeating Alarms

When a repeating appointment alarm is not acknowledged by a key press as it comes due, it is simply rescheduled to the next repeat time and does not become past due. If you would prefer to be notified when you have missed such an alarm, you can set flag -43. This prevents repeating appointment alarms from being rescheduled until they are acknowledged, either by a key press or by one of the acknowledgement commands. A past-due repeating alarm then also leaves the alert annunciator on, and issues the alarm warning when the HP 48 is turned on.

19.2.3 Control Alarms

A *control alarm* is characterized by an execution object that is not a string. When a control alarm comes due, no audible or visible signal is given; rather, the alarm index (section 19.2.4) is placed on the stack, and the execution object is executed. Control alarms allow you to schedule calculator operations to take place without any manual intervention. For example, on an HP 48SX with a RAM card in port 1 configured as independent memory, you might schedule an automatic memory archive into a port variable SAVE to occur every night at 2 a.m. This is done with an alarm set for 2 a.m., with a 1 day repeat interval, and the following execution object:

```
<< DROP :1:SAVE DUP PURGE ARCHIVE >>
```

The initial DROP discards the alarm index, which is not needed in this case. In general, a control alarm execution program can use the index to find the alarm itself in the alarm list. In particular, since control alarms are not automatically deleted from the alarm list after execution, a program might begin by executing DELALARM with the index to delete the alarm.

Like appointment alarms, control alarms do not execute when they come due unless the busy annunciator is off and the HP 48 is ready for key presses. But control alarms are more disruptive than appointment alarms, in that they automatically terminate special environments. In the command line, interactive stack, the catalogs, and the plot environment, a control alarm coming due exits from the environment (as if **ATTN** were pressed), then executes the alarm execution object. If the alarm comes due while the EquationWriter or the MatrixWriter is active, the object currently entered in the environment is saved into level 1 before the alarm index. (For the EquationWriter, the object is saved as a string if it is newly entered or has been modified.)

The program CHIMES provides another example of a program useful as an alarm execution object. It "chimes the hour" by sounding a number of tones equal to the current clock time (truncated to hours). An appropriate use is to make CHIMES the execution object for an alarm set to come due on the hour, with a repeat interval of one hour.

The program **SETCHIMES** in the next section does this automatically.

CHIMES	Hour Chimes	E5D1
<pre><< 1 TIME IP 12 MOD IF DUP NOT THEN DROP 12 END START 500 .5 BEEP .3 WAIT NEXT >></pre>		Get the hour. 12 chimes for 12 o'clock. Sound the chimes.

19.2.4 Alarm Commands

The four commands in the second page of the alarm-set menu provide complete programmable control over the HP48 alarm system. To start with, the alarm setting features of the first page of the menu are combined into a single command, **STOALARM**. This command uses a single argument, a list that combines the four alarm parameters:

$$\{ \textit{date} \textit{ time} \textit{ execution-object} \textit{ repeat} \}$$

Here *date* and *time* are real numbers in the date-number and time-number format, respectively. *execution-object* is the alarm execution object, which may be any object--a string indicates an appointment alarm. The final argument *repeat* is an integer real number that specifies the alarm repeat interval, in *ticks* (section 19.1.1). A repeat interval of zero defines a non-repeating alarm. The maximum repeat interval is 999999999999 ticks, about 3.88 years.

When **STOALARM** executes, it adds an alarm to the current alarm list with parameters matching those in its list argument. For example, the following sets a weekly appointment alarm for 9:00 a.m., starting on February 2, 1992:

```
{ 2.121992 9 "Important Meeting" 4954521600 } STOALARM 6
```

The value 4954521600 for the repeat interval is obtained from $1 \text{ week} \times 7 \text{ days/week} \times 24 \text{ hours/day} \times 3600 \text{ seconds/hour} \times 8192 \text{ ticks/second} = 4954521600 \text{ ticks}$.

The number 6 returned here by **STOALARM** is just an example, which will vary according to the current alarm list in your HP48. The number is the *alarm index* of the new alarm. The index is a real number that indicates the position of the alarm in the alarm list, where an index of 1 corresponds to the chronologically earliest alarm (including date and time). You may not have a very frequent need for the index returned by **STOALARM**, but that number is otherwise hard to determine afterwards, and it is easy

to discard if you don't need it.

SETHIMES is an example of the programmed use of STOALARM to create an alarm. The alarm is defined with the hour-chiming program CHIMES from the previous section in its execution object.

SETHIMES	Set Chime Alarm	OB2E
<pre> << DATE TIME IP IF DUP 23 == THEN 0 SWAP 1 DATE+ SWAP ELSE 1 + END << DROP CHIMES >> 29491200 4 ->LIST STOALARM DROP >> </pre>	Today's date. If the next hour is midnight, then set for midnight next day. Otherwise set for next hour. Alarm execution object. Repeat interval one hour (in ticks). Combine alarm parameters into a list. Set the alarm.	

An alarm index is used as an argument by RCLALARM and DELALARM. RCLALARM returns a parameter list for the specified alarm, or returns the Nonexistent Alarm error if there is no such alarm. If you execute it after the STOALARM example above, with the same index, the original list is returned:

```
6 RCLALARM { 2.121992 9 "Important Meeting" 4954521600 }
```

Another example is given by the program DELCHIMES, which uses RCLALARM to find the hour-chime alarm in the alarm list.

DELCHIMES uses DELALARM to delete the hour-chime alarm once it has been found by RCLALARM. In general, DELALARM deletes the alarm specified by its argument, reporting the Nonexistent Alarm error if no such alarm exists. You can also use DELALARM to remove *all* alarms by using zero for the alarm index. The program DELOLDALARMS is more selective: it deletes only alarms set for times earlier than the current time.

DELCHIMES	Delete Chimes Alarm	CBBC
<pre><< RCLF -55 SF 1 → alarm << WHILE alarm IFERR RCLALARM THEN 0 ELSE 1 END REPEAT IF << DROP CHIMES >> POS THEN alarm DELALARM 0 'alarm' STO ELSE 1 'alarm' STO+ END END >> STOF >></pre>	Disable last argument recovery. Start with the oldest alarm Find the next alarm. If none, then quit. If CHIMES is in the parameter list, then delete it, and quit. Otherwise, get next alarm. Restore last arguments state.	

DELOLDALARMS	Delete Old Alarms	5307
<< DATE TIME "" 0 4 →LIST STOALARM 1 SWAP START 1 DELALARM NEXT >>	Set an alarm for "now." New index is number to be deleted. Delete oldest alarm.	

The final alarm command is **FINDALARM**, which finds an alarm by its date and time. In particular, **FINDALARM** returns the index of the first alarm in the alarm list that is set for a time equal to or later than the time specified by the argument. The latter may be either a date number, or a list { *date time* }. A date number used by itself is equivalent to using a list { *date* 0 }--**FINDALARM** returns the index of the earliest alarm set for that date. Executing 0 **FINDALARM** returns the index of the oldest past-due alarm.

19.2.5 The Alarm Catalog

The HP 48 time system includes an alarm catalog, which is similar in operation to the equation catalog (section 14.2.3.2) and the statistics catalog (section 20.1). The alarm

catalog lets you review, edit, copy, and delete the alarms currently stored in the alarm list. It is activated by the CAT key in the TIME menu, or by TIME from any other menu:

```

{ HOME }           05/01/92 09:39:02A
05/01 09:30A Importa...
▶05/07 09:00A Staff m...
05/25 12:00P * DOIT *
07/01 08:00P Ken's b...

PURG  EXEC  EDIT  +STK VIEW

```

The catalog shown here is typical; the display you obtain naturally depends on the alarms currently stored in your HP 48.

When you first activate the catalog, a triangle *selection pointer* points to the next alarm due--the same one that is displayed in the TIME menu. The alarms are displayed in chronological order, with past alarms above the pointer, and pending alarms below. As in the equation and statistics catalogs, you can move the selection up and down using Δ or ▽--one alarm at a time with the unshifted keys, one "page" of five alarms with Δ and ▽, and to the start and end of the catalog with Δ and ▽. The selected alarm is the subject of the menu operations, with the exception of EXEC. The latter toggles the catalog display between the initial date/time/execution object display as shown above, and a display that shows only the execution objects:

```

{ HOME }           05/01/92 09:39:31A
Important Meeting
▶Staff meeting rm. 72
* DOIT *
Ken's birthday party

PURG  EXEC  EDIT  +STK VIEW

```

The remaining menu operations work as follows:

- PURG deletes the selected alarm from the alarm list.

- **EDIT** also deletes the selected alarm, but copies its parameters to ALRMDAT in the current directory, and activates the alarm-set menu (exiting from the catalog). This allows you to change any of the alarm parameters, then reset the alarm by pressing **SET**. It is important to remember that the alarm is actually deleted by **EDIT**, because even if you decide not to change the alarm, you must still press **SET** to reactivate the alarm.
- **-STK** copies the alarm parameters to the stack in a list, in the format used by STOALARM and RCLALARM (section 19.2.4). The message Copied to Stack is briefly displayed in the top row.
- VIEW displays all of an alarm's parameters, using the same display that you see in the alarm-set menu. The expanded display persists for about two seconds, or as long as you hold the menu key down.

The only keys other than the menu keys that are active in the catalog environment are **ENTER**, which duplicates the action of **-STK**, and **ATTN**, which terminates the catalog.

19.2.6 Automatic Alarm Deletion

By default, both appointment and control alarms are automatically deleted from the alarm catalog when they are acknowledged. For appointment alarms, acknowledgement is accomplished by a key press as the alarm is announced (section 19.2.2), or for past-due alarms, by ACK or ACKALL. For a control alarm, "acknowledgement" is just the execution of the alarm execution object (section 19.2.3). Alarm deletion conserves memory, and helps prevent execution delays associated with long alarm lists.

You may prefer, however, to prevent this automatic deletion by setting flag -44. The old alarms in the alarm list are a record of past alarm events that you may find useful. Furthermore:

- When an appointment alarm is announced, pressing a key to acknowledge it also clears the alarm display. If the alarm is deleted, you can't even use the alarm catalog to review the alarm message afterwards.
- For irregularly scheduled events, it is convenient to keep one copy of the alarm in the alarm list. Then to schedule the next alarm, you can use the alarm catalog **EDIT** to modify one or more of the parameters.
- Old control alarms may contain programs that have other uses (although it is better for sake of time system efficiency to store long programs in variables, and use the variables' names as the alarm execution objects instead of the programs themselves).

20. Statistics

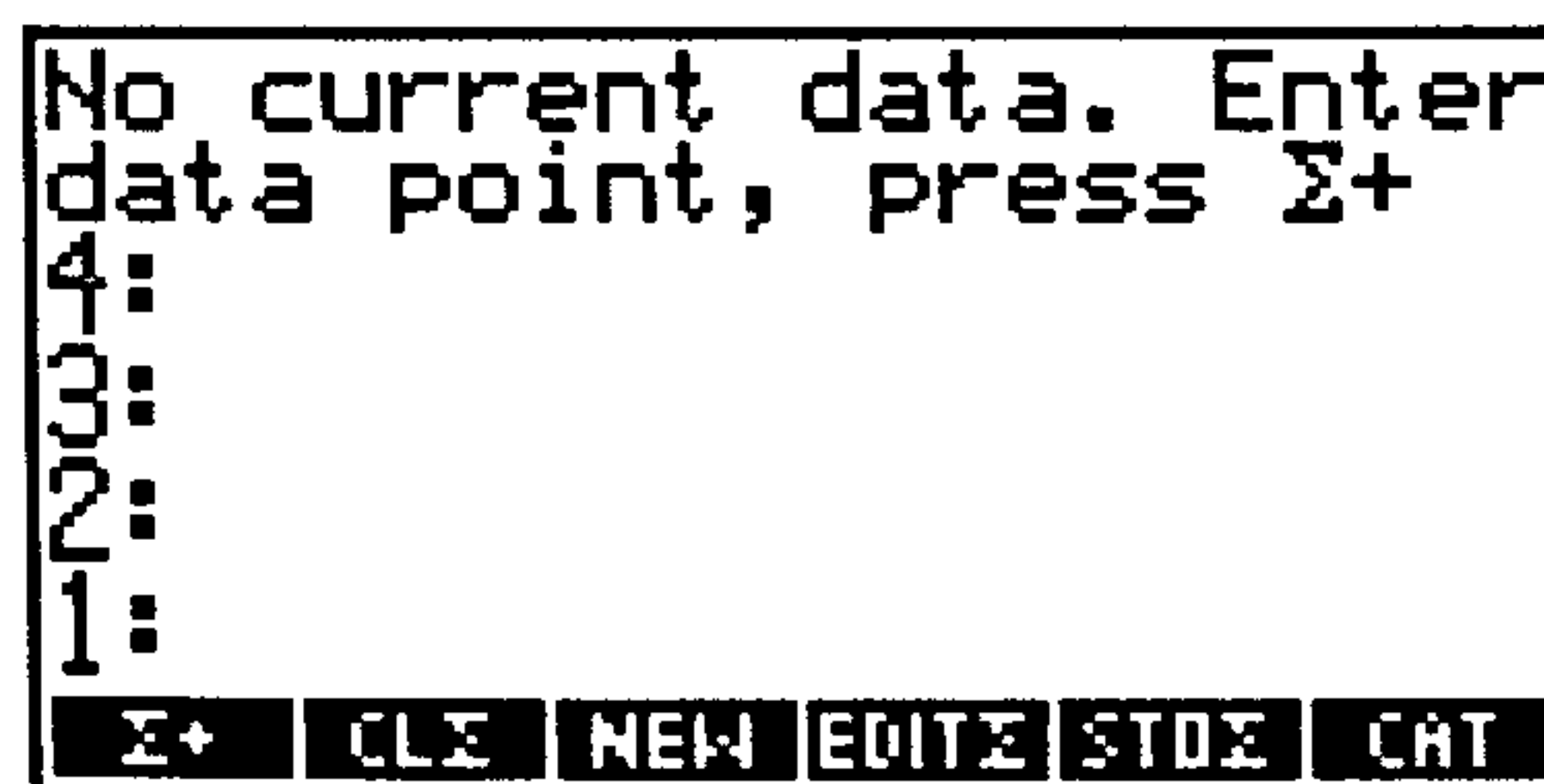
In the realm of statistical computation, the HP 48 provides a capable but not exhaustive set of commands that deal with data entry, simple sample statistics, linear and pseudo-linear regressions, probability, and plotting. The last topic is covered in sections 15.9-15.11; in this chapter we will concentrate on non-graphical analysis tools.

20.1 Data Entry

The traditional HP calculator method of accumulating sample data is the $\Sigma+$ command. In its original versions, $\Sigma+$ takes a number from the stack and adds its value and its squared value to running totals of these values from a series of such entries. $\Sigma+$ also increments a count of the entries. From the three stored values, the total, mean, and standard deviation of the entered data can be computed. Some calculators, such as the HP41, enhance the action of $\Sigma+$ to accumulate pairs of numbers, allowing calculation of cross-correlations and regressions along with the other statistics.

The HP 48 follows the general $\Sigma+$ model of its predecessors, but generalizes the command to work with data points in any number of dimensions. Furthermore, $\Sigma+$ saves the entered data rather than just the summary statistics, which allows for data editing, data transformation, and greater accuracy. The data is saved as an $n \times m$ matrix, each row of which represents the coordinates of a data point in m -dimensional space. The number of rows n indicates the number of points that have been entered. Sample statistics are computed and linear regressions are performed on the columns of the data matrix.

The statistics analysis commands are collected in the statistics menu, activated by  **STAT**. Activating this menu when no data has been stored produces this display:



Notice the deliberate similarity to the plot and HP Solve entry menus. The **NEW** ,

$\overline{\text{EDIT}\Sigma}$, and $\overline{\text{STO}\Sigma}$ keys play the same roles in the statistics menu as $\overline{\text{NEW}}$, $\overline{\text{EDEQ}}$ and $\overline{\text{STEQ}}$ in the plot and solve menus, except that the reserved name variable for statistics is named ΣDAT , and $\overline{\text{EDIT}\Sigma}$ activates the matrix writer for editing. ΣDAT is called the *current statistics matrix*, by analogy with current equation EQ (section 14.2.3). Like EQ, ΣDAT may contain either the object of interest, in this case a matrix, or an indirect reference to the object--the name of another variable containing a matrix. Statistical operations are always directed to the current statistics matrix in the current directory. You can select one of several matrices for these operations by storing its name in ΣDAT , with either STO or $\text{STO}\Sigma$ (which is a shortcut for ' ΣDAT ' STO).

The statistics catalog ($\overline{\text{CAT}}$) works generally like the equation catalog (section 14.2.3.2), except that it displays variables containing matrices. The first three menu keys also differ from those in the equation catalog. Each stores the selected variable name in ΣDAT , and:

- $\overline{\text{1-VAR}}$ activates the second page of the statistics menu, for commands related to single-variable statistics.
- $\overline{\text{PLOT}}$ activates the third page, for commands related to the discrete plot types.
- $\overline{\text{2-VAR}}$ activates the fourth page, for two-variable statistics commands.

The basic action of $\Sigma+$ is to take a real m -element vector from the stack, and append it as a new last row to an $n \times m$ matrix stored or named in ΣDAT . If the designated variable does not exist, it is created by $\Sigma+$, by storing the vector there (as a $1 \times m$ matrix). For 1-dimensional data, you can use $\Sigma+$ either with a single real number or with a one-element vector. $\text{CL}\Sigma$ purges ΣDAT , to prepare for entry of a new set of data.

Once the matrix is established, subsequent entries with $\Sigma+$ can either be vectors (of the right length), or separate real numbers taken from the lowest stack levels. For example, consider the 3-dimensional data points

(11,12,20)
(12,23,23)
(15,27,24)
(14,31,25)

To start entering this data, press $\overline{\text{CL}\Sigma}$, then

[11 12 20] $\overline{\Sigma+}$.

The display shows this entry as $\Sigma\text{DAT}(1)$:

```

ΣDAT(1)=[ 11 12 20 ]
ΣDAT(2)=
4:
3:
2:
1:
Σ+  CLΣ  NEW  EDITΣ  STDS  CAT

```

The incomplete $\Sigma\text{DAT}(2)=$ indicates that the calculator is ready for entry of a second data point. Now that ΣDAT has been established, entering a data point as a vector is no longer necessary. You can enter the coordinates as separate stack entries (maintaining the same order):

12 23 23 $\Sigma+$

15 27 24 $\Sigma+$

14 31 25 $\Sigma+$.

Note that $\boxed{\text{r}}\boxed{\text{D}} \boxed{\text{LASTARG}}$ always returns a vector, even when you have entered separate numbers:

$\boxed{\text{r}}\boxed{\text{D}} \boxed{\text{LASTARG}} \quad \boxed{\text{v}} \quad [14 \ 31 \ 25]$

$\Sigma+$ will also enter several data points together if they are combined into a matrix the same width as the current statistics matrix. For correction of a bad entry, $\Sigma-$ ($\boxed{\text{r}}\boxed{\text{D}} \boxed{\text{Σ+}}$) reverses the action of $\Sigma+$, stripping the statistics matrix of its last row, and returning it to the stack as a vector:

$\Sigma- \quad \boxed{\text{v}} \quad [14 \ 31 \ 25]$

Repeated execution of $\Sigma-$ returns each successive last row, allowing you to edit one or more of them. You can then restore the entries by executing $\Sigma+$ the same number of times.

Although $\Sigma+$ is convenient for entering small data sets, using $\Sigma+$ for each data point may not be the best method for larger sets. When you are using a program to enter data automatically, it is more efficient to combine all of the data into a matrix, then use $\Sigma+$ or $\text{STO}\Sigma$ to store it (one large movement of memory contents is more efficient than several smaller ones). For manual entry, you can use the MatrixWriter (section 6.6) to enter all of the data, taking advantage of the MatrixWriter's facilities for reviewing and correcting the data before storing it. After exiting the MatrixWriter via **ENTER**, you can

- Execute $\Sigma+$ to append the new data to the current statistics matrix.
- Execute $\text{STO}\Sigma$ to replace (or create) the current statistics matrix.
- Use STO to store the matrix, then store its name in ΣDAT .

The last option is combined into a single operation by the statistics menu operation **NEW**, which takes a matrix from the stack and prompts you for a name:

α

RAD	PRG
[HOME]	
Name the stat data, press ENTER	

When you press **ENTER**, the matrix is stored with the designated name, and the name is stored in ΣDAT .

20.2 One-Variable Sample Statistics

The second page of the statistics menu (which is directly accessible with **STAT**) contains five commands for computing simple one-variable sample statistics. (The sixth menu entry, BINS is associated with histogram plotting, and is described in section 15.11). For these commands, the columns in ΣDAT are treated as samples of independent variables, and the statistics of each sample are computed separately. For example, with the data set of the preceding section, we find

MEAN $\rightarrow$ [13 23.5 23]

Each element of the result vector is the average of the numbers in the corresponding

column of ΣDAT . The result object and those of the other commands in this section are single real numbers when ΣDAT contains only one column.

Similarly, you can use **SDEV** to compute the sample standard deviations of the data:

SDEV  [1.82574185835 8.18026079454 2.16024689947]

This standard deviation is defined by the formula

$$\sigma_j = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_{ij} - \bar{x}_j)^2}$$

where σ_j is the standard deviation for the j th column of data, x_{ij} is the i th entry in that column, and $\bar{x}_j$ is the column mean. This definition constitutes an unbiased estimate of the standard deviation of the population from which the data sample was drawn. This differs from the standard deviation of the data itself. The latter is defined by the above formula with n in the denominator rather than $n-1$, making it smaller than the sample standard deviation by a factor of $\sqrt{n/(n-1)}$.

The HP48 also provides the command **VAR** for computing sample variances, which are the squares of sample standard deviations. This command is not found in any menu, but you can execute it by name or via a custom menu.

MAX Σ and **MIN Σ** compute the most positive and most negative values of the data in the separate columns:

MAX Σ  [15 31 25]

MIN Σ  [11 12 20]

The midpoints of the data ranges are easily computed using these two commands:

MAX Σ MIN Σ + 2 /  [13 21.5 22.5].

Because ΣDAT contains the actual entered data rather than summary statistics, it is straightforward to compute statistics other than those explicitly provided in the built-in command set. For example, the following program uses the sorting program **SORT** listed in section 11.5.3 to compute the medians of the data:

MEDIAN	Medians of Σ DAT	E736
		level 1
	☐	median
	☐	[median _i]
<pre><< Σ DAT TRN OBJ→ OBJ→ DROP DUP 2 / DUP FP NOT {} → m n n2 even med << 1 m FOR i n →LIST SORT n2 IF even THEN DUP 1 + SUB OBJ→ DROP + 2 / ELSE GET END 'med' STO+ NEXT med IF 'm>1' THEN OBJ→ →ARRY ELSE 1 GET END >> >></pre>		Transpose the stat matrix, and take it apart. Find the middle. Initialize output list. Create local variables. Repeat for each column: Sort the entries in a column. If n is even, Then average the middle two entries. Otherwise, get the middle entry. Prepend to the output list. If more than one column of data, then convert result to a vector.

20.3 Two-Variable Statistics

The fourth page of the STAT menu contains commands related to *two-variable statistics*, which deal with the interrelationships between pairs of columns of data in Σ DAT. In many applications, the values in one column are considered to be samples of an *independent variable*, which may actually be a noise-free parameter rather than a random variable. The second column then represents a *dependent variable*, the values of which are measured or computed from the first variable. Alternatively, both columns may be treated as independent, where you are interested in the correlations between the pairs of data points.

In either case, you must designate which columns in Σ DAT represent the independent and dependent variables. This is accomplished by means of XCOL and YCOL, respectively (see also section 15.9), the names of which follow the HP 48 plotting convention of

associating the name x or X with the independent variable, and y or Y with the dependent variable. Each command takes a single real number, nominally a positive integer, and stores it as the independent or dependent column choice. The default choices are 1 for x , and 2 for y . These commands' menu keys are in the third page of the statistics menu. When that menu or the fourth page menu is activated, the status area shows the current choices:

```

Xcol:1 Ycol:2 Modl:LIN
-----
4:
3:
2:
1:
XCOL YCOL BARPL HISTP SCATRLINE

```

The Modl: label at the right end indicates the current regression model; see section 20.3.2.

The current two-variable column selections are stored as real numbers in a list stored in the reserved-name variable Σ PAR. The list, which is analogous to PPAR (section 15.2.3), actually contains five entries:

{ *independent dependent intercept slope model* }

These objects specify, in order, the independent column number, the dependent column number, the regression intercept and slope coefficients (or their analogs depending on the regression model) from the most recent regression, and the name of the regression model. The first four objects are real numbers; the *model* object is the command that actually sets the model: LINFIT EXPFIT, PWRFIT, or LOGFIT.

If Σ PAR does not exist in the current directory when a command that uses it is executed, that command automatically creates Σ PAR with the default values

{ 1 2 0 0 LINFIT }.

If the command is one that stores a value in Σ PAR, then that value is immediately substituted in the new list.

20.3.1 Correlations

The *sample covariance* of two sets of data is defined as

$$\text{cov}(x,y) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

where the x_i and y_i are the corresponding data points in the two sets, and $\bar{x}$ and $\bar{y}$ are their averages. The covariance is a measure of the mutual dependence of x and y . It is positive if values $x_i > \bar{x}$ tend to appear with values $y_i > \bar{y}$ and $x_i < \bar{x}$ with $y_i < \bar{y}$, and negative if $x_i < \bar{x}$ appears with $y_i > \bar{y}$ and $x_i > \bar{x}$ appears with $y_i < \bar{y}$. A covariance of zero implies that the fluctuations of x and y around their averages are independent of each other.

The *correlation coefficient* $\text{corr}(x,y)$ is a normalized version of the covariance:

$$\text{corr}(x,y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}}$$

The coefficient has values of +1 for perfectly correlated variables, -1 for perfectly anticorrelated variables, and 0 for totally independent variables.

COV and CORR compute the covariance and correlation coefficient of the data in the two columns of ΣDAT specified in ΣPAR . For example, consider two sets of 100 uncorrelated data points, created using RAND as follows:

```
.12345 RDZ 1 200 START RAND NEXT { 100 2 } →ARRY STOΣ.
```

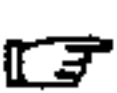
CORR should return a near-zero correlation coefficient:

```
1 XCOL 2 YCOL LINFIT CORR 4.85327453722E-2.
```

On the other hand, if one data set is derived from another by introducing a small amount of noise:

```
.12345 RDZ 1 100 START RAND DUP RAND .1 * + NEXT  
{ 100 2 } →ARRY STOΣ,
```

then CORR should return a coefficient close to one:

CORR  .9961417813403.

LINFIT is included in the sequence above because COV and CORR compute their results based on the Σ DAT data transformed according to the current regression model. See section 20.3.2.1.

20.3.2 Regressions

A *linear regression* is the computation of a linear relationship between two sets of data. If x and y denote the independent (XCOL) and dependent (YCOL) variables, then the linear relationship is $y = ax + b$, and is referred to the *regression of y on x* . In a graph of the line, a is *slope*, and b is the *y-axis intercept*. The regression used in the HP 48 is a *least-squares fit*, where values of a and b are computed such that the sum of the squared differences

$$\sum_{i=1}^n (y_i - ax_i - b)^2$$

is minimized. x_i and y_i are the individual measurements of the variables, and n is the number of pairs of measurements. In effect, the following simultaneous equations are solved:

$$\begin{aligned} an + b \sum_{i=1}^n x_i &= \sum_{i=1}^n y_i \\ a \sum_{i=1}^n x_i + b \sum_{i=1}^n x_i^2 &= \sum_{i=1}^n x_i y_i \end{aligned}$$

To obtain this type of regression, you can execute LINFIT to select the regression model (see the next section), then LR (*Linear Regression*) to compute the coefficients b and a . The latter are returned to the stack as real numbers, with the tags Intercept: and Slope: to identify them. They are also stored as the third and fourth elements in the Σ PAR list.

To illustrate ordinary linear regression, we will first create some noisy “data:”

```
.12345 RDZ 1 100 FOR x x DUP .5 * 1 + RAND + NEXT
{ 100 2 } ->ARRAY STOΣ
```

From this sequence, we expect that a regression will produce values near 0.5 for a and 1.5 for b (1 from the additive 1 plus .5 for the average value returned by RAND). In

fact, we obtain

```
1 XCOL 2 YCOL LINFIT LR
```

```
⌞ Intercept: 1.51222496525 Slope: .500206997492
```

In this example, we are using uniformly distributed noise, which is easy to generate using RAND (section 20.5.2). In the next section, we give an example of a regression applied to normally distributed pseudorandom numbers.

Once you have computed the regression coefficients, you can use them to compute estimates of y for values of x that are not in the data set. This is achieved by PREDY (*PREDicted Y*), which takes an x value from the stack and returns a y value computed from the coefficients stored in Σ PAR. For example, using the data from the current example,

```
200 PREDY ⌞ 101.553624464
```

The HP 48 also provides PREDX, which computes a value of x given a value of y :

```
60 PREDX ⌞ 116.927142819
```

Keep in mind, however, that x is the independent variable and may not even be continuously variable, so that PREDX's result may not be entirely meaningful. Note also that PREDX computes $x = (y - b)/a$ using the coefficients computed from the regression of y on x . This is different in principle from the result that can be obtained by interchanging the roles of x and y and recomputing the regression coefficient.

20.3.2.1 Pseudo-Linear Regressions

The HP 48 stretches the idea of least-squares analysis a little by providing additional regression models in which the data is converted into a *pseudo-linear* form by logarithmic transformations. Since this also distorts the noise associated with the variables, the use of a least-squares approach loses some validity since the simple linear method assumes at least that the noise has a constant variance. However, the accuracy of the resulting fits is likely to be more than sufficient for simple modeling of many physical or other systems.

The linear model and three pseudo-linear models, and the commands that activate each model, are as follows:

Model	Relationship	Command
Linear	$y = ax + b$	LINFIT
Logarithmic	$y = a \ln x + b$	LOGFIT
Exponential	$y = b \exp(ax)$	BESTFIT
Power	$y = bx^a$	PWRFIT

The four model selection commands (plus the automatic model selector BESTFIT) are found in a sub-menu activated by the **≡MODL≡** key in the fourth page of the statistics menu. Each of the four commands, like their plot type counterparts (section 15.2.8), executes by storing itself as the fifth element in the Σ PAR list. LR checks that element to determine which type of transformation to perform prior to calculating a linear regression.

The logarithmic, exponential, and power model regressions are calculated by applying logarithms to the data to transform the model to a linear one, performing a linear regression on the transformed data, then inverse-transforming the computed parameters to the original model form. For example, for the logarithmic model, the data is presumed to have the form $y = a \ln x + b$. If we make the substitution $x' = \ln x$, then the model becomes $y = ax' + b$, which is the normal linear form. Similarly, the exponential and power forms can be transformed to linear by substituting for x , y , and the linear coefficient a . The actual substitutions for the four models are shown here:

Model	a'	x'	y'
Linear	a	x	y
Exponential	$\ln a$	x	$\ln y$
Logarithmic	a	$\ln x$	y
Power	$\ln a$	$\ln x$	$\ln y$

To illustrate a pseudo-linear regression, the following sequence creates data corresponding to a noisy logarithmic curve with $a=5$ and $b=1$, where b is actually a normally distributed random variable with mean 1 and standard deviation 1. The independent variable values x_i are the integers 1 through 100.

.54321 RDZ	Random number seed.
1 100	x from 1 to 100
FOR x x	x_i
1 1 MNORM x LN 5 * +	y_i
NEXT	
{ 100 2 } →ARRY 'LOGDAT' STO	Store the data.

(MNORM is listed in section 12.11.1.2.) The data is stored in the variable LOGDAT. Then,

'LOGDAT' STOΣ LOGFIT LR  Intercept: 1.5848798893 Slope: 4.90171452.

This data and the computed regression line are plotted in section 15.9.1.

■ *Example.* In a high-school chemistry class experiment, the students measure the decay of Ba^{137} using a geiger counter. At 100 second intervals, they record the following count rates, after subtracting the background rate:

Interval	Rate
1	1611
2	1007
3	657
4	510
5	306
6	198
7	113
8	90
9	44
10	29

Find the half-life of the isotope from this data.

■ *Solution.* Radioactive substances exhibit an exponential decay in their radioactivity

$$R(t) = R(0)e^{-\ln 2 \frac{t}{\lambda}}$$

where λ is the half life. This formula matches the HP 48 exponential fit model, where $a = -\ln 2/\lambda$ and $b = R(0)$. Entering the data in the above table into ΣDAT, we find

3 FIX EXPFIT LR  Intercept: 2626.418 Slope: -0.442

The half-life is derived from the slope:

$$2 \text{ LN NEG SWAP } / \quad \boxed{1.567}$$

This result is in units of 100 seconds, so that the calculated half-life is actually 156.7 seconds.

20.3.3 Best Fit

BESTFIT is a variation of LR which performs a regression for each of the four models, and chooses the best fit from among them. The coefficients from that best fit, and the name of the corresponding model selection command are stored in ΣPAR , but no objects are returned to the stack.

“Best” in this context refers to the model for which linear regression returns the largest correlation coefficient. For the first example in the previous section, separate execution of LR for each of the four models yields the following correlations:

Model	Correlation Coefficient
Linear	0.871
Logarithmic	0.976
Exponential	0.636
Power	0.863

From these results it is evident that the logarithmic model provides the best fit, and indeed BESTFIT returns LOGFIT and the coefficients derived from that model to ΣPAR .

A purist might question the value of a “best fit,” since the use of curve fitting at all implies that you already know the nature of the curve you are fitting to the data. The regression process should be to determine the unknown parameters of the model, not the model itself. Despite these reservations, finding a best-fit curve can be quite useful, especially when you are attempting to find an analytic expression to approximate data that isn’t derived from any simple theoretical model, like the price of a security or a sales forecast.

20.3.4 Scatter Plots with Error Bars

An ordinary scatter plot (section 15.9) plots each point as a single dark pixel. It is common in experimental measurement problems to represent the error in a given measurement (x_i, y_i) by drawing a vertical *error bar* through its plotted point, from $y_i + \sigma_i$ to $y_i - \sigma_i$, where σ_i is the standard deviation of the i th measurement. In the next section,

where we discuss weighted measurements, such plots can help to verify the results of a least-squares fit. But even in unweighted cases, you may find it useful to plot with small error bars to make the plotted points stand out against a fitted curve.

The program **DRAWEBAR** listed below is a variation of **DRAW** that makes a scatter plot with error bars. Its arguments are a measurement vector y (level 3), an independent variable vector or $n \times 1$ matrix x , and an object that specifies the half-height of the bars. The latter object can be a real number σ , or a real vector comprised of the different σ_i 's for each point.

For example, for the barium decay problem in the previous section, the measurement error should be equal to the square root of each count, since radioactive decay rates follow a Poisson distribution. To create a plot that reflects these errors, execute the following:

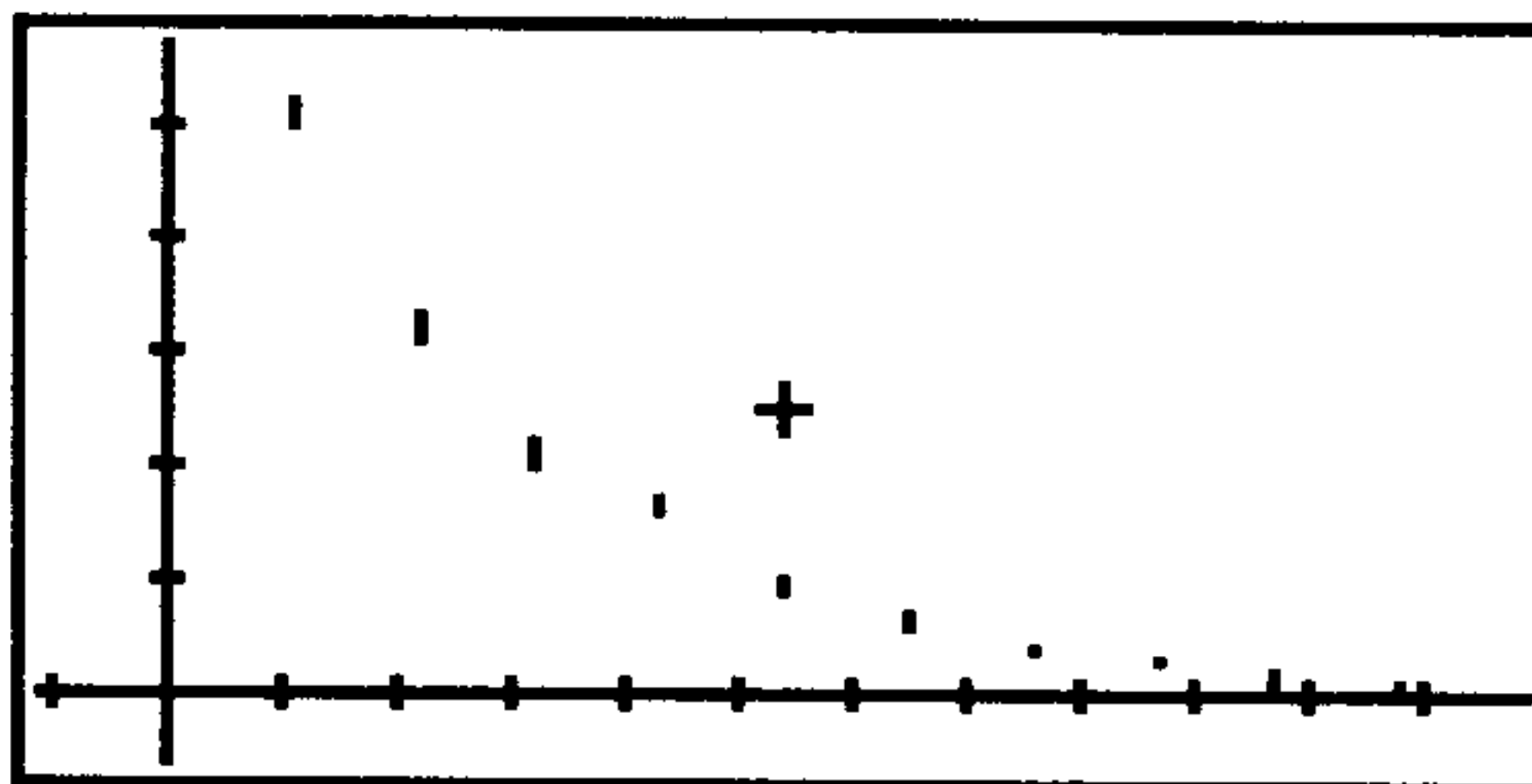
```
RCLΣ CLSPLIT OVER OBJ→
      DROP 1 10 START √ 10 ROLL NEXT 10 →ARRY
```

This creates a vector from the standard deviations. (CLSPLIT is listed in section 20.4.1.)

Then

```
-1 11 XRNG -200 1800 YRNG DRAWEBAR
```

makes this plot:



These bars are relatively small, so you would expect a good fit of a logarithmic curve to the data.

DRAWEBAR	Draw with Error Bars				F544
	level 3	level 2	level 1		
	y	x	σ	σ	
<pre> << 3 ROLLD DUP SIZE 1 1 SUB RDM SWAP (0,1) * + DUP SIZE 1 GET ROT IF DUP TYPE NOT THEN OVER 1 -LIST SWAP CON END (0,1) * -> x n σ << { # 0h # 0h } PVIEW DRAX 1 n FOR i x i GET σ i GET DUP2 - 3 ROLLD + LINE NEXT >> >> </pre>					Make sure x is a vector. Combine x and y into a complex vector. Get number of points. If σ is a number, make a constant vector. Make σ complex. Save data. Watch the action. Next point. Corresponding σ. Connect $(x, y + \sigma)$ and $(x, y - \sigma)$.

20.3.5 Summary Statistics

The last page of the STAT menu contains commands that return one- and two-variable summary statistics from Σ DAT. Each of these commands returns a real number representing a sum:

Command	Sum
ΣX	$\sum x_i$
ΣY	$\sum y_i$
ΣX^2	$\sum x_i^2$
ΣY^2	$\sum y_i^2$
$\Sigma X*Y$	$\sum x_i y_i$
$N\Sigma$	n

where the x_i are the elements in the independent column, y_i are the elements in the dependent column, and n is the number of rows in Σ DAT.

When Σ DAT contains $m > 2$ columns, you can obtain all of the summed squares and

cross products by premultiplying the data matrix by its own transpose. You can also obtain the summed columns as well by grafting a column of 1's to either side of the matrix:

Σ DAT CLADD1 DUP TRN SWAP *

(CLADD1 is defined in section 20.4.1.) The resulting $m+1 \times m+1$ square matrix c has the following elements, where the x_{ij} are the elements of Σ DAT:

<i>Element</i>	<i>Value</i>
$c_{ij} \ (i, j \leq m)$	$\sum_{k=1}^n x_{ki} x_{kj}$
$c_{m+1, j} \ (j \leq m)$	$\sum_{k=1}^n x_{kj}$
$c_{j, m+1} \ (j \leq m)$	$\sum_{k=1}^n x_{kj}$
$c_{m+1, m+1}$	n

The sums obtained this way are not as accurate as those obtained with ΣX , etc., since the built-in commands accumulate the sums and products using 15-digit internal precision.

20.4 General Least-Squares Fitting

The linear regressions described in the preceding sections are simple cases of the application of least-squares fitting. Given the HP 48's powerful array-handling capabilities, it is straightforward to program the calculator to handle a wider variety of curve fitting problems than is provided by the LR command.

The principles of the least-squares method summarized below are presented without proof. You are referred to any standard statistics or data analysis textbook for a more complete discussion. One such source, which has a strong emphasis on numerical analysis, is *Applied Numerical Methods*, by Carnahan, Luther, and Wilkes (John Wiley & Sons, 1969).

The basic purpose of any curve fitting is to determine the values of parameters that can't be measured directly, from measurable quantities that are functions of those parameters. Initially, we will consider the measurement of a variable ϕ that is a linear function of m independent variables x_j :

$$\phi = \sum_{j=1}^m x_j a_j$$

where the coefficients a_j are the parameters that we are to determine. Assume that we have a set of n measurements y_i , taken for various specific values x_{ij} of the independent variables. The differences $y_i - \phi(x_{ij})$ are assumed to be random variables, with standard deviations σ_i . We then want to compute values of the a_j such that the sum of the weighted squared errors is minimized:

$$E = \sum_{i=1}^n \frac{(y_i - \sum_{j=1}^m x_{ij} a_j)^2}{\sigma_i^2}.$$

The notation is simplified if we represent the y_i as an n -element column vector $\mathbf{y}$, the a_j as an m -element column vector $\mathbf{a}$, and the x_{ij} as an $n \times m$ matrix $\mathbf{X}$. The weights $1/\sigma_i^2$ can be combined into a $n \times n$ diagonal matrix $\mathbf{W}$, with $W_{ii} = 1/\sigma_i^2$. Then the least-square error computation is to minimize

$$E = (\mathbf{y} - \mathbf{X}\mathbf{a})^T \mathbf{W} (\mathbf{y} - \mathbf{X}\mathbf{a})$$

To find the minimum, we differentiate with respect to each of the a_j , and set the results equal to zero. The resulting set of simultaneous equations is

$$2\mathbf{X}^T \mathbf{W} (\mathbf{y} - \mathbf{X}\mathbf{a}) = 0.$$

Solving for $\mathbf{a}$,

$$\mathbf{a} = (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{W} \mathbf{y}$$

Notice that if the σ_i are all the same (such as is assumed in the LR linear regressions), $\mathbf{W}$ is a multiple of the identity matrix and so cancels out of this equation. The standard deviations of the parameters $\mathbf{a}$ are the diagonal elements of the covariance matrix

$$\mathbf{V} = (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1}.$$

If $\sigma_i = \sigma = \text{constant}$, then

$$\mathbf{V} = (\mathbf{X}^T \mathbf{X})^{-1} \sigma^2.$$

Moreover, if the σ_i are not known a priori, but are assumed to be uniform, then that standard deviation can be estimated from

$$\sigma^2 = \frac{\mathbf{y}^T \mathbf{y} - \mathbf{y}^T \mathbf{X} \mathbf{a}}{n - m}$$

which is the averaged squared difference between y_i and $(\mathbf{X}\mathbf{a})_i$.

The program LSFIT listed next embodies the principles outlined here:

LSFIT			Least-Squares Fit			B3B2	
level 3	level 2	level 1		level 3	level 2	level 1	
y	X	W		a	V	σ	
<pre> << ROT DUP SIZE 1 + 1 2 SUB RDM ROT DUP SIZE 1 + 1 2 SUB RDM DUP SIZE OBJ→ → w y x n m w? << CASE w 0 SAME THEN 1 'w' STO 0 'w?' STO END w TYPE THEN w SIZE SIZE IF 1 SAME THEN n IDN 1 n FOR i i i 2 -LIST w i GET SQ INV PUT NEXT 'w' STO END END w SQ INV 'w' STO END x TRN w * x * INV DUP x TRN * w * y * DUP m 1 -LIST RDM 3 ROLLD y x ROT * - DUP TRN SWAP * 1 GET n m - / IF w? NOT THEN SWAP OVER * SWAP END ✓ >> >> </pre>				Convert y to a $n \times 1$ array. Likewise for x. w? <i>true</i> means weights are supplied. Unweighted case. If w is not a real number, then if it's a vector of σ's remake into a diagonal matrix: σ_i $1/\sigma_i^2$ w is a number; replace with $1/\sigma_i^2$. Covariance matrix V. Parameter vector a. Convert a to a vector. Estimate σ^2 If unweighted, then multiply V by σ^2 Return σ.			

LSFIT takes the following arguments:

- 3: y Vector or $n \times 1$ matrix
- 2: X Vector or matrix
- 1: W Real number, vector, or square matrix

For cases of uniformly weighted measurements, W is entered as a real number σ that is the standard deviation of each measurement. For non-uniform weights, W may be

entered as a vector of σ_i 's or as a square diagonal matrix where $W_{ii} = 1/\sigma_i^2$. Entering 0 for W indicates unweighted measurements.

LSFIT returns the following:

3: **a** Vector of computed parameters
 2: **V** Covariance matrix
 1: **σ** Standard deviation

The standard deviations of the parameters a_i are given by the square roots of the diagonal elements of the covariance matrix; σ is the average deviation of each y_i from the fitted curve.

■ *Example.* Find the parameters a and b that provide the best fit of the following data to the function $ax^2 + b$:

x	y
5	12.25
4	7.35
3	2.59
2	-.28
1	-1.75
0	-2.88
-1	-1.64
-2	.14
-3	2.42
-4	7.24
-5	12.55

■ *Solution.* In this problem, the independent variables corresponding to the unknown coefficients are x^2 and the constant 1. First, enter the vector y :

[12.25 7.35 2.59 -.28 -1.75 -2.88 -1.64 .14 2.42 7.24 12.55].

Next, enter the matrix X :

[[25 1][16 1][9 1][4 1][1 1][0 1]
 [1 1][4 1][9 1][15 1][25 1]]

Finally, enter the weight matrix, in this case a vector containing the σ_i :

[.1 .2 .2 .3 .3 .4 .4 .4 .3 .3 .2]

Then

2 FIX LSFIT 

[0.60 -2.54] [[5.12E-5 -8.85E-4] [-8.85E-4 0.02]] 0.30

The first result is the coefficient vector, corresponding to $a=0.6$, and $b=-2.54$. The second result is the correlation matrix; looking at the square roots of the diagonal elements we can estimate $\sigma_a = 0.007$ and $\sigma_b = 0.14$. The last result is the average error at each point, which matches reasonably well with the standard deviations of the individual data points.

20.4.1 Utilities

In the previous example, we entered the y , X , and σ arrays individually. For manual entry, it may be more convenient to enter the data point-by-point using $\Sigma+$. That is, for each point i , the entry for $\Sigma+$ might have the form

$$x_{i1} \quad \cdots \quad x_{im} \quad y_i \quad \sigma_i$$

Once the data is accumulated into ΣDAT , it needs to be decomposed into the three separate arrays for LSFIT. The following two programs are utilities that are useful for that purpose. C1SPLIT removes the first column of an $n \times m$ array, returning the column as a vector and the remainder as an $n \times m-1$ matrix. CLSPLIT works similarly, removing the last column.

C1SPLIT		Column 1 Split	ADD9
level 1			level 2
level 1			level 1
[[$n \times m$]]			[n]
[[$n \times m-1$]]			
<pre><< TRN OBJ-> OBJ-> DROP -> m n << m 1 - n 2 ->LIST ->ARRY TRN n 1 + ROLLD n ->ARRY SWAP >> >></pre>		Take data matrix apart. Recombine last $m-1$ rows. Make first row into a vector.	

CLSPLIT	Column Last Split	74D5
	level 1 level 2 level 1	
	$[[n \times m]]$ $\rightarrow$ $[n]$ $[[n \times m - 1]]$	
<pre><< TRN OBJ→ OBJ→ DROP → m n << n →ARRY m 1 - n * 1 + ROLL m 1 - n 2 →LIST →ARRY TRN >> >></pre>		Take apart matrix. Make last row into vector. Roll behind remaining data. Recombine remaining rows.

For cases where one of the unknown coefficients is simply an additive constant, using one of the following two programs will save you from having to enter a 1 in each row of the data matrix. C1ADD1 prepends a column of 1's to an array; CLADD1 appends a column of 1's.

C1ADD1	Column 1 Added with 1's	7343
	level 1 level 1	
	$[[n \times m]]$ $\rightarrow$ $[[n \times m + 1]]$	
<pre><< TRN DUP SIZE OBJ→ DROP → m n << n 1 →LIST 1 CON OBJ→ DROP n 1 + ROLL OBJ→ DROP m 1 + n 2 →LIST →ARRY TRN >> >></pre>		Save dimensions. Enter n 1's. Retrieve original matrix. Combine into new matrix.

CLADD1	Column Last Added with 1's	7343
	level 1 level 1	
	$[[n \times m]]$ $\rightarrow$ $[[n \times m + 1]]$	
<pre><< TRN OBJ→ OBJ→ DROP → m n << n 1 →LIST 1 CON OBJ→ DROP m 1 + n 2 →LIST →ARRY TRN >> >></pre>		Save dimensions. Enter n 1's. Combine with original matrix.

With these utilities available, the data for the example in the previous section can be entered into Σ DAT:

```
CLΣ [ 25  12.25  0.1 ] Σ+
16  7.35  0.2  Σ+
9   2.59  0.2  Σ+
4   -.28  0.3  Σ+
1   -1.75  0.3  Σ+
0   -2.88  0.4  Σ+
1   -1.64  0.4  Σ+
4   .14   0.4  Σ+
9   2.42  0.3  Σ+
16  7.24  0.3  Σ+
25  12.55  0.2  Σ+
```

Then

```
ΣDAT CLSPLIT CLSPLIT CLADD1 ROT
```

sets up the stack arguments for LSFIT.

20.4.2 Polynomial Fits

A common curve-fitting problem is that of fitting data to a polynomial, where there is a single independent variable x , and the fit parameters are the coefficients of successive powers of x . LSFIT is easily adapted to this problem by supplying a matrix X for which the elements are $x_{ij} = x_i^{m-j}$. The width m of the matrix is one greater than the highest power of x that appears in the polynomial.

POLYFIT is an extension of LSFIT specifically for polynomials. It takes the same arguments as LSFIT, plus a real number that specifies the order of the polynomial. The matrix X should be a vector or a one-column matrix, where each element is one value of the independent variable x . POLYFIT creates a new X by computing the necessary powers of each x_i . POLYFIT concludes by calling LSFIT with the modified arguments. Note that the (level 3) result contains the polynomial coefficients in the same descending order that is expected by the polynomial programs in section 17.4.

■ *Example.* Fit a fourth-degree polynomial to the data in the following table, where the standard deviation of each measurement is 0.85:

x	y
5	34.8
10	134.7
14	159.3
18	156.9
24	132.2
30	117.4
35	132.5
41	186.6
50	342.2

(This problem is taken from the *HP 82484A Curve Fitting Pac Owner's Manual* for the HP 71.)

■ *Solution.* Enter y:

[34.8 134.7 159.3 156.9 132.2 117.4 132.5 186.6 342.2]

Enter X:

[5 10 14 18 24 30 35 41 50]

Enter σ and the order, and compute the fit:

.85 4 POLYFIT.

The level 3 result vector contains the following entries, to four places:

<i>Coefficient</i>	<i>Value</i>	<i>Standard Deviation</i>
a_4	-0.0004781	0.000011
a_3	0.07049	0.0013
a_2	-3.270	0.047
a_1	57.48	0.66
a_0	-179.2	2.9

The standard deviations are the square roots of the diagonal elements of the level 2 result matrix. The calculation returns (level 1) $\sigma = 0.811$, which is comparable to the presumed standard deviation of 0.85.

POLYFIT				Polynomial Fit			99BD
level 4	level 3	level 2	level 1		level 3	level 2	level 1
y	x	W	order	σ	a	V	σ
<pre> << 3 PICK SIZE 1 GET → x w o n << 1 n FOR i 1 x i GET 1 1 o FOR j OVER * DUP j 3 + ROLLD NEXT DROP2 NEXT n o 1 + 2 →LIST →ARRAY w LSFIT >> >> </pre>				Store the input data. For each element in X: Get the element x_i. Compute powers of x_i. Combine into x. Compute the fit.			

20.4.3 Non-Linear Least Squares Fits

The least-squares fitting process can be extended to non-linear cases, i.e. cases where the fit parameters a_j are not simple linear coefficients of the independent variables. That is, the model to be fitted to measured data has the general form $\phi = \phi(\mathbf{x}, \mathbf{a})$. We can relate this general case to the linear problem discussed in section 20.4 by expanding ϕ about the point $\mathbf{a}_0$:

$$\phi(\mathbf{x}, \mathbf{a}) \approx \phi(\mathbf{x}, \mathbf{a}_0) + \Delta\phi \cdot \Delta\mathbf{a}$$

where

$$\Delta\phi_j = \frac{\partial\phi}{\partial a_j}(\mathbf{x}, \mathbf{a}_0)$$

and $\Delta\mathbf{a} = \mathbf{a} - \mathbf{a}_0$. This is linear in the parameters $\Delta\mathbf{a}$, and so can be translated to the linear least-squares problem $\phi = \mathbf{X}\mathbf{a}$ by replacing ϕ with $\phi(\mathbf{x}, \mathbf{a}) - \phi(\mathbf{x}, \mathbf{a}_0)$, $\mathbf{X}$ with $\Delta\phi$, and $\mathbf{a}$ with $\Delta\mathbf{a}$.

Therefore, if we take $\mathbf{a}_0$ as a first guess for $\mathbf{a}$, and compute $\Delta\mathbf{a}$ by the least squares method, then $\mathbf{a} + \Delta\mathbf{a}$ should provide a better estimate of $\mathbf{a}$. This process can be iterated until $\Delta\mathbf{a}$ is arbitrarily small. The trick, of course, is to obtain initial values $\mathbf{a}_0$ that are

sufficiently accurate for the linear approximation to be valid. With initial values that are too far from the correct values, the iteration may never converge.

The program NLFIT listed below demonstrates an iterative non-linear least-squares fit, for problems with a single independent variable x . It requires six arguments:

6:	y	Vector of original measurements y_i
5:	x	Vector of values x_i of the independent variable.
4:	W	Weights, entered as for LSFIT.
3:	$\phi(x,a)$	Algebraic object representing ϕ .
2:	$\{a_i\}$	List of parameter names.
1:	$[a_{i0}]$	Vector of initial values of parameters.

To simplify the program itself and argument entry, NLFIT uses the specific name X to represent the independent variable x (you can easily modify the program to use any other global name). This means that the expression entered as the level three argument should contain X , as well as the names of the fit parameters. NLFIT uses the actual variables during program execution, so any previous stored values are overwritten.

After computing one iteration of the non-linear least squares fit, NLFIT halts and displays the corrections Δa (level 2) and the average error per point σ (level 1). You can use that information to decide whether to repeat the iteration. Use ≡MORE≡ in the displayed temporary menu to continue with another iteration, or ≡QUIT≡ to terminate the program and return the final results, which are the parameter vector a (level 3), the covariance matrix V (level 2), and the final σ . The two objects returned at each intermediate halt are not required by the program, so you can drop them, use them in other calculations, or leave them for comparison with the results of the next iteration. If you change menus, you can resume execution by pressing 0 for ≡MORE≡, or 1 for ≡QUIT≡.

The use of NLFIT is best explained by an example.

■ *Example.* The data in the table below (following the program listing) was obtained by measurement of a voltage at uniform time intervals. The voltage is expected to behave sinusoidally with time, i.e.

$$V(t) = V_0 + A \sin(\omega t + \theta)$$

Find the unknown parameters V_0 , A , ω and θ , which are estimated to be $V_0 = 9$ volts, $A = 11$ volts, $\omega = .6$, and $\theta = .9$. The measurement error is ± 1.75 volts.

NLFIT		Non-Linear Fit							D7E6
level 6	5	4	3	2	1		3	2	1
y	x	W	' $\phi(X,a)$ ' {names}	a_0	σ		a	V	σ
<pre> << 5 PICK SIZE 1 GET 3 PICK SIZE RCLMENU {{ "MORE" <<0 CONT>> } { "QUIT" <<1 CONT>> }} → y xm w f a a0 n m cm tm << DO 1 m FOR i a0 i GET a i GET STO NEXT << 'X' >> 'X' STO 1 m FOR i f a i GET .d NEXT m →LIST → da << 1 n FOR i y i GET xm i GET 'X' STO f EVAL - NEXT n →ARRY 1 n FOR i xm i GET 'X' STO 1 m FOR j da j GET EVAL NEXT NEXT {n m} →ARRY W LSFIT → a1 cm sig << a1 'a0' STO+ a1 sig "σ" →TAG tm TMENU IF HALT THEN 1 m FOR i a0 i GET a i GET STO NEXT 'X' PURGE a0 cm sig 1 ELSE 0 END >> >> UNTIL END cm TMENU >> >> </pre>						n = no. of data points. m = no. of parameters. cm = Current menu. tm = temporary menu. Create local variables. Store initial values of a. Prevent evaluation of X. Create list of derivatives: $\partial f / \partial a_i$. Save derivatives in da For each data point: y_i. Store x_i in X. $y_i - f(x_i, a_0)$ For each data point: Store x_i in X. For each parameter: $\partial f / \partial a_j(x_i, a_0)$ Make data array. Compute, save fit parameters. Corrected a. Halt and show results. If done, Store final values of a, and return fit results. Otherwise, repeat. Restore original menu.			

t	$V(t)$	t	$V(t)$
0.0	19.44	3.3	7.02
0.3	23.92	3.6	13.81
0.6	20.09	3.9	19.33
0.9	18.51	4.2	22.86
1.2	16.73	4.5	22.91
1.5	9.12	4.8	22.02
1.8	8.78	5.1	21.86
2.1	5.45	5.4	13.35
2.4	-0.32	5.7	12.22
2.7	3.93	6.0	8.07
3.0	5.76		

■ *Solution.*

1. Enter the measurement vector:

```
[ 19.44  23.92  20.09  18.51  16.73  9.12  8.78  5.45
  -0.32  3.93  5.76  7.02  13.81  19.33  22.86  22.91
  22.02  21.86  13.35  12.22  8.07 ]
```

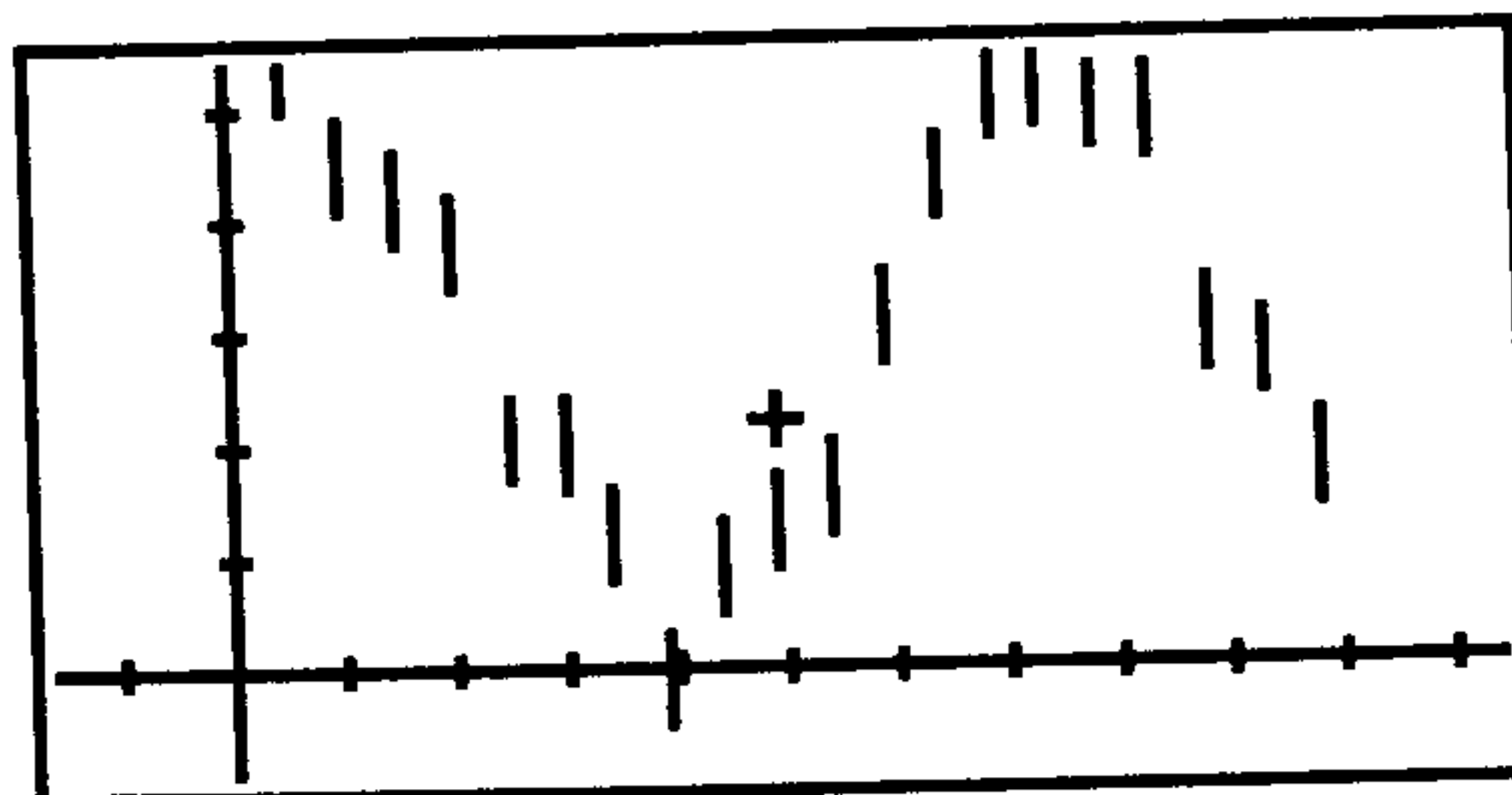
2. Enter the vector of independent variable values. Either type it in, or execute

```
0 6 FOR i i .3 STEP
```

3. Enter 1.75 for the measurement error.

The objects on the stack at this point are also those needed to plot the data using DRAWBAR (section 20.3.4):

```
-1 7 XRNG -4 24 YRNG 3 DUPN ERASE DRAWBAR
```



4. Enter the fit expression:

$$'V0+A*\text{SIN}(\omega*X+\theta)'$$

Notice that the independent variable must be X. ω is $\alpha \rightarrow W$, and θ is $\alpha \rightarrow F$.

So that you can plot this expression later, execute DUP STEQ.

5. Specify the unknown parameter names:

$$\{V0 \ A \ \omega \ \theta\}$$

6. Enter the initial guesses for the parameters:

$$[9 \ 11 \ 1.6 \ .9]$$

7. Execute RAD 2 FIX NLFIT. This produces the following display:

RAD	HALT
{ HOME STAT REGR }	
4:	
3:	
2:	[3.48 -1.16 -0.10...
1:	σ : 1.74
MORE QUIT	

The increments Δa returned to level 2 are substantial fractions of the initial guesses, which suggests continuing with the iteration. Press MORE:

RAD	HALT
{ HOME STAT REGR }	
4:	[3.48 -1.16 -0.10...
3:	σ : 1.74
2:	[0.02 0.34 -0.01...
1:	σ : 1.75
MORE QUIT	

This iteration has not changed σ very much, so further iteration is unlikely to produce a better result. Now press **QUIT**:

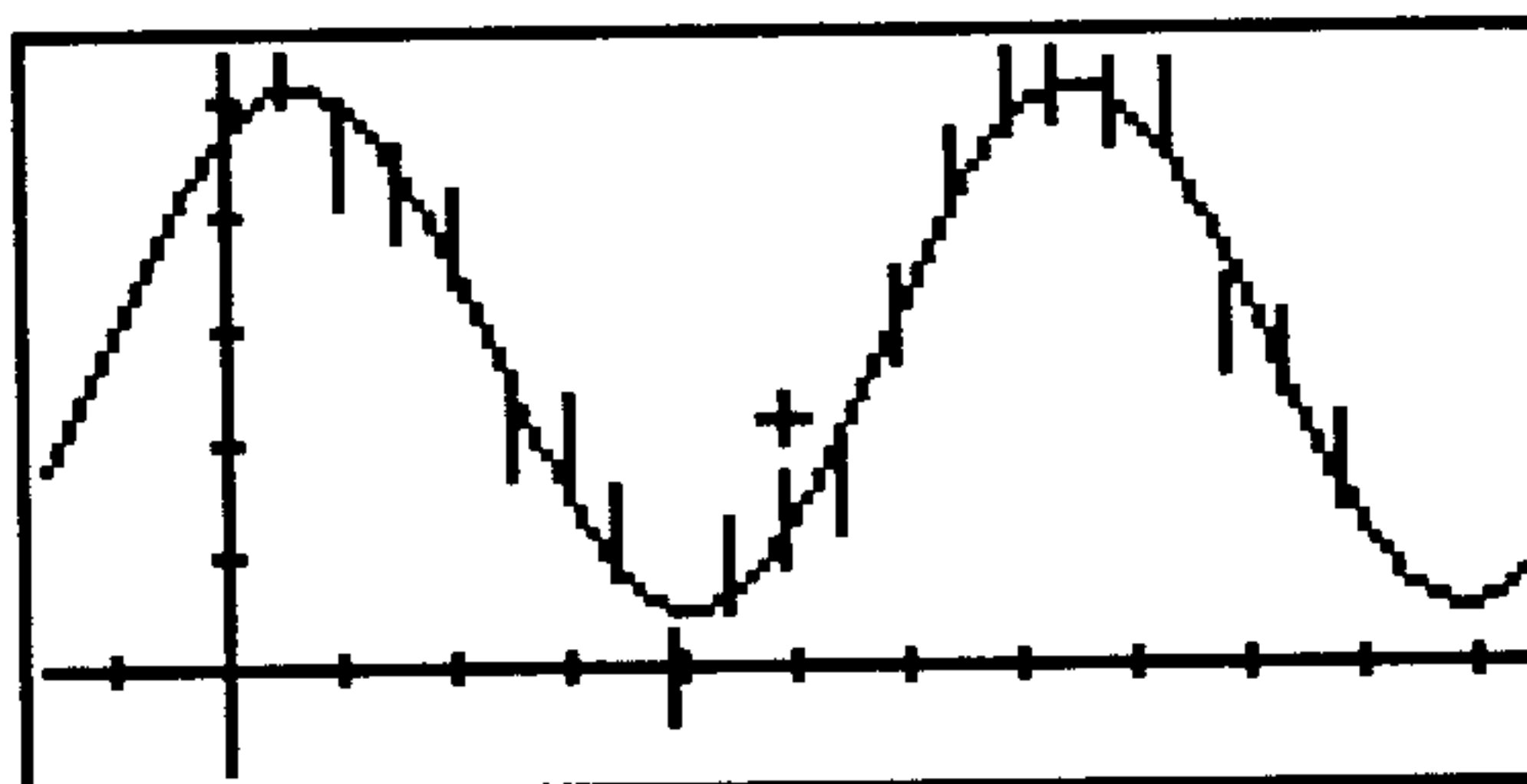
```

RAD
{ HOME STAT REGR }
4:                                      $\sigma$ : 1.75
3: [ 12.50 10.18 1.49...
2: [[ 0.12 -0.04 3.83...
1:                                     1.75
DRAW EQ NLTE NLFIT BARIU LSFIT

```

The final values **a** are also stored in the appropriate global variables. This means that you can plot the expression stored in **EQ** to show it as evaluated with the fit parameters, superimposed on the data:

FUNCTION DRAW 



20.5 Probability Commands

The probability menu (**MTH** **PROB**) contains nine commands that relate to common probability calculations. The commands logically could have been included as part of the statistics menu, but that menu is already five pages long.

20.5.1 Combinations and Permutations

The commands **COMB** and **PERM** compute the number of *combinations* and *permutations*, respectively, of n different objects taken m at a time. Each different ordering of the m selected objects is a different permutation. Since there are n choices for the first object, and $n - 1$ choices for the second, and so on, we obtain

$$P_{n,m} = n(n-1)(n-2) \cdots (n-m+1) = \frac{n!}{(n-m)!}$$

PERM takes its arguments in the order n,m :

10 5 PERM $\Rightarrow$ 30240,

or

'PERM(10,5)' EVAL $\Rightarrow$ 30240.

For combinations, the order of the selected objects doesn't matter; since there are $m!$ ways to arrange m objects, the number of combinations is reduced by that factor from the number of permutations:

$$C_{n,m} = P_{n,m}/m! = \frac{n!}{m!(n-m)!}.$$

COMB takes its arguments in the same order as PERM:

10 5 COMB $\Rightarrow$ 252

The $C_{n,m}$ are the same as the *binomial coefficients* $\begin{Bmatrix} n \\ m \end{Bmatrix}$ in the expansion

$$(x+y)^n = \sum_{m=1}^n \begin{Bmatrix} n \\ m \end{Bmatrix} x^{n-m} y^m.$$

COMB is therefore used to compute the binomial coefficients in the program PPWR, in section 17.4.1.

The algorithms used in computing COMB and PERM do not use the factorial function directly, to minimize numerical overflow problems. Factorial overflows for integer arguments greater than 253, but COMB and PERM can deal with arguments up to 10^{12} . Arguments within that range can still cause the overflow exception (section 9.6.4) when the results are greater than 10^{500} . In general, both arguments for either command must be zero or positive integers less than 10^{12} . Non-integer or too-large arguments return the Bad Argument Value error.

20.5.1.1 Factorial and Gamma Function

The HP48 factorial function ! computes the ordinary factorial of integer arguments $n \geq 0$: $n! = n(n-1)(n-2) \cdots 2 \cdot 1$. However, ! is actually a version of the *gamma function*

$$\Gamma(x) = \int_0^{\infty} e^{-t} t^{x-1} dt.$$

That is, since $\Gamma(x+1) = x!$, HP48 $x!$ computes $\Gamma(x+1)$. If you want an ordinary gamma function, create a user-defined function as follows:

'GAMMA(x) = (x-1)!' DEFINE

The useful range of ! is $-254.1082426465 < x < 253.1190554375$. More negative x causes the underflow exception; more positive x causes the overflow exception. Negative integer values of x also cause overflow.

The factorial is represented as a postfix function for consistency with common written notation. You may also enter it as a prefix function FACT, which is automatically converted to the postfix ! when it is evaluated:

'FACT(X)' EVAL  'X!'

assuming that X has no current value.

20.5.2 Random Numbers

Several examples in this chapter have used the *random number* generator command RAND to create artificial "noise" in data. RAND is designed to produce random numbers *uniformly distributed* between 0 and 1, meaning that any number in that range is equally likely to be returned at each execution. Since there are $10^{12} - 1$ HP48-representable numbers in this range, the probability density is

$$\rho_n = 1/(10^{12} - 1) \quad 0 < n < 1$$

The expectation value of the average of a sample of numbers generated by RAND is obviously 0.5. We can also estimate the standard deviation by treating the distribution function as continuous:

$$\sigma = \left(\int_0^1 (x - .5)^2 dx \right)^{1/2} = \frac{1}{2\sqrt{3}}.$$

RAND uses the *linear congruential method* to create uniform random numbers (cf. Donald Knuth, *The Art of Computer Programming, Vol. II*, Addison-Wesley, Massachusetts 1981). In this method, a sequence of random numbers is generated from

$$x_i = (ax_{i-1} + c) \bmod m$$

Such sequences inevitably repeat, so that they are not truly random. The period of the cycles can be maximized by good choices of the constants a , c , and m . RAND uses values that give a period of 5×10^{13} , using 15-digit internal precision.

The last value returned by RAND is stored in 15-digit form in system memory, so that it can be used for the next execution. You can change this “seed” value by executing RDZ (*RanDomize*), which computes and stores a new 15-digit seed from its argument, which can be any real number. RDZ is useful when you want to repeat a particular series of random numbers--you can restart the same series whenever you want by executing RDZ with the same argument. The examples in this book use RDZ prior to repeated execution of RAND, so that you can try the examples and get the same results. You can also use RDZ to prevent accidental repeat of a series, by using an argument of 0. In that case, RDZ creates a random seed based on a reading of the HP 48 clock.

As a rough check of the uniform distribution of the numbers generated by RAND, we can compute the mean and standard deviation of a sample of 1000 numbers:

```
12345 RDZ 1 1000 START RAND NEXT { 1000 1 } →ARRY STOS,
```

then

```
MEAN  ⌈ .507100107091
STD   ⌈ .289199195881
```

These compare well with the ideal values of 0.5 and .288676134595 ($1/2\sqrt{3}$). The RAND algorithm also passes the *spectral test* (described by Knuth), a much more stringent test of a random number generator.

It is possible to produce random numbers with non-uniform distributions by using functions of uniform random numbers. The programs in section 12.11.1 illustrate the generation of numbers that follow the normal and the Poisson distributions.

20.6 Upper-Tail Probability Distributions

A random variable is characterized by its *distribution function*, which is the (cumulative) probability $P(x)$ that a particular measurement of the variable will have a value less than x . $P(x)$ must be a monotonically increasing function of x (a step function for a discrete variable), with $P(\infty) = 1$. The complement to the distribution function is the *upper-tail distribution function* $U(x)$, which is the probability that the variable is measured with a value greater than x , i.e.

$$U(x) = 1 - P(x)$$

For continuous, differentiable distribution functions, the first derivative is called the *probability density* $\rho(x) = dP(x)/dx$; the probability that a measurement returns a value between x and $x+dx$ is $\rho(x)dx$. The normalization condition for $\rho(x)$ is then

$$\int_{-\infty}^{+\infty} \rho(x) dx = 1.$$

Most probability distributions can be characterized most easily by the probability density, with the distribution functions calculated as integrals of the densities:

$$P(x) = \int_{-\infty}^x \rho(x) dx \quad \text{and} \quad U(x) = \int_x^{\infty} \rho(x) dx.$$

For example, the *normal*, or *Gaussian* distribution is defined by the density

$$\rho(x) = \frac{1}{(2\pi)^{1/2}\sigma} e^{-\frac{(x-\bar{x})^2}{2\sigma^2}},$$

where $\bar{x}$ is the average of x , and σ is the standard deviation. Common probability densities generally have a long “tail” that asymptotically approaches zero as $x \rightarrow \infty$ —this is the origin of the “upper tail” term that describes the HP 48 distribution functions.

Such densities are easily calculated on the HP 48 from their definitions; in section 20.6.5 we list user-defined function programs for the densities associated with the four HP 48 upper-tail probability commands. The distributions present a harder calculation problem since they involve numerical integrals with $\pm\infty$ as one limit, as shown here for the upper-tail probability of the normal distribution:

$$U(x) = \frac{1}{(2\pi)^{1/2}\sigma} \int_x^{\infty} e^{-\frac{(t-\bar{x})^2}{2\sigma^2}} dt$$

Because of the difficulty of computing such integrals, the HP 48 provides four commands that use special algorithms for calculating upper-tail probabilities for common distributions. The upper-tail probabilities are chosen because they are used more frequently by statisticians than the distribution functions, but of course you can compute $P(x)$ from $1 - U(x)$. The four commands are UTPN (*Upper-Tail Probability--Normal*), which embodies the formula above, plus UTPF, UTPC, and UTPT, for the F -, χ^2 -, and t -distributions.

That these commands were not encoded as HP 48 functions was simply a matter of conserving ROM space in the HP 48. You can easily make user-defined functions for these commands, e.g.

```
<< → n1 n2 x << n1 n2 x UTPF >> >> 'FUTPF' STO
```

20.6.1 UTPN

UTPN takes three real number arguments: the distribution average $\bar{x}$, the variance σ^2 , and the number x . It returns the probability that a measurement of x will return a value greater than the specified x , given a normal distribution described by $\bar{x}$ and σ .

■ *Example.* On a standardized test with a mean score of 500 and a standard deviation of 100, what fraction of students is expected to score 750 or better, assuming a normal distribution of test scores?

■ *Solution.* The variance is the square of the standard deviation, so

```
500 100 SQ 750 UTPN ⏎ .0062
```

20.6.2 UTPT

The sample mean

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

(section 20.2) is itself a random variable. Its distribution is usually expressed in terms of the normalized parameter $t = \bar{x}n^{1/2}/s$, where s is the square root of the sample variance. t is described by the probability density

$$\rho(t) = \frac{\Gamma(\frac{f+1}{2})}{\Gamma(\frac{f}{2})\sqrt{f\pi}} (1 + t^2/f)^{-\frac{f+1}{2}},$$

where $f = n - 1$ is the number of degrees of freedom.

UTPT takes f and t as its arguments, and returns the probability that a measurement of t will yield a value greater than the specified t .

20.6.3 UTPC

The sample variance (section 20.2)

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

is itself a random variable. If the sample is drawn from a normal distribution with variance σ^2 , then the random variable

$$\chi^2 = \frac{n-1}{\sigma^2} s^2$$

follows the so-called χ^2 -distribution with $f = n - 1$ degrees of freedom. This distribution has the probability density

$$\rho(\chi^2) = \frac{(\chi^2)^{\lambda-1} e^{-\chi^2/2}}{2^\lambda \Gamma(\lambda)}, \quad \chi^2 \geq 0$$

where $\lambda = f/2$. The density is zero for $\chi^2 < 0$.

UTPC takes f and χ^2 as its arguments, and returns the probability that a measurement of χ^2 will yield a value greater than the specified χ^2 .

20.6.4 UTPF

The F -test compares the sample variances of populations that have equal means. The random variable in this case is the variance ratio

$$F = s_1^2 / s_2^2$$

where s_1 and s_2 are the two sample variances, computed for n_1 and n_2 measurements, respectively.

The probability density for F is given by

$$\rho(F) = \left(\frac{f_1}{f_2} \right) \frac{\Gamma\left(\frac{f_1 + f_2}{2}\right)}{\Gamma(f_1/2)\Gamma(f_2/2)} F^{\frac{1}{2}f_1 - 1} (1 + f_1 F / f_2)^{-\frac{1}{2}(f_1 + f_2)}, \quad F \geq 0.$$

$$= 0, \quad F < 0$$

where $f_1 = n_1 - 1$ and $f_2 = n_2 - 1$ are the numbers of *degrees of freedom* of the two distributions.

UTPF takes f_1 , f_2 , and F as its arguments, and returns the probability that a measurement of F will yield a value greater than the specified value.

20.6.5 Probability Density Functions

The following user-defined functions compute probability densities for the normal, χ^2 , t , and F distributions.

NDIST	Normal Probability Density				7874
	level 3	level 2	level 1		level 1
	$\bar{x}$	σ	x		$\rho(x)$

```
<< - m sigma x 'EXP(-(x-m)/sigma)^2/2)/(sqrt(2*pi)*sigma)'
>>
```

CDIST	χ^2 Probability Density				223C
	level 2	level 1		level 1	
	f	χ^2		$\rho(\chi^2)$	

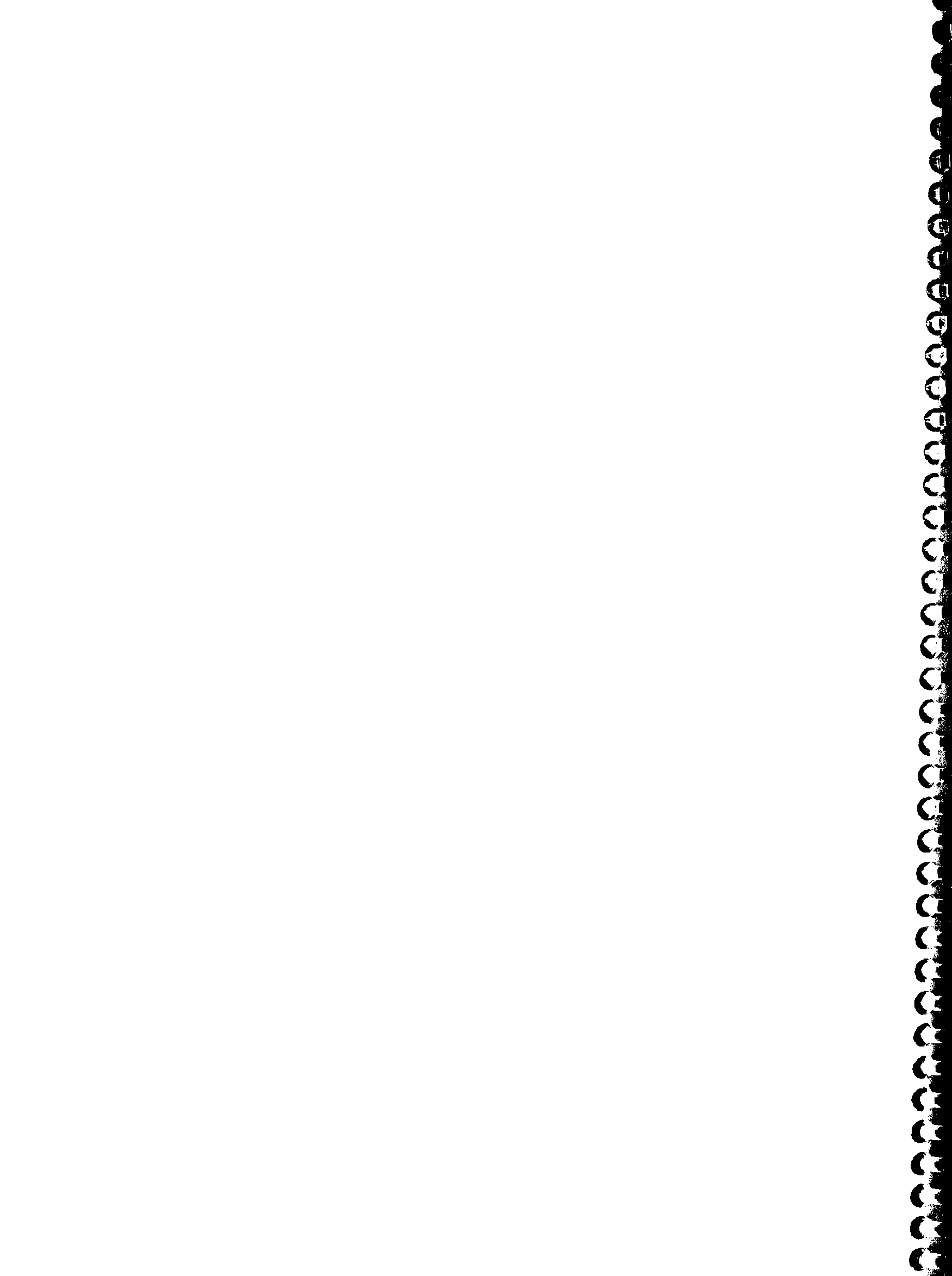
```
<< - f x
      'IFTE(x>=0,x^(f/2-1)*EXP(-x/2)/(2^(f/2)*(f/2-1)!),0)'
>>
```

TDIST	t Probability Density				C980
	level 2	level 1		level 1	
	f	t		$\rho(t)$	

```
<< - f t
      '((f-1)/2)!/((f/2-1)!*sqrt(f*pi))*(1+t^2/f)^-((f+1)/2)'
>>
```

FDIST	F-Distribution Probability Density				6EDD
	level 3	level 2	level 1		level 1
	f_1	f_2	F		$\rho(F)$

```
<< - f1 f2 F
      'IFTE(F>=0,(f1/f2)^(f1/2)*(((f1+f2)/2-1)!/((f1/2-1)!
      *(f2/2-1)!))*F^(f1/2-1)*(1+f1/f2*F)^-((f1+f2)/2),0)'
>>
```

21. Unit Management

A good deal of the effort involved in solving practical engineering and scientific problems comes from dealing with physical units. The results of most measurements are not simple dimensionless numbers, but combine a numerical magnitude with various *units* of length, time, mass, etc. Calculations with these values require keeping track of these units, including converting between values expressed with different units. Calculators have generally been unable to provide much assistance here. Many calculators contain tables of multiplicative conversion factors, which allow you to convert a number representing a measurement in one unit to another unit, such as multiplying a number of inches by 2.54 to obtain the equivalent number of centimeters. But the real accounting work here--collecting and canceling units and checking for dimensional consistency--is still left to pencil and paper (reminiscent of the pre-calculator era, where we used a slide-rule to perform operations on the mantissas of numbers, but had to keep track of the exponents by hand). The HP 48 removes this burden from you by providing a *unit management* system, incorporating physical units into a wide range of its calculation facilities.

In the HP 48, a magnitude and its associated units are combined into a single object called a *unit object* (section 3.4.9). These objects are entered and displayed in the format

magnitude_dimensions,

where *magnitude* is an ordinary real number, and *dimensions* is an expression combining products, powers, and (at most) one quotient, of unit names. The magnitude is displayed in the same manner as a number within an algebraic object, following the current number display format (STD, FIX, etc.), except that digit separators are not used, and integers less than 1000 are displayed without any decimal part. The dimension part is displayed in a manner similar to an algebraic object. Since only products and simple powers are allowed, plus one quotient, parentheses are used only to enclose a denominator that contains a product.

When you enter a unit object in the command line, you can use same rules that are used in the object display. Note that you must enclose a compound denominator in parentheses, because of the usual algebraic operator precedence. That is,

1_m/s*s **ENTER** $\rightarrow$ 1_m.

The entry is equivalent to 1_(m/s)*s, so that the two s units cancel each other.

Parentheses are also important when you enter a unit object within an algebraic object, to separate the unit object from the rest of the expression:

$$\begin{aligned} 'Y+1\_m/s*X' &\quad \Rightarrow \quad 'Y+1\_m*X/s' \\ 'Y+(1\_m/s)*X' &\quad \Rightarrow \quad 'Y+(1\_m/s)*X'. \end{aligned}$$

Whether a unit object is entered in the command line, or created through operations on other unit objects, the dimension part is always simplified as follows:

- Any real numbers are multiplied together, then multiplied times the number part of the unit object. Thus 1_5*m becomes 5_m .
- There are only positive powers (any unit with a negative power is shown in the denominator).
- There is at most one $/$ symbol; if the denominator is a product of units, it is enclosed in a single pair of parentheses.
- If a unit operation returns a result with a null dimensions, the result is converted to a real number. For example,

$$1\_m \ 1\_m \ / \quad \Rightarrow \quad 1.$$

- Unit objects entered within expressions may be entered with symbolic magnitudes; such entries are automatically converted to the product of the symbolic magnitude times a normalized unit object. Thus $'A\_m'$ becomes $'A*1\_m'$

21.1 Types of Units

The HP 48 unit system is based on the *System Internationale*, commonly abbreviated as *SI*. This system specifies *base units*, representing seven independent physical dimensions:

Dimension	Base Unit
Length	meter
Time	second
Mass	kilogram
Temperature	kelvin
Electrical current	Ampere
Luminous intensity	candela
Quantity of substance	mole

These units are considered as fundamental; all other units are expressed in terms of the base units.

The HP 48 system actually uses eight base units. The extra dimension is unspecified, i.e. it is not related to any physical dimension, and so is reserved for use in user-defined units (see section 21.1.3 below). One built-in unit named ? has this dimension.

21.1.1 Prefixes

A natural extension to the built-in unit table is the *prefixing* of unit names with any of the following symbols:

Exponent	Word Prefix	SI symbol	HP 48 symbol(s)
+18	exa	E	E
+15	peta	P	P
+12	tera	T	T
+ 9	giga	G	G
+ 6	mega	M	M
+ 3	kilo	k	k or K
+ 2	hecto	h	h or H
+ 1	deka	da	D
- 1	deci	d	d
- 2	centi	c	c
- 3	milli	m	m
- 6	micro	μ	μ
- 9	nano	n	n
-12	pico	p	p
-15	femto	f	f
-18	atto	a	a

These symbols are used immediately preceding the unit names they modify, without any intervening symbols or spaces. Thus 1_cm represents 1 *centimeter*, and 1_ft/ns represents 1 *foot/nanosecond*. Built-in unit names are given precedence over prefixed unit names when there is an ambiguity. For example, 1_min is 1 *minute*, not 1 *milli-inch*. This resolution requires single-character prefix symbols for simplicity, so the SI prefix *da* is replaced in the HP48 by D. You may also use K for k and H for h when entering unit objects, but the objects will always be displayed with the lower-case prefixes.

21.1.2 Built-in Units

The HP 48 contains 122 units, which means that it has permanently stored their dimensions and conversion factors relative to the SI base units. For example, the unit mph is recorded with the dimensions $length \cdot time^{-1}$, and the conversion factor 0.44704 that expresses units of *mph* in base units *m/s*. We will refer to unit objects corresponding to the built-in units as *simple units*, since each has a dimension part that is a single unit name. A *compound unit*, such as 10_cm/s, contains a dimension part that is a composite object similar to an algebraic object, comprised of unit names, prefixes, powers, and special *, /, and ^ operators. For either type of unit object, the command UBASE converts an object into an equivalent where the dimension part of the result contains only base units. If the original magnitude is 1, then the result magnitude is the conversion factor from the original units to base units:

1_mph UBASE $\Rightarrow$.44704_m/s

You can use the following program to determine the actual dimensions of a unit object:

UDIMS		Unit Dimensions	8A8E
level 1			level 1
magnitude_units		→	{ dimensions }
<pre><< { } 1 1 1 1 1 1 1 1 → list kg m s A K cd mol ? << UBASE OBJ→ →STR 4 OVER SIZE SUB "" SWAP + OBJ→ 1 8 FOR i i { kg m s A K cd mol ? } SWAP GET 10 OVER STO OVER EVAL LOG 1 ROT STO 'list' SWAP STO+ NEXT DROP list + >> >></pre>		Initial values. Make local variables. Convert normalized unit object to string. Strip off "'1_". Convert to an expression. For each base unit: Get the <i>i</i>th base unit. Assign it value 10. Get the power of the base unit. Restore value 1. Add the power to the list. Discard the expression. Retrieve the list. Add the unit magnitude at the head.	

UDIMS converts a unit object into a list, where the first element is the equivalent

magnitude in base units, followed by the object's dimensions expressed as (integer) powers of length, mass, time, current, temperature, luminous intensity and quantity of matter, in that order. (UDIMS only works for integer dimension powers--see section 21.4.1).

```
1_mph  UDIMS  { .44104  0  1  -1  0  0  0  0  0 }
```

21.1.3 User-Defined Units

You can represent any unit that can be reduced to SI base units by using compound units. As an additional convenience, you can also create *user-defined units*, to add new unit names that can be used like simple units. A user-defined unit is any global variable that contains a single unit object: the stored object specifies the unit magnitude and dimensions, and the variable name may be used as a unit name in the same manner as any built-in name. Thus

```
.125_mi  'furlong'  STO
```

and

```
14_d  'fortnight'  STO
```

create new units furlong and fortnight; then 1_furlong/fortnight is a valid unit object with the dimensions of speed. User-defined units may also be used to define new units, e.g.

```
1_furlong/fortnight  'slow'  STO
```

defines a user-defined unit named slow. Any user-defined unit is subject to the same operations as built-in units:

```
1_slow  UBASE  1.6630952831E-4_m/s
```

The names of user-defined units should not match any built-in unit, with or without a prefix, as the latter are given precedence in the interpretation of unit objects entered in the command line.

21.1.4 Unit Object Mechanics

As for other object types that contain more than one part, the HP48 provides simple tools for taking unit objects apart, and for assembling them from other objects:

OBJ→ decomposes a unit object into its magnitude (a real number returned to level 2) and a normalized (magnitude 1) unit object of the same dimensions (level 1).

600_W/sr **OBJ→**  600 1_W/sr

→UNIT replaces the magnitude of a unit object in level 1 with a real number taken from level 2:

50 100_km/s **→UNIT**  50_km/s

UVAL extracts the magnitude of a unit object:

1.6E-19_C **UVAL**  1.6E-19

UVAL and **→UNIT** are available in the  **UNITS** menu; **OBJ→** is found in the  **PRG**  menu.

21.2 Unit Conversions

The HP 48's list of conversion factors for simple units to SI base units makes it straightforward for the calculator to convert any simple or compound unit into any other dimensionally consistent unit. To do so, the HP 48 computes a net conversion factor for the original unit by applying the arithmetic in its dimension part to the conversion factors of the simple units represented there. This value is divided by a net conversion factor similarly computed for the output units, then multiplied by the original unit magnitude. The result is the magnitude of the converted unit.

The most obvious application of this process is in the command **CONVERT**, which converts one unit (level 2) into an equivalent with its dimension part specified by a second unit object:

1_b*Mpc 1_cm^3 **CONVERT**  3.08567818585_cm^3

Note that the magnitude of the second argument unit object is ignored; that argument is only used to specify the new dimensions for the first argument.

If the dimensions of the two units are not consistent, i.e. they do not correspond to the same powers of SI base units, **CONVERT** returns the Inconsistent Units error. An attempted conversion of a unit object that contains undefined unit names returns the Invalid Unit error.

21.2.1 The Unit Menus

The *unit menus* are the sixteen built-in menus associated with the $\boxed{\leftarrow} \boxed{\text{UNITS}}$ key. All of the built-in simple units, plus a number of compound units, are represented in these menus, organized by common dimensionality or at least common application. For example, the length menu ($\boxed{\equiv} \boxed{\text{LENG}} \boxed{\equiv}$) contains keys for 22 simple units of length. The speed menu ($\boxed{\equiv} \boxed{\text{SPEED}} \boxed{\equiv}$) has seven entries with the dimensions *length/time*, plus one, *ga*, that is a unit of acceleration. (It is more practical to include *ga* in the speed menu than to have a separate acceleration menu with only one entry). The first three speed menu entries, m/s, cm/s, and ft/s, are compound units. Compound units are included in the units menus as typing aids and for convenient conversions.

The units menus are useful catalogs of the built-in units and their dimensions. Since unit names use upper and lower case letters, but the key label characters are upper-case only, you may occasionally wish to use $\boxed{\leftarrow} \boxed{\text{REVIEW}}$ to see the correct spelling of a unit name. For example, in the second page of the electricity menu ($\boxed{\equiv} \boxed{\text{ELEC}} \boxed{\equiv}$), $\boxed{\leftarrow} \boxed{\text{REVIEW}}$ produces:



To determine the value of a particular unit in SI base units, you can use the menu keys like this:

$\boxed{\leftarrow} \boxed{\text{UNITS}} \quad \boxed{\text{NXT}} \quad \boxed{\equiv} \boxed{\text{ENRG}} \boxed{\equiv} \quad 1 \quad \boxed{\equiv} \boxed{\text{BTU}} \boxed{\equiv}$
 $\boxed{\rightarrow} \boxed{\text{UNITS}} \quad \boxed{\equiv} \boxed{\text{UBASE}} \boxed{\equiv} \quad \boxed{\rightarrow} \quad 1055.05585262\_kg*m^2/s^2$

Here we entered the unit object 1_BTU by entering 1 then pressing $\boxed{\equiv} \boxed{\text{BTU}} \boxed{\equiv}$. This behavior is characteristic of units menu keys. Their design arises from the observation that the conversion of a unit object from one unit representation to another has certain similarities to an HP Solve problem. Consider the conversion of square centimeters to square inches. You could achieve such conversions by using a current equation

$$CM2*1\_cm^2=IN2*1\_in^2.$$

To convert 100 cm^2 to in^2 , you would use the following keystrokes:

$\boxed{\rightarrow}$ **SOLVE** 100 $\equiv$ **CM2** $\equiv$ $\boxed{\leftarrow}$ $\equiv$ **IN2** $\equiv$ $\boxed{\rightarrow}$ IN2: 15.5000310001.

Similarly, to convert a number of in^2 to cm^2 , you would store a value for IN2 and solve for CM2.

The HP 48 unit menus are deliberately modeled on the HP Solve system to provide convenient automated conversions among dimensionally consistent units. To achieve the above cm^2 -to- in^2 conversion, you can use these keystrokes:

$\boxed{\leftarrow}$ **UNITS** $\equiv$ **AREA** $\equiv$ 100 $\equiv$ **CM^2** $\equiv$ $\boxed{\leftarrow}$ $\equiv$ **IN^2** $\equiv$ $\boxed{\rightarrow}$ 15.5000310001_in^2.

Notice that the keystrokes after the initial menu selection are identical to the previous example. The result this time is a unit object with the same magnitude as the HP Solve result.

The unit menu behavior demonstrated here is derived from specific menu key actions. An *unshifted* unit menu key executes the sequence:

1. **ENTER** (if there is a command line);
2. 1_unit **ENTER**, where *unit* is the unit indicated by the menu key label;
3. $\boxed{\times}$ (multiply).

A *left-shifted* ($\boxed{\leftarrow}$) menu key works the same way, except that the final multiply is replaced by **CONVERT**.

Returning to the area conversion example, 100 $\equiv$ **CM^2** $\equiv$ enters 100, then multiplies it by 1_cm^2, yielding 100_cm^2. $\boxed{\leftarrow}$ $\equiv$ **IN^2** $\equiv$ enters 1_in^2, then executes **CONVERT**, returning 15.5000310001_in^2.

There are variations on the standard sequence of unit menu keys illustrated above. The multiply performed by the unshifted keys allows you to create compound units by pressing successive menu keys. For example, to convert 2500 *dyne-cm* to *joules*, press


```

2500  [↵] [UNITS] [NXT] [FORCE] [DYN]
[↵] [UNITS] [LENG] [CM]
[↵] [UNITS] [NXT] [ENRG] [↵] [J] [F] .00025_J.

```

A *right-shifted* ([↵]) unit menu key works like an unshifted key, except that it performs a final / instead of *. Thus, with the previous result still in level 1,

```

[↵] [UNITS] [TIME] [↵] [S] [F] .00025_J/s.

```

The unit menu keys literally execute the commands *, CONVERT, and /--if an error occurs, one of these commands is identified in the error message.

The various actions of the unit menu keys described here apply only in immediate-execute mode (section 6.4.1). In algebraic entry mode (ALG) or program entry mode (PRG), an unshifted menu key acts as a simple typing aid for the corresponding unit name (note that this means that to enter a unit object, you must use the [↵] [] key as well as the menu key). The shifted keys are inactive.

21.2.2 Using ?

The ? unit allows you to create new kinds of units that aren't tied to any ordinary base units, but for which automatic dimensional consistency checking is still available. One useful example is provided by currency conversions. Suppose you want to convert between dollars and pounds, where the conversion rate is 1.91 dollars to 1 pound. Define user-defined units as follows:

```
1_? '$' STO
```

```
1.91_? '£' STO
```

(The ? symbol is [α] [↵] [↵]; \$ is [α] [↵] [4]; £ is [α] [↵] [5].) With these definitions, you can use CONVERT to translate between the two currencies, e.g.

```
25_£ 1_$ CONVERT [F] 47.75_$.
```

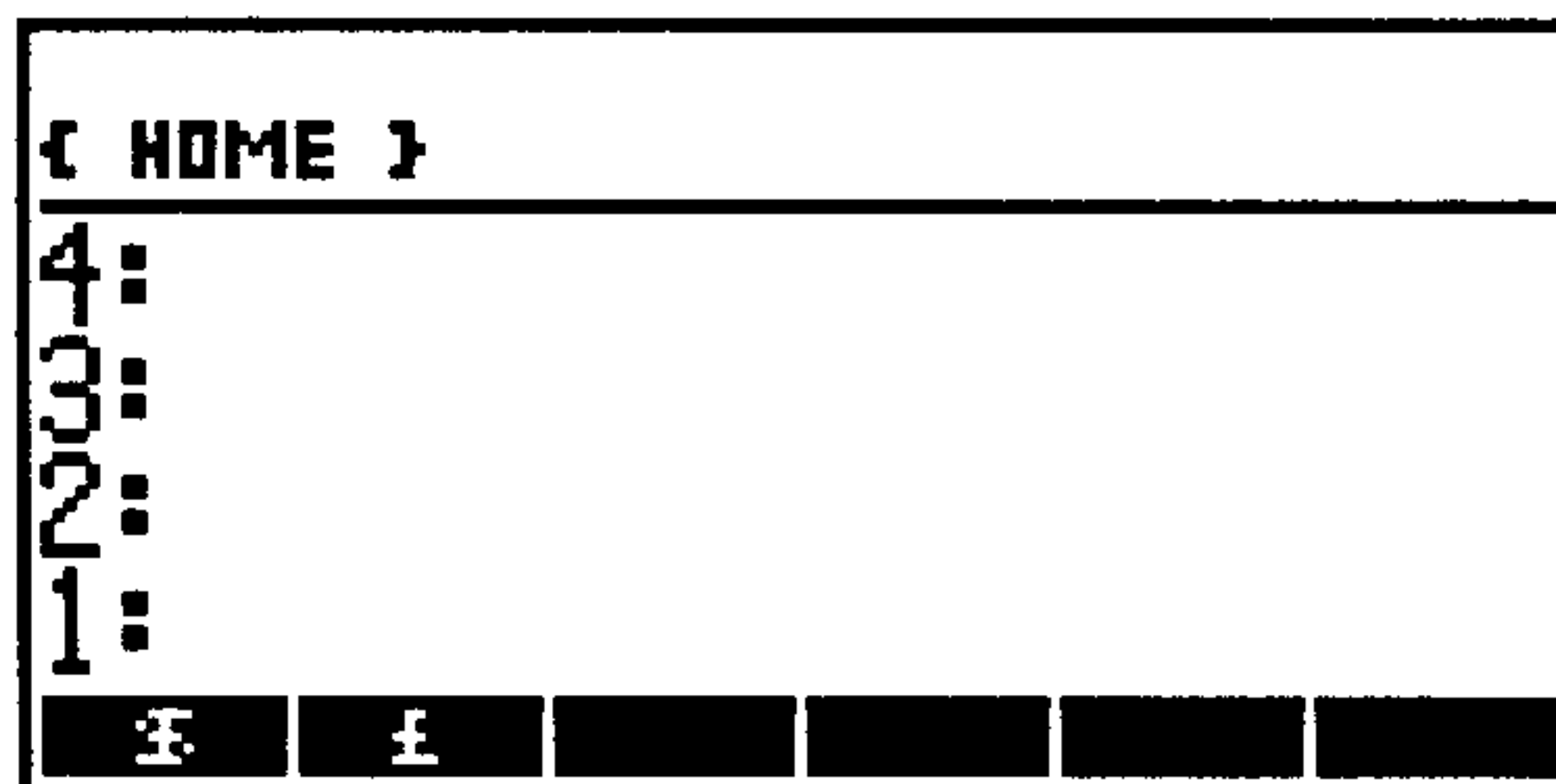
21.2.3 Units in Custom Menus

When a custom menu key (section 7.3) is specified by a unit object in the CST list, the key exhibits the same actions as the menu keys in any of the units menus. This is

particularly useful when you are performing repeated conversions among units that don't happen to be in the same unit menu, or include one or more user-defined units. For example, you can create a dollars/pounds menu as follows, using the user-defined units from the previous section:

{ 1_\$ 1_£ } MENU

yields this display



Here you can use the menu keys for conversions:

50 $\equiv$ \$ $\equiv$ $\leftarrow$ $\equiv$ £ $\equiv$ $\rightarrow$ 26.18_\$

21.3 Unit Object Mathematics

The real power of the HP 48 unit management system comes from the unit conversions that occur automatically when you apply various functions to unit objects. The conversions entail a certain speed penalty compared with the same functions applied to the real number magnitudes alone, but you gain a significant problem-solving efficiency because the calculator does the dimensional bookkeeping for you.

21.3.1 Unit Operations Requiring Dimensional Consistency

A typical unit operation is the addition of two objects with different dimension parts:

1_mi 1_km + $\rightarrow$ 2.609344_km

In effect, + applies CONVERT to the two arguments to convert the first argument to the same units as the second, then adds the two magnitudes. The result units are thus always the same as those of the second argument, which is consistent with CONVERT.

If you want a result expressed in the units of the first argument, execute **SWAP** prior to executing the combining function.

Automatic conversion is performed by all of the following functions:

- **+** and **-**.
- **%CH**, which computes $100 \cdot (x - y) / y$ from two objects x and y .
- **%T**. Here the first argument is converted to the dimensions of the second, then the magnitudes are divided. The result is always a real number.
- **=**. In numerical evaluation mode (section 3.5.5), **=** is equivalent to **-**.
- **==**, **≠**, **<**, **>**, **≤**, **≥**. These six predicate operators compare the magnitudes of two objects after converting them to common units. Note that **SAME** does not perform any conversions; two unit objects must have the same magnitudes and dimension parts for **SAME** to return *true*. See section 9.3.2.

If one of the two arguments of any of these functions is the real number 0, it is automatically replaced by **0_units** before the function is applied, where *units* is the dimension part of the non-zero argument. considered to have the dimensions of the other argument. Thus

$$1\_m \ 0 \ + \ \Rightarrow \ 1\_m$$

Besides being a simple convenience, this feature allows **ISOL** (section 16.4) to work with expressions containing unit objects. **ISOL** starts with 0, then accumulates terms as the various inverse functions are applied to the argument expression.

21.3.2 Unit Functions with Simplification

Multiplicative functions automatically simplify the dimension part of unit object results so that no unit name appears more than once. Thus

$$1\_cm/s \ 1\_cm^2/s \ * \ \Rightarrow \ 1\_cm^3/s^2$$

$$1\_g/l \ * \ 25\_l \ \Rightarrow \ 25\_g.$$

Unit names in the result's numerator and denominator are collected, and unit names common to the numerator and denominator are canceled. Note that only identical unit names with common prefixes are collected or canceled. When the net power of any name is zero, the name is removed from the dimension part of the unit. If all of the names in the dimension part cancel, the result is converted to a real number:

28_s 4_s / 7.

Unit simplification is performed by the following functions:

- *, %, /, and INV.
- ^, SQ, √, and XROOT. For ^ and XROOT, the level 1 argument is a real number, which is multiplied or divided into the powers of the unit names in the dimension part. If the argument is zero for XROOT, the Bad Argument Value error is reported (unlike the case where both arguments are real numbers, which returns the underflow exception).

The functions do *not* attempt to simplify specific combinations of units into equivalent units. 1_N*m, for example, does not automatically convert to 1_J. While this might be convenient in some cases, there is no unambiguous set of rules that the calculator could use to identify desirable combinations. The gasoline mileage of automobiles is an obvious case in point. We commonly express this quantity in *miles/gallon* or *kilometers/liter*, but it actually has the dimensions $length^{-2}$. It would require a very intelligent conversion algorithm to resist returning mileage in units of m^{-2} .

When you do want to convert a unit object to a specific form, you can use CONVERT, where the second argument contains the desired dimension part. For cases where you do not know exactly what form is required, but you do want to extract a particular combination unit, use UFACT. This command takes a unit object (level 2) and converts it such that its dimension part becomes the product of the dimension part of a level 1 unit object times whatever base unit factors are left over. Suppose you want to know what a *watt* is in terms of *newtons*:

1_W 1_N UFACT 1_N*m/s

UFACT is equivalent to the sequence

OBJ→ 3 ROLLD / OVER / UBASE *.

21.3.3 Operations on the Unit Magnitude

The following functions alter the magnitude of a unit object without regard to its dimensions: ABS, CEIL, FLOOR, FP, IP, NEG, RND, SIGN, and TRNC. All of these except SIGN apply a function to the unit magnitude, and combine the result with the original dimension part of the argument. SIGN returns a real number -1, 0, or +1, according to the sign of the magnitude.

RND and TRNC accept a unit object as the level two argument. The level one

argument, which specifies the rounding, must be a real number or a symbolic object.

21.3.4 Trigonometric Unit Functions

The trigonometric functions SIN, COS, and TAN can accept unit object arguments, provided the units are simple angle units--*radians* (r), *degrees* (°), *arc-minutes* (arcmin), *arc-seconds* (arcs), or *grads* (grad). The results are independent of the current angular mode. However, differentiation and integration of expressions containing these functions with unit object arguments will produce correct results only in radians mode. Note also that there are no corresponding inverse functions that return unit objects.

21.3.5 Examples of Calculations with Units

■ *Example.* A mass of 15 g moves in a circle of radius 7 cm with a speed of 35 cm/s. Compute its various kinematic and dynamic properties.

■ *Solution.* Start by storing the parameters:

```
15_g 'M' STO 7_cm 'R' STO 35_cm/s 'V' STO
```

Now compute the angular velocity:

```
V R / 5_1/s
```

```
'ω' STO
```

Moment of inertia:

```
'M*R^2' EVAL 735_g*cm^2
```

```
'I' STO
```

Angular momentum:

```
I ω * 3675_g*cm^2/s
```

Kinetic Energy:

```
'I*ω^2/2' EVAL 9187.5_g*cm^2/s^2
```

```
1_J CONVERT .00091875_J.
```

■ *Example.* How large must a parallel-plate air-filled capacitor be to have a capacitance

of 1 *farad*, if the plate separation is 0.1 *mm*?

■ *Solution.* The capacitance of a parallel-plate capacitor is given by $C = \epsilon_0 A/d$, where A is the plate area, d is the separation, and $\epsilon_0 = 8.9 \times 10^{-12} \text{ coul}^2/\text{N}\cdot\text{m}^2$. Rearranging, $A = dC/\epsilon_0$. Thus A can be calculated by

$$.1\_mm \quad 1\_F \quad * \quad 8.9E-12\_C^2/(N*m^2) \quad / \quad UBASE \quad \Rightarrow \quad 1.1E7\_m^2.$$

21.3.6 HP Solve

The current equation used by HP Solve and any of its variables may contain unit objects, either explicitly in expressions or implicitly as the values of variables within those expressions. The only constraint is that you must supply a guess for the unknown variable which is a unit object with the correct dimensions for that variable. See section 14.3 for more detail and examples.

When the root-finder evaluates the current equation at each iteration, it evaluates it in the normal way, then strips the units from the result for comparison with the previous iteration. If the current equation happens to change its units at some point in its domain, this would appear as a discontinuity to the root-finder, leading possibly to an invalid result. This situation arises from terms like this:

$$\text{IFTE}(x < 0, 1\_cm, 10\_mm).$$

Such terms are unlikely to appear in practical problems.

21.3.7 Plotting

The automatic plotting provided by DRAW (Chapter 15) does not fully support units, in that the plotting ranges may only be specified as simple real numbers. However, you can still create function plots from current equations that contain unit objects, with the following provisions:

- You must store a unit object with appropriate dimensions in the independent variable. The magnitude of the object is not used, but the dimension part determines the units of the independent variable at each evaluation of the current equation as the plot progresses across the horizontal domain.
- The horizontal and vertical ranges are taken to be the magnitudes of unit objects. That is, if the current equation evaluates to y_units_y , then the vertical range is assumed to be $y_{\min\_units_y}$ to $y_{\max\_units_y}$, where $y_{\min}$ and $y_{\max}$ are the (dimensionless) vertical plot limits. The horizontal range is effectively $x_{\min\_units_x}$ to $x_{\max\_units_x}$, where $units_x$ is the dimension part of the unit object stored in the

independent variable.

- The left and right sides of an equation should evaluate to the same units. Otherwise they will be plotted with different vertical scales. `≡ROOT≡` and `≡ISECT≡` will still give valid results, but the graph will show the intersections in the wrong places.

As in the case of HP Solve, an artificial discontinuity can be introduced if the current equation for plotting returns different units at different points in the plot domain.

21.3.8 Differentiation

Consider the simple conservation of energy for a falling body: $\frac{1}{2}v^2 = gh$, where v is the body's speed after it falls a distance h . Entering this equation into the HP 48, you can use QUAD (section 16.4.3) to solve for V :

```
'V^2/2=g*h' 'V' QUAD 1 's1' STO EVAL COLCT ⏏ 'V=√(2*g*h)'
```

Then you can store values for g and h , and evaluate again:

```
1 FIX 9.81_m/s^2 'g' STO 100_m 'h' STO EVAL UBASE ⏏ 44.3_m/s.
```

But if you repeat the exercise, this time with the unit objects already stored, you get an unexpected error:

```
'V^2/2=g*h' 'V' QUAD ⏏
```

```
+ Error:
Inconsistent Units
4:          -981_m^2/s^2
3: '∂V(V)*2*V^(2-1)/2'
2:          0_m
1:          0_m/s^2
COLCT EXPR ISOL QUAD SHOW TAYLR
```

The problem arises from the differentiation performed by QUAD as it computes the quadratic coefficients. During the course of chain-rule differentiation, an expression is evaluated one or more times, which means that variable names are replaced by their values. Try differentiating the right-hand side of the equation with respect to V :

```
'∂V(g*h)' EVAL ⏏ '∂V(g)*h+g*∂V(g)'
```


At the next evaluation, both of the remaining derivatives are zero, but the values of h and g are also substituted in the expression, so that you will be evaluating $'0*100\_m+9.81\_m/s^2*0'$. This fails because the two terms have different unit dimensions.

This problem affects ∂ , QUAD, and TAYLR applied to expressions with variables containing unit objects. There are two ways to avoid the difficulty:

- Carry out the differentiations before assigning values to the variables, or switch to a directory where the variables are undefined.
- Replace the variable names by their values before differentiating. For the variable of differentiation, separate its units from the variable.

To illustrate the second method, start again by entering the equation $'V^2/2=g*h'$. First, evaluate g and h using SHOW (section 16.4.2):

$'V^2/2=g*h'$ { V } SHOW  $'V^2/2=981\_m^2/s^2'$

Now replace V by $'v*1\_m/s'$:

{ V $'v*1\_m/s'$ } $\uparrow$ MATCH DROP  $'(v*1\_m/s)^2/2=981\_m^2/s^2'$

Solve for v :

$'v'$ QUAD EVAL  $'v=44.3'$

You can multiply this result by $1\_m/s$ to obtain V .

21.3.9 Integration

For numerical evaluation of an integral, both the integrand and the integration limits may contain unit objects. There is nothing special that you need to do to compute an integral with units, except to insure that the units of the limits are compatible with the units assumed for the independent variable:

- The units of the lower limit, determined by evaluating it numerically, are used for the integration variable at every point at which the integrand is evaluated during the approximation process. The limit's units must therefore have suitable dimensions for successful numerical evaluation of the integrand.
- The units of the upper limit must be dimensionally consistent with those of the lower limit.

- The first time the integrand is evaluated, the units of the integrand are saved. Subsequent evaluations of the integrand are converted to the dimensions of the saved units. This prevents any variation of units at different points in the integral domain from causing artificial discontinuities (integration is thus a little “smarter” than DRAW or HP Solve).
- The units of the result are the product of the integrand units and the units of the limits.

■ *Example.* Compute $\int_{0s}^{10s} (10\_cm/s + 5cm/s^2 \cdot t) dt$.

■ *Solution.*

$$' \int(0\_s,10\_s,10\_cm/s+(5\_cm/s^2)*t,t)' \rightarrow \text{NUM} \quad \Rightarrow \quad 350\_cm$$

Symbolic integration with units presents a somewhat different problem than numerical integration. There are integrals that appear to be computable but which return an error on the HP 48. Consider the integral

$$\int_a^b \frac{1}{k \cdot x + v} dx$$

where k has the dimensions s^{-1} , a , b and x are expressed in m , and v is a speed in m/s . Evaluating this on the HP 48 would return an error, because $\ln$ requires a dimensionless argument. To compute the integral correctly, you must factor the units out of the denominator before integrating.

21.4 Unit Management Idiosyncrasies

The HP 48 unit management system is not perfect, in that in a few areas it compromises universality for the sake of execution speed or simplicity. This may lead to some surprises, but there are actually very few practical calculations that you can not carry out as long as you allow for the system's limitations.

21.4.1 Non-integer Unit Powers

The units in the dimension part of a unit object may be raised to arbitrary non-integer powers. CONVERT computes its conversion factor accordingly, so that you can convert between units with non-integer dimensions:

$$1\_m^{.5} \quad 1\_cm^{.5} \quad \text{CONVERT} \quad \Rightarrow \quad 10\_cm^{.5}$$

However, for the sake of dimension checking, unit powers are rounded to integers in the range -128 to $+127$. This means that while you can successfully add $1\_m^{.5}$ and $1\_cm^{.5}$, you can also add $1\_m^{.5}$ and $1\_cm^{.6}$ with no error report. The dimension checking limitation also carries over to UBASE, and UFACT. Neither of these commands will return meaningful results for units containing non-integer exponents.

This limitation is a deliberate design choice intended to make unit object arithmetic as fast as possible. Representing base unit powers with small integers provides a considerable efficiency compared with using floating-point numbers. The loss of ability to handle non-integer exponents in all circumstances is presumed to be a modest price to pay for the speed gain, given that the great majority of physical problems involves integer powers of units.

There are some cases where non-integer exponents appear in the course of a calculation even though the final result has only integer powers. You can perform such calculations on the HP 48, as long as you take care to simplify quantities to base units whenever possible, before any conversions are done. For example, evaluating

$$2\_atm \ 8\_lb/gal \ * \ \sqrt{\ } \ UBASE \Rightarrow 13937.812022\_kg^2/(m^3*s),$$

which is dimensionally incorrect. The UBASE should be applied before the square root to obtain the correct result:

$$2\_atm \ 8\_lb/gal \ * \ UBASE \ \sqrt{\ } \Rightarrow 13937.812022\_kg/(m^2*s).$$

21.4.2 Temperature

The temperature units $^{\circ}F$ and $^{\circ}C$ have ambiguous meanings that prevent them from fitting easily in the unit management system. They can be interpreted as units of thermodynamic temperature; with that definition, $^{\circ}F$ and $^{\circ}C$ are interchangeable with $^{\circ}R$ and K , respectively. Conversions associated with this interpretation work properly, such as converting J/K to $BTU/^{\circ}F$.

An alternate meaning for these units is as points on a thermometer scale. These are not units in the usual sense, and intermixing thermometer points with thermodynamic temperature units leads to problems. The HP 48 unit conversion scheme is based on multiplicative conversions. This means that a conversion between two dimensionally consistent units can be achieved with a simple multiplication. $^{\circ}F$ and $^{\circ}C$ considered as thermometer points are not expressible as simple multiples of the base unit K . Additive constants are needed as well, so the ordinary conversion scheme is inadequate for these units. The thermometer points do not, for example, obey ordinary arithmetic rules. For

example, multiplication is not very meaningful--returning 20°F as the value of $2 \times 10^{\circ}\text{F}$ is not a very useful result. Also, addition is not commutative. If you write $5^{\circ}\text{C} + 10^{\circ}\text{F}$, you probably mean "increment a thermometer reading of 5°C by 10°F (i.e. $5/9 \times 10^{\circ}\text{C}$).” But this does not yield the same result as $10^{\circ}\text{F} + 5^{\circ}\text{C}$.

The HP 48 does make some provision for the use of $^{\circ}\text{F}$ and $^{\circ}\text{C}$ as thermometer points. In particular, when CONVERT is applied to two unit objects whose dimension parts are each any one of the four temperature units, the conversion accounts for the additive constants. This allows CONVERT, including the automatic CONVERT built into the TEMP menu keys, to be used for simple temperature conversions:

$98.6_{\circ}\text{F} \quad 1_{\circ}\text{C} \quad \text{CONVERT} \quad \Rightarrow \quad 37_{\circ}\text{C}.$

The HP 48 computes this result by converting the first unit object to kelvins, then converting back to the units of the second object, including additive constants in both conversions. For better or worse, a similar logic is applied to addition and subtraction. Thus we have the somewhat surprising result

$10_{\circ}\text{C} \quad 5_{\circ}\text{C} \quad + \quad \Rightarrow \quad 288.15_{\circ}\text{C}.$

Both arguments are converted to *kelvins*, added, then converted back to $^{\circ}\text{C}$. This result is justifiable, but it is hardly useful. The consolation is that the physical laws used in practical problems are generally written in terms of absolute temperatures, and the HP 48 units K and $^{\circ}\text{R}$ always work in calculations related to these laws.

A safe strategy when you do want to enter temperatures in $^{\circ}\text{C}$ or $^{\circ}\text{F}$ is to apply the UBASE function to the temperatures prior to any further calculations. For example, in problems of thermal expansion, you might have an expression of the form

$$\text{Length} = L_0 * (1 + \alpha * (T - T_0)),$$

where L_0 is the length of a bar at the reference temperature T_0 , and α is the coefficient of expansion. Calculations with this formula will not give correct results if the temperatures are entered in $^{\circ}\text{C}$ or $^{\circ}\text{F}$, unless it is modified to

$$\text{Length} = L_0 * (1 + \alpha * (\text{UBASE}(T) - \text{UBASE}(T_0))).$$

Now the temperatures may be entered in any units. α may also be expressed in units K^{-1} , $^{\circ}\text{R}^{-1}$, $^{\circ}\text{F}^{-1}$, or $^{\circ}\text{C}^{-1}$. Note that applying UBASE to the temperature difference, e.g. $\text{UBASE}(T - T_0)$, does not solve the problem, because the subtraction is performed before the conversion to base units.

21.4.3 Angle Units

The angle units in the ANGL menu are *dimensionless*, which presents another problem for the unit management system. Consider the radians unit *r*. A typical use of radians is in formulae such as the definition of angular speed, $\omega = v/R$. Here v has dimensions $length \cdot time^{-1}$ and R has dimension $length$, and so ω must have the dimension $time^{-1}$. But instead we like to assign ω the dimensions $radians \cdot time^{-1}$ to tie it to angular motion. Evidently, $UBASE(1_r)$ must have the value 1 (dimensionless) to make calculations like $\omega = v/R$ return correct results. But if this is the definition of 1_r , then $1_r/s$ would convert to 1_{Hz} , which is certainly wrong by a factor of $1/2\pi$. In order to make this conversion correct, $UBASE(1_r)$ must have the value $1/2\pi$.

This contradiction is insoluble within the scope of the HP 48 unit management system, which can not deal consistently with dimensionless units that appear or disappear depending on what the visual appearance of an expression is supposed to be. The HP 48 makes the somewhat arbitrary choice that $UBASE(1_r)$ is $1/2\pi$ (actually .159154943092). With analogous definitions for $1_{\circ}$ ($1/360$), 1_{grad} ($1/400$), 1_{arcmin} ($1/21600$), and 1_{arcs} ($1/1296000$), conversions among all these units and Hz give correct results.

To deal with formulae like the angular speed example where angle units can appear and disappear, you can construct expressions in which each term must explicitly or implicitly be uniform in angle units--specifically, each should have the same power of angle units. For example, if the angular frequency equation is represented by the object ' $\omega = (v/R) * 1_r$ ', you can use it with values of ω expressed in *r*, Hz, or any other units *angle/time* (but you can't use units of $1/time$). Alternatively, you can leave the angular units out entirely, since they do not contribute anything to the computation.

By extension from the plane angle case, and for the benefit of conversions involving photometric units (see the next section), the solid angle unit *sr* is defined such that $UBASE(1_{sr})$ returns $1/4\pi$.

21.4.4 Photometric Units

The photometric units included in the LIGHT menu provide various measures of *luminous* power. The term *luminous* refers to the incorporation of the spectral response of the idealized human eye into the units, so that they are not directly convertible into ordinary units of power. For example, *luminous intensity* is the luminous power emitted per steradian by a body. The base unit of luminous intensity is the *candela*, abbreviated *cd*. It is defined as the luminous intensity of a body emitting $1/683$ watt/sr of monochromatic radiation at $5.4 \cdot 10^{12}$ Hz (5550 angstrom), which is the wavelength of maximum sensitivity of the eye). Because of the spectral response factor, it takes more power at any other wavelength to produce the same luminous intensity.

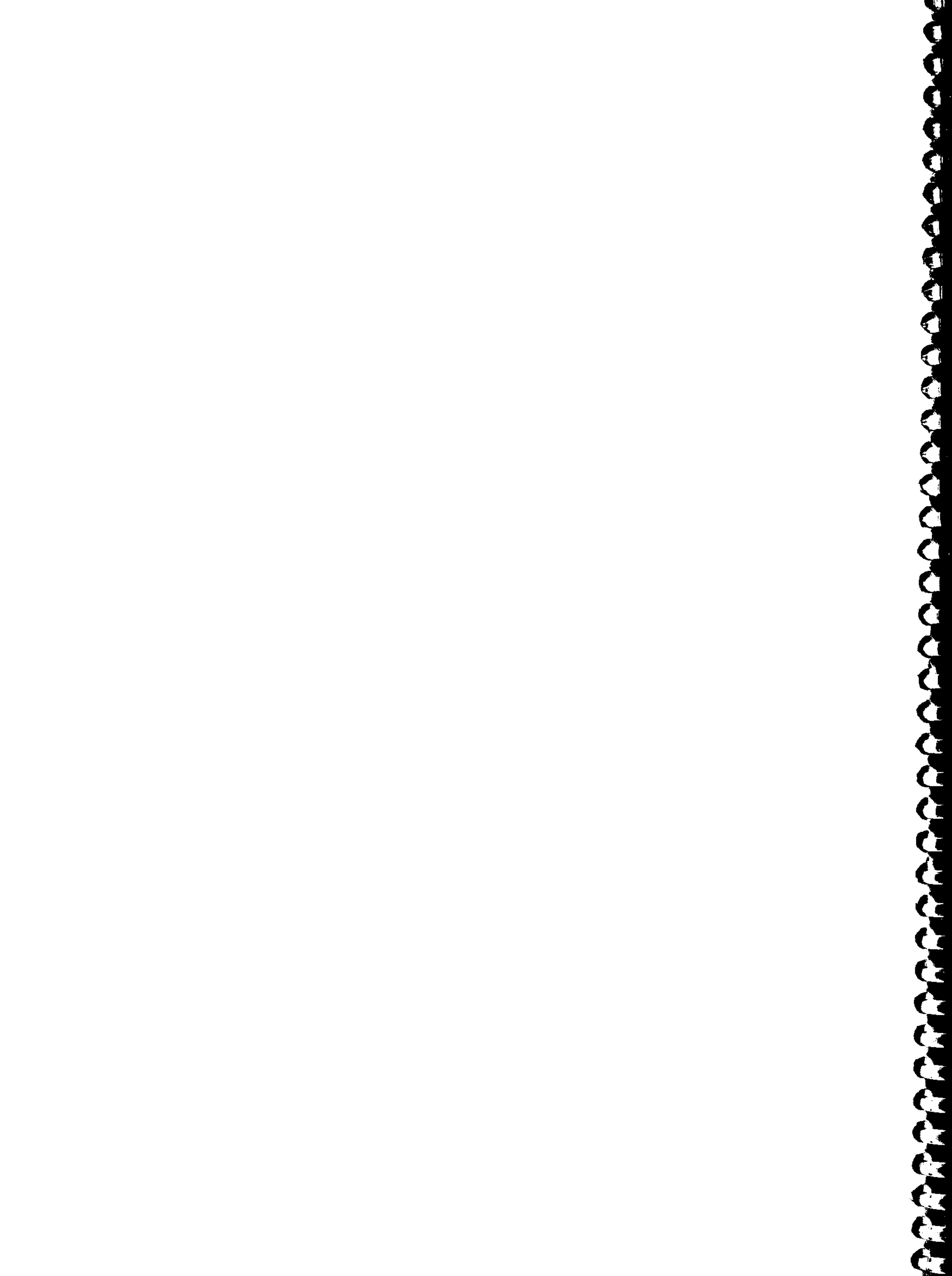
The fact that the definition of candela includes solid angle makes conversions among related units problematic for the HP 48 unit management system, again because of the elusive nature of angle units. For example, a *lumen* (*lm*) is a measure of *luminous flux*, which is the power emitted by a surface per unit solid angle. A *lumen* is defined as the flux from 1 *cd* into 1 *sr*. However, if you execute 1_lm 1_cd CONVERT, you will obtain 7.95774715459E-2_cd. The numerical magnitude of the result is $1/4\pi$; this factor is included to cancel the $1/4\pi$ that is included in the definition of the HP 48 sr unit. That is, a more proper conversion between *lumens* and *candelas* should include an explicit solid angle factor:

1_lm/sr 1_cd CONVERT  1_cd.

A conversion between cd and lm without including solid angle is essentially invalid, but the HP 48 can not detect this because solid angle is dimensionless.

The LIGHT menu also includes three units of luminous *surface brightness*--*stilb* (sb), *lambert* (lam), and footlambert (flam), and three units of *illuminance*--*phot* (phot), *lux* (lux), and *footcandle* (fc). Surface brightness is the power emitted by a unit surface area per unit solid angle, whereas illuminance is the power received by a unit surface area. Dimensionally, the two quantities are related to each other in the same manner as luminous intensity and luminous flux. Therefore conversions between the two types of units must include a solid angle factor, either multiplied times the surface brightness unit, or divided into the illuminance unit:

1_stilb*sr 1_ph CONVERT  1_ph.

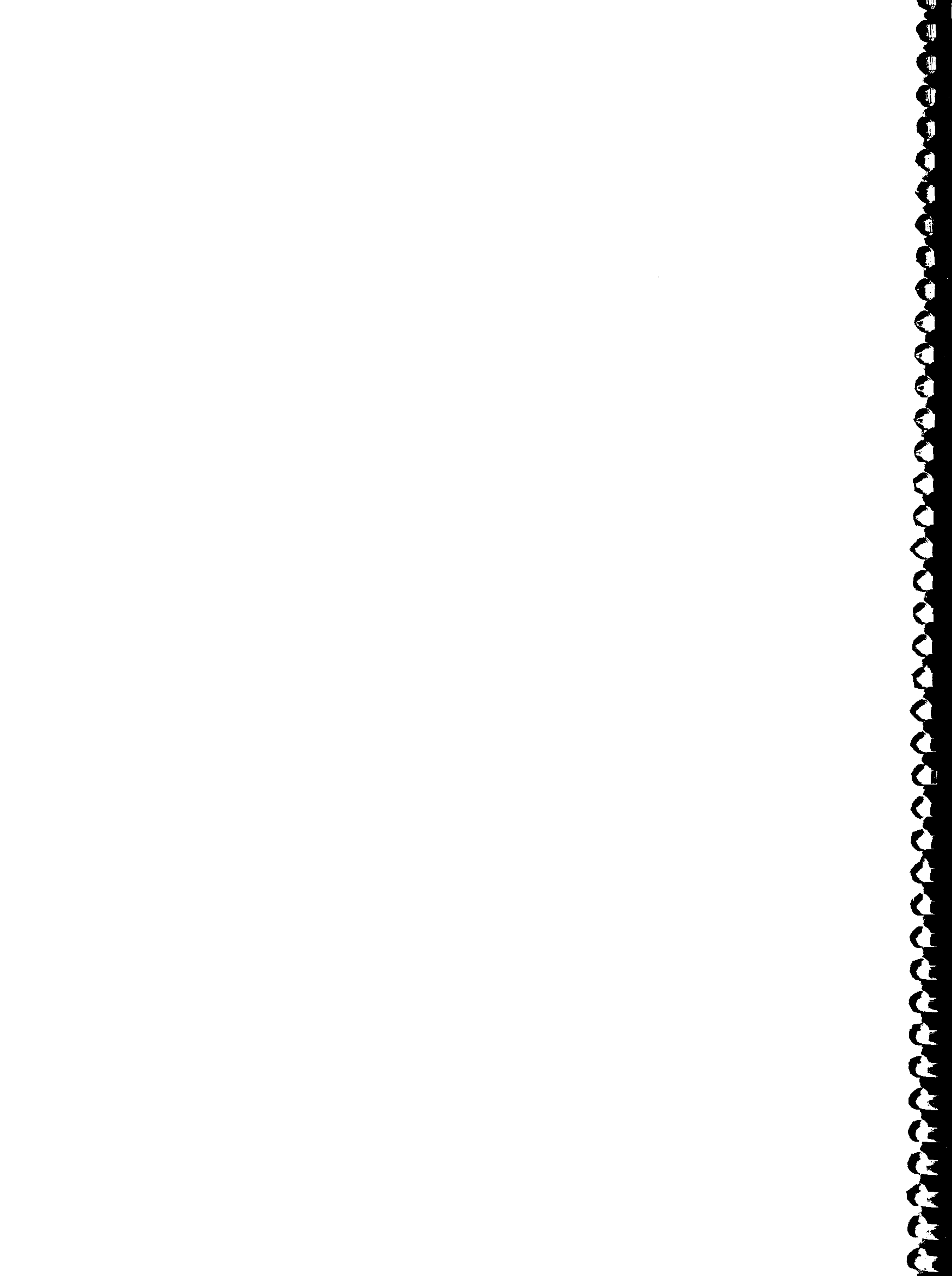


Program Index

ADD&T	Add Date and Time	627
ADDV	Concatenate Vectors	300
AGXOR	Animate with GXOR	264
APLY1	Apply Program to 1 Symbolic Array	304
APLY2	Apply Program to 2 Symbolic Arrays	304
APVIEW	Animation with PVIEW	268
AREPL	Animation with REPL	268
ASN41	ASN HP 41-style	182
ASTO	Animation with STO	268
BINCALC	Binary Integer Calculator	192
BOUNCE	Bouncing Ball Demo	269
BS?	Bit Set?	178
C1ADD1	Column 1 Added with 1's	661
C1SPLIT	Column 1 Split	660
CB	Clear Bit	178
CDIST	χ^2 Probability Density	677
CEQN	Characteristic Equation	310
CHARDISP	Display HP 48 Characters	258
CHIMES	Hour Chimes	636
CHKINPUT	Prompt and Check Input	331
CHVAR	Change of Variables	614
CI	Cosine Integral	369
CINT	Circle in a Triangle	248
CLADD1	Column Last Added with 1's	661
CLSPLIT	Column Last Split	661
COSSUM	Cosine of a Sum	573
COUNT4	Count in 4 Ranges	229
CROSSF	CROSS Function	289
CSC	Cosecant Function	537
DATENAME	Create a Name from the Current Date	107
DD&T	Delta Days and Time	627
DELCHIMES	Delete Chimes Alarm	638
DELOLDALARMS	Delete Old Alarms	638
DELROW	Delete a Matrix Row	280
DFACT	Double Factorial	234
DIM	Symbolic Array Dimensions	302
DOTF	DOT Function	288
DOW	Day of Week	625
DRAWEBAR	Draw with Error Bars	655
DRAWPIX	DRAW using PIXON	271
FDER	Formal Derivative	593
FDIST	F-Distribution Probability Density	677
FIB	Fibonacci Series Generator	296
FIND	Find a Variable	121
FRACALC	Fraction Calculator	194
FRAME	Frame the Graph Screen	273

FUNOF?	Function Of?	607
GCD	Greatest Common Divisor	238
GSAMP	Graphics Samples	261
GSORT	General-purpose Sort	298
INDEF	Indefinite Integral	600
INFSUM	Compute an Infinite Sum	349
INTEVAL	Integrand Evaluation	603
INTSAMP	Integration Samples	610
ITDRAW	Interruptible Truth DRAW	503
JULIA	Julia Plot Utility	505
KEEP	Keep n Objects	78
KEYHALT	Halt if a Key is Pressed	340
KEYTIME	Wait a Specific Time for a Key	340
LCM&GCD	LCM and GCD	345
LINEAR?	Linear Form Test	606
LPWR	List to Power	584
LSFIT	Least-Squares Fit	658
MAND.EQ	Mandelbrot Current Equation	504
MANPLOT	Manual Plot Program	481
MEDIAN	Medians of Σ DAT	646
MINFSUM	Compute an Infinite Sum (Monitor)	349
MINISTK	Small-font Stack Display	266
MINL	Minimum of a List	322
MINOR	Minor of a Determinant	280
MNDROP	DROP m through n	78
MNORM	Modified Normal Distribution Generator	357
MOVE	Move a Variable	122
MSGSHOW	Show Messages	343
NDIST	Normal Probability Density	677
NEXTDAY	Next ...Day Alarm Utility	634
NLFIT	Non-Linear Fit	666
NORM	Normal Distribution Generator	357
NSEGINT	Numerical Segment Integral	613
N→S	Numeric to Symbolic	303
OLABEL	Object Labeling Utility	259
PADD	Polynomial Add	580
PDIVD	Polynomial Divide	582
PEXPR	Polynomial Expression	582
PLIST	Polynomial to List	584
PMUL	Polynomial Multiplication	581
PNEG	Polynomial Negate	581
POIS	Poisson Generator	356
POLY	Polynomialize Expression	585
POLYF	Polynomializing Function	536
POLYFIT	Polynomial Fit	664
PPWR	Polynomial Power	583
PREST	Plot Restore	455
PRGINT	Program to Integral	615
PRIMES	Find Prime Numbers	360
PROMPTCONT	Prompt with CONT Display	330
PSAVE	Plot Save	456
PSUB	Polynomial Subtract	580

PTINFSUM	Infinite Sum from Previous Term	368
QU	Quadratic Root Finder	319, 326
RC→R	Real/Complex to Real	227
RMINL	Recursive Minimum of a List	353
→RPN	Convert to RPN	534
SADD	Add Symbolic Arrays	305
→SA	Stack to Symbolic Array	302
SA→	Symbolic Array to Stack	302
SB	Set Bit	177
SCDRAW	Save Coordinates and Draw	482
SCOF	(Unsigned) Symbolic Cofactor	308
SDET	Symbolic Determinant of a Matrix	308
SEGINT	Segment an Integral	612
SETCHIMES	Set Chime Alarm	637
SETTICKS	Store Tick Spacings	463
SI	Sine Integral	369
SIMEQ	Simultaneous Equations	364
SKETCH	Sketch Lines	273
SLVDRAW	Solve and Draw	483
SMINOR	Minor of a Symbolic Matrix	309
SMS	Scalar Multiply Symbolic Arrays	306
SMUL	Multiply Symbolic Arrays	306
SORT	Sort a List in Increasing Order	297
ΣPATTERNS	Summation Patterns	597
SSUB	Subtract Symbolic Arrays	305
STAR	Draw a Star	272
STRN	Transpose Symbolic Array	305
SUBCOL	Subtract Columns	282
SUM4	Sum $1/x^4$	347
SUMTERM	Compute an Infinite Sum from TERM	347
S→N	Symbolic to Numeric	303
TAYLRX0	Taylor's Polynomial at x_0	595
TDIST	t Probability Density	677
TICKAXES	Draw Axes with Ticks	464
TIMED	Timed Execution	350
TOCOSSUM	To the Cosine of a Sum	574
TPIX	Toggle a Pixel	272
UDIMS	Unit Dimensions	682
VANGLE	Angle Between Two Vectors	283
VARPAR	Variable δt Parametric DRAW	500
VARPOL	Variable $\delta\theta$ Polar DRAW	496
VSUM	Sum Vector Elements	234
XARCHIVE	Extended Archive	108
XFORM	Coordinate Transformation	289
XINT	Extended Integration	607
XJULIA	Julia Set Utility with Exit	506
XPTINFSUM	Infinite Sum in x from Previous Term	368
XSUM	Extended Sum	597



Subject Index

- aborting programs 316
- ABS 283
- ACK 632, 640
- ACKALL 632, 640
- acknowledge alarm 179, 631
- action 30, 32
 - user key 179
- activation 30
- add fractions 569
- alarm 627
 - acknowledge 179, 631
 - announcement 630
 - appointment 630
 - beep 179
 - catalog 135, 628, 638
 - commands 636
 - control 635
 - deletion 637, 640
 - execution object 628, 629, 636
 - index 635, 636
 - list 627, 636, 639
 - parameters 628
 - repeat interval 636
- alarm-set menu 628
- ALG annunciator 141
- alert annunciator 631
- algebraic 48, 515
 - calculator 21
 - entry mode 141, 143, 207
 - evaluation 49
 - object 3, 21, 25, 29, 30, 32, 47, 63, 200, 515, 534
 - syntax 16, 48
- algebraic/program mode 142
- ALOG 570
- alpha key action 179
- ALRMDAT 630, 632, 640
- analytic function 26, 527
- angle mode 175, 178, 286
- annunciator, alert 631
 - ALG 141
 - busy 145, 631
 - PRG 141
 - USER 180
 - 1USR 180
- applications 402
- APPLY 536
- appointment alarms 627, 640, 629, 630, 636
- arbitrary integer 530, 531, 532
 - sign 530, 531, 532
- ARC 273, 274, 475
- ARCHIVE 106
- AREA 487
- argument 16
 - disappearing 79
 - recovery 72, 87, 179, 242, 312
 - saving 87
- array 40, 277, 301
 - entry 151
- >ARRY 72, 277, 288
- ARRY-> 278
- ASCII files 215
- ASN 180, 182, 184
- assignment, key 175, 180
- associate left 565
 - right 565
- association 546, 547, 562, 565
- attach library 101
- ATTACH 102, 104
- ATTN 133, 241, 315, 339
- automatic linefeed 179
 - mode change 142
 - repeated operations 561
 - simplification 52, 57
- automating calculations 195
- AUTO 449, 453, 508, 510, 512
- AXES 454, 462, 493
- >A 565
- A-> 565
- backspace 168
- backup object 46
- Bad Argument Type 49
- Bad Argument Value 690
- Bad Guess(es) 428
- BARPLOT 510
- BAR 464, 509
- base 165
- base units 680
- BASIC 2, 32, 201, 245, 516
- beep, error 179
- BEEP 630
- best fit 653
- BESTFIT 651, 653
- BIN 178
- binary integer 40
- binary transfer 179
- binomial coefficients 670

Subject Index

- BINS 512, 644
- BLANK 262
- BOX 273, 274, 475
- branch 225
 - unconditional 225
- built-in object 59
 - program object 31
 - units 682
- busy annunciator 145
- BYTES 10, 41, 224, 318, 327, 406
- calculus 587
- CASE structure 228
- catalog, alarm 638
 - equation 421, 638
 - statistics 642
- cell 150
 - cursor 150
- CENTR 138, 457, 469
- CENT 454, 469
- CF 55, 176, 222
- %CH 590, 689
- chain-rule 587
- change of sign 430
- changing variable contents 117
- character code 38, 39
- characteristic equation 310
- checksum 327
- CHR 39
- CIRCL 475
- Circular Reference 87
- clear flag 175
- CLEAR 68, 135
- clearing 68
- clipping 275
- CLKADJ 624
- CLLCD 257, 258
- clock 621
 - setting 622
- closing subexpression 160
- CLUSR 96
- CLVAR 96, 142, 420
- CLΣ 642
- code object 30
- cofactor 307
- COLCT 519, 525, 544, 545, 559, 572, 576, 579, 601, 604
- column number 116
- column vector 282
- COLΣ 507
- combinations 669
- combining RPN and algebraic 22
- COMB 669, 670
- comes due 628
- command 26, 59, 100
 - test 221, 223
- command line 63, 139, 140, 144, 146, 202, 335
- command stack 145
- comment 146, 36, 216
- common notation 17
- commutation 564
- commutative property 562
- compact format 9, 405
- complex array, MatrixWriter 151
- complex number 34
 - operators 568
 - result 35
- composite object 30, 51, 63, 289, 299
- compound unit 682
- CON 117, 120, 279, 294
- concatenation 37
- conditional 32, 221, 225
- configuration program, library 103
- conic plots 459, 488
- CONIC 464, 489
- CONJ 568
- conservative approach 519
- constant, symbolic 57, 170, 178, 211
- Constant? 429
- CONT 183, 290, 314, 316, 317, 329, 333
- continuous plot 449, 459
 - type 449
- contravariant vector 282
- control alarms 627, 640, 629, 635
- conversions, unit 684
- CONVERT 684, 686, 687, 690
- coordinate mode 285
- coordinate system 178
- coordinates, logical 269, 456
 - cylindrical polar 284-286
 - polar 34, 284-286
 - rectangular 284
 - spherical polar 284-286
- copying stack objects 70
- correlation coefficient 648
- CORR 648
- counted string 37
- counter 119
- covariant vector 282
- COV 648
- CRDIR 89
- CROSS 283, 288
- CST 185, 687
 - menu 185, 410
- current directory 89, 111, 608

- equation 414, 453
- path 89, 101, 592
- statistics matrix 642
- cursor 457
 - cell 150
 - graphic 179
 - subexpression 170
- curve filling 179, 499
 - flag 499
- custom error 242
- custom menu 185, 186, 175, 443, 687
- customization 175
- cylindrical polar coordinates 284, 286
- C→PX 270, 274
- C→R 35
- ←D 566
- data object 32
- date display 622
 - format 179, 623
 - flag 623
 - numbers 622
- DATE+ 626
- DATE 624, 626
- DATE 624
- DEBUG 317, 319
- DDAYS 626
- debugging 317
- DEC 178
- decimal digits 179
- DECR 117, 119
- default guess 433
- DEFINE 86, 205, 208, 210, 526
- defining expression 206
 - procedure 11, 246, 407, 538
- definite integral 600
- definite loop 230, 234
- definition equation 478
- definition expansion 570
- definition, object 27
- DEF 570
- degrees mode 589
- degrees of freedom 676
- ←DEL 147
- DELALARM 635, 637
- deleting suspended program 316
- delimiter 27, 127, 144, 214
- DELKEYS 183
- DEL→ 147
- denominator 163
- dependent column number 647
 - variable 416, 646
- depn 454
- DEPND 460, 462, 489, 502
- DEPTH 293
- der 590
- derivative 436, 538, 587
- DETACH 104
- determinant 307
- differentiation 436, 538, 587
 - with units 693
- digit-group commas 34
- digitize 430
- dimensions 679
 - consistency 688
- directory 45, 60, 88, 420
 - current 89, 111
 - PURGE 94
- Directory Not Allowed 94, 95
- Directory Recursion 95
- disappearing argument 79
- discrete plot types 455, 449
- DISP 258, 332, 344
- display 255
 - freeze 258
 - graphics 123, 124, 260
 - mode 525
 - standard 126, 256
- distribute-prefix-operator 567
- distribution 546, 547, 566, 568
 - function 673
- divide bar 163
- DO loop 236
- DO 238
- DOERR 100, 104, 241, 242
- DOT 282, 288
- DOT+ 473
- DOT- 474
- double guess 430, 431
 - integral 616
 - inverse 568
 - negate and distribute 567
 - negative 568
 - quote 36, 62
- double-space mode 179
- double-where 599
- DRAW 53, 249, 271, 417, 449, 452, 453, 459, 461, 464, 469, 478, 480, 488, 489, 493, 495, 497, 498, 501, 502, 507, 509, 511, 619, 654, 692
- DRAX 452, 461, 463
- DROP 68, 134
- DROP2 69
- DROPN 69
- DUP 70, 82, 134
- DUPN 72

Subject Index

- D→ 566
- ECHO 148
- EDEQ 413
- EDIT 76, 147, 154, 311
- editing 554
 - menu 147, 154
 - program 311
- edit/view 148
- ELSE 225
- else-sequence 225
- empty 79
- END 225
- endless execution 61, 87
- ENG 608
- ENTER 19, 26, 72, 134, 140
 - explicit 140
 - implicit 140, 145
- α ENTER 191
- β ENTER 191, 265
- entry mode 141, 142, 187, 189, 550
 - algebraic 141, 143, 207
- entry, array 151
- entry, text 128
- ENTRY 143
- environment 12, 126, 132, 408
 - plot 12, 126, 465
 - standard 12, 408
- EQ 118, 410, 413, 414, 420, 450, 453, 478, 509, 642
- EQ→ 527
- equality 223
 - logical 223
 - physical 223
- equation catalog 421, 638
- equation 534
- EquationWriter 134, 148, 156, 266, 521
- ERASE 449, 452, 469
- ERR0 241
- ERRM 100, 104, 240, 241, 242, 316
- ERRN 240, 241, 242
- error 239
 - bar 653
 - beep 179
 - trap 239, 242
 - custom 242
- error-sequence 240
- EVAL 31, 47, 51, 55, 60, 64, 97, 114, 290, 427, 572, 589, 600
- evaluate 16, 25, 31
 - algebraic 49
- exception 243
 - action flag 243
 - overflow 178, 244, 670, 671
 - underflow 178, 244, 671, 690
- exchange of arguments 69
- execution 25, 30, 31, 60, 116
 - by address 64
 - endless 87
 - local name 61
 - numerical 52, 55, 56, 58, 598
 - postponed 146
 - preventing 31
 - symbolic 52, 58, 225
- exit 234, 225
- EXP^ 570
- EXPAN 50
- expansion 541, 547
- EXPAN 525, 544, 545, 547, 559, 572, 576, 601
- EXPFIT 647
- explicit ENTER 140
- exponent 33
- exponentiated 165
- EXPR 170
- EXPR= 418, 419, 428, 445
- expression 16, 48, 534
 - defining 206
 - manipulations 541
 - rearrangement 521
 - simplification 545
- EXP 570
- extensive manipulations 544
- Extremum 429, 438
- EXTR 436, 486
- F(X) 484
- F-test 675
- FACT 671
- factorial 670, 671
- factors 546, 563
- fast catalog 179
- FC? 223
- FC?C 177, 223
- FCN menu 436, 483
- filled curves 478
- FINDALARM 638
- first guess 429
- FIX 526, 608
- flag 55, 175, 220, 333
 - 2 56
 - 3 56, 428, 598
 - 15 285
 - 19 36
 - 20- -26 243-244
 - 30 478, 492
 - 31 499

- 41 622
- 42 622
- 42 623
- 51 34
- 53 539, 559, 565, 574
- 55 73, 243
- 60 129
- 61 129, 180
- 62 180
- clear 175
- exception action 243
- stack 177
- system 175
- user 176, 222
- floating-point 33
- font 265
- FOR 230
- FOR...NEXT 246, 230
- FOR...STEP 233, 246
- formal derivatives 592
- formal variable 61, 249
- format, 24-hour 179
 - compact 9
 - date 179
 - decimal number 179
 - linear 156
- FORM 559, 561
- FORTH 5, 67
- fraction mark 179
- FREE 105
- freeze display 258
- FREEZE 257, 329, 332, 344
- FS 222
- FS? 223
- FS?C 177, 223
- function 16, 25, 26, 48, 205, 535, 598
 - analytic 26
 - execution mode 598
 - menu 132, 483
 - plot 459, 477
 - user-defined 86, 161, 196, 204, 205, 210, 288
- FUNCTION 464, 509
- gamma function 671
- GCD 345
- generations, calculator 2
- GET 116, 117, 186, 278, 281, 291, 295, 299
- GETI 116, 117, 278, 291
- global name 30, 59, 60, 61, 83, 90, 245, 327
 - variable 45, 59, 83, 111, 245, 248, 329
- GOR 263, 267, 270
- GOTO 217
- GO→ 151
- GO→ 151
- grads mode 589
- graph screen 126, 255, 260, 266, 465
- GRAPH 126, 134, 256, 257, 465, 551
- graphics 256
 - cursor 179, 466
 - display 123, 124, 260, 465
 - display of expression 156
 - object 42, 260
- greatest common divisor 345
- GROB 265
- GROB 42
- guillemets 62
- GXOR 263, 267, 270
- GXOR 263, 267, 270mantissa 33
- *H 454, 457, 472
- HALT Not Allowed 480
- HALT 109, 290, 313, 315, 316, 317, 320, 329, 332, 335, 480
- helvetica 8, 403
- HEX 178
- hidden operation 133
 - parentheses 179
- HISTOGRAM 464, 511, 512
- HISTPLOT 512
- HMS+ 626
- HMS 625
- HMS 625
- HMS→ 625, 626
- home directory 88
- HOME 91, 114, 121
- HP Solve 53, 138, 196, 201, 409, 523, 524, 530, 532, 692
 - equation entry menu 412
 - solver menu 410, 416, 418, 443, 532
 - variables menu 410
- HP 15C 608
- HP 17B 13
- HP 19B 13
- HP 27S 13
- HP 28 617
- HP 34C 409
- HP 35 2
- HP 41 2-5, 76, 140, 180, 181, 202, 216, 217, 641
- HP 65 2
- HP 71B 180
- HP 80 409
- i 35
- identity operations 559, 568
- IDN 117, 120, 279
- IERR 608, 609
- IF structure 135, 225

Subject Index

- IF 225
- IFERR structure 317, 339
- IFT 32, 51, 227
- IFTE 32, 51, 227
- IM 36
- immediate entry mode 141, 143
- immediate-execute key 141
- implicit integration 615
 - ENTER 140, 145
 - GET 117
 - parentheses 165
- implied multiplication 162
- Improper Definition 209
- IM 568
- Incomplete Subexpression 161, 551
- Inconsistent Units 684
- INCR 117, 119, 237
- indefinite loop 230, 234, 235
- independent column number 647
 - RAM 99
 - variable 415, 417, 459, 492, 646
- INDEP 417, 454, 459, 460, 478, 489, 493, 497, 502
- index for GET 116
- index wrap 179
- infinite result 178
 - action flag 243
- infix notation 17
 - operator 162
- inner product 283
- input and output 329
- input list 294
- INPUT 335, 341
- insert mode 148
- integral 598
 - derivatives 602
 - limits 600
 - numerical 608
 - programs 615
 - strategies 611
 - symbolic 598
 - with units 694
- interactive stack 74, 134, 148
- intercept 647
- Intercept: 649
- intermediate result list 296
- intermix binary and real 41
 - real and complex 36
- internal accuracy 33
- Invalid Array Element 151
- Invalid Card Data 105
- Invalid Date error 626
- Invalid PTYPE 483
- Invalid Syntax 161
- Invalid Unit 684
- Invalid User Function 210
- Inverse of power or inverse of inverse-product 568
- INV 568
- IOPAR 118
- IP 590
- IR port 179
- ISECT 485, 486
- ISOL 207, 249, 439, 519, 521, 523, 525, 526, 527, 530, 531, 541, 598, 689
- italics 8, 404
- iteration 225, 230
- KEEP 76
- Kermit message 179
- Kermit overwrite 179
- key assignment 175
 - buffer 339
 - code 341
 - format 8
 - menu 8
 - plane 180
 - shifted 8
 - type 141
 - typing 141, 142, 190
- KEY 335, 339, 341
- key-per-function 125
- keyboard 127
 - standard 126
- keycode 180, 339
- keys, format 404
- keys, menu 404
- keys, shifted 404
- KILL 316
- known variables 520
- label 217
- LABEL 462, 473, 478, 493
- last arguments 72, 87, 179, 242, 312
 - command 176
 - error message 123, 124
 - error number 123, 124
 - menu 129
 - stack 72, 144, 176
- LAST 72
- LASTARG 72, 87, 242, 460
- LCD 257, 260
- LCM 345
- least-common-multiple 345
- least-squares fit 649, 656
- LET 86
- LEVEL 76
- library 30, 96, 99, 100, 123, 124, 401

- attach 101, 102, 104
- ID 100
- title 102
- LIBRARY menu 46, 96, 105, 137
- LIBS 104
- limits of integration 600
- LINE 272, 274
- linear 598
 - coefficient 599
 - format 156
 - regression 649
- linear congruential method 672
- LINE 474
- LINFIT 647, 649
- LISP 5
- list 40, 277, 289
 - object 25, 30, 32, 51, 289
 - input 294
 - output 296
- LIST 290, 293
- LIST→ 291
- LNP1 601
- LN 570
- local memory 109, 110, 123, 124, 245, 248
 - name 30, 109, 111, 206, 208, 245, 538
 - name execution 61
 - name resolution 249
 - variable 79, 59, 61, 74, 109, 205, 206, 231, 245, 248, 315, 538
 - variable structure 111, 246
- LOGFIT 647
- logical coordinates 269, 456
 - equality 223
 - operator 221
- logical units 460
- loop 230
 - index 231, 246
 - definite 230, 234
 - DO 236
 - endless 61
 - indefinite 230, 234, 235
 - WHILE 236-238
- loop-sequence 236, 237
- LR 345, 509, 649, 651, 653
- Łukasiweicz, Jan 17
- ←M 567
- MacLaurin's formula 594
- magnitude 679
- manual operation 26
- mark, plot 468
- MATCH 572, 574, 604
- ↑MATCH 525, 572, 593
- ↓MATCH 525, 572, 605
- mathematical approximation 433
- mathematical function 207
- matrix 40, 277
- MatrixWriter 134, 148, 149, 176, 642
- MAXΣ 645
- MEAN 644
- MEM 96, 105, 184
- Memory Clear 124, 125
- memory reset 123, 124, 622
- memory, local 109, 110, 123, 124, 245, 248
 - user 59, 88
 - VAR 59
- MENU 130, 185, 333
- menu, custom 185, 175
 - exit 132
 - key label 126
 - port 99
 - screen 255
 - subexpression 170
 - VAR 83, 112, 137, 185, 245
- MERGE 105
- merging 541, 567
- message, prompt 179
- message, table 242
- minor 307
- MINΣ 645
- mode 12, 175, 408
 - coordinate 285
 - entry 141, 142, 187, 189
 - insert 148
 - numeric 225
 - program entry 76, 143, 214
 - server 138
 - user 123, 124, 129, 158, 180
- mode-dependent key 141
- Modl: 647
- MOD 590
- multiple integral 616
 - plots 454
 - roots 529
- M→ 567
- name, global 30, 59, 60, 61, 83, 90, 245, 327
 - local 30, 109, 111, 206, 208, 245
 - object 29, 58, 116, 233
 - port 97
 - quoted 62
 - resolution 90, 110, 249
 - XLIB 30, 46, 61, 103, 181
- NEG 260, 262
- negative pixel coordinates 274
- NEG 568

Subject Index

- NEW 413
- newline 151
- NEWOB 100, 297, 300
- next event 622
- non-analytic function 26
- Non-Empty Directory 94, 96
- Nonexistent Alarm 637
- normal distribution 673
- normal-sequence 239
- NOT 238
- notation 8, 403
 - common 17
 - infix 17
 - Polish 17
 - prefix 17
 - Reverse Polish 15
- NUM 32, 55, 56, 427, 598, 608, 617
- NUM 39
- numbered register 59
- numerator 163
- numerical execution 52, 55, 56, 58, 598
 - integration 608
 - mode 225
- numeric/symbolic execution 178
- NXEQ 485
- object 25, 139
 - class 32
 - composite 30, 51, 63, 289, 299
 - data 32
 - definition 27
 - entry 127
 - graphics 42, 260
 - name 29, 58, 116, 233
 - program 30
 - string 36, 144
 - symbolic 63
 - system 27
 - tagged 42, 345
 - type 27
 - unit 44
 - untagged 43
 - value 27
- Object In Use 99, 104
- object-to-grob conversion 265
- OBJ→ 35, 38, 43, 44, 69, 112, 191, 278, 288, 291, 574, 578, 605
- obtaining guesses 433
- OCT 178
- operation 25, 59
 - manual 26
 - logical 221
- ORDER 85, 112, 422, 423
- output, display 344
 - list 296
- OVER 71
- overflow 178, 244, 670, 671
- Owner's Manual 5, 402
- π 56, 211
- page, menu 129
- PARAMETRIC 464, 497
- parent 90
- parsing 144
- past due alarms 628
- path, current 89, 101
- path name 113
- PATH 89, 113
- PDIM 267, 458
- pencil-and-paper 18
- pending alarm 628
- permanent custom menu 185
- PERM 669
- PGDIR 96, 106
- physical equality 223
- PICK 70
- PICT 267, 476
- picture 255
- PIX? 272
- PIXOFF 271, 274
- PIXON 270, 274, 473
- plot cursor 466
 - environment 12, 126, 408, 451, 465
 - resolution 460
 - ranges 692
 - with units 692
- PMAX 456
- PMIN 456
- polar coordinates 34, 284
 - plots 491
- POLAR 464, 491
- Polish notation 17
- polynomial fits 662
- polynomialize 576
- polynomials 575
- port 96, 97, 99, 100, 105
 - menu 99
 - name 97
 - variable 97, 96
- POS 39, 293
- postponed execution 146
- PPAR 118, 138, 269, 270, 454, 456, 458, 459, 460, 461, 464, 507, 511, 647
- PR1 138
- precedence 17, 179, 565
- predicate operator 689

- PREDX 650
- PREDY 650
- prefix notation 17
 - operations 567
 - operators 567
 - unit 681
- preventing execution 31
- PRG annunciator 141
- principal value 178, 531
- printer port 179
- probability density 673
- probability menu 669
- problem solving 195, 198, 516, 520
- procedure 47, 200, 201
 - defining 11, 246
- program 29, 47, 52, 196, 201, 213
 - as argument 331
 - body 214, 215
 - content 47
 - definition 47
 - editing 311
 - entry mode 76, 141, 143, 201, 214
 - legibility 205
 - object 30
 - optimization 323
 - quotes 62
 - quoted 63
 - structure 47, 202, 213, 215, 219, 326
 - structure word 32, 142, 213, 219
 - suspended 313, 329
 - unquoted 63
- programming 195, 202
 - structured 203, 216, 217, 313
- prompt message 179
- PROMPT 290, 313, 315, 317, 320, 329, 330, 332, 335, 344, 480
- PRTPAR 118
- PRVAR 97, 318
- pseudo-linear fits 650
- pseudo-linear regression 650
- PTYPE 448, 489
- PURGE 86, 92, 96, 97, 99, 100, 267
 - directory 94
 - recovery 87
- PUT 117, 119, 120, 186, 278, 281, 291, 295
- PUTI 117, 119, 120, 278, 291
- PVARS 96
- PVIEW 257, 270, 345
- PWRFIT 647
- PX→ 270, 274
- Q 193
- QUAD 249, 489, 519, 521, 525, 526, 528, 530, 531, 541, 693
- quadratic equations 528
- qualifying message 412, 427, 428
- quotation mark delimiters 62
- quote, name 62
 - program 62
 - single 62
 - string 62
 - tagged object 99
- quoted argument 535, 617
- QUOTE 535, 538, 601
- Q 525, 608
- Qπ 525, 526
- radians mode 570, 589
- random number 671
- RAND 648, 649, 650, 671
- RATIO 164
- RCEQ 138, 452
- RCL 60, 97, 100, 113, 116, 222, 267
- RCLALARM 183, 628, 637, 640
- RCLF 177
- ROLKEYS 183
- RCLMENU 186
- RCWS 178
- RDM 117, 120, 278
- RDZ 672
- RE 36
- real number 33
- recentering 469
- Recover RAM 124, 125
- recovery 314
 - argument 72, 87, 179, 242, 312
 - from PURGE or STO 87
 - stack 72
- rectangular coordinates 284
- referenced 99
- register 115
 - numbered 59
 - storage 59
- reordering terms 541
- repeat interval 628
- REPEAT 237, 238
- REPL 39, 172, 264, 267, 268, 270, 292
- REPL 476, 545
- Replace RAM, Press ON 105
- reschedule 179
- RESET 451, 455
- resolution, name 110, 460, 477, 507
 - local name 249
- RESTORE 106, 107
- result 16
- RES 454, 499

Subject Index

- Reverse Polish Notation 15
- RE 436, 568
- right hand 9, 405
- RND 690
- ROLL 69
- ROLLED 69
- root 409, 693
- ROOT 201, 249, 413, 426, 438, 440, 482, 485
- root-finder 409
- ROT 70
- row number 116
 - order 277
 - vector 282
- RPL 5, 12, 21, 22, 60, 64, 82, 112, 518
- RPN 3, 15
 - command 26
 - principle 15, 50
- RULES 132, 149, 157, 172, 522, 525, 541, 544, 545, 550, 556, 557, 560, 561, 563, 572, 575
- R→C 35
- S 183
- Σ 587, 595, 601
- $\Sigma +$ 510, 660, 641
- $\Sigma -$ 641
- SAME 44, 223
- sample covariance 648
 - standard deviations 645
- saving, argument 87
- SCALE 454, 457
- scatter plot 507, 653
- SCATTER 464, 507, 654
- SCI 608
- SCL 508
- SCONJ 117, 120
- screen 126, 255
 - graph 126, 255, 260, 266, 465
 - menu 255
 - text 126, 255, 260
- Σ DAT 118, 453, 507, 510, 642
- SDEV 645
- selection arrow 421
 - environment 550
- separator 144
- sequence 9, 213, 230, 233, 405
- server mode 138
- set flag 175
- setting an alarm 628
- SF 55, 176, 222
- SHOW 425, 439, 440, 528, 537, 599, 694
- signal flag 176, 244
- SIGN 499, 690
- simple units 682
- simplification, automatic 52, 57
- SIN 175
- single guess 430, 431
- single quote 62
- single-step 316, 319
- SINV 117, 120
- SIN 589
- SI 680
- SIZE 38, 262, 267, 278, 291
- SKEY 184
- SKIP→ 147
- ←SKIP 147
- slope 647, 649
- Slope: 649
- SNEG 117, 120
- solve variables menu 197
- solver menu 197, 410, 416, 418, 443, 532
 - modifying 443
- space, in EquationWriter 161
- Σ PAR 118, 509
- spectral test 672
- spherical polar coordinates 284, 286
- square root 165
- SQ 568
- SST 316, 319
- stack 9, 19, 21, 67, 125, 405
 - diagram 10, 406
 - flag 177
 - level 29, 67
 - recovery 72, 312
 - roll 69
 - interactive 74, 134, 148
 - unlimited 77
 - display 126, 256
 - environment 12
 - keyboard 126
- standard environment 12, 408
- start 230, 231, 233
- START...NEXT 234
- START...STEP 234
- starting and stopping 313
- statistics 641
 - catalog 642
 - matrix 149, 641
 - one-variable 644
 - summary 655
 - two-variable 646
- status area 126
- STD 525
- step 233
- step-wise substitution 60
- STEQ 138, 410, 413, 452

- STO 44, 83, 87, 92, 97, 100, 267, 268, 281, 476, 642
- STO recovery 87
- STO* 118
- STO- 118
- STO+ 118
- STO/ 118
- STOALARM 630, 636, 640
- STOF 177, 183
- STOKEYS 182
- STO Σ 642, 644
- stop 230, 231, 233
- storage arithmetic 118
- storage register 59
- STR 38
- string object 36, 144
 - counted 37
- stripping tags 44
- structure 50
 - local variable 111, 246
 - program 47, 202, 213, 215, 219, 326
 - word, program 32, 142, 213, 219
- structured programming 203, 216, 217, 313
- STR→ 112
- STWS 41, 178
- SUB 39, 265, 266, 267, 292
- subdirectory 90, 420
- subexpression 50, 158, 170
 - cursor 170
 - environment 521
 - level 50
 - menu 170, 551
 - mode 550
 - closing 160
- subroutine 217, 330
- substitution 64, 523
- SUB 476, 545, 603
- summands 546
- summary statistics 655
- summation 231, 595
 - patterns 596
- suspended program 313, 329
- SWAP 69, 70, 134
- symbolic array 301
 - calculator 3
 - constant 57, 170, 178
 - execution 52, 58, 225, 598, 598, 608
 - fraction entry 163
 - integration
 - manipulations 196
 - math 197, 515
 - object 63, 515
 - solutions 439, 524
- syntax 140
 - algebraic 16, 48
- SYSEVAL 64
- system flag 175
- system halt 88, 100, 123, 124, 622
- System Internationale 680
- system object 27
 - ←T 560, 563
 - T→ 563, 560
 - %T 591, 689
- table-filling evaluation 427
- tag 42
 - stripping 44
- TAG 43
- tagged object 42, 345
 - object, quoting 99
- Taylor's polynomial 594, 617
- TAYLR 529, 579, 587, 694
- temporary custom menu 186
- term movement operations 560
- test 221
 - command 221, 223
- test-sequence 225, 236, 237
- text entry 128
 - screen 126, 255, 260
- TEXT 126, 257
- THEN 225
- then-sequence 225
- 3D 287
- tick marks 463
- ticking clock 179, 621
- ticks 62, 622, 624, 636
- TICKS 623
- time arithmetic 625
 - menu 621
 - numbers 625
 - number 622
 - system 621
 - value of money 409
- TIME 623, 624
- TIME 623, 624
- time-adjust menu 623
- time-set menu 623
- TLINE 272, 274, 474
- TMENU 130, 186, 333
- Too Few Arguments 210
- transformation 555
- trigonometric functions 570, 691
- triple guess 430, 431
- TRN 117, 120, 279
- TRNC 690
- truth plot 501

Subject Index

- truth-valued function 501
- TRUTH 464, 501
- TSTR 624
- TVAR 121, 122
- TVM 409
- 24-hour format 179, 622
- 2D 35, 287
- two-variable statistics 646
- type number 27
- TYPE 27, 44, 52, 63
- type, object 27
- type-ahead 339
- typing 188
 - key 141, 142, 190
- UBASE 224, 442, 682
- UFACT 224, 690
- Unable to Isolate 527
- unconditional branch 225
- Undefined Local Name 538
- Undefined Name 53, 573
- underflow 178, 244, 671, 690
- uniformly distributed 671
- unit built-in 682
 - conversions 684
 - in custom menu 687
 - differentiation 693
 - integration 694
 - magnitude 44
 - management 44, 679
 - menu 685
 - object 44, 679
 - plotting 692
 - prefixes 681
 - simplifications 689
- UNIT 684
- unknown variable 415, 417, 520
- unlimited stack 77
- unquoted program 63
- untagged object 43
- UNTIL 236
- UPDIR 90
- upper-tail distribution function 673
- user annunciator 180
 - flag 176, 222
 - key action 179
 - key assignment 180
 - memory 59, 88
 - mode 123, 124, 129, 158, 180
 - 1USR 180
- user-defined derivatives 590
- user-defined function 86, 161, 196, 204, 205, 210, 288, 538, 590, 592, 600
- user-defined units 683
- UTPC 674, 675
- UTPF 674, 676
- UTPN 674
- UTPT 674, 675
- UVAL 684
- V2 35, 287, 288
- V3 287, 288
- value, object 27
- VAR 645
- VAR menu 83, 112, 137, 185, 245, 410, 421, 532
- VAR memory 59
- variable 16
 - formal 61, 249
 - global 45, 59, 83, 111, 245, 248, 329
 - local 79, 59, 61, 74, 109, 205, 206, 231, 245, 248, 315
 - port 97, 96
- VARS 121, 122
- VEC 154
- vector 40, 277, 282
 - contravariant 282
 - covariant 282
- vectored ENTER 144, 145, 179, 190
- vertical lines 478
- VIEW 76
- VISIT 76, 147, 311
- VTPE 27, 52, 122
- V→ 35, 287
- *W 454, 457, 472
- WAIT 341
- where 168, 599, 617
- WHILE 238
- WHILE loop 236-238
- ←WID 150
- WID→ 150
- wild card names 572
- wordsize 41
- χ^2 distribution 675
- $x_{\max}$ 454
- $x_{\min}$ 454
- x_{axes} 454
- XCOL 453, 507, 509, 511, 512, 646
- XLIB name 30, 46, 61, 103, 181
- XRNG 454, 457
- XROOT 160, 166, 690
- $y_{\min}$ 454
- $y_{\max}$ 454
- y_{axes} 454
- YCOL 453, 507, 646, 649
- YRNG 454, 457
- Zero 419, 428

zoom 470
 ZOOM 469, 471
 Z-BOX 470

Delimiters and Punctuation

" " 36, 62
 # 41
 [] 40
 <<>> 213, 214
 :: 43
 _ 44
 ' ' 62
 {} 40
 < 24, 285
 @ 36, 146
 → 206, 210, 246, 290
 ? 687
 ☞ 9, 405

Functions

+ 262
 ! 671
 % 590
 ∫ 596, 598, 608
 ∂ 538, 587, 590, 694
 √ 165
 | 160, 168, 599
 = 224
 == 223, 224
 ≠ 223
 > 223
 < 223
 ≤ 223
 ≥ 223
 ≠ 223

RULES Operations

←A 565
 A→ 565
 AF 569
 COLCT 572
 ←D 566
 D→ 566
 DINV 568
 DNEG 568
 E^ 570
 E() 570
 L* 570
 L() 570
 ←M 567

Subject Index

M→ 567
←T 563
T→ 563
TRG* 570
→TRG 570
(← 566
→) 566
↔ 565
() 566
→() 567
-() 567
1/() 568
+1-1 569
/1 569
*1 569
^1 569



HP 48 Insights

II. Problem-Solving Resources

The HP 48S/SX Scientific Calculators provide powerful symbolic and numerical computing features that are applicable to a wide range of problems. There is special emphasis on six application areas, with special prompting and operating environments to facilitate interactive manual calculations. *HP 48 Insights II* focuses on these areas, which are indicated by the legends above the top six keys of the number pad: SOLVE, PLOT, ALGEBRA, TIME, STAT, and UNITS.



This book is the second volume of a two-part series by Dr. William Wickes on the operation and application of the HP 48. *Part I* concentrates on the underlying unified principles of HP 48 operation, and the tools and techniques for programming the calculator, including calculation methods, object storage, display management, and customization. *Part II* focuses on problem-solving in the six special application areas, including discussions of design motivations, "how-to" descriptions and examples for interactive operation, and methods for extending the built-in functionality for custom applications. More than 60 programs are included as examples and extensions, including two multi-program systems for polynomial manipulations and least-squares fitting.

Part II of *HP 48 Insights* is self-contained, and can be appreciated by anyone with a rudimentary knowledge of the HP 48. *Part I* is not a prerequisite, although many of its principles and methods are applied in *Part II*. Together, the two volumes constitute the most comprehensive and effective exposition available of the principles and methods of the remarkable HP 48.

Chapter Headings for Part II:

13.	Introduction to Part II	401
14.	HP Solve	409
15.	Plotting	447
16.	Symbolic Objects and Solutions	515
17.	Expression Manipulations	541
18.	Calculus	587
19.	The Time System	621
20.	Statistics	641
21.	Unit Management	679

Scan Copyright ©
The Museum of HP Calculators
www.hpmuseum.org

Original content used with permission.

Thank you for supporting the Museum of HP
Calculators by purchasing this Scan!

Please to not make copies of this scan or
make it available on file sharing services.