

Compute $\int_{-\sqrt{2}}^{\sqrt{2}} dx \int_{x^2}^{4-x^2} dy$

HP-28 Insights

Principles and Programming of the HP-28C/S

IGL

« 4 ROLL →NUM 4 ROLL

→NUM 3 →LIST ACC J

IF 0 <

THEN

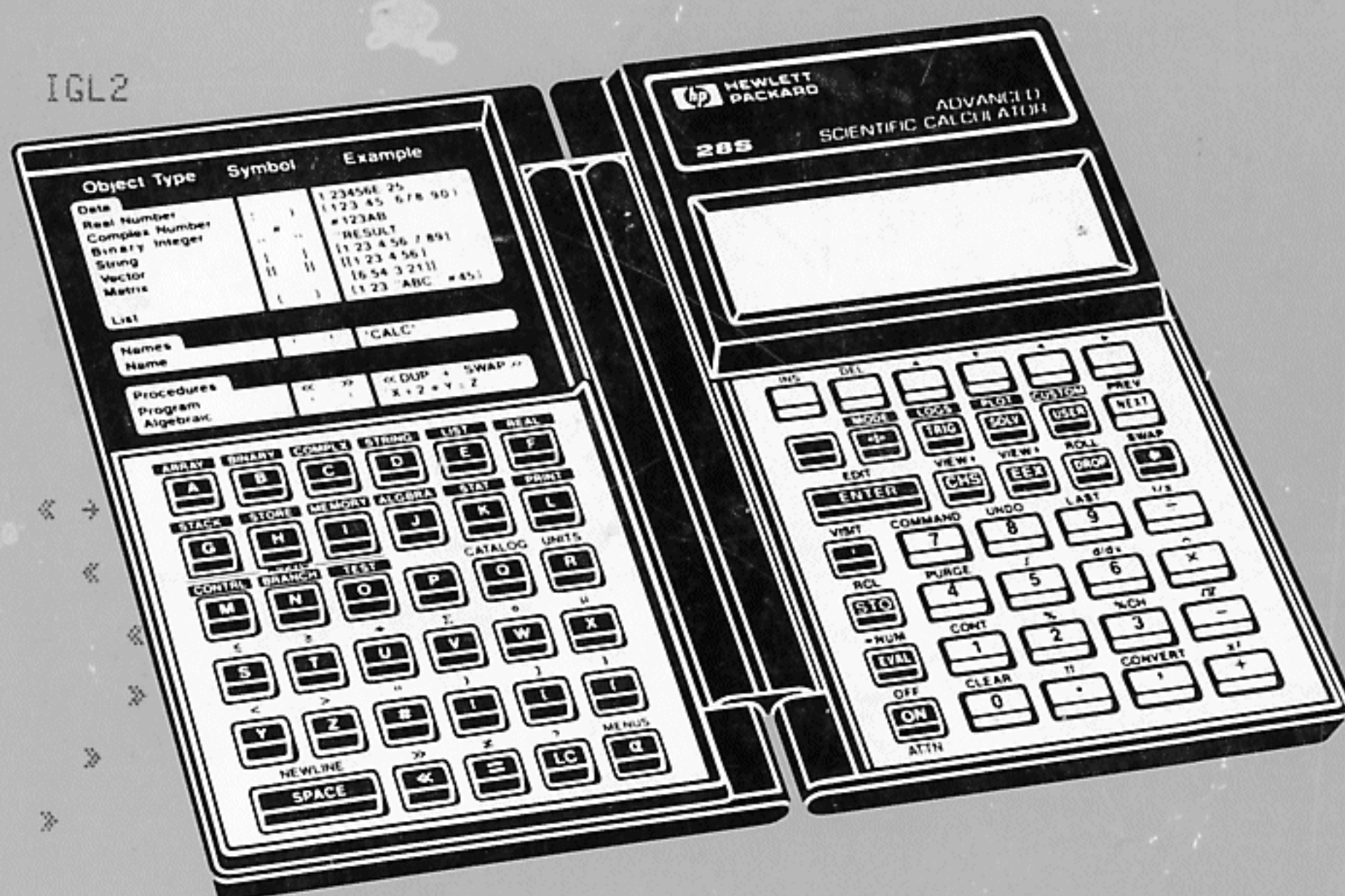
"Bad Integral" 1

DISP

END

»

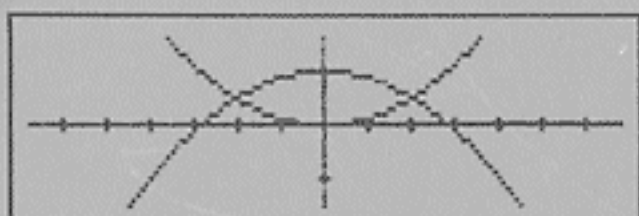
IGL2



'-√2' '√2' 'X' 'X^2' '4-

X^2' 1 'Y' IGL2 ENTER

1: 7.54247233265



William C. Wickes

HP-28 Insights

Principles and Programming of the HP-28C/S

William C. Wickes

*Larken Publications
4517 NW Queens Avenue
Corvallis, Oregon 97330*

9/1/88

Copyright © William C. Wickes 1988

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, May 1988

Acknowledgements

I thank my wife, Susan, and my children, Kenneth and Lara, for their help in the preparation of this manuscript. I am also grateful to Robert Moore for his encouragement and assistance.

Thanks also to the originators: Bob, Charlie, Gabe, Laurence, Max, Paul and Ted, who turned ideas into silicon.

To Susan



CONTENTS

1. Introduction	1
1.1 Some History	1
1.1.1 HP-28C and HP-28S	3
1.2 About This Book	4
1.3 Notation	7
1.4 Easy to Use or Easy to Learn?	9
2. Understanding RPN	10
2.1 The Evaluation of Mathematical Expressions	11
2.2 Calculator RPN	14
2.3 HP-28 RPN	16
3. Objects and Execution	19
3.1 Operations	19
3.2 Objects	21
3.2.1 Operations as Objects	23
3.3 Execution and Evaluation	23
3.3.1 When are Objects Executed?	24
3.4 Data Objects	25
3.4.1 Real Numbers	25
3.4.2 Complex Numbers	26
3.4.3 Binary Integers	27
3.4.4 Arrays	28
3.4.5 Strings	28
3.4.6 Lists	29
3.5 Procedure Objects	29
3.6 Name Objects	30
3.7 Quoted Names	33
3.8 Quotes in General	33
3.9 EVAL	35
3.10 SYSEVAL	35
3.11 The Meaning of ENTER	36
3.11.1 Key Actions and Entry Modes	38
3.11.2 Controlling the Entry Mode	40
3.11.3 ENTER in Detail	42
4. The HP-28 Stack	46
4.1 Clearing the Stack	47
4.2 Rearranging the Stack	48
4.2.1 Exchanging Two Arguments	48
4.2.2 Rolling the Stack	48

4.2.3	Copying Stack Objects	49
4.2.4	How Many Stack Objects?	50
4.3	Recovering Arguments	51
4.4	An Example of HP-28 Stack Manipulations	53
4.4.1	The Easy Way: Local Variables	56
4.5	Managing the Unlimited Stack	57
4.5.1	Stack Housekeeping	57
4.5.2	A Really Empty Stack	59
4.5.3	Disappearing Arguments	60
4.5.4	Postfix Commands	61
4.5.5	Stack-lift Disable	63
4.6	HP-28 Translations of HP-41 Stack Commands	65
5.	Variables	66
5.1	Creating Global Variables	67
5.2	Recalling Values	68
5.3	Altering the Contents of Variables	69
5.3.1	HP-28 Storage Arithmetic	70
5.3.2	Additional Storage Commands	71
5.4	Purging Variables	72
5.5	Grouping Variables	73
5.6	The USER Menu	74
5.7	HP-28S Directories	74
5.7.1	Name Resolution in Detail	78
5.7.2	Directories as Variables	80
5.7.3	USER Memory Utilities	82
6.	Problem Solving	87
6.1	The Solver	88
6.2	User-Defined Functions	89
6.3	Symbolic Math	90
6.4	Programs	90
6.5	Summary	93
6.6	Memory Limitations of the HP-28C	93
6.7	HP-28S Memory	94
7.	The Solver	95
7.1	Basic Solver Operation	96
7.2	Interpreting Results	99
7.3	Independent, Dependent, and Unknown Solver Variables	100
7.4	First Guesses	101
7.5	How Many Guesses?	102
7.5.1	Examples Using $x(x-2)(x+2) = 0$	103

	7.5.2	Solver Guess Summary	104
7.6		Obtaining Guesses	104
	7.6.1	Single Guesses	105
	7.6.2	Double Guesses	107
	7.6.3	Triple Guesses	107
7.7		Finding Extrema	108
	7.7.1	Using the Derivative	109
7.8		Evaluating Rather than Solving	111
	7.8.1	Using ISOL with the Solver	112
		7.8.1.1 Limitations of ISOL	112
7.9		Secondary Results	113
7.10		Differences between the HP-28 and other HP Solvers	114
8.		User-Defined Functions	117
	8.1	User-Defined Function Structure	118
	8.2	User-defined Functions as Mathematical Functions	120
	8.3	User-defined Functions Defined by Programs (HP-28S)	121
	8.4	Beyond User-defined Functions	122
	8.5	Additional Examples	122
		8.5.1 Permutations and Combinations	122
		8.5.2 Geometric Formulae	123
9.		Symbolic Math	125
	9.1	The Nature of Algebraic Objects	129
		9.1.1 Expression Structure	130
	9.2	Function Execution	132
		9.2.1 Automatic Simplification	132
		9.2.2 Symbolic and Numerical Evaluation; $\rightarrow$ NUM	133
	9.3	Symbolic Constants	134
		9.3.1 Other Symbolic Constants	137
		9.3.2 Evaluation of Symbolic Constants	137
	9.4	General Problem Solving with Symbolics	138
	9.5	Symbolic vs. Numerical Solutions	142
	9.6	Automated Symbolic Solutions: QUAD and ISOL	143
		9.6.1 ISOL	143
		9.6.2 QUAD	146
	9.7	Multiple Roots	146
		9.7.1 Using the Solver to Select Roots	149
	9.8	Expression Manipulations	150
		9.8.1 COLCT	153
		9.8.2 EXPAN	155
		9.8.3 Simplifying Polynomials	157
		9.8.4 FORM	157

	9.8.4.1	Commutation: $\leftrightarrow$	158
	9.8.4.2	Association: $\leftarrow A$ and $A \rightarrow$	159
	9.8.4.3	Distribution: $\leftarrow D$ and $D \rightarrow$	159
	9.8.4.4	Merging: $\leftarrow M$ and $M \rightarrow$	160
	9.8.4.5	Prefix Operations: $\rightarrow()$, $-()$, $1/()$, DNEG and DINV	160
	9.8.4.6	Identities: $*1$, $/1$, ^{1} , and $+1-1$	161
	9.8.4.7	Adding Fractions: AF	162
	9.8.4.8	Logarithms: L^* and $L()$	162
	9.8.4.9	Exponentials: E^* and $E()$	162
	9.8.5	Subexpression Substitution	162
9.9	Calculus		165
	9.9.1	Differentiation	165
	9.9.2	Taylor's Polynomials	167
	9.9.3	Integration	169
	9.9.3.1	Integration with an Implicit Variable	173
	9.9.4	Double Integrals	175
10.	Programming		178
	10.1	Program Basics	179
	10.1.1	The $\ll \gg$ Delimiters	179
	10.1.2	The Program Body	180
	10.1.3	Structured Programming	181
	10.1.4	Comparing HP-28 and HP-41 Programs	184
	10.2	Program Structures	186
	10.3	Tests and Flags	188
	10.3.1	HP-28 Test Commands	192
	10.3.2	SAME, ==, and =	192
	10.4	Conditional Branches	193
	10.4.1	Nested IF Structures	195
	10.4.2	Command Forms	196
	10.5	Loops	198
	10.5.1	Definite Loops	198
	10.5.1.1	Varying the Step Size	199
	10.5.1.2	Looping with No Index	200
	10.5.1.3	Exiting from a Definite Loop	201
	10.5.1.4	Comparison with the HP-41	202
	10.5.2	Indefinite Loops	203
	10.5.2.1	DO Loops	203
	10.5.2.2	WHILE Loops	205
	10.5.2.3	DO vs. WHILE	206
10.6	Error Handling		206

10.6.1	The Effect of LAST	207
10.6.2	Comparison with HP-41 Error Handling	208
10.6.3	Exceptions	209
10.7	Local Variables	211
10.7.1	Comparison of Local and Global Variables and Names	214
10.8	Local Name Resolution	216
11.	Program Development	220
11.1	Program Documentation	220
11.2	Program Editing	222
11.2.1	Low Memory Editing Strategies	223
11.3	Starting and Stopping	224
11.3.1	WAIT	226
11.3.2	KEY	226
11.3.3	The ON key, ABORT and KILL	227
11.3.4	System Halt and Memory Reset	229
11.3.5	Single-Stepping	229
11.4	Debugging	230
11.5	Program Optimization	235
11.6	Memory Use	238
11.7	Input and Output	240
11.7.1	Prompts and Labels	240
11.7.1.1	Prompted Input	240
11.8	Using The USER Menu for Input and Output	244
11.9	The HP-28S CUSTOM Menu	244
11.9.1	Output Menus	245
11.9.2	Input Menus	247
11.9.3	Local Names in Custom Menus	248
11.9.4	Saving Custom Menus	249
11.9.5	Programmable Built-in Menu Selection	250
11.10	Programs as Arguments	250
11.11	Timing Execution	255
11.11.1	Erratic Execution	256
11.12	Recursive Programming	257
11.13	Additional Program Examples	259
11.13.1	Random Number Generators	259
11.13.1.1	Poisson Distribution	259
11.13.1.2	Normal Distribution	260
11.13.2	Prime Numbers	262
11.13.3	Simultaneous Equations	264
11.13.4	Infinite Sums	267
11.13.4.1	Sine Integral	268

	11.13.4.2 Cosine Integral	268
	11.13.4.3 Sum Programs	270
12.	Arrays and Lists	272
12.1	Arrays	272
12.2	Lists	275
12.2.1	List Operations	276
12.3	List Applications	277
12.3.1	Input Lists	279
12.3.1.1	HP-28S Command List Arguments	280
12.3.2	Output Lists	281
12.3.3	Lists of Intermediate Results	281
12.4	Lists and Memory	284
12.5	Indexed Variables on the HP-28S	286
12.6	Symbolic Arrays	287
12.6.1	Utilities	287
12.6.2	Symbolic Array Arithmetic	291
12.6.3	Determinants and Characteristic Equations	293
13.	Plotting	297
13.1	Function Plots	297
13.1.1	Notation	299
13.1.2	The Plot Procedure EQ (Current Equation)	300
13.1.2.1	Evaluation of EQ	302
13.1.2.2	Missing Points and Errors	302
13.1.3	The Independent Variable	303
13.1.4	Reducing the Number of Points Plotted	305
13.2	Digitizing	305
13.2.1	HP-28C Digitizing	306
13.2.2	HP-28S Digitizing	307
13.3	Storing Pictures (HP-28S Only)	307
13.3.1	The Image String	310
13.4	Plot Ranges	311
13.4.1	Conversions Between Plot Ranges and Pixel Numbers	314
13.4.2	Adjusting the Plot Ranges	315
13.4.2.1	Moving the Center of the Plot	315
13.4.2.2	Digitizing New $P_{\min}$ and $P_{\max}$	316
13.4.2.3	Rescaling with *H and *W	318
13.4.2.4	Positioning the Axes	321
13.5	Scatter Plots	321
13.6	Generalized Plotting using PIXEL	324
13.6.1	Polar Plots	326
13.6.2	Drawing Lines	329

13.6.3	Elaborating DRAW	331	
13.7	Printing Plots		334
13.7.1	Extended Plots	334	
	Program Index		340
	Subject Index		343



1. Introduction

Welcome to the world of the HP-28 calculator. If you are a new or prospective owner of this calculator, you may well be a little overwhelmed or intimidated by the sheer extent of the HP-28's capabilities. This is not surprising--the HP-28 represents the most powerful and comprehensive set of mathematical functionality ever built into a handheld device. Because the HP-28 presents so much that is new or different from previous high-capability calculators, you might imagine that it will take you a long time to master its use. Fortunately, this shouldn't be true.

In this book we intend to give you some *insight* into the theory and operation of the HP-28. The most important message to convey is that the HP-28 is actually quite easy to use once you understand a few fundamental ideas and try a few examples. The beauty of this calculator is that it applies its basic rules and procedures uniformly and consistently across the entire range of its capabilities. You don't have to learn new methods for each general area of calculator operation.

For example, here's how you add two numbers on the HP-28:

1. Key in the first number. 5×10^3
2. Key in the second number.
3. Press $\boxed{+}$.

If you're familiar with traditional HP scientific calculators, you will recognize this as the standard "RPN" keystroke sequence for addition. If you have only used so-called "algebraic" calculators, the sequence may seem a little awkward--but we'll postpone explanation and justification to Chapter 2. The point here is that once you've learned this sequence for ordinary numbers, you immediately know also how to add, for example, two complex numbers or two vectors. Just take the above instructions and substitute "complex number," or "vector," everywhere you see "number." You follow the same logical sequence, and press the same $\boxed{+}$ key, for all of the kinds of addition that the HP-28 provides. This is what we mean by consistency and uniformity.

1.1 Some History

In 1972, Hewlett-Packard introduced the HP-35, an "electronic slide-rule" that revolutionized the world of numerical calculations. It offered high-precision arithmetic, logarithmic, and trigonometric functions at the press of a key, obsoleting slide-rules and thick function tables. The HP-35 was followed by a stream of products, from HP and from other manufacturers, that expanded on the HP-35 theme by offering more functions and more data storage registers.

A second generation of calculators was started by the HP-65, the first programmable calculator. This calculator allowed you to customize it by creating programs, which effectively extended the built-in command set. Like the HP-35, the HP-65 was followed by numerous variations of the programming theme, including handheld computers programmable in BASIC. Perhaps the most successful of these is the HP-41, which has become the standard engineering calculator.

All of the calculators of the first two generations share two common limitations. First, they are optimized only for dealing with real floating-point numbers. Some calculators allow you to work with character strings, complex numbers, and/or matrices, but typically each additional data type has its own special commands or working environment, requiring you to learn new calculation methods and making it hard to combine different data types in the same calculation.

Second, none of these calculators allow you deal with programs as *unevaluated* mathematical quantities. For example, you can write programs to calculate $a + b$, and $c + d$, but there is no way for you to manipulate the program results to produce a new result like $a + b + c + d$ except by running the programs to produce numerical results, then combining the numbers.

The HP-28C, introduced in early 1987, removes both of these limitations. It allows you to work with a variety of data types, including the strings and matrices mentioned above, using exactly the same logic and keystrokes that you use for ordinary numbers. The most important of these new data types is the *algebraic object*. You can enter algebraic objects that represent $a + b$ and $c + d$ symbolically, then press the $\boxed{+}$ key to return the new symbolic result $a + b + c + d$. The variables don't have to have numeric values *before* you can add them. Most HP-28C mathematical functions, in fact, can accept symbolic inputs and return symbolic results. Not only does this mean that you can perform symbolic algebra, and even calculus, right on the HP-28C, but at a stroke, much of the work of programming disappears. These capabilities represent such a dramatic advance over previous calculator technology that they merit the name "third generation." To the built-in functions of the first generation, and the customizability of the second, this new third generation of calculators adds the ability to apply programs to other programs and expressions.

The HP-35 also introduced a standard "user-interface," called RPN (short for *Reverse Polish Notation*), that has been the hallmark of HP calculators ever since. RPN calculators are organized around a stack of number registers, using a last-in-first-out logic that is optimal for the key-per-function operation. Throughout the evolution of HP calculators from the HP-35 up through the HP-41, that standard RPN interface has remained virtually unchanged. If you are familiar with one HP calculator, you can pick up any other and use it right away--that is, until the advent of the HP-28C. The HP-28C, while

definitely an RPN calculator, makes some fundamental departures from the standard RPN interface of the past.

The changes to standard RPN made by the HP-28C are actually a generalization of the RPN interface to handle a variety of new data types, most particularly including *variables* and *expressions* for symbolic mathematics. As we explain the various aspects of HP-28 operation throughout this book, we will also try to show how HP-28 techniques compare with those of its HP RPN calculator predecessors. We shall use the HP-41 as the standard of comparison, since its alphanumeric capability means that its commands and programs have legible names, instead of the key names and keycodes used by calculators like the HP-11C and the HP-15C. However, most of what we say about the HP-41 also applies to the other RPN calculators.

The HP-28 “language,” which includes the operating logic as well as the specific command set, is called *RPL*. Computer languages are known for their whimsical names; RPL is no exception—it stands for *Reverse Polish Lisp*. This name suggests the HP-28’s derivation from HP calculators (and from FORTH, another language that uses reverse Polish logic) and from the computer language LISP, which is frequently used in computer symbolic mathematics systems. Note that the HP-41 language was never given a name, so many people call HP-41 programming “RPN programming,” which is unfortunate since, properly speaking, RPN is a mathematical logic that is not specific to any calculator or computer.

1.1.1 HP-28C and HP-28S

One year after the introduction of the HP-28C, Hewlett-Packard announced a new model, the HP-28S. The primary difference between the two calculators is that the HP-28S has 32K bytes of random access memory (*RAM*) for calculation and storage, compared to the 2K bytes of the HP-28C. The HP-28S represents a major redesign of the HP-28C electronics, which permitted the inclusion of the extra RAM at no increase in price, so that the HP-28S effectively replaces the HP-28C.

The additional RAM in the HP-28S makes a profound difference in your ability to take advantage of the capabilities of the HP-28. There is enough memory to store dozens of procedures and data objects for customizing your calculator, while still leaving sufficient working memory to perform operations that fail even on an “empty” HP-28C. Furthermore, the HP-28S command set is extended over the HP-28C’s in various ways to make good use of the bigger memory, including commands to organize the user memory with a directory system and to save, retrieve, and combine display pictures.

Nevertheless, the HP-28C is still a capable calculator, and this book is designed for both the HP-28C and the HP-28S. There are, of course, discussions of HP-28S-only features,

but the majority of the ideas and techniques described here apply to both calculators. We shall use the name "HP-28" without a suffix to refer to both calculators; for topics that are specific to either the HP-28C or the HP-28S we will append the appropriate letter to the name.

1.2 About This Book

The HP-28 comes with an *Owner's Manual* (called the *Getting Started Manual* for the HP-28C) and a *Reference Manual*. The first book gets you off and running with the HP-28, providing lots of sample keystroke sequences that give you a quick introduction to many HP-28 features. It does not go very deeply into any topic. The *Reference Manual*, on the other hand, gives detailed specifications for all of the calculator's operations, but it is almost devoid of examples.

What's missing from these two manuals, and what this book hopes to provide, is a little more *motivation*, and some more elaborate examples. By motivation, we mean the purpose and use of many of the operations, and the connections between various features of the calculator. The scope of the HP-28 is so broad that we cannot show you how to use it for every imaginable problem, but we can try to help you understand it enough to solve your own problems. We delve quite deeply into the HP-28's principles of operation, with the expectation that if you know the principles, you will learn and remember keystrokes and methods much more easily.

We assume that you have read most of the HP-28 Owner's Manual, so that you at least know how to perform simple keystroke calculations, enter various object types, and find a command in a menu. In some cases, where there are crucial ideas that we want to communicate, we will show some actual keystroke sequences and perhaps even repeat some material that is in one of the HP manuals. But for the most part we will assume that you know the rudiments of HP-28 operation so that we can concentrate on ideas and connections.

HP-28 Insights breaks roughly into two parts. In the first part, Chapters 1 through 5, we discuss primarily the principles and concepts of HP-28 operation, starting with the mathematical ideas that underlie the HP-28's use of Reverse Polish Notation and the object stack. The second part, Chapters 6 through 13, is an extended discussion of problem solving techniques, starting with the equation solver, running through the development and application of program objects, and concluding with a description of plotting methods.

Here's a summary of the chapter topics:

Chapter	Topics
1. Introduction	Introductory material, notation conventions.
2. Understanding RPN	The theory of RPN, and its electronic implementation.
3. Objects and Execution	Operations, objects, execution and evaluation, quotes, ENTER.
4. The HP-28 Stack	Stack operations, comparison with the HP-41, postfix logic.
5. Variables	Creating, storing, recalling, evaluating and purging variables; storage arithmetic; the USER menu; HP-28S directories.
6. Problem Solving	Introduction to HP-28 problem-solving methods.
7. The Solver	The HP-28 automatic equation solver; interpreting results; obtaining guesses; finding extrema.
8. User-defined Functions	HP-28 user-defined functions as a simplified programming technique.
9. Symbolic Math	Performing symbolic mathematics on the HP-28; function execution; simplification; symbolic constants; problem solving; automated solutions; expression manipulations; calculus.
10. Programming	The principles of program objects; comparison with HP-41 programming; tests and flags; conditional branches; loops; error handling; local variables.

11.	Program Development	The art of program construction; editing and debugging; starting and stopping; optimization; input and output; programs as arguments; recursion.
12.	Arrays and Lists	The application of array and list objects; HP-28S indexed variables; symbolic arrays.
13.	Plotting	General principles; function plots; digitizing; storing pictures; plot ranges; scatter plots; polar plotting and line drawing; printing pictures.

You may find the first five chapters to be a bit tedious with their emphasis on theory and terminology. Nevertheless, we recommend that you read those chapters through at least enough to insure that you have a grasp of the definitions and terms that we introduce there, which are used throughout the second part of the book. In particular, the concepts of *operations*, *objects*, *execution*, and *evaluation*, described in Chapter 3, are used extensively in all of the material that follows.

The presentation of the book's subject matter is not always exactly linear. That is, occasionally we make use of or refer to concepts or techniques that are not explained until later sections. This occurs frequently in programming examples. Wherever possible, examples that illustrate a concept are chosen to have practical uses as well. This often requires combining more techniques into an example than just the one being demonstrated. For instance, in section 5.7.3, we list some useful programs for working with HP-28S directories. The programs are relevant to the section in which they are listed, but at that point in the book we haven't yet discussed programming at all. To alleviate this kind of problem, we include many cross-references between the sections, and a subject index.

You might notice that this book is not exhaustive of HP-28 features. For example, there are minimal references to HP-28 statistics, and none to unit conversions. Rather than try to cover every possible topic, we try to concentrate on the key ideas of calculator operation, especially where there are important differences from other calculators and computers. If you master all of the topics covered here, you will have no trouble applying the principles to the remaining HP-28 features.

1.3 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-28 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-28 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 menu key like this: **↔**.
- A shifted key is shown with the key name in a box preceded by a black square ■ representing the shift key, e.g. ■ **BINARY**, or ■ **PURGE**.
- Menu keys for commands available through 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: **SIN** or **-LIST**.
- When the six menu keys are acting as cursor keys, they are shown as **INS** **DEL** **△** **▽** **◀** and **▶**.

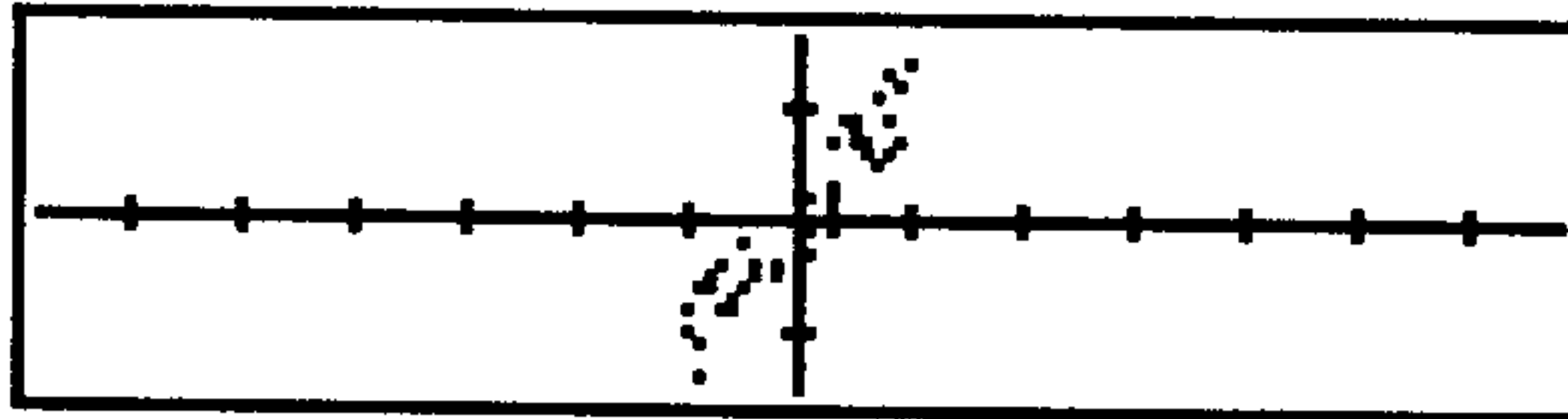
Examples of HP-28 operations take several forms. When appropriate, we will give step-by-step instructions that include specific keystrokes and show the relevant levels of the stack, with comments, as in the following sample:

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 1 2 3. We show key boxes for the 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-28-generated picture of the calculator display, such as this picture from Chapter 13:



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-28 commands and data that you are to execute, together with the stack objects that result from the execution. The “right hand” symbol  is used as a shorthand for “the HP-28 returns...” In the compact format, the addition example is written as

123 456 +  579

The  means “enter the objects and commands on the left, in left-to-right order, and the HP-28 will give back--*return*--the objects on the right.” If there are multiple results, they are listed to the right of the  in the order in which they are returned. For example,

A B C ROT SWAP  B A 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-28, 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  symbol.

The  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  are the “input” objects, and the objects following the  are the program outputs.

In Chapter 9 there are several of examples of the use of FORM. These examples

represent the “before” and “after” FORM displays in a format similar to ordinary stack command examples. The following typical example demonstrates the use of the M→ operation:

$$A*B \boxed{+} A*B \quad \overline{\overline{M \rightarrow}} \quad \boxed{(A+A)} \boxed{*} B$$

The example indicates that starting with the expression $A*B + A*B$, with the FORM cursor (represented by a box around an object) on the +, pressing $\overline{\overline{M \rightarrow}}$ converts the expression to $(A+A)*B$, with the cursor on the *.

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 format is reasonably self-explanatory, and is described in detail in section 11.1.

1.4 Easy to Use or Easy to Learn?

It would be nice if you could pick up the HP-28 and use all of its facilities without ever referring to a manual. Some other recent HP calculator products, the HP-17B, HP-18C, HP-19B, and HP-27S, come much closer to this ideal than the HP-28. But these calculators trade for this convenience by having very limited computational capabilities and flexibility compared to the HP-28. If your problem “fits” on one of these other calculators, then they’re easy to use as well as easy to learn. But if you want to do something just a little different, forget it.

The HP-28 approach is to provide a broad, very flexible set of computational capabilities, many of which have never before been available on a handheld calculator. Furthermore, it is expressly designed for “linking” calculations together--the results of one calculation are always ready to be used as input for another, even if you didn’t know in advance that your work would proceed that way, and even if the calculator designers didn’t expect you to make that particular combination of calculations. These ideas are what the HP-28 means by “ease-of-use.”

“Ease-of-learning” is a different story. Unfortunately, the HP-28’s rich capability set doesn’t leave enough built-in memory to provide “no-manual” learning. And there’s no doubt that the HP-28 does work differently from other calculators, even from its RPN calculator predecessors like the HP-41. You have no choice but to spend some time reading the manuals and learning new procedures. But learning the basic ideas doesn’t take long, and once you master them, a wide range of truly easy-to-use calculating capabilities is available to you.

2. Understanding RPN

The HP-28, like most of its Hewlett-Packard calculator predecessors, presents a user interface centered around a logic called "RPN," short for *Reverse Polish Notation*. If you are unfamiliar with this logic, particularly if you are accustomed to so-called "algebraic" calculators, RPN may seem as bizarre as its name. In this chapter, we will explain how RPN works, and why its virtues make it the choice for the HP-28.

Many people use a calculator in a style that you might call "fingers in, eyes out." That is, they manually type in all of the data for a calculation and read out the result visually from the display, perhaps writing it down on paper. For this type of use, a calculator that uses "algebraic" entry seems desirable, because in at least simple cases the key-strokes follow more-or-less the order of common written mathematical notation.

The algebraic style, however, is not well suited for exploratory calculation, where you don't necessarily know what to do next until you see the results of previous calculations--and you need those results as part of the next calculation. When you press an algebraic calculator's [=] key to complete a calculation, you had better be sure that you're finished, because the result you see in the display may vanish at the next key-stroke.

The choice and design of an RPN system for a calculator arises from consideration of one central principle:

- *The result of any calculation, no matter how complicated, may be used as an input for a subsequent calculation.*

RPN calculators are designed to embody this principle, by providing a mechanism (the "stack") whereby you can apply mathematical operations to data already entered into the calculator. The results of the operations are also held indefinitely, so that they, in turn, can be the input data for subsequent operations.

In the calculator world, the term *Reverse Polish Notation*, or more specifically, the abbreviation "RPN", has come to mean "the way HP calculators work." RPN actually is a mathematical notation; HP calculators provide an electronic implementation of the notation. In RPN, mathematical functions are written *after* their arguments, not before or between the arguments as in ordinary written expressions. The notation appears strange, because we are not used to visualizing or writing expressions this way. However, when you actually evaluate an expression to a numerical value using pencil and paper, you must revert to an order of operation that exactly corresponds to RPN. We will illustrate this point by examining how mathematical expressions are evaluated.

2.1 The Evaluation of Mathematical Expressions

A mathematical *expression* is an abstract representation of the calculation of a single value. An expression combines data (numbers or other explicit quantities), variable names, and functions. When you *evaluate* an expression, you perform all of the calculations represented by the expression. Examples of expressions are:

$$1 + 2$$

$$x + y + 2z$$

$$\sin [\ln (x + 2)]$$

$$x^3 + 4x^2 - 6x + 2$$

We will confine our attention to expressions that can be formed from the mathematical functions included in the HP-28: arithmetic operations, powers, roots, transcendental functions, etc. Expressions like these have the property that they are equivalent to a single value. That is, if you perform the calculations represented by an expression, you end up with a single value as the result.

In our discussions, we will be using the following terms:

- A *function* is a mathematical operation that takes zero, one, or more values as input, and returns one value.
- A value used by a function as “input” is called an *argument*.
- A value returned by a function as “output” is called a *result*.
- A mathematical *variable* is a symbol that stands for a value. Evaluating a variable replaces the symbol with the value.
- *Algebraic syntax* is the set of rules that governs how data, variables, and functions may be combined in an expression.

As an example of these concepts, consider the following expression:

$$\sin [123 + 45 \ln (27 - 6)]$$

The expression contains the *functions* $\sin$, $\ln$, $+$, $-$, and $\times$ (implied multiply between the 45 and the $\ln$), and the *numbers* 123, 45, 27, and 6. The expression is written in common mathematical notation, but notice that the order in which you read or write the expression, i.e., left to right, does not correspond very well to the order you would use if you were actually going to evaluate the expression with pencil and paper and function tables. For example, although the $\ln$ function *precedes* the quantity $(27 - 6)$, you can't actually compute (or look up) the logarithm until *after* you have computed the difference

27-6. Similarly, the *sin*, which is the *first* function that appears in the expression, is actually the *last* that you will execute. You have to wait until the entire rest of the expression $[123 + 45 \ln(27 - 6)]$ is evaluated.

The common notation that we have been using here has been developed over the centuries to present a readable picture of a mathematical expression that takes advantage of a human's ability to view an entire expression at once and draw general conclusions from its structure. But the notation is not a very good prescription for actually evaluating an expression--as you step through a calculation, you have to jump back and forth, match parentheses, etc. to find the next step. As we will show now, converting an expression into an orderly procedure for evaluation leads directly to RPN. First we'll adopt a uniform structure that treats all functions alike, then we'll turn it around to match actual calculation order.

Common notation is not uniform because the notation differs for one-argument and two-argument functions. In our sample expression, the one-argument functions *sin*, *ln*, and *cos*, are written *in front* of their arguments ("prefix" notation), whereas the two-argument functions *+* and *-* are written *between* their arguments ("infix"). Furthermore, there is an implied multiply between the 45 and the *ln* that is not explicitly written. Infix notation also leads to ambiguity. For example, does $1+2\times 3$ evaluate to 9 or 7? You either have to introduce extra parentheses, e.g. $(1+2)\times 3$ or $1+(2\times 3)$, or use so-called *precedence* conventions that specify which functions are executed first in ambiguous situations. One of the drawbacks of non-RPN calculators is that there is no universal standard for precedence, so you have to memorize the precedence rules of each calculator you use.

A general-purpose form for functions is to write each function name followed by its arguments contained in parentheses, as in $f(x)$, $g(x,y)$, etc. You can make expressions more uniform by writing all of its functions in this prefix form:

$$\sin(+(123, \times(45, \ln(-(27,6))))))$$

In this notation, $+(1,2)$ means "add 1 and 2"; $\times(1,2)$ means multiply 1 by 2; etc.

Writing expressions this way is called *Polish notation*, honoring the Polish logician, Jan Łukasiewicz. Unfortunately, this notation appears practically unintelligible to people accustomed to conventional notation. But it does show explicitly the hierarchical structure of the expression, which we will discuss later (section 9.1.1). Also, it is useful because it is a step towards RPN. That is, you can obtain a form that corresponds more closely to the actual order of evaluation of an expression by rewriting the Polish form so that the function names *follow* their arguments' parentheses. For example, rewrite $+(1,2)$ as $(1,2)+$. The example expression now becomes:

$$((123, (45, ((27, 6) -) \ln) \times) +) \sin$$

You have replaced Polish notation with *Reverse Polish Notation*. In this form, the expression represents a step-by-step evaluation prescription for pencil-and-paper or electronic calculation, that follows the left-to-right order of the expression. To see this, consider an orderly pencil-and-paper method for evaluation:

- Start at the left of an RPN expression, and work to the right.
- When you come to a number, write it down below any previous numbers.
- When you come to a function, compute its value using the last number(s) you wrote as its arguments. Erase the argument number(s), and then write the function value.

Apply this procedure to calculate the example expression (keeping two decimal place accuracy):

Object	What to do	What you see
123	Write 123	123
45	Write 45	123 45
27	Write 27	123 45 27
6	Write 6	123 45 27 6
-	Subtract 6 from 27	123 45 21
ln	Find ln(21)	123 45 3.04
×	Multiply 45 and 3.04	123 137.00

+	Add 123 and 137.00	260.00
sin	Take the sine of 260°	-.98

There are two things you can notice from this exercise:

- Whenever you encounter a function, you can execute it immediately because you have already calculated its arguments.
- You can ignore parentheses. When you write an expression in RPN form, you don't need parentheses, because there is no ambiguity of precedence--functions are always executed left-to-right.

The latter point means that you can eliminate parentheses from the notation. Doing so, the example becomes:

123 45 27 6 - ln × + sin

2.2 Calculator RPN

An RPN calculator allows you to substitute an electronic medium for paper. The calculator's **ENTER** key is the equivalent of "write it down" in paper calculations. You "write" a number by pressing the appropriate digit keys, then **ENTER**, which terminates digit entry and enters the number into the calculator's memory. The memory takes the place of paper.

For cases where you need to have more than one number written down at a time, calculator memory is organized into a "stack." You can visualize the stack as a vertical column of numbers, where the most recently entered numbers are at the bottom of the column, and the oldest numbers at the top. Each new entry "pushes" previous entries to higher stack levels. A function always operates on the latest stack entry or entries, and replaces those entries with its result, where it is ready for use by the next function to come along. If one or more entries are removed from the stack, older entries drop down to fill in the vacant levels. Again, this is quite analogous to the pencil-and-paper technique you used in the example.

To illustrate calculator RPN, redo the previous example on the HP-28. Start by setting the numerical display mode for two decimal places:

Keystrokes:		Stack:
MODE 2 FIX		
123 ENTER	1:	123.00
45 ENTER	2:	123.00
	1:	45.00
27 ENTER	3:	123.00
	2:	45.00
	1:	27.00
6 ENTER	4:	123.00
	3:	45.00
	2:	27.00
	1:	6.00
-	3:	123.00
	2:	45.00
	1:	21.00
LOGS LN	3:	123.00
	2:	45.00
	1:	3.04
x	2:	123.00
	1:	137.00
+	1:	260.00
TRIG SIN	1:	-0.98

Note how

- each *number* entered goes into level 1, raising the preceding numbers to higher levels;
- each *function* removes its argument or arguments from the stack, and returns a new result to the stack.

Here you can see how a *stack* provides for the realization of the principle stated at the start of Chapter 2, namely, that every result can be an argument. The stack acts as central exchange, where each function expects to find its arguments. Since each function also returns its results to the stack, those results are automatically ready to be used as arguments for the next function.

2.3 HP-28 RPN

The HP-28 stack is a more thorough implementation of RPN principles than those of its predecessors. The HP-28 is the only calculator that does not limit the number of stack levels; the stack grows and shrinks as needed. Other calculators provide only a fixed stack of four levels, adequate for many calculations, but still a handicap. With only four levels:

- You can't routinely convert any expression into RPN, then execute it left to right. Instead, you have to study the expression, looking for ways to avoid piling up more than four stack entries at a time.
- Some calculations intrinsically require more than four entries, no matter how clever you are. This means that you have to save one or more intermediate results in storage registers, then recover them later for further stack operations.

The unlimited HP-28 stack allows you to concentrate on the results of a calculation without requiring extra mental effort to rearrange it to fit the constraints of a four-level stack.

Most "algebraic" calculators have a limit on the number of parentheses that you can nest in a calculation. This limitation is exactly analogous to the restriction in the number of RPN stack levels, and is perhaps even worse since algebraic entry does not lend itself well to passing the results of one calculation on to another.

The HP-28 also provides an important new capability in its ability to intermix expressions, entered in algebraic form, with RPN operations. This ability is provided through the use of *algebraic objects*, which are representations of expressions that you can enter into the stack as entire units. We discuss algebraic objects in great detail in later sections of this book; for now, you can consider them as the means by which you can calculate with algebraic notation.

In section 2.1 we showed how RPN is derived by considering the manner in which expressions are actually evaluated. However, we do not mean to imply that a completely RPN approach is always the most convenient method of calculation. In fact, to evaluate a known expression like our example $\sin[123 + 45 \ln(27 - 6)]$, it is arguably simpler to key in the expression in a manner that corresponds as nearly as possible to

the written form, than to figure out the more efficient RPN keystrokes. RPN, on the other hand, is most useful for exploratory calculation, when you're not merely evaluating a predetermined expression. The HP-28 design allows you to have the best of both worlds, by combining algebraic and RPN logic as follows:

- If you know in advance the complete mathematical form of a calculation, enter it as an algebraic object.
- If you are working out the solution to a problem, and don't know in advance all of the steps, work through the problem with an RPN approach, applying functions to the results as they appear.
- In both cases, the results are held on the stack ready for use in further calculations.

Our sample problem was originally expressed as an expression, so you can enter it as an algebraic object.

'SIN(123+45*LN(27-6))' **ENTER**

puts the algebraic object representing the expression into stack level 1. (Note that it is the expression itself that is present, not its evaluated value; the ability to handle expressions without first evaluating them is one of the unique and most powerful HP-28 capabilities.) In this example, you are interested in the numerical value, so press **EVAL**. This replaces the algebraic object with its value $-.98$. Actually, if this result were all that was of interest, you could omit pressing **ENTER**, and use **EVAL** to take the expression directly from the command line and evaluate it.

Suppose, however, that at the beginning of the calculation you were only interested in the expression $123 + 45 \ln(27 - 6)$. In that case, you would compute the value by entering

'123+45*LN(27-6)' **EVAL**  260.00

Then, after obtaining this result, you realize that in addition to the value itself, you also need to know the sine of the value. Because the result of the initial calculation is on the stack, it is ready for further calculation. In this case, you can execute **DUP** to make a copy of the number for later use, then **SIN** to compute the sine.

The HP-28 is unique in its ability to hold the results of algebraic expression evaluation in a manner that allows you to apply additional operations to the results after they are calculated. Algebraic entry calculators require that you know the entire course of a calculation before you start; RPN calculators overcome that problem, but you must always

mentally rearrange an expression into reverse Polish form as you proceed. The HP-28 allows you to proceed with any mix of the two approaches that is appropriate for the problem at hand.

In most of the examples in this chapter, you calculated by entering real, floating-point numbers into the stack, then executing mathematical functions that replaced those numbers with new ones. We have just shown that a stack level can also hold an entire expression, as a unit, the same way it holds a single number. In the next chapter, we will show how the HP-28 allows you to use the same process for a variety of mathematical or logical objects, including complex numbers, matrices, text strings, and even programs.

3. Objects and Execution

In Chapter 2, we demonstrated how you perform calculations on the HP-28 by applying functions to numbers that are present on a stack, which acts as the electronic equivalent of a sheet of scratch paper. This RPN system is very uniform and flexible, and there is no particular reason to restrict its use to real numbers and ordinary mathematical functions. The HP-28 generalizes the RPN approach to problem solving in two ways:

- Real numbers are just one of several types of *objects* that the HP-28 can manipulate on the stack and store in memory.
- Mathematical functions are just one of several classes of HP-28 *operations* that can be applied to numbers and other types of objects.

The terms *object* and *operation* are key terms for any discussion of the HP-28, and we will study them in detail in this chapter. In addition, we will introduce the concept of object *execution*, and the closely related term *evaluation*. In rough terms, *operations* are “what things the HP-28 can do,” and *objects* are “what the HP-28 can do things to.” *Execution* and *evaluation* are the actual “doing.”

We will use these four words extensively throughout this book to make general statements about HP-28 principles, so it is important that you understand the meanings of each. If you find occasionally that the statements are too abstract, you can relate them to more familiar ideas by substituting concrete examples for the general terms. For example, when we refer to an object, you can think of a number as an example; for an operation, think of an ordinary math function like + or sine. Execution is the “activation” of an object--think of executing a program. Evaluation differs from execution only for algebraic objects: executing an algebraic object treats it as symbolic data; evaluation actually performs the calculations defined by the object.

(Several other English words might be substituted for *object*; item, unit, element, etc. The use of *object* for this purpose is common in mathematical jargon, and so that word is adopted for HP-28 terminology.)

3.1 Operations

“What things the HP-28 can do” make up a very long list, and constitute the subject matter of most of this book. Here we will concentrate on defining the different types of operations, to facilitate later discussions.

We use the term *operation* to mean any of the built-in capabilities of the calculator. Most calculator manuals use the term *function* for this purpose. HP-28 literature uses

the term *operation* instead, so that *function* can be reserved for the specific group of operations that correspond to the mathematical meaning of *function*.

There are two basic methods by which you can make the HP-28 "do" something; that is, perform an operation.

- Find the key that is labeled with the name or symbol for an operation, and press it. Many important operations, such as the arithmetic operators, or STO and RCL, are permanently available on the keyboard. The remaining operations are available as menu keys.
- Spell out the operation's name in the command line, then press **ENTER**. ENTER on the HP-28C plays a role that combines its original RPN calculator purpose of ending number entry with a more sophisticated meaning of "do these commands." ENTER is explored in detail in section 3.11.

Operations are classified as follows:

1. An operation can be a *command* or a *keyboard operation*, according to whether it is programmable or non-programmable, respectively. A command has a specific name, so that you can
 - execute the command by typing its name into the command line.
 - include the command in a program that you write.

Keyboard operations don't have names that you can spell out or include in a program; you can only execute a keyboard operation by pressing a key. Examples are **ENTER**, **LC**, and **SOLVR**.

2. Programmable operations--commands--are sorted into two classes. If a command can be included in the definition of an algebraic object, it is called a *function*. Examples of functions are +, SIN, LOG, and NOT. Other commands, which are not allowed in algebraics are called *RPN commands*. These commands, such as DUP, STO, or RDZ (randomize), are typically stack or memory operations that make no sense in the context of an algebraic object, which is the HP-28 calculator representation of a mathematical expression or equation. The logic of expressions demands that every part of an expression (including the entire expression itself) can be evaluated to a single value. So for an HP-28 command to be included in an algebraic object, it must act like a mathematical function--take zero or more values as input, and always return exactly one result.
3. The final classification of HP-28 operations is the division of functions into two categories: *analytic* and *non-analytic*. Analytic functions are just those for which the HP-28 knows the derivative and inverse. "Knowing" the inverse of a function

f means the HP-28 can automatically solve the equation $f(x) = y$ for x . (In mathematics, an analytic function is continuous and differentiable, which corresponds more-or-less to the HP-28 meaning of analytic function. For various reasons, the HP-28 does not provide derivatives and/or inverses for every function that is analytic mathematically. % is an example of a well-behaved function for which no built-in derivative is provided. On the other hand, the function ABS can be differentiated on the HP-28, even though it is not properly differentiable at zero.)

The main reasons for sorting HP-28 operations into these categories is to make possible general statements about various classes of operations, and to provide information about individual operations without unnecessary repetition. Thus when we refer to DUP as an RPN command, we are reminding you that DUP is programmable, but not allowed in an algebraic expression.

3.2 Objects

The HP-28 recognizes 11 distinct types of objects, as listed by type number (as returned by the TYPE command) in Table 3.1 (next page).

The word *object* is the collective term for all of the different items listed in the table. This list does not contain all imaginable object types; these are just the types that you can create and use on the HP-28. In the abstract, an object is a collection of data or procedures that can be treated as a single logical entity. In practical HP-28 terms, this means that an object is something that you can put on the stack. [The HP-28 system actually recognizes some additional types that it uses internally. Occasionally one of these slips through a crack in the system and appears on the stack, where it is displayed as System Object. If you see one of these, you should just drop it from the stack.]

Most objects are identified in the HP-28 by their characteristic *delimiters*, which are just the symbols #, ", ', etc., which you enter to tell the calculator what type of object you are entering, and where it starts and stops. Similarly, the calculator uses the same delimiters when it displays an already entered object so that you can recognize its type. If you enter a string of characters without any delimiters, the HP-28 attempts to interpret it as a real number, or failing that, as a name or command.

An individual object is characterized by its type and its value. The *type* (number, array, etc.) indicates the general nature and behavior of the object. The *value* distinguishes one object from another of the same type. For a real number object, the value is its simple numerical value. For a string, the value is the text characters in the string. For a program, the "value" is the sequence of objects and commands that make up the program. For lists, programs, and algebraic objects, which are made up of other objects,

we will use the term *definition* rather than *value*.

Table 3.1. HP-28 Objects

TYPE Number	Object Type	Identification
0	Real numbers	<i>digits</i>
1	Complex numbers	<i>(real number, real number)</i>
2	Strings (text)	<i>"characters"</i>
3	Real arrays (vectors and matrices)	<i>[real numbers]</i>
4	Complex arrays (vectors and matrices)	<i>[complex numbers]</i>
5	Lists	<i>{ objects }</i>
6	Global names	<i>characters[†]</i>
7	Local names	<i>characters[†]</i>
8	Programs	<i><< objects >></i>
9	Algebraic objects	<i>'objects'</i>
10	Binary integer numbers	<i>#digits</i>

[†] Names can be entered with or without ' delimiters. See section 3.7.

A central theme of the HP-28 is the uniform treatment of different object types. This means that the basic calculation process--applying operations to objects on the stack--is the same for every object type:

- Each stack level holds one object, regardless of type.
- The stack commands to copy, reorder, and discard objects are the same for all object types.
- The processes of storing (naming), recalling, and executing are the same for all object types.

- The same mathematical operation can be applied to as many different object types as make sense for the operation.

These points have the very practical consequence of simplifying the learning and use of the HP-28, for once you learn how an operation works for one object type, you automatically know how to use it for any other object types to which it might apply. For example, if you learn RPN arithmetic for real numbers, you don't have to learn anything new to do arithmetic with complex numbers or arrays--the steps and logic are the same. There is no such thing as "complex mode" or "matrix mode" on the HP-28.

3.2.1 Operations as Objects

You might ordinarily think of operations as actions, and objects as the targets or results of the actions. However, the existence of object types that are not simple data--names, algebraic objects, and programs--blurs this distinction. As a matter of fact, all HP-28 commands are just built-in program objects. To demonstrate that a command is an object, you can put it on the stack. Try this:

1 2 { + } LIST→ DROP

4:	
3:	
2:	1
1:	2
	+

In level 1, you see the *object* +. (You have to enter the + originally in a list to prevent its execution when you press **ENTER**.) If you press **EVAL**, the + is executed, adding the 1 and 2 you entered previously and leaving the result 3. This technique works for any command.

This brings us to the subject of *execution*: when is an object "passive"--it just sits on the stack, for example--and when is it "active?"

3.3 Execution and Evaluation

We have generalized the concept of an object to include not only data objects but also user-defined programs and expressions, and built-in operations. We now similarly define *execution* as the general term for the *activation* of an object: to execute an object means to perform the "action" associated with that object. In the next sections, we will look at the various actions associated with the different object types.

Along with execution, we also need to define the closely related term *evaluation*. The need for two terms that mean almost the same thing arises from the use of algebraic objects as data in some situations, and as “programs” in others.

- For an algebraic object, to *execute* the object means to put it on the stack as symbolic data. To *evaluate* the object means to compute the value that it represents, by treating the object as a program, and executing the program.
- For all other object types, *evaluate* and *execute* are synonymous.

[The HP-28 owner's manuals do not make this distinction between execution and evaluation. We make it here to emphasize the dual nature of algebraic objects, and because it provides for simpler explanations of program and command line execution, quoted and unquoted objects, and global name execution.]

3.3.1 When are Objects Executed?

Before studying the execution actions of the various object types, it is helpful to review the circumstances under which objects are executed or evaluated. It is not unreasonable to say that object execution takes place all the time while the HP-28 is on, since virtually any HP-28 activity--interpreting keystrokes, displaying objects, printing, etc.--can be viewed as the automatic execution of built-in program objects. However, of most interest are the times when objects are executed under *your* direction, particularly objects that you have created. These times are as follows:

1. *Execution*

- When you execute ENTER (section 3.11), each object specified in the command line is executed, in the order in which it appears in the command line. You can *prevent* execution of names or programs in the command line by enclosing them in their respective delimiters ' ' or << >>, as discussed in sections 3.7 and 3.8 below.
- When a program is executed, the objects that make up the program are executed, following the same rules as command line execution (section 3.5).
- When a global name (section 3.6) is executed, the object stored in the corresponding variable is executed. (Execution of a local name merely recalls the stored object.)

2. *Evaluation*

- EVAL removes the object in level 1 from the stack and evaluates it. This is the most common means for evaluating an object *after* it is placed on the stack.
- →NUM, ROOT, QUAD, TAYLR, ∂, and ∫ also evaluate their stack arguments.

- The Solver and DRAW cause the object stored in the variable EQ to be evaluated.
- HP-28S commands such as PUT or $\int$ that use a list containing real numbers as an argument numerically evaluate ($\rightarrow$ NUM) the objects in the list to convert them to real numbers.

The HP-28 manuals identify three classes of objects: *data*, *name*, and *procedure*. This classification is made according to an object's behavior when it is evaluated. Data objects put themselves on the stack; names execute other objects; and procedures cause the sequential execution of the objects contained in the procedures. We will look at these classes of behavior in the next sections. You might note that this classification is made according to object *evaluation*; if the sorting were done according to *execution* properties, algebraic objects would be classed as data objects.

3.4 Data Objects

The idea of a *data object* should be quite familiar to you, since data objects are the only quantities that can be manipulated as objects by other calculators and BASIC computers. The archetype data object is a *floating-point real number*. More generally, an HP-28 data object is the calculator's representation of a mathematical or logical data entity such as a number, a vector, or a character string.

You would not expect a data object to be able to *do* anything; rather, it exists to have things done to it. Nevertheless, data objects do have an execution action: they just enter themselves onto the stack. When you type in a number, for example, and press **ENTER**, the number object is executed and so ends up in level 1. When a data object is already on the stack and you execute EVAL, nothing apparently happens. Actually, EVAL removes the object and executes it, which puts it right back on the stack.

The HP-28 data object class includes the following types: real number, complex number, binary integer, real array, complex array, string, and list.

3.4.1 Real Numbers

A real number object is the HP-28's version of an ordinary real decimal number. The number value of the object is stored in *floating-point* representation, as a combination of a 12-digit *mantissa* ($x/10^{\text{IP}(\log |x|)}$) between 1 and 9.99999999999, and a 3-digit *exponent* ($\text{IP}(\log |x|)$) between -499 and +499. That is, a number is represented as

$$\text{mantissa} \times 10^{\text{exponent}}.$$

When the HP-28 is in scientific number display format (execute 12 **MODE** **SCI**),

you can see the mantissa and exponent explicitly; for example, the number 1.234×10^{23} is displayed as 1.23400000000E23. The E is a one-character symbol for “ $\times 10$ to the power...”

When the HP-28 performs internal calculations during the execution of mathematical functions, real numbers are expanded to fifteen-digit mantissas and five-digit exponents, and all of the calculations are carried out to that accuracy. Functions' results are rounded back to twelve-digit mantissas and three-digit exponents when they are returned to the stack. Note that this does not imply that calculations involving multiple functions are always accurate to twelve digits. The error derived from rounding intermediate results to twelve digits accumulates as each new function executes on the result of the previous one.

3.4.2 Complex Numbers

Complex number objects consist of two real numbers combined in an ordered pair (x,y). They have two primary uses:

- To represent complex numbers, where the first number in each ordered pair is the real part of a complex number, and the second number is the imaginary part. A complex number object (x,y) corresponds to the complex number $z = x + iy$, where $x = \text{Re}z$ and $y = \text{Im}z$. The object (3,2) represents the complex number $3+2i$. Complex number objects obey the rules of complex number arithmetic; for example,

$$(1,2) \quad (3,4) \quad + \quad \Rightarrow \quad (4,6).$$

- To represent the coordinates of points in two dimensions, such as points used in conjunction with HP-28 plotting (Chapter 13). For this purpose, the real part (the first number of the pair) of the complex number is the horizontal coordinate of the point, and the imaginary part (the second number) is the vertical coordinate.

HP-28 mathematical functions treat real number and complex number objects in a very uniform manner. That is, you can intermix the two object types in almost any calculation involving arithmetic, trigonometric, logarithmic, or exponential functions. Two-argument functions return complex results if either argument is complex:

$$3 \quad (2,3) \quad * \quad \Rightarrow \quad (6,9).$$

The result of a single-argument function may be real or complex, according to the argument type and the appropriate mathematics. The functions RE (real part), IM (imaginary part), ARG, and ABS always return real number objects. Trigonometric, logarithmic, exponential, power and root functions applied to a complex argument always return a complex results, e.g.:

Objects and Execution

$$(0,2) \quad \checkmark \quad (1,1).$$

These functions applied to real arguments may return either a real or a complex result. For example,

DEG .5 ASIN 30,

but

2 ASIN (1.57079632679, -1.31695789692).

On most other calculators, the last example would cause an error. The HP-28's integrated treatment of real and complex numbers means that you can write programs that work equally well for real and complex inputs and outputs. However, it also means that you may have to include explicit range testing in a program that you *want* to stop when a calculation strays out of the real number domain.

You should note that the last example gives the same result regardless of whether the HP-28 is in degrees mode or radians mode. Trigonometric functions consider all complex arguments and results to be expressed in radians.

3.4.3 Binary Integers

Binary integer objects represent unsigned integer numbers stored as a sequence of binary bits (rather than decimal digits as for floating-point numbers). The maximum value of an integer is the hexadecimal number FFFFFFFFFFFFFFFF, corresponding to 64 binary bits. The results of arithmetic and logical operations applied to binary integers are truncated to the current *word size*, which you can set between 1 and 64 bits by executing *n* STWS (*n* is a real number argument).

You can control the entry and display of binary integers by executing one of the mode commands BIN (binary, base 2), OCT (octal, base 8), DEC (decimal, base 10) or HEX (hexadecimal, base 16). To enter a binary integer, you type the # delimiter followed by the number digits. The digits are interpreted according to the current base; in hex mode, for example, you can use digits 0 - 9 and A - F. (On the HP-28S, you can override the current base by adding a lower-case letter b, o, d, or h immediately after the number digits. See also section 11.2.) The objects are always displayed in the current base regardless of how they were entered.

In addition to ordinary addition, subtraction, multiplication, and division of binary integers, the HP-28 provides a modest set of bit-shifting and logical operations as commands in the BINARY menu. For the four arithmetic operations, you can intermix

binary integer and real number arguments--the results will be binary integers.

3.4.4 Arrays

Array objects are the HP-28 representation of real or complex vectors (one-dimensional arrays) and matrices (two-dimensional). Arrays are identified by the square-bracket delimiters []. You enter a vector as a sequence of numbers surrounded by a single pair of brackets. To create a matrix, you enter an initial [, then each row of the matrix as a "vector" of numbers surrounded by [], with an extra closing] after the last row. If any of the numbers in an array entry is complex, the resulting object will be a complex array.

As in the case of number (scalar) objects, you can intermix real and complex arrays in calculations. You can also combine numbers and arrays for many operations, where it makes mathematical sense. For example,

$$2 \ [1 \ 2] * [2 \ 4].$$

However, you can't add a number to an array, since that is not a mathematically defined operation.

The ARRAY menu contains various commands relevant to arrays. Arrays are discussed at more length in Chapter 12.

3.4.5 Strings

String objects are character sequences that are interpreted as simple text. Strings are identified by the double quote delimiters ". The characters within the quotes can be any HP-28 characters, including the other delimiter characters, which have no special meaning in a string. You can use string objects to prompt for input or label output (section 11.7), or as data to be processed logically, such as names to be alphabetized by a sorting routine (section 12.3.3). The sequence "text" DROP can act as a program "comment" that has no computational significance but helps you document a portion of a program.

The STRING menu contains commands for performing simple string manipulations. These commands are straightforward, and are explained adequately in the HP-28 manuals. The use of string objects for saving and restoring display pictures on the HP-28S is explained in section 13.3.

3.4.6 Lists

A list object consists of a series of any types of objects entered between { } delimiters, which allows the series of objects to be manipulated as a single data object. Lists are described in detail in Chapter 12, and are used in numerous program examples throughout this book.

3.5 Procedure Objects

The concept of execution taking an object off the stack and performing the object's action is pretty trivial in the case of data objects, but is more complicated for name and procedure objects. We'll look at the latter first.

A *procedure* is an object defined as a series of other objects intended for sequential execution. A procedure corresponds to the conventional calculator concept of a "program." In an HP-41, a program is a series of numbered steps that are executed in numerical order, with occasional jumps due to GTO, XEQ, conditionals, etc. Each step either enters data, or performs a built-in command. The line numbers indicate the order of execution, but they really have no meaning other than for visual reference. If you take the line (step) numbers out of an HP-41 program, for example, and write more than one program line on a display line, you will end up with something that looks very similar to an HP-28 program.

You can therefore understand HP-28 procedure evaluation as the HP-28 equivalent of HP-41 program execution. In simplest terms, the action associated with a procedure is to execute in turn each object in the procedure's definition. Usually, execution proceeds in the order in which the objects were originally entered, but as in the case of HP-41 programs, the order can be altered by branches, loops, subroutines, etc.

There are two types of HP-28 procedures: *program* and *algebraic*. *Program* objects correspond most closely to HP-41 programs, as arbitrary sequences of any HP-28 objects and commands (see Chapter 10). *Algebraic* objects are special procedures that are restricted to contain only certain commands and objects, organized in a specific mathematical form. But in their internal representation, they are indistinguishable from programs that make the equivalent calculations using RPN logic (for example, the algebraic 'A+B' is the same as the program << A B + >>). When you evaluate an algebraic object, you execute its internal definition exactly as if it were a program. This concept is explored further in Chapter 9.

In all previous calculators, there is a distinction between user programs and commands:

- *Programs* are written in the user programming language, and are executed by means of a command like RUN, XEQ, GSB, etc., together with a program name or label

number. Programs can call other programs (subroutines), but there is a restriction on the number of pending returns of which the calculator can keep track (6 in the HP-41).

- *Commands*, on the other hand, are executed or entered into a user program by name, with no prefix command. In most calculators, “naming” a command consists of pressing the key that has the command name on it. This either executes the command, or enters a function code or the name itself into a program. The HP-41 allows you also to name a command by spelling it out using the **XEQ** **ALPHA** *name* **ALPHA** sequence.

The fact that the commands themselves are just internal programs is invisible to the user. First, the programs can't be viewed or edited. Second, they are written in the calculator's assembly language, which would require much more information for the typical user to understand and apply than can be provided in owners' manuals.

The HP-28 philosophy is that the distinction between user programs and built-in commands is artificial and unnecessary, at least as regards their use from the keyboard and as subroutines. That is, when you write a program and name it, you should be able to use it exactly as if it were a built-in command. When you enter a program name into the command line, and press **ENTER**, or include a program name in another program definition and execute the latter program, or just press a **USER** menu key labeled with the program name--the program should execute. The central idea underlying the execution of HP-28 name objects follows from these ideas.

3.6 Name Objects

So far we have seen that the execution of data objects just returns the same objects, and the evaluation of procedures “runs” the procedures. *Name* objects are designed to let you access an object indirectly, by specifying the name of an object rather than the object itself.

The concept of name objects is closely tied to that of HP-28 *variables* (Chapter 5). A variable is a combination of one name and one other object (the variable's *value*) stored together in memory. In many cases, it is not necessary to distinguish between a variable and the object stored in it, so that you can speak of a stored object as being named by the variable's name. Thus if you store a program in a variable named **PROG**, you can call the program **PROG**.

There are two types of name objects, which work with different types of variables:

- Names that are available in the **USER** menu are called *global* names, since the corresponding *global variables* are available at any time, from the command line or

in a procedure.

- *Local variables* are created for temporary use by programs. The corresponding *local names* can be executed only within the program that creates the variables.

Global variables are intended for storing data for general access, and for containing named programs that act to extend the HP-28 command set. With this in mind, global name objects are designed to work like commands:

- *Execution of a global name causes execution of the object stored in the global variable with that name.*

The net result of the execution of a global name follows directly from the execution action of the object stored in the corresponding variable:

- If the stored object is a data object or an algebraic object, it is returned to level 1.
- If the stored object is a program, the program is executed.
- If the stored object is another global name, then the object stored in the variable corresponding to the stored name is executed.

The fact that executing the name of a stored algebraic object returns the object to the stack without evaluation makes possible "step-wise" algebraic substitution. For example, consider evaluating 'A+B', where A has the value 'C+D', B is 5, C is 10, and D is 20. The HP-28 will return 'C+D+5' at the first use of EVAL, and 35 at the next. If an algebraic stored in a variable was automatically *evaluated* when the variable's name was executed, you would lose the intermediate step and obtain only the final result 35 at the first EVAL. The details of algebraic evaluation are discussed in Chapter 9.

In the case where a global name is evaluated for which no variable currently exists, the action is simple--the name itself is just returned to the stack as if it were a data object. This behavior is necessary for symbolic operations; it means the HP-28 can deal with symbols (names) even when no value has yet been established. Thus 'A+B', where A is undefined and B is 10, evaluates to 'A+10'. Evaluation of A returns 'A', B returns 10, and + combines the symbolic 'A' and the number 10 into a new symbolic 'A+10'. We call A a *formal* variable, meaning you can work formally with the name in calculations just as if there were an existing variable named A.

If a variable contains a global name, the stored name is executed when the variable's name is executed. Thus if the number 8 is stored in the variable A, and 'A' is stored in B, evaluating B returns 8. This property of names leads to the possibility of "endless loops"--if 'A' is stored in B, and 'B' is stored in A, evaluating either A or B will start an unending circle of evaluations. Pressing **ON** does not stop execution--the HP-28 acts as

if it were “locked up.” The lockup occurs because name evaluation is optimized for speed, and no **[ON]** key check is performed during that process. Your only recourse in this situation is to perform a system halt by pressing **[ON]** and **[Δ]** together (section 11.3.4).

Local variables are intended primarily for temporary storing and naming of stack objects, in order to simplify argument manipulations in programs. Local name execution is simpler than that of global names:

- *Execution of a local name recalls to level 1 the object stored in the corresponding local variable, without executing the object.*

The creation and use of local variables is described in Chapter 8, section 10.5.1, and section 10.7.

The properties of name evaluation listed here explain why **[RCL]** is relegated to a shifted key position on the HP-28 keyboard. For variables containing all object types except programs and names, you can use either **[EVAL]** or **[RCL]**. When a variable contains a program or a name, you are more likely to execute the stored object by executing the variable name than to recall the object. The primary purpose of **RCL**, therefore, is to recall a stored program to the stack without evaluating it, a relatively infrequent need.

You can view name objects as the HP-28 version of storage register numbers, but this simple picture doesn't really do justice to their power. Register numbers are purely passive labels, of the most primitive sort--they don't tell you anything about what is stored in the register. Names, on the other hand, label their variable contents with readable names that can help you remember what each variable does, and which make programs more legible. Furthermore, HP-28 names are active instead of passive: when you execute them, they cause automatic recall or execution of another object. In the simplest case, the other object is a data object, in which case the name execution is the same as **RCL**. But if the other object is a program, global name execution plays the role of **XEQ** on the HP-41.

Unquoted global names act just like built-in commands, so that you can define your own command set by storing programs in global variables. You can execute a global name by:

- Typing the name into the command line and pressing **[ENTER]**; or
- Pressing the **USER** menu key labeled with that name; or
- Including the name in a procedure (a program or an algebraic), and evaluating the procedure.

These three methods are identical for global name objects and HP-28 commands.

[As it happens, HP-28 commands are also programs written in a language that is a superset of the HP-28 user RPL, so there really is no structural difference between user programs and commands. The practical difference is that since built-in commands are fixed in read-only memory, they can be encoded in a program by a memory address and hence executed more quickly than user variables. The latter are referenced by name, and must be searched for in user memory whenever their names are executed.]

3.7 Quoted Names

We have shown that names automatically replace themselves with their associated variable values when executed. However, in many cases, you need the name itself on the stack, not the value, so that you can use the name as an argument for a command like STO or GET. You can accomplish this by enclosing the name within single quote delimiters, e.g. *'name'*. The quotes around a name instruct the HP-28 to return the literal name itself, and not to execute it.

To store the value 10 into a variable X, the correct sequence is 10 'X' STO. If you omit the quotes, as in 10 X STO, you may very well get an error, since the *value* of X is returned before the STO executes, rather than the name X. You *can* use 10 X STO if the variable X does not yet exist, since that case executing X just returns to the stack the name 'X', which is a suitable argument for STO. In general, you should keep the habit of entering the quotes around the name when you want to store, just to avoid uncertainty. However, if you're primarily performing symbolic calculations, you may want to take the trouble to purge all of the variables you want to work with, just so you can put the names on the stack without bothering with the quotes.

3.8 Quotes in General

There are three sets of quotation marks that are used as HP-28 delimiters:

- Single quotes ' ', (called "ticks," for short) which identify algebraic objects, and also create name objects on the stack;
- Double quotes " ", which create strings; and
- Program quotes (*guillemets*) << >>, which create programs.

All three types of quotation marks have a common theme in the HP-28. They mean "put this object on the stack--don't execute it yet." Preventing execution of a string object is not particularly meaningful, since strings are data objects, but we include the double quotes " " in this discussion for completeness. The double quotes primarily

distinguish text strings from names.

We stated in section 3.7 that placing single quotes around a name creates an unexecuted name object. The quotes play the same role for algebraic objects--the same symbol is used for the two different object types (name and algebraic) because it makes sense in many contexts to treat a name object as an algebraic expression consisting of just one variable name. As you will see in Chapter 9, an algebraic object is an executable program that happens to be in algebraic form rather than RPN. Again, the quotes mean "don't execute this program, just put it on the stack." The HP-28 doesn't allow you to specify an immediate-execute algebraic object (i.e., without quotes)--if you want the expression to be executed immediately, you have to enter it in RPN form.

Although the same delimiters are used for algebraics and names, and for many cases you can treat them the same, they are still different object types. The distinction is maintained for the sake of commands like PUT and RCL, which would make no sense with an expression or an equation as an argument. The HP-28 insures a smooth interaction between names and algebraics by treating them uniformly (as a general *symbolic* object type) as arguments for functions, and by automatically converting algebraics containing only a variable name into actual name objects.

Understanding the meaning of quoted and unquoted *programs* starts with the recognition that the contents of the command line constitute a program--an arbitrary series of objects for sequential execution. When you're carrying out keyboard calculations, the execution is immediate as soon as you execute ENTER (section 3.11). The command line program is created, then executed right away. However, you can postpone execution of the command line by inserting a << delimiter at the start. ENTER then creates a program object containing the command line objects.

Because the command line is a program, and programs are deferred-execution command lines, it follows that whatever you can do in the command line, you can also do in a program (and vice-versa). Thus programs can contain quoted objects: names, algebraics, and even other programs. For example, here is a program named TEST that creates a global variable containing yet another program:

```
<< ... << 10 * >> 'X10' STO ... >> 'TEST' STO
```

Executing TEST executes this program, which in turn creates a variable X10 containing the program << 10 * >>. Because of the surrounding << >>, the sequence 10 * is not executed, but is put on the stack as a program, where it and the quoted name X10 are the arguments for STO. X10 then appears in the USER menu; in this manner a program can create new named programs for later execution.

3.9 EVAL

As you saw in the preceding section, the various types of quote delimiters cause objects to be placed on the stack without being evaluated. The EVAL command is provided so that you can later evaluate these “pending” objects, particularly programs, names, and algebraics. Applying EVAL to a data object does in fact evaluate the object, but that just returns the same object.

EVAL is the closest HP-28 equivalent to the HP-41 “run” ($\boxed{R/S}$) key. To “run” a HP-28 program you can put it on the stack, then press $\boxed{EVAL}$.

Perhaps the most common use of EVAL arises in symbolic calculation, where you have entered an algebraic object and want to substitute values for the variable names that appear in the object’s definition. The evaluation of algebraic objects is described in detail in Chapter 9. The $\boxed{EVAL}$ key also provides a handy way of making a keyboard calculation in algebraic syntax, acting like an algebraic calculator’s $\boxed{=}$. Just press $\boxed{}$ to start algebraic entry, enter an expression, then press $\boxed{EVAL}$. For example,

$\boxed{}$ 1+2*3 $\boxed{EVAL}$ $\boxed{\rightarrow}$ 7.

The command $\rightarrow$ NUM is closely related to EVAL; it is described in section 9.2.2.

3.10 SYSEVAL

Built-in HP-28 program objects--commands--are permanently stored in the calculator. These objects are always in the same place in memory; any such object could in principle be executed by specifying its memory *address* rather than its name. In fact, this execution-by-address does take place within HP-28 system programs. Furthermore, the HP-28 contains many hundreds of objects that are not named, and which are consequently not directly executable from the keyboard. The majority of these objects are not useful for common HP-28 operations--those that are most useful have names to make them commands. However, some unnamed objects do have practical uses.

The SYSEVAL command provides for execution of any system object by means of its address. That is, you enter the object address as a binary integer object, then execute SYSEVAL, which in turn executes the specified system object. From time to time, in response to customers’ requests, Hewlett-Packard has published the addresses of a few system objects that help solve certain common programming problems.

For example, if a program creates a temporary display by means of DISP or other commands, that display will persist until the end of the program. You can cause the calculator to restore the normal stack display while a program is running by executing a certain system object with SYSEVAL. The address of the object depends on the HP-28 *version*:

Version	Address (hexadecimal)
HP-28C 1BB	#13A32
HP-28C 1CC	#13B2C
HP-28S 2BB	#25AFC

Two additional examples of using SYSEVAL with HP-published addresses are given in sections 11.11 and 13.2.1.

You must use extreme care when using SYSEVAL, for execution with an incorrect address may cause a system halt or a memory reset (section 11.3.4). When you execute SYSEVAL from the command line, or enter it in a program, you should do the following:

- Be sure that the address you are using is the correct one for the calculator version that you have. You can determine the version number by executing

■ **BINARY** **≡DEC≡** #10 SYSEVAL **ENTER** .

(Address #10 is the same on all HP-28's.)

- Be sure you enter the address correctly. This means not only getting all digits right, but also making sure that the number is correct for the current binary integer base. All of the SYSEVAL addresses listed in this book are given in hexadecimal, so you should execute HEX before entering the binary integer address. (Remember that including HEX in the command line does *not* affect the interpretation of binary integers entered in that same command line). On the HP-28S, you can append the character "h" to the address to insure that it is treated as a hexadecimal number.
- Do not attempt to single-step (section 11.3.5) programs containing SYSEVAL. If you need to do this, replace the sequences *#address SYSEVAL* with global names, where each name corresponds to a variable containing a program

<< *#address SYSEVAL* >> .

3.11 The Meaning of ENTER

The unmistakable hallmark of RPN calculators has always been the double-wide key **ENTER↑** that you see instead of the algebraic calculator **≡** key. The HP-28 also has that familiar key (minus the arrow), but the action of the key has changed along with the HP-28's other alterations of traditional RPN.

In the HP-41, ENTER↑ plays a dual role:

1. Pressing **ENTER** terminates digit entry, and enters a newly keyed number onto the stack. This is the fundamental purpose of **ENTER**.
2. **ENTER**, which is a programmable command, copies the X-register into Y and disables stack-lift. Note that this second role really has no necessary connection with the first; it just happens to be the way it was designed on the HP-35, and has remained ever since on HP calculators until the HP-28C.

In the HP-28, the two roles have been separated. **ENTER** itself retains the fundamental purpose of terminating entry and entering new objects onto the stack. The secondary role of duplicating the object in level 1 is provided separately by the command **DUP**. To preserve more keystroke consistency with the HP-41, the HP-28 **ENTER** key executes **DUP** if you press it when no command line is present, but you should recognize this as only a keyboard convenience, not a property of **ENTER** itself.

The basic definition of the HP-28 operation **ENTER** is:

Take the text in the command line, check it for correct object syntax, then treat it as a program and execute the objects defined there.

This is a much-elaborated version of the old “terminate-digit-entry and enter a number onto the stack,” but in simple cases, it amounts to the same thing. If you press a series of digit keys, then **ENTER**, you will end up with a number in level 1. The same key sequence on an HP-41 yields the same result, except that the number is also copied into the Y-register.

On the HP-41, you can also terminate digit entry by pressing any key other than a digit key. In effect, the non-digit key both terminates digit entry and executes its own key definition. But it is not correct to say that the key performs **ENTER**, because no second copy of the number is created, nor, in general, is stack-lift disabled. In the HP-28, however, many keys do literally execute **ENTER**, then their own definitions. This feature, called *implicit ENTER*, is provided for keystroke similarity with previous RPN calculators, and for keystroke efficiency. Pressing the **ENTER** key itself is called *explicit ENTER*.

An example of the use of implicit **ENTER** is the sequence **1** **ENTER** **2** **+**. This adds the 1 and the 2, just as it always has in HP RPN calculators. At the time you press **+**, the 2 is still in the command line; the implicit **ENTER** performed by **+** puts the 2 on the stack before the addition is performed.

Note, however, that the sequence **1** **ENTER** **2** **ENTER** **+** does not give the same results on an HP-41 and on the HP-28. The second **ENTER** on the HP-41 duplicates

the 2, so that the $\boxed{+}$ adds $2+2$ and leaves the 1 in the Y-register. In general, when you key in a number, then press $\boxed{\text{ENTER}}$ n times, you get $n+1$ copies of the number (up to four). On the HP-28, the same sequence produces only n copies--the first $\boxed{\text{ENTER}}$ moves the number from the command line into level 1; each subsequent press executes DUP once and makes one copy of the number.

[The remaining material in Chapter 3 is quite detailed; you may consider it as reference information, and skip these sections at a first reading.]

3.11.1 Key Actions and Entry Modes

Whether or not a particular key performs ENTER depends on the key type and the current entry mode. Any key does one of two things when you press it:

- The key acts as an *alpha* key, merely adding one or more characters to the command line.
- The key acts as an *immediate-execute* key, causing any other kind of action.

Given these definitions, we can sort HP-28 keys (including menu keys) into three types:

1. Keys that are always alpha keys. These include the letter, digit, and delimiter keys and the other miscellaneous symbols on the left keyboard, plus the program structure words in the BRANCH menu (section 10.2).
2. Keys that are always immediate-execute keys. These are keys that never add characters to the command line. Examples are $\boxed{\text{ENTER}}$, $\boxed{\text{LC}}$, $\boxed{\alpha}$, $\boxed{\text{SST}}$, and $\boxed{\text{CONT}}$.
3. Keys that may act as either immediate-execute keys or alpha keys, according to the current entry mode. These *mode-dependent* keys are the most common key type in the HP-28; all are command keys.

There are three entry modes that affect the mode-dependent keys. The HP-28 manuals call these *immediate entry mode*, *algebraic entry mode*, and *alpha entry mode*. The modes might more mnemonically be called *immediate mode*, *expression mode*, and *program mode*, respectively; these names help describe the primary purpose of the modes. Immediate mode is for ordinary keyboard arithmetic and command execution. Expression mode is for entering algebraic objects. Program mode is for entering programs.

The active entry mode determines whether pressing a mode-dependent key executes its command immediately or adds its command name to the command line. The rules are simple:

- In immediate entry mode, all mode-dependent keys execute their commands.
- In alpha entry mode (program mode), all mode-dependent keys add their names to the command line.
- In algebraic entry mode (expression mode), keys corresponding to *functions* (commands allowed in expressions) add their names to the command lines. A left parenthesis is added after the names of functions that use one or more arguments. Keys for *RPN commands* execute those commands.

Returning to the question of which keys perform ENTER and when, the general rule is that only immediate-execute keys *may* do ENTER. There are no cases where a key acting as an alpha key adds characters to the command line, and then also does ENTER. The next “rule” is that the great majority of immediate-execute command keys *do* perform ENTER. For example, all keys for commands that use stack arguments do ENTER. The implicit ENTER insures that pressing a command key applies the command to the most recently entered arguments, including those that are still pending in the command line. This saves you the extra **ENTER** keystroke that you would otherwise need.

The command keys that *do not* perform ENTER regardless of the entry mode are menu keys for the commands that control calculator numerical modes, and which require no arguments: **STD**, **DEG**, and **RAD** in the MODE menu, and **DEC**, **HEX**, **OCT** and **BIN** in the BINARY menu. Because the modes can affect the interpretation of command line numbers, these exceptions to the general implicit ENTER rule are provided to allow you to change the modes *after* you have started a command line.

Whether or not an immediate-execute non-programmable operation key performs ENTER generally depends on whether the operation uses or affects the stack. Those that do ENTER are:

- **ENTER**.
- **EDIT** and **VISIT** (these use arguments).
- **COMMAND** (the first time you press it).
- Solver menu keys, including **EXPR=**, **LEFT=**, and **RT=**.
- HP-28S CUSTOM menu keys.
- **SST** and **CONT**.

Those that don't do ENTER are:

- α , α LOCK (HP-28C) or **MENUS** (HP-28S), **LC** , **CHS** , **EEX** , and  .
- Cursor menu keys **INS** , **DEL** ,  ,  ,  , and  .
- **VIEW↑** and **VIEW↓** .
- Menu selection keys, **NEXT** , and **PREV** .
- **CATALOG** and **UNITS** .
- Non-programmable mode keys:

HP28C **+CMD** , **-CMD** , **+LAST** , **-LAST** , **+UND** , **-UND** , **+ML** , **-ML** , **RDX.** , **RDX** , **TRACE** , and **NORM** .

HP-28S **CMD** , **UNDO** , **LAST** , **ML** , **RDX** and **TRAC** .

- **OFF** (you can turn the HP-28 off, then on, without affecting the command line).

CLUSR is a special case, because its effects are so drastic. The key's action is mode-dependent, but its immediate-execute behavior is to spell CLUSR into the command line. It is not an ordinary alpha key, because it does perform ENTER in immediate- or algebraic-entry mode before adding CLUSR to the (new) command line. In alpha mode, it adds CLUSR to the existing command line without doing ENTER.

3.11.2 Controlling the Entry Mode

The preceding key-behavior rules may appear elaborate, but in actual use they are generally not difficult to master (in fact, you seldom need to think about them at all). This is due in large part to the fact that the HP-28 *automatically* changes its entry mode to match the objects that you enter. Also, you can manually change the entry mode for those cases when the HP-28's automatic choice is not what you want.

1. The default mode following an ENTER is immediate entry mode. This choice is derived from the traditional behavior of RPN calculators, where pressing a function key causes immediate execution of the function. When you type digits or letters to start a new command line, the HP-28 remains in immediate entry mode, as shown by the open box cursor:

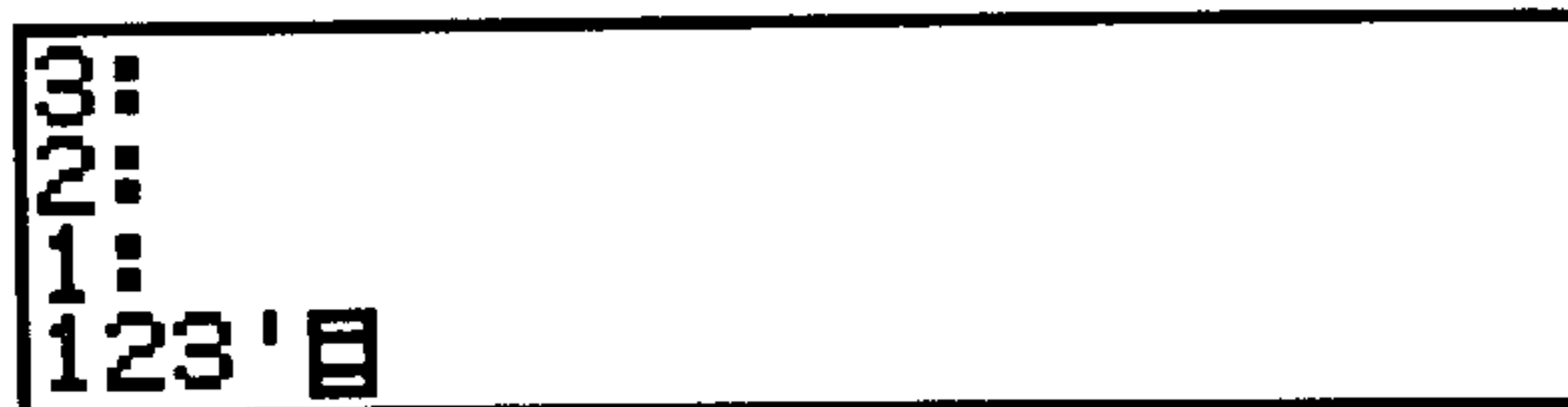
```

3:
2:
1:
123□

```

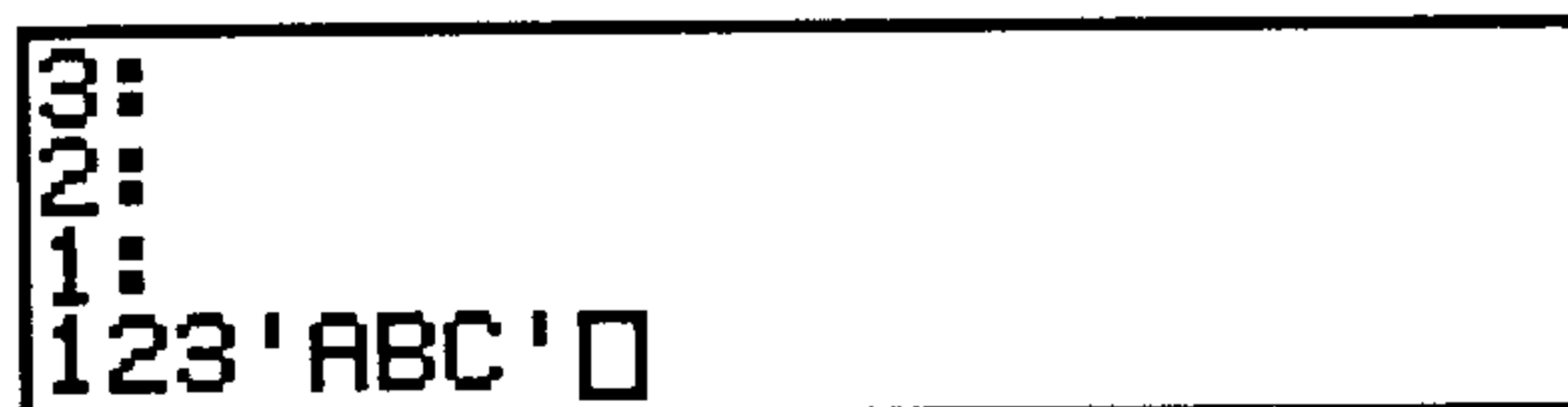
2. The HP-28 automatically changes to algebraic entry mode when you press  to

start entry of an algebraic object:

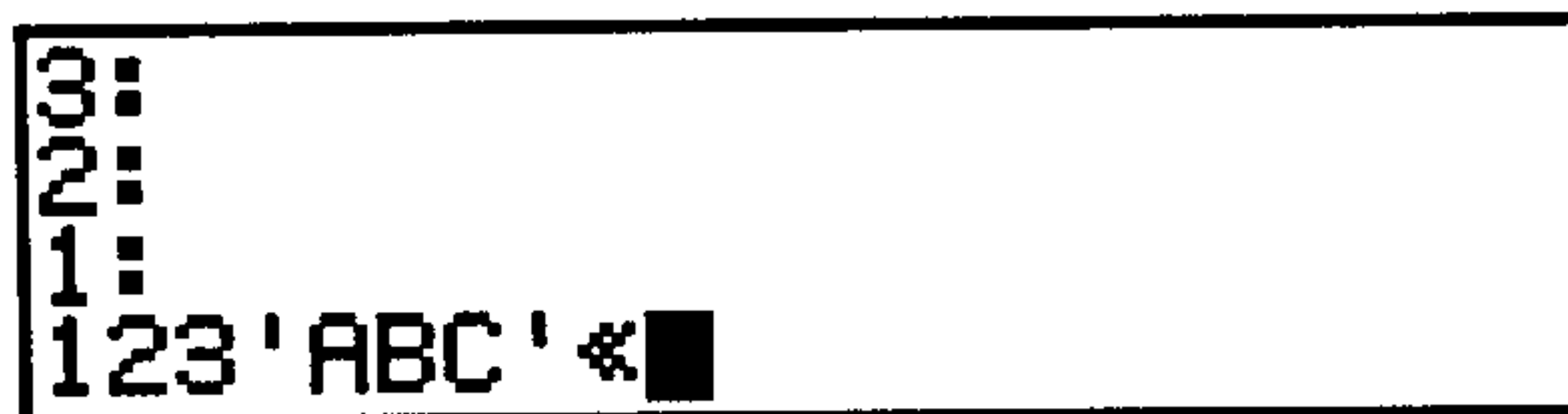


The cursor is now a box containing an = sign.

3. If you press $\boxed{=}$ again to terminate an algebraic object, the HP-28 reverts to immediate entry mode:



4. If you press $\boxed{\ll}$ to start entry of a program, or $\boxed{\text{"}}$ to begin a string object, the HP-28 automatically switches to alpha entry mode, indicated by the solid box cursor:



While the HP-28 is in alpha entry mode, the $\boxed{=}$ key does not change the mode.

This progression works reasonably well to spare you from having to control the entry mode yourself, especially if you are entering one object at a time. However, there are circumstances in which you need to override the HP-28's automatic choice. For example:

- The start list delimiter $\{$ does not activate alpha entry mode, but there are many cases where you need that mode to enter objects into the list. Creating a list of names by pressing USER menu keys is such a case.
- You are entering a program containing an algebraic object, and would like to switch to algebraic entry mode to suppress the automatic entry of spaces and to obtain the leading (after function names.
- You want to accumulate a series of commands into the command line without creating a program object.

The α key in combination with the $\square$ key allows you to change entry modes any time the command line is present. The specific keystrokes are listed in Table 3.2.

Table 3.2. Changing Entry Modes

Current Mode	Desired Mode	HP-28C Keys	HP-28S Keys
Immediate	Algebraic	$\square$ ($\square$)	α α
Immediate	Alpha	α	α
Algebraic	Alpha	α	α α
Algebraic	Immediate	$\square$ ($\square$)	α
Alpha	Immediate	α	α α
Alpha	Algebraic	α $\square$ ($\square$)	α

The backspace $\square$ is needed after the $\square$ on the HP-28C if you want to change modes without actually entering the algebraic delimiter.

3.11.3 ENTER in Detail

Now that we've established at some length which keys perform ENTER, and under what circumstances, we can return to the precise definition of ENTER. The simple use of ENTER to move a single number from the command line onto the stack is more straightforward on the HP-28 than on the HP-41--there is no second copy of the number, and no stack-lift disable. ENTER, however, can do much more than just put a number onto the stack, thanks to the flexibility of the HP-28 command line. The HP-28 manuals give a rough list of the steps involved in ENTER; here we will elaborate on those steps. Also, those manuals describe only the action of the $\overline{\text{ENTER}}$ key specifically, omitting the additional step that occurs when ENTER is performed implicitly.

Here are the steps that follow an explicit or implicit ENTER:

1. The "busy" annunciator is turned on. When this annunciator is on, it indicates that the HP-28 is not ready to process additional keys. Keys that are pressed (up to 15 keys) during this time are stored for processing after the current operations are complete.

2. The current stack is saved for UNDO. It is important to note that the UNDO save is performed *before* the command line is processed. If the ENTER is caused by an immediate-execute operation key, the stack save also *precedes* execution of the operation. This means that although breaking up a series of commands with ENTER (either explicit or implicit) gives the same computed results as executing all of the commands at once in a single command line, the results of pressing UNDO at the ends of the series will differ. For example, each of the following keystroke sequences adds 1+2 and returns 3 to the stack. But UNDO gives a different result in each case (assume an empty stack to start with):

Keystrokes:	Stack after UNDO:	
1 ENTER 2 ENTER +	2:	1
	1:	2
1 ENTER 2 +	1:	1
1 SPACE 2 +	(empty)	

3. The command line is converted into a series of objects--in effect, into a program. First, the command line is broken into *object strings*, individual portions of the command line text that will become objects. The object strings are defined by *delimiters* and *separators*:

- A *delimiter* is one of the symbols (,), ', ", [,], {, }, <<, >>, and #, that identify the different object types.
- A *separator* is either a space, a newline, or the *non-radix*--whichever of "." or "," is not the current radix mark. Separators are used to separate objects like real numbers, commands, and names, where no delimiter is used. Unlike delimiters, however, separators can be repeated--extra ones are ignored.

For example, the command line

12345.789 'FRED' "123" << DROP 'SAM' STO >> PETE

is broken into the object strings

12345.789
'FRED'
"123"
<< DROP 'SAM' STO >>
PETE

The process is repeated as necessary within algebraic objects, programs and lists,

which contain other objects. In the above example, the program object is further broken into the object strings DROP, 'SAM', and STO.

4. Each object string is checked against the syntax rules appropriate for its object type. As each object string passes its tests, an object is created from the string and pushed onto the stack. (This step is invisible--you won't see a stack display again until all of the new objects have been executed.) If any object string is found to violate a syntax rule, all of the newly created objects are dropped from the stack, and the command line is reactivated, with the cursor placed at the position in the command line where the error was encountered.
5. When the command line has successfully been converted into stack objects, a copy of the original text string is saved in the command stack (unless it has been disabled). Normally, this only happens if there are no syntax errors. However, if the HP-28 runs out of memory while it is creating the command line objects, the command line is saved, giving you a chance to try again after you have cleared some additional memory (see section 11.2). If the command stack is disabled, the command line text is never saved.
6. The new stack objects are combined into a program, which is then executed just like any other program.
7. If the ENTER was implicit, the operation associated with the key that started the ENTER is executed.
8. When the command line program plus the implicit ENTER key operation are finished, the HP-28 checks to see if there have been any keys pressed since the ENTER. If there have, those keys are processed. If one or more of the keys performs an ENTER, the "busy" annunciator blinks off momentarily and whole command line processing cycle starts over.
9. Finally, when all execution is complete, and no unprocessed keys remain, the stack is displayed (unless some special display supersedes the normal stack display) and the "busy" annunciator is turned off. Since the stack display can take an appreciable amount of time, the display is postponed when keys are pending to speed up the overall process.

This elaborate HP-28 ENTER process may seem like overkill compared to the relatively simple actions of HP-41 ENTER[†]. Remember, however, that you can use ENTER on the HP-28 pretty much the same as you would on the HP-41, after every data object that you key in--explicitly by pressing **ENTER**, or implicitly by pressing a command key. To make use of the many-objects-at-a-time capabilities of the command line, just think of the command line as a "instant" program. You write this program, execute it, and purge it all in one operation.

Here are three advantages of using command lines instead of immediate-execute command keys:

- You can repeat a sequence of commands without having to make the sequence into a program. Each time you execute the sequence, you can recover the command line with **■** **COMMAND** , then press **ENTER** to execute it again. You can also modify the sequence each time you execute it.
- If you get an unexpected result, you can press **■** **UNDO** to recover the stack, then **■** **COMMAND** to reexamine what you did. (Since **■** **COMMAND** itself does an implicit **ENTER**, you should press **■** **UNDO** *before* **■** **COMMAND** to recover the original stack.)
- It's the fastest way to execute the command sequence from the keyboard, since you don't have to wait for the stack display after each object is executed.

Because the command line is a program, you can do anything within the command line that you can in a program--create local variables, use program branch structures, **HALT**, single-step, set error traps, etc. On the HP-41, certain commands, like **ISG** or **X=0?**, are meaningless when executed from the keyboard, since they depend on a "next program line" for meaning. In the HP-28, all program commands are usable within the command line.

You can also turn the picture around and imagine a program as a command line for which execution is postponed. You can take any command line, insert a **<<** at the start, and then you have a program that enters level 1, rather than executing, when you press **ENTER** .

4. The HP-28 Stack

The HP-28 *stack* is the center of all calculator operations. It is the place where the great majority of commands find their arguments and return their results. It's also the primary and most efficient means for commands and programs to transfer data and instructions so that small calculations can be linked together. In this chapter, we'll describe the fundamental stack operations by which you can manipulate the objects on the stack. There are numerous practical examples of stack manipulations in the program examples in later chapters.

The stack consists of series of numbered *levels*, each of which contains one object of any type. The stack is always filled from the lowest level up, so that there are never any empty levels between full ones. ENTER always moves new objects from the command line into level 1, pushing previous stack objects up to higher levels. Most commands remove their argument objects from the lowest levels, whereupon the objects in higher levels drop down. The only exceptions are some of the stack manipulation commands, which can move objects to or from arbitrary stack levels. There is no limit on the number of objects or levels of the stack; you can enter as many objects as available memory will permit.

The HP-28 provides an extensive set of stack manipulation commands, some permanently assigned to keys, and the rest contained in the STACK menu. If you are familiar with previous HP calculators, you'll notice certain commands as renamed versions of traditional commands, and others which are new. Many of the discussions in this chapter deal with the transition from traditional HP-41-style RPN to the HP-28.

If you have no previous experience with HP RPN calculators, a good way to get used to the RPN stack is to view it at first as a "history" stack, which keeps a record of your calculations. That is, you can calculate in "algebraic" style, as we showed in sections 2.3 and 3.9, by entering algebraic expressions surrounded by ' ' delimiters, and pressing **EVAL** to calculate the results. The successive results pile up on the stack, where you can then start to get the "feel" of RPN by executing RPN commands to combine the results into new values.

Table 4.1 lists the stack operations found on the keyboard and in the STACK menu. The individual operations are explained in subsequent sections. [Most of the HP-28 stack commands are adapted from the FORTH computer language. Indeed, many key HP-28 features are based on FORTH, with its unlimited data and return stacks, RPN logic, and structured programming.]

Table 4.1. HP-28 Stack Manipulations

<i>Stack Clearing</i>	DROP DROP2 DROPN CLEAR
<i>Reordering Arguments</i>	SWAP ROT ROLL ROLLD
<i>Copying Objects</i>	DUP OVER PICK DUPN
<i>Counting Objects</i>	DEPTH
<i>Object Recovery</i>	LAST UNDO

4.1 Clearing the Stack

Perhaps the most common stack operation is “clearing” one or more objects, either to discard unnecessary objects so that others are moved to lower levels, or just to clear the decks for a new calculation. The standard command is **DROP**, which removes the object in level 1, and “drops” the remaining stack objects one level to fill in the empty level. Every time you press **DROP**, another object is discarded, and the stack drops one level. You can remove the entire contents of the stack in a single operation with **CLEAR**.

DROP and **CLEAR** correspond roughly to the HP-41 commands **CLX** and **CLST**, respectively. However, as we will discuss further in section 4.5.2, the HP-41 and the HP-28 differ considerably in their concepts of a “clear” stack. **CLX** clears the X-register, and **CLST** the entire stack, by replacing the contents of those registers with zeros. **CLX** is primarily intended for replacing the contents of X with a new value. By disabling stack lift, the zero **CLX** enters can be overwritten by a following entry. The HP-28 takes a simpler approach: **DROP** discards the level 1 object and doesn’t replace it at all. Since the HP-28 has no stack-lift disable, the next entry always replaces the dropped object.

CLST is of very limited value in the HP-41, serving only to provide a “supply” of zeros. HP-28 CLEAR, on the other hand, removes all objects from the stack, and recovers the associated memory.

There are two additional HP-28 stack clearing commands, DROP2 and DROPN, that are less drastic than CLEAR. DROP2 does just what its name implies: it removes two objects, from level 2 and level 1, then drops the remaining objects down two levels to fill in. The closest HP-41 equivalent to DROP2 is the sequence CLX RDN CLX RDN. DROPN drops $n+1$ objects, including a number n in level 1 that specifies how many objects to drop (see section 4.2.4 for a discussion of stack depth parameters).

4.2 Rearranging the Stack

4.2.1 Exchanging Two Arguments

The simplest form of stack rearrangement is the exchange of the positions of the objects in levels 1 and 2, which is accomplished by SWAP. SWAP is used for switching the arguments for a two-argument command, or more generally for changing the order in which the level 1 and 2 objects may be used. SWAP is easy to illustrate:

A B SWAP  B A.

The HP-41 counterpart to SWAP is $X<>Y$ (here at least is one case where an HP-28 command and an HP-41 command produce identical results).

4.2.2 Rolling the Stack

A stack “roll” is an exchange of stack positions involving objects in two or more stack levels. One object is moved to or from level 1, and other objects move up or down together to make room for it.

The concept of a stack roll is simple on the HP-41: you move all of the stack register contents by one level, up for roll up ($R\uparrow$), and down for roll down ($R\downarrow$ or RDN). The object that spills off the top or bottom of the stack is moved to the other end. Thus on the HP-41:

Register	Register contents		
	Before	After $R\uparrow$	After $R\downarrow$
T:	t	z	x
Z:	z	y	t
Y:	y	x	z
X:	x	t	y

Which way is “up” and which is “down” depends on how you picture the stack. HP calculator manuals have always pictured the stack with the X-register at the bottom, so that “up” means towards Y, Z, and T in that order. “Roll up” means x goes to Y, y to Z, z to T, and t rolls around to X.

The HP-28 provides a more general roll up/down capability with ROLL (roll up) and ROLLD (roll down). These work analogously to $R\uparrow$ and $R\downarrow$, respectively, but you must specify the number of stack levels you want to roll by placing the number in level 1. Each command drops the number from the stack, then rolls that number of the remaining stack objects. So 4 ROLL is equivalent to HP-41 $R\uparrow$, and 4 ROLLD is the same as $R\downarrow$:

Level	Stack Contents		
	<i>Before</i>	<i>After 4 ROLL</i>	<i>After 4 ROLLD</i>
4:	t	z	x
3:	z	y	t
2:	y	x	z
1:	x	t	y

Although ROLL and ROLLD move several objects at once, the primary purpose of these commands is still focused on level 1:

- n ROLL means “bring the n th level object to level 1.” That is, ROLL retrieves a previously entered or computed object that has been pushed up the stack by subsequent entries.
- n ROLLD means “move the level 1 object to level n .” ROLLD moves the level 1 object “behind” other objects that you want to use first.

SWAP and ROT (rotate) are one-step versions of ROLL. SWAP is equivalent to 2 ROLL; ROT is the same as 3 ROLL. 0 ROLL and 1 ROLL do nothing.

4.2.3 Copying Stack Objects

One of the strengths of RPN calculators is their ability to make copies of an object on the stack, so that you can reuse it without having to stop and create a variable. The simplest example of this facility is the HP-28 command DUP, which makes a second copy of the object in level 1, pushing the original copy to level 2, and all other stack objects up one level. The HP-41 counterparts of DUP are RCL X and ENTER $\uparrow$, although the use of the latter command is complicated by its extra feature of disabling stack lift.

The HP-28 also lets you copy a block of stack objects with DUPN. The sequence n DUPN, where n is a real integer, makes copies of the first n objects on the stack. The order of the objects is preserved; for example

X Y Z 3 DUPN $\Rightarrow$ X Y Z X Y Z.

DUP2 is a one-command version of 2 DUPN:

X Y DUP2 $\Rightarrow$ X Y X Y.

In some cases it is desirable to copy an object that is not in level 1, by bringing a copy to level 1 while leaving the object in its original position relative to other objects. In the HP-28, this combination of ROLL, DUP, and ROLL is represented by PICK, the general purpose stack copy command. PICK works like ROLL, returning the n th level object to level 1, but it leaves the original copy behind. The original therefore ends up in level $n + 1$:

W X Y Z 4 PICK $\Rightarrow$ W X Y Z W.

DUP is the same as 1 PICK, and OVER is a one-step version of 2 PICK:

X Y OVER $\Rightarrow$ X Y X.

Generally, you use PICK and ROLL when you are carrying out a complicated calculation entirely with stack objects. When you need to use a certain object repeatedly, you use PICK to get each new copy of the object. For the *final* use of the object, use ROLL instead of PICK; then you won't leave an unneeded copy around after the calculation is complete.

The HP-41 analogs of n PICK are RCL X, RCL Y, RCL Z, and RCL T, which are intended for making copies of stack numbers (without disabling stack lift). Note that RCL T is equivalent to $R\uparrow$, since the original contents of T are pushed off the stack.

4.2.4 How Many Stack Objects?

Several HP-28 stack commands require you to supply an argument that specifies how many stack levels the command will affect. Because this argument is always taken from level 1, you might be uncertain about what the argument should be--should you count level 1, which contains the argument? The answer is no--always count the stack levels you need before the count is entered into level 1.

For example, suppose the stack looks like this:

4:	D
3:	C
2:	B
1:	A

To roll D to level 1, execute 4 ROLL. But notice that at the point when ROLL actually executes, the stack is:

5:	D
4:	C
3:	B
2:	A
1:	4

Here D is actually in level 5. But don't try to compensate for this by using 5 as the argument to ROLL. ROLL removes its argument from the stack *before* it counts levels for the roll. All other similar commands, such as DUPN, PICK, ROLLD, →LIST, etc., work the same way.

DEPTH, which returns the number of objects currently on the stack, works in conjunction with this class of commands. The count returned by DEPTH does not include itself--it counts the objects before the new count object is pushed onto the stack. (Every time you execute DEPTH, the depth increases by one.) Thus DEPTH ROLL rolls the entire stack, DEPTH →LIST packs up all the stack objects into a list, etc.

4.3 Recovering Arguments

HP-41 LASTX recovers the X-register argument used by a previous command, for two general purposes:

1. To allow you to re-use the same argument for a new command.
2. To help you reverse the effect of an incorrect command, by applying the inverse of the command to the same argument.

These two purposes are split into separate operations on the HP-28:

1. The capability of recovering an argument for reuse is provided by the LAST command. It is important to note, however, that whereas HP-41 LASTX returns only the X-register argument, LAST returns *all* of the arguments used by most recent command--up to three arguments. (No HP-28 command uses more than three arguments. Commands like DUPN or →ARRY, which appear to use an indefinite

number of arguments, are considered for this purpose to use only one argument, which is the number or list in level 1 that specifies the number of stack levels that are involved.)

The arguments saved by HP-28 commands are kept in a special area of memory that is only accessible via LAST, rather than in an L-register that is almost an extension of the stack as in the HP-41. Also, a wider variety of HP-28 commands use stack arguments than is the case in the HP-41, so that the objects returned by LAST change more frequently in the HP-28. For example, DROP and SWAP both affect LAST, whereas HP-41 CLX and $X \leftrightarrow Y$ do not affect the L-register. Only commands like STD or HEX, that use no arguments at all, leave the LAST objects unchanged.

2. The use of HP-41 LASTX in recovering from incorrect commands is replaced in the HP-28 by the UNDO operation. At the start of ENTER, a copy of the entire stack is saved (see section 3.11.3). When all of the objects processed by ENTER have completed execution, you can cancel the stack effects of the objects by pressing  UNDO. This discards the new stack and replaces it with the stack contents saved by ENTER.

The objects saved for recovery by LAST and UNDO can consume a substantial amount of memory if the objects are numerous or large. In some cases, particularly on the HP-28C, where memory is in short supply, this use of memory can actually prevent you from carrying out various operations. For this reason, the HP-28 gives you the option of disabling either or both of these features (and also the command stack), by means of the appropriate keys in the MODES menu. You can also enable or disable LAST by setting or clearing flag 31, respectively.

Two notes:

- Disabling LAST prevents commands that error from returning their arguments to the stack. This makes it harder to recover from an error, and also affects the design of error traps (section 10.6.1).
- If there is insufficient memory available to save the current stack for UNDO, the HP-28 shows the error message No Room for UNDO, and *automatically* disables UNDO. This is necessary, since you would otherwise be unable to do anything until some memory was made available, yet any commands you tried to execute to free memory would fail because the HP-28 tries to save the stack first, which it can't do.

4.4 An Example of HP-28 Stack Manipulations

The following illustrates the use of several of the HP-28 stack commands. By following the steps, you can observe how to copy, move, and combine stack objects.

■ *Example.* Write a program that computes the three values

$$\begin{aligned} P + A + B \\ P + B \cdot F + A/F \\ P + B/F + A \cdot F, \end{aligned}$$

leaving the results on the stack. Assume that P is in level 4, A in level 3, B in level 2, and F in level 1.

■ *Solution:*

```
<< 4 ROLLD 3 DUPN 3 DUPN + +
    8 ROLLD 7 PICK * SWAP 7 PICK
    / + + 5 ROLLD 4 PICK
    / SWAP 4 ROLL * + +
>>
```

To help you understand this program, we will show the stack contents at each step, using using symbolic names throughout the calculation.

Steps:		Results:
(Start)	4:	'P'
	3:	'A'
	2:	'B'
	1:	'F'
4 ROLLD	4:	'F'
	3:	'P'
	2:	'A'
	1:	'B'
3 DUPN	7:	'F'
	6:	'P'
	5:	'A'

	4:	'B'
	3:	'P'
	2:	'A'
	1:	'B'
3 DUPN	10:	'F'
	9:	'P'
	8:	'A'
	7:	'B'
	6:	'P'
	5:	'A'
	4:	'B'
	3:	'P'
	2:	'A'
	1:	'B'
+ +	8:	'F'
	7:	'P'
	6:	'A'
	5:	'B'
	4:	'P'
	3:	'A'
	2:	'B'
	1:	'P + (A + B)'
8 ROLLD	8:	'P + (A + B)'
	7:	'F'
	6:	'P'
	5:	'A'
	4:	'B'
	3:	'P'
	2:	'A'
	1:	'B'
7 PICK	9:	'P + (A + B)'
	8:	'F'
	7:	'P'
	6:	'A'
	5:	'B'
	4:	'P'
	3:	'A'
	2:	'B'

	1:	'F'
* SWAP 7 PICK	9:	'P + (A + B)'
	8:	'F'
	7:	'P'
	6:	'A'
	5:	'B'
	4:	'P'
	3:	'B * F'
	2:	'A'
	1:	'F'
/ + +	6:	'P + (A + B)'
	5:	'F'
	4:	'P'
	3:	'A'
	2:	'B'
	1:	'P + (B * F + A / F)'
5 ROLLD	6:	'P + (A + B)'
	5:	'P + (B * F + A / F)'
	4:	'F'
	3:	'P'
	2:	'A'
	1:	'B'
4 PICK	7:	'P + (A + B)'
	6:	'P + (B * F + A / F)'
	5:	'F'
	4:	'P'
	3:	'A'
	2:	'B'
	1:	'F'
/ SWAP	6:	'P + (A + B)'
	5:	'P + (B * F + A / F)'
	4:	'F'
	3:	'P'
	2:	'B / F'
	1:	'A'
4 ROLL	6:	'P + (A + B)'

	5:	'P + (B * F + A / F)'
	4:	'P'
	3:	'B / F'
	2:	'A'
	1:	'F'
* + +	3:	'P + (A + B)'
	2:	'P + (B * F + A / F)'
	1:	'P + (B / F + A * F)'

4.4.1 The Easy Way: Local Variables

The preceding example illustrates the use of stack manipulation commands, but it does not necessarily represent the best way to solve the problem. Keeping track of numerous objects on the stack takes considerable care when you are writing or editing a program. In general, manipulating objects on the stack in a purely RPN manner yields the most efficient programs. However, there are other programming techniques that are easier and produce more legible programs.

If you were writing the above program on the HP-41, you would have to store the initial values in registers, then recall each to the stack as it is needed in the calculations. You can do the same thing in the HP-28, using variables instead of registers (Chapter 5), but this has the disadvantage of cluttering up user memory with a lot of variables, which you may or may not need after the program is finished. The cleanest method is to use *local* variables.

The solution to the problem in the preceding section, written using local variables, is

```

<<  → p a b f
    << 'p+a+b' EVAL
      'p+b*f+a/f' EVAL
      'p+b/f+a*f' EVAL
    >>
  >>

```

→ p a b f takes the four initial values off the stack and assigns them to local variables p, a, b, and f (here we are using the convention of lower-case characters for local names). The rest of the program computes the three results, then discards the local variables. The obvious advantage of this method is that you can write the program “instantly,”

since the program so closely resembles the written form of the expressions you are trying to compute. The use of local variables is explored in detail in Chapter 8 and section 10.7.

4.5 Managing the Unlimited Stack

In our review of HP-28 stack manipulation commands, we have described the differences between the HP-28 commands and the similar commands provided by the HP-41. In addition to these individual command differences, there are also several general aspects of the use of the HP-28 stack that will require some adjustment if you are used to the HP-41-style stack. The hardest part, perhaps, is changing keystroke and programming practices that you have developed to use the advantages and to overcome the disadvantages of a four-level stack. In the following sections, we will outline some of the significant differences in stack management between a four-level stack and an unlimited one.

4.5.1 Stack Housekeeping

The key difference, of course, between the HP-28 and HP-41 stacks is that the HP-28 stack can contain an unlimited number of objects, whereas the HP-41 and all other previous RPN calculator stacks have been limited to four registers. An important advantage of the unlimited stack is that objects are never lost by being pushed off the end of the stack when a new object is entered. This is also a mild disadvantage--if you don't clear objects from the stack when you're through with them, more and more objects will pile up. This not only wastes memory, but can even slow down execution, since the calculator has to keep track of all of the stack objects. It can also be distracting to see old objects appear in the display when you've long since forgotten their purpose.

A general recommendation for HP-28 stack management is to clean up the stack after a calculation is complete. By all means pile up as much as you want on the stack while you are working through a problem--that's what it's there for. But when you're finished, empty the stack. You can do this either at the beginning or the end of each calculation. We recommend the latter, since at that point you will best remember what each object is, and whether it's all right to throw it away.

"Clean up the stack" doesn't always mean to DROP every object, or to CLEAR the stack. You may very well want to keep certain objects, in which case you can store the desired objects in variables. Notice that the STO command removes the object being stored from the stack, reducing the number of objects on the stack.

KEEP1 and KEEPn are two programs that supplement the commands in the STACK menu. (This is our first example of a named program; you may wish to refer to the

description of the program listing format in section 11.1.) KEEP1 discards all objects on the stack *except* the object in level 1. For example,

A B C KEEP1  C

KEEP1		Keep 1 Object	
...		level 1	level 1
objects...		object ₁	 object ₁
<pre><< -> x << CLEAR x >> >></pre>		Save object₁. Clear the stack, recover object₁.	

KEEPN discards all objects after the first *n*, where you supply *n* in level 1.

KEEPN		Keep N Objects	
...		level 1	...
objects		<i>n</i>	 <i>n</i> objects
<pre><< ->LIST KEEP1 LIST-> DROP >></pre>		Combine <i>n</i> objects in a list. Discard all but the list. Put the saved objects back on the stack.	

Occasionally you may need to interrupt one ongoing keyboard calculation in order to perform another, and wish to resume the suspended work later. In this case it is not appropriate to clear the stack with CLEAR to provide an empty stack for the new calculation. You could take the trouble to save each object in a variable, but this can be tedious, and doesn't guarantee that you can reconstruct the stack order of the objects. The solution is to preserve the entire stack in a single variable by combining the stack objects into a list.

All of the commands you need are in the second level of the STACK menu. Press  **STACK**  **NEXT**  **DEPTH** to return the number of objects on the stack, and  **->LIST** to combine all of the objects into a single list object. Then you can store the list into a variable named OLDST (for example) by typing 'OLDST'  **STO**. The stack is now cleared for another calculation. After completing any number of subsequent operations, you can restore the old stack by pressing  **USER**  **OLDST**  **LIST**  **LIST->**  **DROP**. (The DROP removes the object count returned by LIST->.)

If you expect to do this kind of stack save frequently, you can write programs to save keystrokes:

```
<< DEPTH →LIST 'OLDST' STO >>
```

saves the stack, and

```
<< OLDST LIST→ DROP >>
```

restores it.

4.5.2 A Really Empty Stack

An important property of the HP-28 stack not shared by other RPN calculators' stacks is its ability to be *empty*. That is, when you clear the stack with DROP or CLEAR, there's *nothing* left. If you try to execute a command that requires arguments, you'll get an outright error--Too Few Arguments. The HP-28 makes no attempt to supply default arguments.

On an HP-41, the stack is never empty. CLX puts a zero in the X-register; CLST (clear stack) fills all four stack registers with zeros. This is handy if you happen to use a lot of zeros in your calculations, but the primary reason that the HP-41 works this way is that the stack registers are fixed memory registers that are always present. "Clearing a register" means resetting all the memory bits to 0, which happens to correspond to the internal representation of the floating-point number zero.

Zero is not such an obvious choice for a default value in the HP-28, since the calculator doesn't know what type of calculation you are working on. A null matrix, an empty string or list, or the complex number (0,0) are just as good choices as floating-point zero, depending on your current work. So the HP-28 avoids the problem by never supplying a default. When the stack is empty, it's really empty.

You can turn this property to advantage. The following sequence adds all of the numbers on the stack, no matter how many there are:

```
IFERR DO + UNTIL 0 END THEN END
```

The sequence DO + UNTIL 0 END is an endless loop (section 10.5.2.1) that repeats + indefinitely. Every time + executes, there is one fewer number on the stack, until eventually there is only one (the total) left. Then + will error (Too Few Arguments), but the error is intercepted (section 10.6) by the IFERR, and the program stops without any error display. You can embellish this sequence to label the result by inserting a display

message

"SUM= " OVER →STR + 1 DISP

between the THEN and the final END.

Notice that if an empty stack were treated as if it were filled with zeros, there would be no way for the program to know when to stop adding.

4.5.3 Disappearing Arguments

The HP-28 itself takes some steps to insure that unnecessary objects don't pile up on the stack. In particular, most commands that use stack arguments remove those arguments from the stack. You shouldn't find this surprising; for example, you wouldn't expect the sequence 1 2 + to leave the 1 and the 2 on the stack as well as the answer 3. But it may be a little disconcerting the first time you use STO on the HP-28, to see that the object you just stored disappears from the stack, especially by contrast with HP-41 STO, which leaves the stored object on the stack.

If commands did not remove their arguments from the stack, then you would have to take the trouble to drop them when you no longer need them. On the other hand, since HP-28 commands do remove their arguments, you must remember to duplicate them before executing the commands on those occasions when you want to reuse the arguments. The HP-28 chooses this approach for these reasons:

- Consistency with mathematical functions. You *never* want math functions to leave their arguments on the stack--otherwise, the whole RPN calculation sequence would be disrupted.
- Stack "discipline." The fewer objects that are on the stack, the easier it is to keep track of what they are.
- Efficiency. It's easier to duplicate or retrieve a lost argument than it is to get rid of an unwanted one.

To illustrate the last point, consider obtaining a substring from a string:

"ABCDEFGH" 3 4 SUB ⌵ "CD".

This sequence returns only the result string "CD"; the original string "ABCDEFGH", and the 3, and 4 that specify the substring are discarded. If you want to keep the original string, add a DUP to the start of the sequence:

DUP "ABCDEFGH" 3 4 SUB ⌵ "ABCDEFGH" "CD".

If SUB left its arguments on the stack, the original sequence would yield a final stack like this:

4:	"ABCDEFGH"
3:	3
2:	4
1:	"CD"

In that case, to leave only the result on the stack, you would have to add 4 ROLLD 3 DROPN to the sequence. If you only want the two strings, you would have to add ROT ROT DROP2. As we stated, either of these is more complicated than adding a DUP to the start of the sequence.

When you use STO to preserve an intermediate result in the middle of a calculation, you may prefer to keep the result on the stack so that you can continue the calculation. In this case, just execute DUP (press **ENTER** if you're just working from the keyboard) before you enter the variable name for the STO. If you forget, the stored object is always available by name in the USER menu.

4.5.4 Postfix Commands

Perhaps the single HP-28 feature that takes the most adjustment by HP-41 users is the extension of *postfix* syntax to commands that use a *prefix* form in the HP-41. Prefix commands are those like STO, SF (set flag), FIX, etc., where the command is specified *before* its arguments.

For these commands, the "arguments" don't go on the stack at all. When you're working with a fixed stack, you can't afford to lose objects from the end of the stack whenever you store an object or set the number of display digits. Instead, these HP-41 commands use a syntax that combines the argument with the command into a single unit. For example, when you press **STO**, the display shows STO _ _ , indicating that STO expects an argument in the form of a two-digit number (identifying the data register). You must then press two digit keys, following which the store is actually executed. When you include STO in a program, the STO *nn* is treated as a single program line.

One important reason for the HP-28 to abandon the prefix syntax is the inflexibility of that form. In the HP-41, STO always requires a two-digit prefix, which is a nuisance on a calculator with up to 319 data registers since you can directly store only in registers 00 through 99. It's not so bad for FIX, SCI, and ENG, since a single-digit argument allows you to specify all possible display formats. But since the HP-28 has 8-byte floating-point numbers, you can specify up to 12 mantissa digits, so that a 1-digit prefix syntax for these commands is inadequate.

The HP-28 uses a postfix syntax for all of its commands, including those that correspond to HP-41 prefix functions. This provides:

- Consistency--all commands work the same way. You don't have to remember which commands are prefix and which are postfix.
- Flexibility--no restrictions on the number of digits or characters in the arguments. FIX, SCI, ENG, SF, CF, etc., use floating-point numbers as arguments, so they are not limited to one or two digits. STO, RCL, etc., work with variable *names* with any number of letters.
- Computed arguments--arguments can be computed using any other HP-28 operations, as well as entered manually.

The last point means that there is no necessity in the HP-28 for "indirect" addressing as it is defined in the HP-41. Indirect addressing is the case for which a command argument is not contained as part of a prefix command syntax, but is stored in a register. The number of the register is specified as the argument for the indirect form of the command. For example, in the HP-41, the indirect command STO IND 01 means "store the number in the X-register into the register named in register 01." You use indirect addressing when you don't know a command argument in advance, but must compute it. This often occurs when you are working with a set of sequentially numbered data registers, as in matrix operations.

To reproduce the effect of indirect addressing in the HP-28, you can define a variable named INDEX to play the role of an index register. You store the name of the indirectly referenced variable in INDEX. Then the sequence 'INDEX' RCL STO is equivalent to STO IND 01 (where we have arbitrarily chosen register 01 to be the index register) on the HP-41, or to STO(i) on the HP-11C or the HP-15C. Assuming that the object you want to store is already in level 1, executing 'INDEX' RCL puts the name stored in INDEX onto the stack, pushing the first object to level 2. Then STO is ready to go, with its arguments correctly positioned on the stack.

You will probably find that there are few occasions when you need to use indirect addressing of this form on the HP-28, because it already provides indexed structures--arrays and lists. For example, suppose you want to create a sequence consisting of the reciprocals of the integers 1 through 10. In the HP-41, you would write a program as follows:

01 LBL"RECIP"	
02 1.010	Set up an index for 1 through 10
03 LBL 01	
04 ENTER↑	Copy the index
05 INT	Take the integer part
06 1/X	Reciprocate
07 STO IND Y	Store the reciprocal
08 RDN	Discard the reciprocal
09 ISG X	Increment the index
10 GTO 01	Loop
11 END	

This program stores the 10 reciprocals into registers 01 through 10. An HP-28 equivalent is the following program **RECIP**. This program returns the reciprocals together as the elements of a list object.

RECIP <i>Compute 10 Reciprocals</i>	
<i>level 1</i>	
☐ { <i>reciprocals</i> }	
<pre><< 1 10 FOR x x INV NEXT 10 →LIST >></pre>	Put loop parameters on the stack Start a loop with x as the index Compute the reciprocal of x Loop. Pack up the 10 reciprocals into a list.

4.5.5 Stack-lift Disable

"Stack lift" is the RPN calculator process whereby entering a new object "lifts" the previous contents of the stack into higher levels. In the HP-41, it is possible to disable stack lift, so that a no-argument operation that returns a result to the stack will overwrite the contents of the X-register with the result rather than lifting the stack. This feature, which originally derived from the memory limitations of the first RPN calculators and has carried over into later products, is not present in the HP-28.

The purpose of stack-lift disable centers around the behavior of CLX in a one-line display calculator. The first RPN calculators were characterized by very limited memory. They could not afford the luxury of having a separate register for digit entry, so numbers were keyed directly into the X-register. Usually you don't want to overwrite the number already in X, so the calculators provided that entry of a new number "lifts" the existing stack contents into higher registers. Now suppose that you start to enter an

incorrect number, and start pressing $\leftarrow$ to remove it. What happens if you press the back-arrow too many times, at least once after the last remaining digit of the newly entered number? You wouldn't want that to erase the number that was previously in the X-register, so $\leftarrow$ was designed to stop erasing in this case and just enter a zero into the X-register. But since that zero has nothing to do with your calculation, and will most likely just be in the way, the zero entry is coupled with a stack-lift disable. The next number you enter overwrites the phantom zero. For similar reasons, CLX works the same way (and, in fact, $\leftarrow$ just performs CLX when digit entry is not active).

Why doesn't CLX just drop the stack? The answer is a psychological one. Many people, especially if they're used to algebraic calculators (which typically require you to press the clear key to start new calculations), like the sense of "a clean slate" suggested by a zero. An alternative might be to show a blank display, but then you wouldn't know if the calculator was turned on. Although with an RPN stack it's never necessary to clear anything to begin a new calculation, a zero in the display when you start is less intrusive than some other number that you might not recognize.

The HP-28 eliminates the need for stack-lift disable by providing a command line, where you can enter an object without disturbing the stack during entry. Since an object in the command line is not on the stack, you can abandon it (ON), or $\leftarrow$ repeatedly to empty the line) without having to leave a zero as a placeholder on the stack. Furthermore, if you want to get rid of an object that's already on the stack, you can use DROP, which discards the object and drops the remaining objects on the stack one level.

The HP-41 commands that disable stack-lift are $\text{ENTER}\uparrow$, CLX, $\Sigma+$, and $\Sigma-$. The first two commands each serve dual purposes. CLX can be used either to eliminate an unwanted number from the stack, or just to enter a zero. $\text{ENTER}\uparrow$ acts either to terminate digit entry, or to copy the X-register into Y. On the HP-28, the two roles of each of these operations are separated, as shown in Table 4.2 in the next section. The HP-41 summation commands $\Sigma+$ and $\Sigma-$ disable stack-lift in order to return a running count of the number of data points that have been accumulated to the X-register, while allowing you to overwrite the count with the next data entry if you wish. The HP-28 does not attempt to reproduce this behavior; when you execute $\Sigma+$, it just removes the data from the stack and adds it to the ΣDAT matrix; no count is returned. If you like seeing a running count, you can use the following program instead of $\Sigma+$:

```
<<   $\Sigma+$   N $\Sigma$   1  DISP  >>.
```

This program displays the count as a temporary display in display line 1 after each entry.

4.6 HP-28 Translations of HP-41 Stack Commands

Table 4.2 lists the nearest HP-28 equivalents of the HP-41 stack manipulation commands.

Table 4.2. HP-41 and HP-28 Stack Commands

HP-41 Command	Purpose	Nearest HP-28 Equivalent
CLX	Remove last entry Enter a 0	DROP 0
CLST	Clear the stack Enter four 0's	CLEAR 0 DUP DUP2
X<>Y	Exchange X and Y	SWAP
R↑	Roll up four levels Roll up the entire stack	4 ROLL DEPTH ROLL
RDN	Roll four levels down Roll down the entire stack	4 ROLLD DEPTH ROLLD
ENTER↑	Terminate digit entry Duplicate X into Y	ENTER or any delimiter or separator character DUP
LASTX	Recover last argument Correct an error	LAST UNDO
RCL X	Copy X	DUP
RCL Y	Copy Y	OVER
RCL Z	Copy Z	3 PICK
RCL T	Copy T	4 PICK

5. Variables

The HP-28 stack has evolved from the fixed-size number registers of the HP-41 and its predecessors into a more flexible mechanism for operations with objects of varying types and sizes. For similar reasons, the HP-28 introduces a general storage system in which objects are stored in named *variables*. Variables are similar to stack levels in that each holds one object of any type. But whereas the stack levels are numbered, and the order of stack objects is important, HP-28 variables are named with words, and their order is unimportant. The names are the means by which you access the objects stored in variables, and they also serve to label the variables. Variables replace the fixed-size numbered data storage registers of the HP-41; the variable names act as a mnemonic form of register “number.” With numbered registers, you must keep a mental or written record of what is stored in each register. By choosing appropriate variable names, you can substantially reduce the need for such records on the HP-28.

There are two kinds of HP-28 variables: *global* and *local*. Most of the discussion in this chapter focuses on global variables, which are those variables that constitute the USER menu and are generally accessible from any program. Local variables are “local” to specific programs; they are discussed in detail in Chapter 8 and section 10.7 .

In HP-28 terminology, a variable is “a combination of a name object and any other object that are stored together in memory.” Looking at each part of this definition:

- *A combination...* A variable has two parts, the name and the other object. You can visualize a variable as a labeled box--the name is the label, the other object is the content of the box. The label is permanent, but you can change what is stored in the box.
- *of a name...* You always refer to a variable by its name. The name fills the role of the storage register number in the HP-41.
- *and any other object...* The contents of a variable can be any single object, of any size or complexity. You can only store one object in a variable, but if you want to store more, you can combine any number of objects into a list, then store the list in a variable.
- *that are stored together...* The name and the object are in fact contiguous in memory, but it is the logical combination that is significant, not the details of the storage.
- *in memory.* Global variables are all stored in a portion of memory we'll call USER memory because of its association with the USER menu. Local variables are stored in one or more temporarily assigned sections of memory that are reclaimed when the program that created the variables is finished executing.

HP-28 variables are the electronic version of mathematical variables. In an expression like $x + 2$, you understand that x stands for some specific quantity (the value of x). The implication is that at any time you can substitute the value for the symbol, and actually carry out the calculation prescribed by the expression. In the HP-28, name objects are the symbols for variables. Executing a name replaces the symbol with the variable's value, which is the object stored in the variable.

You can only use an object stored in a variable by means of the variable name. For this reason, it is often convenient to blur the distinction between *names* and *variables* by using the terms interchangeably, particularly in a mathematical context. We speak, for example, of the variables in an equation--actually, only the names are literally present in the equation. A *formal* variable in the HP-28 is a name for which no variable exists. The name potentially represents a value of some kind; the value becomes definite when you store an object in the variable.

Also, in many situations it is convenient to refer to the object stored in a variable by the name of the variable. For example, if you create a program object, then store that program in a variable named PROG, you can say that you have named the program "PROG." Strictly speaking, this is not correct since the program has an existence independent of any name (for instance, you can recall the program object to the stack, where it is nameless). What you have done is *associated* the program and the name by combining them into a variable. The association may be temporary, or you may keep it indefinitely; but as long as it is current it makes sense to refer to the program simply as PROG rather than as "the program stored in the variable named PROG."

5.1 Creating Global Variables

HP-28 global variables are created by the command STO, which takes a name and one additional object as its arguments. STO moves the object from the stack to USER memory where it and the name are added to the current set of variables that appear in the USER menu. HP-28 variables have no particular object type associated with them; you might create a variable initially with one type of object, but later you can store any other type of object into the same variable.

STO expects to find the name of the target variable in level 1, and the object to store in level 2. As described in section 3.7, you must *quote* the variable name in order to enter it as an object onto the stack. To store 25 in a variable X, for example, you execute 25 'X' STO. If you forget the quotes and execute 25 X STO, you will most likely see an error message, since X will be executed rather than entered as a name, before the STO is executed. (If you know that there is no current variable X, then you can omit the quotes, since executing the name of a non-existent variable just returns the name back to the stack. This can save you a couple of keystrokes when you're creating a *new*

variable.)

There is no direct way to change the name of a variable. However, you can easily move the contents of a variable to a new variable with a different name by executing

'old-name' *'new-name'* OVER RCL ROT PURGE SWAP STO

The order of these operations ensures that there is only one copy of the stored object at a time.

5.2 Recalling Values

There are two fundamental ways to “recall” the value of a variable:

- *Execute the variable's name.* Executing a global name executes the object stored in the named variable (section 3.6). For data objects and algebraic objects, this just recalls the object to the stack. For example, if you have stored the number 25 in a variable named X, pressing **X** **ENTER** returns the number 25 to level 1.
- *Use RCL.* *'name'* RCL returns the object stored in the variable *name* to the stack, without executing the object. RCL is primarily used for variables containing programs and names, in cases where you just want to put a copy of the stored object on the stack. For data and algebraic objects, RCL has the same effect as simple execution of a variable's name, which requires fewer steps.

The commands GET and GETI allow you to recall individual elements from arrays and lists stored in variables, without having to recall the entire object to the stack. For GET, the stack use is

object index GET  *element*,

where *index* specifies the element to retrieve:

- For a list, the index is a real number (the HP-28S also allows the real number itself to be in a list, which is handy when you're converting between lists and vectors).
- For a vector, the index is a list containing one real number (the HP-28S also allows a real number not in a list).
- For an array, the index is a list of two real numbers representing the row and column numbers of the element (the HP-28S also allows a real number representing the element number, counting in “row order”-- left to right, top to bottom).

The *object* in the above sequence can either be the list or array itself, or the name of a

variable in which the list or array is stored. Thus,

$$\{ A \ B \ C \} \ 2 \ GET \Rightarrow 'B',$$

or

$$\{ A \ B \ C \} \ 'D' \ STO \ 'D' \ 2 \ GET \Rightarrow 'B'.$$

GETI is designed for sequential recall of the elements in a list or array, and returns the object or its name, and the index incremented to the next element, as well as the recalled element. The general form of GETI is

$$object \ index \ GETI \Rightarrow object \ index+ \ element,$$

where *object* and *index* are the same as for GET, and *index+* is the same as *index* except that its value is incremented to represent the next element. Thus,

$$\{ A \ B \ C \} \ 2 \ GETI \Rightarrow \{ A \ B \ C \} \ 3 \ 'B',$$

If *index* points to the last element, GETI returns either 1, { 1 }, or { 1 1 } for *index+*, as appropriate to cycle back to the first element. [In the HP-28S, GETI also sets flag 46 when this occurs, or clears the flag otherwise, so that a program can easily determine when it has come to the end of a list or array.]

5.3 Altering the Contents of Variables

The most straightforward means of changing the contents of a variable is to store a new object into the variable using STO. However, there are a number of commands that let you modify the stored object short of replacing it entirely, without having to recall the object to the stack.

The STORE menu contains seven commands of this type. The first four are the "storage arithmetic" commands STO+, STO-, STO*, and STO/, which are modeled after their HP-41 counterparts. The remaining commands SNEG, SINV, and SCONJ are single-argument commands that are not present in the HP-41.

In addition to the commands in the STORE menu, the four array commands CON, IDN, RDM, and TRN can be applied to arrays stored in variables. PUT and PUTI are the storing counterparts of GET and GETI, allowing you to alter individual elements in a stored list or array.

Finally, there are several commands associated with the reserved-name variables EQ, PPAR, ΣDAT, and ΣPAR associated with the Solver, plotting and statistics operations.

We will discuss these commands in the chapters describing their respective applications.

5.3.1 HP-28 Storage Arithmetic

Storage arithmetic commands let you combine a number in level 1 with a number stored in a variable, without having to recall the latter to the stack. For example, 25 'X' STO+ adds 25 to a number stored in X. More generally, STO+, STO−, STO*, and STO/ use a syntax similar to that of STO:

object 'name' STO●,

where the ● stands for any of the symbols +, −, *, or /. However,

'name' *object* STO●

is also allowed. Either sequence combines the *object* in level 2 with the object stored in the variable *name*, leaving the result stored in the same variable. The object and the name are dropped from the stack. The two objects that are combined must be numerical--real or complex numbers or arrays--and suitable for the ● operation (you can't add a matrix to a real number, for example). For STO−, STO* (for arrays) and STO/, the order of the *object* and the 'name' is important:

- *object* 'name' STO● computes

$$(\text{new value}) = (\text{stack object}) \left\{ \begin{array}{c} + \\ - \\ * \\ / \end{array} \right\} (\text{old value}).$$

In this case, STO● is equivalent to

DUP RCL ROT SWAP ● SWAP STO.

If X has the value 1, then 3 'X' STO− stores 2 in X.

- 'name' *object* STO● computes

$$(\text{new value}) = (\text{old value}) \left\{ \begin{array}{c} + \\ - \\ * \\ / \end{array} \right\} (\text{stack object}).$$

Here $\text{STO} \bullet$ is equivalent to

$\text{OVER RCL SWAP} \bullet \text{ SWAP STO.}$

With 1 stored in X, 'X' 3 $\text{STO}-$ stores -2 in X.

You can understand these two choices by realizing that $\text{STO}-$, $\text{STO}*$, and $\text{STO}/$ combine a named object with a stack object just as if you replaced the name on the stack with the named object, then executed $-$, $*$ or $/$, respectively. For $\text{STO}+$, the order doesn't matter.

The flexibility in the order of the stack arguments is possible because these commands only work with numerical arithmetic--the stack object can't be a name. For ordinary STO , the object to be stored can be a name; to distinguish between the variable name and the stored name, STO requires the variable name to be in level 1 and the stored object to be in level 2.

5.3.2 Additional Storage Commands

In addition to the four storage arithmetic commands, the HP-28 has three storage commands that alter a stored number or array without requiring any stack argument other than the variable name. In each case, the result replaces the original object:

SNEG negates the stored object.

SINV computes the reciprocal of a stored number or square matrix.

SCONJ computes the complex conjugate of the stored object.

Like the four storage arithmetic commands, these commands are provided to save keystrokes and/or program memory when compared to the equivalent stack object commands used in conjunction with STO and RCL . However, the single-argument commands are particularly useful for arrays. Because their mathematical operations are applied to the stored arrays "in place"--replacing the stored values as the computation proceeds, the storage commands offer the most memory-efficient method of finding the negative, inverse, or conjugate of an array. If, for example, you recall an array from a variable to the stack, then invert it, you will need enough memory to hold both the stored array and its inverse at the same time. By inverting the array in place with SINV , you only need enough room for the original array.

Similar considerations apply to certain array commands that work equally well when you replace the array on the stack with the name of a variable containing an array:

- CON converts an arbitrary array into a constant array (all elements are the same), where the constant number is specified on the stack.
- IDN converts a square matrix into the identity matrix.
- TRN transposes and conjugates an array.
- RDM redimensions an array according to the dimensions specified by a list of one or two real numbers. Note that RDM can change the total size of an array if the new dimensions correspond to more or fewer elements than are in the original array.

PUT and PUTI allow you to store individual elements into an existing array or list, using a syntax similar to that of GET and GETI (section 5.2).

For example,

```
{ A B C } 2 'D' PUT → { A D C }.
```

Here the target list itself is on the stack. The target can also be identified by name:

```
'MAT' { 3 3 } 25 PUTI → 'MAT' { 3 4 }
```

stores the number 25 in the 3-3 element of a matrix stored in the variable MAT, and leaves the name and the incremented index (here assumed to indicate the 3-4 element) on the stack.

5.4 Purging Variables

In section 4.5.2, we discussed the differences between an “empty” stack in the HP-28, and a “clear” stack in the HP-41. Similar considerations apply to HP-28 variables compared with HP-41 storage registers.

In the HP-41, when you reserve memory for a certain number of storage registers using the SIZE command, the registers are “created” each with the initial value zero. This choice of a default value is often convenient, such as for cases where you wish to use a register as a counter or an accumulator that starts at zero.

An HP-28 variable, on the other hand, doesn't exist until you create it with STO. Furthermore, when you purge a variable, it is entirely removed from memory. If you try to execute 'name' RCL, when no variable with that name exists, you get a Undefined Name error message, the analog of the HP-41 NONEXISTENT message. Because variables can hold any objects, not just numbers, there is no such thing as a “clear”

variable--no empty version of a purged variable left in USER memory. You can certainly store zero in a variable if you want to use it as a counter or accumulator, but in general, the real number zero is no better choice as a default value than any number of other objects--an empty list, a null matrix, etc.

The ordinary use of PURGE follows the syntax '*name*' PURGE. This sequence removes the variable called *name* from USER memory. If the USER menu is active, with the *name* label showing in the display, you will see that label disappear, and the other labels move over to fill in the vacant position. As an added convenience, you can execute PURGE with a list of names: { *name*₁ *name*₂ ... *name*_{*n*} } PURGE simultaneously removes the variables *name*₁, *name*₂, ..., *name*_{*n*}. The easiest way to purge several variables is to press **USER** **α** **[]**, then press the menu key for each variable you wish to purge, followed by **ENTER** **■** **PURGE**. The **ENTER** turns off alpha mode so that **■** **PURGE** executes PURGE rather than just adding the command name to the command line.

5.5 Grouping Variables

An advantage of numbering rather than naming variables is that a number scheme makes it easy to index the variables and to "group" variables into logical blocks. For example, the HP-41 Extended Functions module (built into the HP-41CX) includes two commands for manipulating a blocks of registers: REGMOVE copies the contents of one block of registers into another; REGSWAP exchanges the contents of two blocks. There are two general purposes for these commands:

1. To save the contents of a block of registers in a second block, so that a new program can use the original registers without destroying the first program's data.
2. To allow you to write programs that access data in a specific block of registers, then use those programs with different sets of data stored elsewhere in the calculator without having to rewrite the programs. For example, a program might use data in registers 0 through 9. You could have alternate data in registers 10 - 19, 20 - 29, 30 - 39, etc. Prior to each execution of the program, you would use REGSWAP or REGMOVE to move one of the alternate data blocks into registers 0 - 9.

There are no direct equivalents for these commands in the HP-28. The flexibility of HP-28 variables allows you to achieve the above two purposes by using different approaches:

1. By making variable names unique to each program, there is never any reason for conflict between programs trying to use the same variables (unless the programs are deliberately exchanging data via variables). Better yet, you can use local

variables for temporary storage that is guaranteed to be unique to a program, without worrying at all about two programs using the same variable names.

2. All of the data used by a program can be combined in a list (or in an array if all of the data are real or complex numbers). By writing the program to use a list as an input, you can easily supply alternate data sets by choosing different lists, each of which can be stored in an appropriately named variable.

Lists and arrays provide all of the advantages of numbered registers for purposes of data indexing, and have the additional advantage that the data they contain can easily be manipulated as a unit. These ideas are explored in more detail in Chapter 12.

5.6 The USER Menu

The USER menu is a visible and operational listing of the contents of USER memory--the current collection of global variables. When you create a variable, its name automatically appears in the USER menu; when you purge a variable, its name disappears from the menu. Pressing a USER menu key automatically executes the name that appears in the menu label above the key.

New variable names are always added to the beginning of the menu, the left end of the first menu level that appears when you press **USER**. In this sense, the USER menu is a "last in, first out" arrangement almost like the stack (except that executing a name doesn't drop it from the menu). "First out" also means "found first." When a name is executed, the HP-28 searches USER memory for the corresponding variable, starting with the newest variable and continuing in the reverse order of creation.

You can reorder the variables at any time by using ORDER. ORDER takes a list of names, and moves the variables in USER memory so that the order of names in the USER menu matches the order in the list. Variables not named in the list remain in their current order, following those variables that were named. If you have a lot of variables in USER memory, you can obtain a slight improvement in a program's execution speed by using ORDER to move the variables named in the program to the start of the USER menu.

5.7 HP-28S Directories

The simple linear structure of the USER menu becomes inadequate once you exceed more than a few levels (groups of six) of variables. The HP-28S, which has enough memory to hold dozens or even hundreds of variables, addresses this problem via *directories*.

A directory is a special type of variable that contains, in effect, a complete independent USER memory consisting of a sequence of additional variables, possibly including more directories. When a directory name is executed, the contents of that variable become the active USER menu. The following illustrates this meaning by way of example.

Imagine that you have performed a memory reset, so that there are no global variables. If you press **USER**, all you see is six blank menu key labels. Create three variables A, B, and C using **STO**, so that the labels C, B, and A appear in the USER menu. Now execute 'DIR' **CRDIR**--a new label DIR appears. If you press **DIR**, the menu key labels become blank again. Pressing **DIR** switches the USER menu to the new DIR directory, which is empty since you have not yet created any variables there. When you do create more variables, say D, E, and F, these appear in the menu.

Meanwhile, what's become of A, B, and C? They still exist--in fact, if you press **A** **ENTER**, for example, the value of A is returned as usual. Now press **MEMORY** **HOME** **USER**; the labels C, B, A, and DIR reappear. If you press **D** **ENTER**, the name 'D' ends up on the stack, just as if the variable D didn't exist. All of this behavior is determined by the HP-28S rules governing name *resolution*, described below.

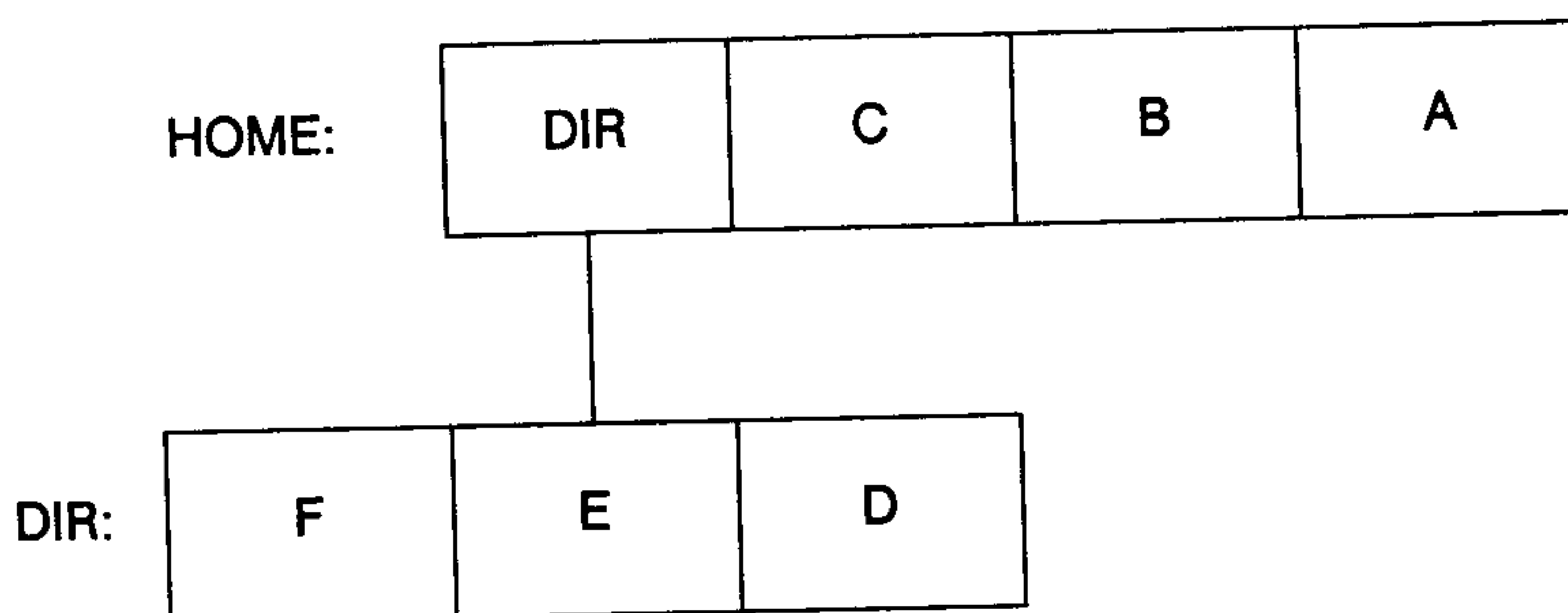


Figure 5.1. Example USER Memory

Figure 5.1 is a schematic representation of our hypothetical USER memory. The top level, containing DIR, C, B, and A, is called the *home directory*. This directory always exists--it's the "base" USER memory. The command HOME acts like a built-in directory name: when you execute HOME, the home directory becomes active, or *current*. The current directory is the directory

- that is shown in the USER menu; and
- is where most commands that deal with variables seek those variables.

The next level down in the picture is called the DIR directory. Executing DIR makes that the current directory. DIR is said to be a *subdirectory* of the home directory; similarly the home directory is called the *parent* directory of DIR. DIR itself can contain additional directories, and so on indefinitely. Any directory can be a parent or a subdirectory (the home directory can only be a parent), depending on the vantage point.

The purpose of directories is threefold:

- To let you organize the USER menu such that variables of common purpose appear together;
- To permit the creation of utility variables that are used by programs in one or more directories, but are not visible in the directory where the keyboard application variables are collected;
- To provide a measure of variable protection, by giving you a place to put programs and data that you can use while reducing the chance that you might accidentally change or replace them.

These purposes are achieved by a) the hierarchical structure of the directories as they are organized into parents and subdirectories, and b) the rules of *name resolution* used by the HP-28S. Name resolution is the method by which the HP-28S finds a variable when its name is executed or used as a command argument.

You can understand the basic rules of name resolution by considering the two most fundamental name operations: execution and STO. Name *execution* is the means by which you “use” (execute) the contents of a variable by specifying its name (unquoted, or quoted with EVAL or →NUM). The idea of a “utility” variable implies that you should be able to use it even when it’s not in the current directory. Therefore, when a name is *executed*, the HP-28S searches not only the current directory, but also its parent directories all the way up to the home directory, until it finds a variable with the right name.

STO, on the other hand, overwrites (or creates) a variable. In the HP-28S, you are prevented from storing into a variable that you can not “see”—that is, a variable that is not in the current directory and so is not visible in the USER menu. Instead of searching up through parent directories, STO looks only in the current directory. If it can’t find the named variable there, it creates a new variable, regardless of whether there is one or more identically named variables in other directories.

These principles of variable creation and execution suggest that you use a USER memory organization like that shown in figure 5.2. The home directory in this scheme contains

- “Application” variables, which when executed by name activate third-level subdirectories containing groups of related programs.
- Subdirectories containing application programs plus utility programs specific to the application.
- General purpose variables that may be used (executed) by any program in memory, or used in the command line no matter what directory is current.

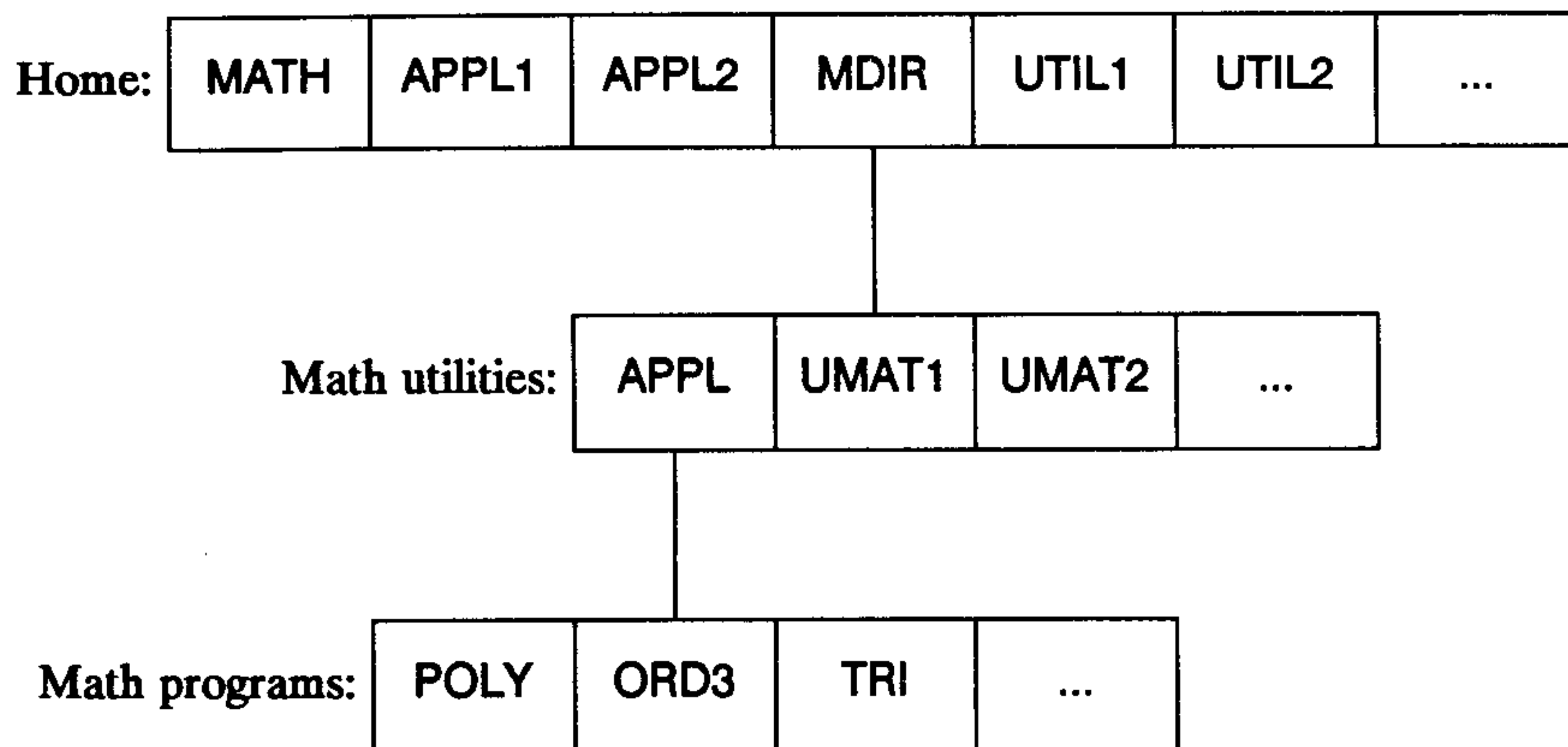


Figure 5.2. Example USER Memory Organization

The example home directory “application” variable in the figure is MATH. Pressing MATH in the USER menu activates the third-level subdirectory APPL. This subdirectory contains the keyboard-useful application programs, POLY, ORD3, TRI, etc., that are associated with the MATH key. These programs use subroutines named UMAT1, UMAT2, etc., which are stored in the directory MDIR that is the parent directory for APPL. The variable MATH contains the program << MDIR APPL >>, which first makes MDIR the current directory, then APPL. When you execute MATH, therefore, you bypass the math utility subdirectory MDIR containing the programs and activate the APPL subdirectory.

The programs in the APPL directory are those you are likely to use from the keyboard. These programs can use any of the utility programs in MDIR or in the home directory. But while the APPL directory is current, the USER menu contains only keyboard-useful programs--you’re not distracted by seeing utility programs in the menu. Also, while APPL is current, you don’t have to worry about unwittingly overwriting one of the utility programs.

5.7.1 Name Resolution in Detail

We have looked so far at two types of name resolution, typified by execution and STO. The great majority of HP-28S operations that deal directly or indirectly with variables follow the STO model, even if they only "use" the contents of variables without altering them. For these operations, name resolution is very simple: only the current directory is searched. The execution model of resolution--searching parent directories as well--is very much the exception.

There are a number of operations that you might consider as variations of STO, in that they create or alter the contents of global variables. Without exception, all of these operations affect only variables found in the current directory. This is true for operations that specify a variable by means of an explicit name argument, like STO, PUT or CON; and for operations that affect certain specially named variables without having the names on the stack, such as STEQ (variable EQ) or SCLΣ (PPAR). The Solver and ROOT are included in this general category, both when you store a value in a variable by means of the Solver menu, and when the root-finder computes and stores a new value for the unknown variable. Table 5.1 is a list of all such operations.

Table 5.1.
Operations That Create or Alter Global Variables

- *Variables named on the stack:* CON, IDN, PURGE, PUT, PUTI, RDM, ROOT, SCONJ, SINV, SNEG, STO, STO*, STO+, STO-, STO/, TRN, VISIT. Also STORE menu keys.
- *Variable EQ:* STEQ.
- *Variable PPAR:* *H, *W, AXES, CENTR, DRAW, DRAX, DRWΣ, INDEP, PMAX, PMIN, PIXEL, SCLΣ.
- *Variable ΣDAT:* CLΣ, Σ+, Σ-.
- *Variable ΣPAR:* COLΣ, CORR, COV, LR, PREDV.

Most, but not all, commands that *use* without altering the contents of a variable also restrict their search for the specified variable to the current directory, as listed in Table 5.2.

Table 5.2
Operations That Use the Contents of Global Variables in the Current Directory

- *Variables named on the stack:* GET, GETI.
- *Variable EQ:* RCEQ.
- *Variable PPAR:* DRAW, DRAX, DRWΣ, PIXEL.

- *Variable* Σ DAT: CORR, COV, DRW Σ , MEAN, MAX Σ , MIN Σ , N Σ , RCL Σ , SCL Σ , SDEV, TOT, VAR, Σ +, Σ -.
- *Variable* Σ PAR: CORR, COV, DRW Σ , PREDV, SCL Σ .

COV, CORR, DRAW, DRAX, DRW Σ , PIXEL, and SCL Σ appear in both of the preceding tables, since although they nominally don't affect the contents of PPAR, Σ DAT, or Σ PAR, they will create default versions of those variables in the current directory if the variables don't already exist.

There are three exceptions to the general rule of searching only the current directory for a named variable: 1) the execution of a name, either directly in a procedure or the command line, or by EVAL or $\rightarrow$ NUM; 2) RCL; and 3) PRVAR. We will focus on execution, but keep in mind that the following discussion applies to RCL and PRVAR.

When a name is executed, the HP-28 attempts to find the corresponding variable in the current directory. However, if a variable with the specified name is not in the current directory, the search continues up into the parent directory. If there's no luck there, the parent of the parent is searched, and so on up to the home directory if necessary. As soon as the first occurrence of a variable with the specified name is found, the name is said to be *resolved*, and the object stored in the variable is executed. If no variable of the correct name is found anywhere in the search, the name itself is returned to the stack.

Viewed in the reverse order of a name resolution search, a sequence of directories starting with the home directory and continuing with a series of subdirectories leading to a particular directory, is called a *path*. A path is effectively a prescription for making the last directory in the series the current directory. The PATH command returns the path to the current directory, called the *current path*, as a list of directory names. If you execute each name in the PATH list in turn, you'll end up reactivating the directory that was current when PATH was executed. (See the program DOPATH in section 5.7.3.)

The rules of global name resolution can be summarized concisely in terms of the path concept:

Global name execution (EVAL and $\rightarrow$ NUM), RCL, and PRVAR, search the current path upwards for a named variable. All other global name operations search only the current directory.

It is important to note that a path search for a name never goes "sideways." When a directory is searched, all of the variables explicitly named in that directory are checked, from left to right in the menu. However, the variables contained in subdirectories are

not checked. This makes it possible for more than one variable to have a particular name; only the first such variable in the current path will be found. If a variable is in a directory not in the current path, it will not be found.

5.7.2 Directories as Variables

[The material in this section is quite detailed; you may consider it as reference information, and skip this section at a first reading.]

At the beginning of section 5.7, we referred to a directory as a special type of variable. The HP-28S owner's manuals avoid any suggestion that a directory is a variable--an object and a name stored together. But directory names are just global names, just like those of ordinary variable names, and directory names and other variable names are mixed indistinguishably in the USER menu. In this sense, you can think of a directory as a variable containing a new object type. This type of object is essentially an ordered collection of other variables. When you execute it, its execution action is to make those variables the current directory.

Most commands that work with named variables do not work with directories. The situation is analogous to that of commands like CON or STO+, which fail if their argument specifies a variable that contains an inappropriate type of object (e.g. other than an array for CON). It is the current contents of the named variable that determine whether such commands will succeed, not the name. Two reasons dictate that most HP-28S variable commands must fail with directories:

1. The command doesn't make sense. Mathematical commands obviously can not act on directories. This eliminates commands like CON, STO+, and ROOT.
2. Memory protection. A directory may contain a significant number of objects that you wouldn't want to destroy accidentally. Thus, commands like STO that might overwrite the directory contents are not allowed to apply to directories.

Attempting to apply an inappropriate operation to a directory will return one of several error messages:

1. For operations that otherwise work with variables containing any type of object, the Directory Not Allowed error is returned: **PPAR**, PRVAR, RCEQ, RCL, RCLΣ, ROOT, STEQ, STO (and STORE menu keys, which execute STO), STOΣ, and VISIT. CRDIR itself returns this error if the specified directory already exists.
2. If you create a directory named with one of the reserved names, any command that works with the reserved name will fail:
 - Invalid PPAR (i.e., PPAR is a directory): *H, *W, AXES, CENTR, DRAW, DRAX, DRWΣ, INDEP, PMAX, PMIN, PIXEL.

- Invalid Σ DAT (i.e., Σ DAT is a directory): CORR, COV, DRW Σ , LR, MEAN, MAX Σ , MIN Σ , N Σ , SCL Σ , SDEV, TOT, VAR, Σ +, Σ -.
- Invalid Σ PAR (i.e., Σ PAR is a directory): COL Σ , CORR, COV, DRW Σ , LR, PREDV, SCL Σ .
- Certain commands that work with any name require specific object types to be stored in the target variable. These will return Bad Argument Type. CON, GET, GETI, IDN, PUT, PUTI, RDM, SCONJ, SINV, SNEG, STO*, STO+, STO-, STO/, TRN.

Notice that even a command as innocuous as RCL does not work with a directory. There's no fundamental reason for this--except that the HP-28S is not able to display the *contents* of a directory on the stack. The substitute for RCL in this context is VARS, which returns a list containing the names of all of the variables in the current directory, in the same order as they appear in the USER menu. Thus for example, if you want to print the contents of all of the variables in a directory, you just make that directory current by executing its name, then execute VARS PRVAR.

The only operations that *do* work with directory names are the following:

Execution by name, or by EVAL or $\rightarrow$ NUM, makes a directory current.

PURGE will remove a directory from memory, but only if it is empty. This is a precaution to prevent accidental erasure of the directory's contents. You can purge a non-empty directory by making it current, then using CLUSR, or PURGE on a list of the variables in the directory, then going back to the parent directory and purging the empty directory.

When PURGE is applied to a list of names, it proceeds left-to-right through the list, purging the named variables until it finishes the list or encounters a non-empty directory. In the latter case, the Non-Empty Directory error is returned, and the remaining variables named in the list are not purged.

CLUSR purges all of the variables in the current directory. You can view CLUSR as equivalent to VARS PURGE--again, if there is a non-empty subdirectory in the current directory, only those variables that precede the subdirectory in the USER menu are actually purged, and the Non-Empty Directory error is returned.

ORDER rearranges variables in the current directory so that their order matches that specified in a list of names. ORDER works perfectly well if one or more names in the list refer to subdirectories, with the important reservation that in order to move a subdirectory, there may need to be enough

room in memory to make a copy of the entire subdirectory. A copy is required when any of objects stored in variables anywhere in the subdirectory (including in its subdirectories) have been copied to the stack or the UNDO stack, the last argument stack, or to a local variable; or when any of the objects are currently executing procedures.

If ORDER fails (Insufficient Memory) because there is no room in memory to copy a large subdirectory, your easiest recourse is to perform a System Halt ($\boxed{\text{ON}}$ - $\boxed{\Delta}$), then try ORDER again.

CL Σ purges the Σ DATA variable in the current directory. It is equivalent to ' Σ DATA' PURGE, and will work if Σ DATA happens to be an empty directory.

The fact that directories are variables can lead to results that may appear surprising but are not errors. For example, a variable containing a directory can not appear in the Solver variables menu, nor can it act as the independent plot variable. If a directory variable is executed during the course of execution of a command that evaluates a user procedure, the resulting change of directory often leads to unexpected results. If A is a directory, and you execute 'A+B' EVAL, A becomes the current directory, and the contents of B are added to whatever happens to be in level 1 (remember that the expression is equivalent to the sequence A B +).

5.7.3 USER Memory Utilities

The following programs are utilities for use with HP-28S directories.

- CURR** returns the name of the current directory.
- SUBD** returns a list of the subdirectories in the current directory.
- UP** activates the parent of the current directory.
- DOPATH** "executes" a path--activates the directory that was current when the list in level 1 was returned by PATH.
- FIND** searches the current directory and all of its subdirectories for a variable specified by its name, and returns a path list for the directory in which it finds the variable. An empty list indicates that the variable was not found.
- UFIND** searches all of USER memory for a specified variable, returning a path list for the directory in which the variable is found. An empty list indicates that the variable does not exist anywhere in memory. UFind does not change the current directory.

Purge works just like PURGE, except that it *will* remove a non-empty directory.

Clusr works just like CLUSR, except that it *will* remove non-empty subdirectories in the current directory.

CURR	<i>Return Current Directory Name</i>
	level 1
	☐ 'name'

```
<< PATH DUP SIZE GET
>>
```

SUBD	<i>Find Subdirectories</i>
	level 1
	☐ { names }

```
<< { }
  31 SF
  VARS
  IF DUP2 SAME
  THEN DROP
  ELSE
    1
    DO GETI
      IFERR RCL
      THEN 4 ROLL SWAP +
        3 ROLL
      ELSE DROP
      END
    UNTIL 46 FS?
    END DROP2
  END
>>
```

Start an empty list.
 Activate LAST.
 Get the list of all variables.
 If the VARS list is empty...
 ...then we're done.
 ...otherwise, examine the list.
 Initialize an index.
 Get the next variable name.
 If RCL fails, this is a directory.
 Add the name to the list.

 Discard the recalled object.

 Keep going to the end of the VARS list.
 Discard the VARS list and the index.

SUBD sets flag 31.

UP	<i>Activate the Parent Directory</i>

```
<< PATH
  DUP SIZE 1 - 1 MAX
  GET EVAL
>>
```

Get the current path.
 Index of the parent name, or HOME.
 Activate the parent.

DOPATH <i>Execute a Path</i>	
<i>level 1</i>	<i>level 1</i>
{ <i>path</i> } 	
<pre><< 1 DO GETI EVAL UNTIL 46 FS? END DROP2 >></pre>	Initialize the list index. Execute the next name. Keep going to the end of the list. Discard the list and index.

Purge <i>Purge Any Variable</i>	
<i>level 1</i>	<i>level 1</i>
' <i>name</i> ' 	
<pre><< 31 SF IFERR PURGE THEN DUP EVAL Clusr UP PURGE END >></pre>	Activate LAST. If PURGE fails, this is a non-empty subdirectory. So switch to that subdirectory, empty it, and then go back and purge it.

Purge sets flag 31.

Clusr <i>Clear Entire Directory</i>	
<pre><< VARS LIST→ 1 SWAP START Purge NEXT >></pre>	Get a list of variables. Set up a loop. Purge each variable.

Note that Purge and Clusr call each other recursively (see section 11.12).

FIND <i>Find a Variable</i>	
<i>level 1</i>	<i>level 1</i>
<i>'name'</i>	<i>{ path }</i>
<pre> << 0 → name dodir << << CURR 1 DISP VARS DUP IF SIZE THEN 1 DO GETI DUP 2 DISP IF DUP name SAME THEN DROP 1 SF PATH 'name' STO ELSE IFERR RCL THEN EVAL dodir EVAL UP ELSE DROP END END UNTIL 1 FS? OVER 1 == OR END DROP2 ELSE DROP END >> 'dodir' STO 1 CF dodir EVAL IF 1 FS? THEN name ELSE {} END CLMF >> >> </pre>	Save the name.† Create the dodir subprogram: Show the directory name‡ Get a list of variables in this directory. If the list is not empty... ...Initialize an index. Get the next name. Display the name.‡ Is this the one?... ...set flag 1 to indicate success, ...and replace the search name with the path. ...Otherwise, Is it a directory? ...then search it ...and come back. ...otherwise, drop the name. Keep going until found or the list is exhausted. Discard the list and index Empty list, discard it. End of subprogram. Name the program dodir. Now start the actual search: Set flag 1 when found. Make the search. If found... ...Else return empty list. Restore the normal display.‡ Flag 1 is set if the variable was found, clear otherwise.

†Here, besides saving the name, we are creating a local variable dodir, into which we will store the program object to be created next. We create the variable before the program, so that the name dodir will be interpreted as a local name in the program, which calls itself.

‡These lines are optional. Include them if you want the program to display directory and variable names as it searches.

UFIND		<i>Universal Find a Variable</i>	
<i>level 1</i>			<i>level 1</i>
'name'		→	{ path }
<< PATH SWAP HOME FIND SWAP DOPATH >>		Get the current path. Find the variable. Restore the original current directory.	

6. Problem Solving

The rich command set of the HP-28 allows you to solve many problems merely by pressing a few keys. However, where the HP-28 really excels is in the ease with which you can link command sequences together into procedures. This allows you to solve complex problems by breaking them down into simple pieces. Once the procedure corresponding to a problem's solution is developed and stored, you can execute it any number of times while you vary the input data.

The term *programming* is conventionally used for the process of recording a sequence of calculator instructions in such a manner that you can later replay the sequence any number of times without having to reenter the instructions. Here, we will use the more general term *problem solving* to describe the various HP-28 solution strategies, of which programming--creating program objects--is just one of several.

A problem solution generally consists of three parts:

1. Data input;
2. Data processing and calculations;
3. Results output.

Each of these stages can be simple or complicated. To input data, for example, you can use a program that just takes one or more objects from the stack that are presumed to be there when the program is executed. Or, your program can prompt for each required value by halting with a text display that asks you for a specific input. Similarly, a program can return its results to the stack, or it can display each result with an identifying text label.

Regardless of the complexity of a calculation, in most calculators the only method of automating calculations is to create a program, complete with labels and line numbers. While this restriction has the virtue of simplicity in that there are no alternatives, the process can be cumbersome for simple procedures, particularly for straightforward mathematical expression evaluation. The HP-28 provides a series of problem solving alternatives, ranging from simple expression evaluation to programs with loops, branches, recursion, etc. Problem solving can be both simpler and more complicated than in other calculators. In general, it is easier to program any given calculation on the HP-28; additional complication only arises really when you are dealing with problems that are not soluble at all on other calculators.

The HP-28 problem solving alternatives sort roughly into four approaches:

- The Solver;
- User-defined functions;
- Symbolic math;
- Programs.

These are listed roughly in order of increasing complexity; not so much in the complexity of the mathematics involved but rather in the amount of mental effort you need to translate a real problem into HP-28 terms. The classification is somewhat imprecise because there's a great deal of overlap, such as programs that contain user-defined functions; Solver exercises that use programs; even algebraic objects that execute programs. With all of these options, your challenge is to determine which approach is most appropriate for a particular problem.

In the remainder of this chapter, we will show which types of problems are suitable for each general problem solving method. Then in subsequent chapters, we'll review each method in detail.

6.1 The Solver

The "Solver," which is essentially a combination expression-evaluator and root-finder, provides perhaps the easiest method of problem solving on the HP-28. It is suitable for any problem that can be reduced to a single equation relating all of the variables in the problem, and for which a real-valued numerical answer is sufficient. The greatest benefit of the Solver is that you don't have to solve the equation formally for the unknown--all you have to do is enter any equation that relates the unknown to the known variables. Furthermore, you can interchange the roles of known and unknown variables as you go along, without doing any additional work to restate the problem.

A prototype problem ideal for the solver is the simple "cost-of-travel" equation:

$$\text{COST} = \text{DISTANCE} \times \text{PPG} / \text{MPG},$$

where PPG stands for "price per gallon," and MPG stands for "miles-per-gallon." This single equation relates all the relevant parameters, and has the virtue of containing only simple arithmetic operations, so that there is only one possible solution for any choice of values for any three of the variables. To address this problem with the Solver, all you have to do is enter the equation in algebraic form as written above, press **SOLV** **STEQ** to select it as the current equation, then press **SOLVR**. The calculator presents you with the *Solver menu*, which provides a menu key for each of the four variables, that you can use to store values in any three of the variables and solve for the fourth.

Contrast this simplicity with the process you have to follow on a calculator like the HP-41. For each choice of unknown variable, you have to

- a. Solve the equation formally (on paper) for the unknown;
- b. Translate the solved equation to RPN logic;
- c. Enter the RPN form as a series of program steps, where the variables are represented by numbered registers;
- d. Add input prompting steps to the start of the program, and output labeling to the end.

If you're very clever, you can figure out how to combine the four separate programs into one, where the program figures out from the inputs which variable is to be calculated and thus which branch of the program to use--in other words, to duplicate what the HP-28 Solver does for you automatically.

The Solver is described in detail in Chapter 7.

6.2 User-Defined Functions

The subjects of *local variables* in general and *user-defined functions* in particular are not given a great deal of attention in the HP-28 owner's manuals. Local variables provide a means of naming stack arguments, and thereby substantially increase the ease of writing programs. User-defined functions, which are a special case of the use of local variables, represent a simple yet powerful problem solving tool for:

- Evaluating algebraic expressions as functions of one or more arguments;
- Creating new functions for use in algebraic objects.

Creating a user-defined function is only a matter of

1. Naming the stack arguments, by using $\rightarrow$ followed by a sequence of (local) names.
2. Specifying the function in HP-28 algebraic notation, using the local names already listed.

Consider the trip cost example from the previous section. You can represent a solution for the trip cost by a function of three variables, e.g. $\text{COST}(\text{distance}, \text{price-per-gallon}, \text{miles-per-gallon})$. COST is defined as follows:

```
<<  $\rightarrow$  dist ppg mpg 'dist*ppg/mpg' >> 'COST' STO.
```

COST is a function that takes three arguments from the stack, and returns the cost

computed from those arguments according to the formula in the definition. For example, 100 .75 35 COST returns the price of a trip of 100 miles, at \$.75/gallon, in a car that gets 35 miles per gallon. COST is not as convenient for keyboard use as the Solver version, since it doesn't prompt you for input, and you have to know the correct order for the arguments. However, it can be used in programs and algebraic objects, and will accept symbolic arguments as well.

Chapter 8 contains a more detailed look at user-defined functions.

6.3 Symbolic Math

As we will show in section 9.1, algebraic objects are procedures that are internally the same as programs. So just creating any algebraic object is equivalent to writing a program. Its "inputs" are the values stored in the variables named within the algebraic object; its "output" is the symbolic or numeric result that is returned to the stack. The beauty of an algebraic object as a program is that you can treat it as a symbolic quantity, to which you can apply additional mathematical operations, obtaining new algebraic objects--programs--automatically.

The best time to use algebraic objects as programs is when you have already defined a set of user variables, and wish to make calculations using their values. You can, of course, use the values directly by evaluating the variables as you go and using RPN commands and functions to combine the values. But if a calculation is defined in algebraic terms, you'll do better to enter the appropriate formula as an algebraic object, so that you can verify its definition before substituting specific values.

For example, to add the values of variables A and B, you can press **[A] [B] [+]**. Or you can type 'A+B' **[EVAL]**. The advantage of the latter is that you can see the entire calculation symbolically before making numerical substitutions. This advantage becomes more important as the complexity of a calculation increases. You are also relieved of the necessity for translating the calculation into RPN logic.

The HP-28 approach to symbolic mathematics is described in Chapter 9.

6.4 Programs

If you can't use any of the simplified problem solving methods outlined so far, your final option is to write a program. There is a wide range of problems that don't fit the requirements for using the previous methods, including many that are mathematically very simple. For example, all of the previous methods have the common limitation of being able to return only one result at a time. If you want to automate a process as trivial as returning the square and the cube of an argument, you must write a program.

Here are three HP-28 programs that make those calculations:

```
<<  DUP  SQ  SWAP  3  ^  >>
```

```
<<  →  x  <<  x  SQ  x  3  ^  >>>>
```

```
<<  →  x  <<  'x^2'  EVAL  'x^3'  EVAL  >>>>
```

The last two versions illustrate that you don't have to give up the advantages of the alternate problem solving methods when you create program objects; you just incorporate them into your programs. Even the Solver's root-finding capabilities can be programmed, via the ROOT command.

The HP-28 is unusual among calculators in that it has no "program mode." In other calculators, you create a program by activating a mode where the keystrokes you press are recorded sequentially as program steps or lines. A consecutive sequence of such steps constitutes a program. To execute the program, you must leave program mode and invoke the program by means of a command like RUN or XEQ (execute).

In the HP-28, programming differs from interactive calculating only in that you don't execute sequences of objects individually, but instead combine them into procedure objects--programs or algebraics--for later execution. You treat the procedure objects the same as any other objects: you enter and identify them by characteristic delimiters (<< >> or ' '), and you can edit, visit, store, recall, evaluate, and purge them, or just move the objects around on the stack using standard commands.

Many BASIC language computers share with the HP-28 the property of lacking a special program mode. By placing a line number at the beginning of a command line, you tell the computer to include the program line in the current program. However, that style of program entry is very context-dependent: you must be sure that the line number you assign is appropriate. It must be in the proper sequence relative to other lines, and you must have somehow established that you are adding the line to the right program. Some computers solve that problem by only holding one program in memory at a time; others permit multiple programs but you must use various means to select a particular program for editing.

Other calculator programming also uses more "program-only" concepts, like GTO (Go To), labels, line numbers, RTN (return), and commands that behave differently when used in a program than when they are executed from the keyboard. An example of the latter is the HP-41 command FS? (flag set?). From the keyboard, this command returns a temporary display of YES or NO; when executed in program, FS? acts as a

“skip-if-false” operation, where the next program line is executed if the flag is set, and skipped if it is clear.

These concepts are part of what makes programming a calculator a mysterious art for many people. When you are solving a problem mentally, or with pencil and paper, you don't consider line numbers, GTO's, program modes, etc. Instead, you think in terms of a series of operations that you apply to data or symbols, which produces results that may in turn be the input for additional operations. This translates nicely to key-per-function interactive use of an RPN calculator; the operations become keystrokes, and the data is kept in front of you on the stack. “Keystroke programming” on calculators originated as a process of preserving a series of keystrokes as a program. Unfortunately, as calculators became more powerful, their programming languages required you more and more to rethink a problem in order to cast it as a program.

The HP-28 is designed to minimize or eliminate the differences between interactive keystroke operations and programming. It does this in several ways:

- The command line is a program that is executed immediately; a program is a command line for which execution is deferred (see section 3.11.3).
- Anything you can do in program you can do in the command line, including halting, single stepping, using local variables, branches, loops, etc.
- Commands work the same way in programs as they do from the keyboard.
- Programs contain no constructs that are artificial from the standpoint of the problem being solved--no line numbers, no labels, no GTO's. The only things that appear in a program are objects and commands relevant to the calculation being performed, plus certain program structures (conditionals, loops, etc.), that are local to a particular program.

The absence of GTO's and the corresponding labels and line numbers is a manifestation of the HP-28's insistence on *structured* programming. Every program is a self-contained module, with a single “entry” and a single “exit”. A program can, of course, “call” (execute by name) other programs, but only as subroutines that always return to the same point in the same program that called them. These rules promote a programming style whereby you break down a large programming task into smaller programs which are easily written and understood. As you write each “building block” program, you can test it independently before it is included in any larger program.

Methods and principles of program object design and application are covered in Chapters 10 and 11. Chapters 12 and 13 contain many additional program examples.

6.5 Summary

Table 6.1 summarizes HP-28 problem solving choices, and can act as a guide to selecting an appropriate method for a given problem.

Table 6.1. HP-28 Problem Solving Methods

<i>Method</i>	<i>Type of Problem</i>	<i>Advantages</i>
Solver	• Numerical evaluation of an algebraic expression for many values of its variables. • Symbolic substitution for variables. • Numerical solution of an algebraic expression, especially in combination with DRAW. • "What if" problems where the independent/dependent roles of variables are interchanged.	• Automatic input prompting and labeling; automatic numerical equation solving. • Lets you interchange known and unknown variables.
User-defined Functions	• Evaluation of algebraic functions, with arguments taken from the stack. • Creation of new symbolic functions.	• Can be used in RPN or algebraic calculations. • Does not require "permanent" user variables.
Symbolic Math	• Algebraic calculations using existing user variables. • Symbolic manipulations.	• Symbolic results can be used as new programs. • Calculations can be verified before they are performed.
Programs	All problems, especially those for which the other methods are insufficient: • Multiple results. • Non-mathematical problems. • Special prompting or labeling. • Iteration. • Complicated decisions and branching.	All calculator resources are available, including the algebraic evaluation features of the other programming methods.

6.6 Memory Limitations of the HP-28C

It is important to note that the modest amount of user memory (RAM) included in the HP-28C places significant limitations on the size and number of objects, especially programs, that you can store in memory. Although 2 Kbytes (1 Kbyte = 1024 bytes) of RAM is roughly the same as the main memory in an HP-41CV, and four times as large as that in the HP-15C, the HP-28C uses memory more rapidly and for more purposes than either of those calculators. The basic program "unit" in an HP-41 is one byte;

most program steps require one byte, some two or three, and a few containing alpha characters use four or more bytes. In the HP-28, the basic unit is $2\frac{1}{2}$ bytes (five nibbles); all built-in commands in a procedure take this amount of memory each. Thus in rough terms, the HP-28 consumes memory $2\frac{1}{2}$ times as fast as the HP-41 (see section 11.6 for more detail). Furthermore, the HP-41 has no built-in commands that need any of its program/register memory just to execute, whereas the HP-28 has numerous commands like ∂ or EXPAN that may use large amounts of memory and so can fail due to lack of memory. On the other hand, the built-in capabilities of the HP-28C far exceed those of the HP-41, so in many cases a complicated HP-41 program reduces to a few keystrokes or to a short program in the HP-28C.

In any case, if you try to do any extensive program development on the HP-28C, you are bound to be frustrated by running out of memory frequently. The HP-28C has enough memory to work most simple problems and to accumulate a few programs, but its power encourages you to try more and more things, especially those that you can't do on another calculator regardless of how much memory it has. The appropriate programs for the HP-28C are short utilities that you use to enhance its interactive keyboard operation, not major application programs. You can generally afford to devote about 1200 bytes of memory to USER menu objects, leaving about 500 bytes for working space. If you reduce the latter much below 500 bytes, you will frequently run out of memory when editing or executing programs or performing complicated algebra.

6.7 HP-28S Memory

The HP-28S takes advantage of newer electronic memory technology than that of the HP-28C, to provide 32 Kbytes of RAM instead of the 2 Kbytes of the HP-28C. This amount of memory is well matched to the capabilities of the calculator; you can store over 25 Kbytes of USER memory objects and still have several thousand bytes for working space and temporary objects. By choosing a "permanent" object storage limit of about 25 Kbytes, you insure that there is generally enough free memory left to always leave LAST, COMMAND, and UNDO enabled, and to perform even quite complicated calculations and stack manipulations without running out of memory.

7. The Solver

The “Solver,” the interactive equation-solving system that has become an important feature on several advanced HP calculators, was first introduced on the HP-18C. The HP-28C was the second calculator to have this capability. The HP-28 Solver uses internal calculation processes very similar to those of the HP-18C Solver, although the two user interfaces are somewhat different. Historically, the Solver is derived from two sources: the famous “time-value-of-money” (TVM) key system that originated with the HP-80 calculator and has been a fixture on all subsequent HP financial calculators; and the SOLVE function that was originated by the HP-34C.

The original SOLVE function, as it still exists in the HP-15C and the HP-41 Advantage Pac (and as the HP-71B BASIC function FNROOT), is a system designed for the numerical solution of problems that can be expressed as a single equation containing one unknown variable. 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. SOLVE starts with two guesses of the unknown value that you supply to define a region of values of the unknown variable in which to start searching for a root. It then adjusts the variable’s value iteratively until it finds a value that is a root. This is a very powerful system, since it relieves you of the need to rearrange or solve an equation by hand before entering it into the calculator. (On an HP-15C or an HP-41, you do have to translate the mathematical expression of the problem into RPN form.)

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

$$PV + (1 + i^p)PMT \left[\frac{1 - (1 + i)^{-N}}{i} \right] = -FV(1 + i)^{-N}$$

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.

An important contribution of the TVM system is its keyboard interface, that lets you use the same key (for N , I , PV , PMT , or FV) to enter a value for a variable or to solve for it if it is unknown. Pressing any one of the TVM keys following data entry causes a number to be stored from the X-register into the variable corresponding to the key. But pressing two consecutive TVM keys activates the solving algorithm, which is applied to the TVM equation for the variable designated by the second key. This simple interface makes “what if?” analysis of loan and savings amortization very efficient, and has become a standard in the financial industry.

The HP-18C made an important contribution to the calculator art with its synthesis of the TVM interface and a general-purpose solve function. In the HP-18C, you can specify an arbitrary equation in any number of variables. The calculator then creates a set of labeled menu keys corresponding to the equation variables, with the same sort of store/solve logic as the TVM system. Again, you can store values by entering each and pressing a menu key; when you press two consecutive menu keys, the calculator automatically invokes the root-finder and finds a value of the variable last pressed that satisfies the equation. Since the HP-18C allows direct entry of equations in algebraic form, you are spared the work of translating the equation into RPN. The HP-18C also has an additional innovation: whenever possible, it solves the equation symbolically rather than using the numerical iterative root-finder. The result returned to the user is the numerical evaluation of the symbolic solution.

7.1 Basic Solver Operation

The general procedure for using the HP-28 Solver is as follows:

1. Enter the expression or equation representing a problem, and store it in a variable named EQ. You can use 'EQ' STO for this purpose, but the command STEQ (in the SOLV menu) does this in one step. Typically, you use an algebraic object to represent the problem, but you can also use an equivalent RPN program. The use of a specially named global variable to hold the Solver equation saves you from having to keep the equation on the stack while you perform the Solver operations. We refer to the object stored in EQ as “the current equation,” even if it is not actually an algebraic equation object.
2. Press **SOLV** **SOLVR** to activate the *Solver menu*. This menu contains a menu key for each independent variable in the equation.
3. 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. If you want to make any variable a “constant,” enclose the constant value in program delimiters << >> before you store it. Then the variable name will disappear from the menu, and you can't accidentally change the

value.

4. Store a guess for the value of the unknown variable, again by entering a value and pressing the variable's menu key.
5. Solve for the unknown: press the shift key, 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, and stores that value in the unknown variable. You will also see a *qualifying message* in the display that can help you interpret the result.
6. Verify the result, using the evaluation key $\overline{\overline{\text{EXPR}=\overline{\overline{\quad}}}}$ that appears in the menu for cases where EQ contains an expression or a program; or the menu keys $\overline{\overline{\text{LEFT}=\overline{\overline{\quad}}}}$ and $\overline{\overline{\text{RT}=\overline{\overline{\quad}}}}$ that appear for an equation.
7. Repeat steps 2 through 6 with new values for the variables, and perhaps a new choice of the unknown variable.

A nice property of the HP-28 Solver is that you can use programs, expressions, and equations almost interchangeably for solving purposes. In effect, the Solver always solves $f(x) = 0$. In the case of an HP-28 algebraic expression object, $f(x)$ is the expression represented by the object, where x is the name of the unknown variable. For a program, $f(x)$ is the expression that is equivalent to the (RPN) program. For an equation object representing $g(x) = h(x)$, the Solver solves $f(x) = g(x) - h(x) = 0$.

As an example of the basic use of the Solver, consider this problem from the HP-41 Advantage Pac manual (it is also in the HP-15C manual):

■ *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.* Start by entering the equation:

$$'H=5000*(1-EXP(-T/20))-200*T' \quad \boxed{\text{SOLV}} \overline{\overline{\text{STEQ}}}.$$

Now activate the Solver menu: press $\overline{\overline{\text{SOLVR}}}$. The display looks like this on an HP-28S:



(On the HP-28C, the menu keys are white characters on black. The appearance of the Solver menu keys on the HP-28S are changed from the HP-28C to match the menu keys in a custom store menu--see section 11.9.) Here you see menu keys for the two variables in the problem, T and H, and the $\overline{\text{LEFT=}}$ and $\overline{\text{RT=}}$ keys for evaluation of the two sides of the equation. For this example, you want H to have the value zero, so press $\boxed{0} \overline{\text{H}}$.

Now you need to enter an initial guess for the solution, to give the Solver some idea of where to start looking for the root. 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

1. 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.
2. If t is large, you can ignore the exponential term, so that the equation approximates to

$$5000 - 200t = 0,$$

which has the solution $t = 25$.

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

Finally, press $\blacksquare \overline{\text{T}}$ to start the root-finder. After a few seconds, the Solver 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 $\overline{\text{LEFT=}}$ and $\overline{\text{RT=}}$, both of which return 0, demonstrating that both sides of the equation have the same value.

7.2 Interpreting Results

Not all Solver problems work out as nicely as the ridget example. The equation may have multiple solutions, no solution, discontinuities, infinities, etc. It is important when you use the Solver 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 the EXPR=, LEFT=, and RT= menu keys, and of the qualifying message.

The message Zero returned in the ridget example is the most welcome of the possible qualifying messages. Zero means that the HP-28 succeeded in finding a value of the unknown variable for which the EQ expression or program exactly evaluates to zero, or, if EQ contains an equation, both sides have exactly the same value. “Exactly” in this sense means exact to 12-digit precision, the numerical accuracy of the HP-28. When you see the Zero message, you know that pressing EXPR= or LEFT= RT= $\boxed{-}$ will return zero.

When the Solver returns the qualifying message Sign Reversal, it means that it was 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*. In principle, 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, the Solver 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 the evaluation keys. In the case of 'IFTE(X>0,1,-1)', LEFT= returns 1, while RT= returns 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 LEFT= returns 1.99999999999; RT= returns 2.00000000000. The near-equality of these two values indicates that you have a good solution.

Other possible Solver messages are as follows:

Bad Guess(es) The root-finder can't get started, 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 ' $\sqrt{X}=2$ ', and start with a guess of $X=-5$ (for which $\sqrt{X}$ is imaginary).

- Constant?** At every value of the unknown tried by the root-finder, the equation returns the same value. 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 7.7.

7.3 Independent, Dependent, and Unknown Solver Variables

Imagine that you are using the Solver with an equation ' $A+B=C+D$ '. If any of the four variables takes only numeric values, it is 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 are contained in the current equation, only those that are independent variables appear in the Solver menu. To see this, key in ' $A+B=C+D$ ', and press **SOLV** **STEQ** **SOLVR**. You will 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 enter ' $B+C+F$ ' 'A' **STO**. The A menu key disappears, and is replaced by an F key after B and C. 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 ' $B+C+F$ '.

If you are working with a current equation with lots of variables, you may notice some delays in calculator operation while the Solver menu is active. The HP-28 has to rebuild the menu at every ENTER, because any of the commands you execute may change one or more of the solver variables or the current equation itself. If the current equation is complicated, this process can take a noticeable amount of time.

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 (Chapter 13). 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 the Solver, you should use INDEP (section 13.1.3) to select the Solver 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.

7.4 First Guesses

You may find it disappointing that a system as sophisticated as the Solver requires you to supply a "first guess" of the answer in order to get the root-finder off to a good start and insure that it returns the "right" answer. After all, the HP TVM system, which is a specialized solver, doesn't require a first guess, yet it always returns the right result.

The problem is that 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 arises from the fact that 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} in steps of 10^{-499}). This is obviously not practical. On the other hand, if the Solver 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 should the Solver find?

These considerations lead to the requirement that you must specify an initial guess for the Solver. Your guess serves to give the Solver an idea of where to start looking, and to guide the Solver to a particular root in cases of multiple roots.

7.5 How Many Guesses?

The HP-28 root-finder works 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 gives 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), or failing that, two neighboring points where the equation has opposite signs (Sign Reversal). You will obtain the best results from the Solver by specifying a good initial search region, by means of the initial guess.

The HP-28 gives you the option of making single, double, or triple initial guesses for the Solver. 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 guess, the Solver 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. The Solver takes your single guess and makes its own second guess by 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's never guaranteed that the Solver 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 the Solver 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 the Solver by pressing **ON**, it returns a list of three numbers. This list is intended for use as a new triple guess, in case 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 the Solver 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 define the search region as in the double guess case.

7.5.1 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 (see the plot in section 7.6.1) has roots at $x = -2$, $x = +2$, and $x = 0$, so you know what to expect from the Solver. To get an idea of how quickly the Solver 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 using this program as the current equation:

```
<< '(X-2)*(X+2)*X' 1 'N' STO+ >>
```

Each time the program is executed, the value of N is incremented by 1, so the difference in N before and after solving is the number of root-finder iterations. N will appear in the Solver menu, but that 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.

{ -2 9 }	-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."

7.5.2 Solver Guess Summary

Table 7.1 summarizes the meaning and application of the single, double, and triple guess options for the Solver.

Table 7.1. Solver 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.

7.6 Obtaining Guesses

So far we have discussed the need for guesses and how many guesses to supply the Solver. This still leaves the problem of *how to find* good values to use as guesses. 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.

Of these three methods, we recommend the last, since it is the most reliable.

If you don't explicitly store a guess in the unknown variable before solving, the Solver 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 the Solver 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 Solver will return that root, as we discussed in section 7.5.

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 7.1. For 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 ' $\text{COS}(X)=X$ ', the Solver 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.

In the preferred method of obtaining guesses, you *plot* the current equation, and *see* where the roots are in the picture. You can use a plot to obtain single, double, or triple guesses and ensure that the Solver finds the root you want. This application of plotting is an important reason for the existence of the plotting feature in the HP-28.

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-28 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. So the roots of equations appear as the intersections of the two curves.

7.6.1 Single Guesses

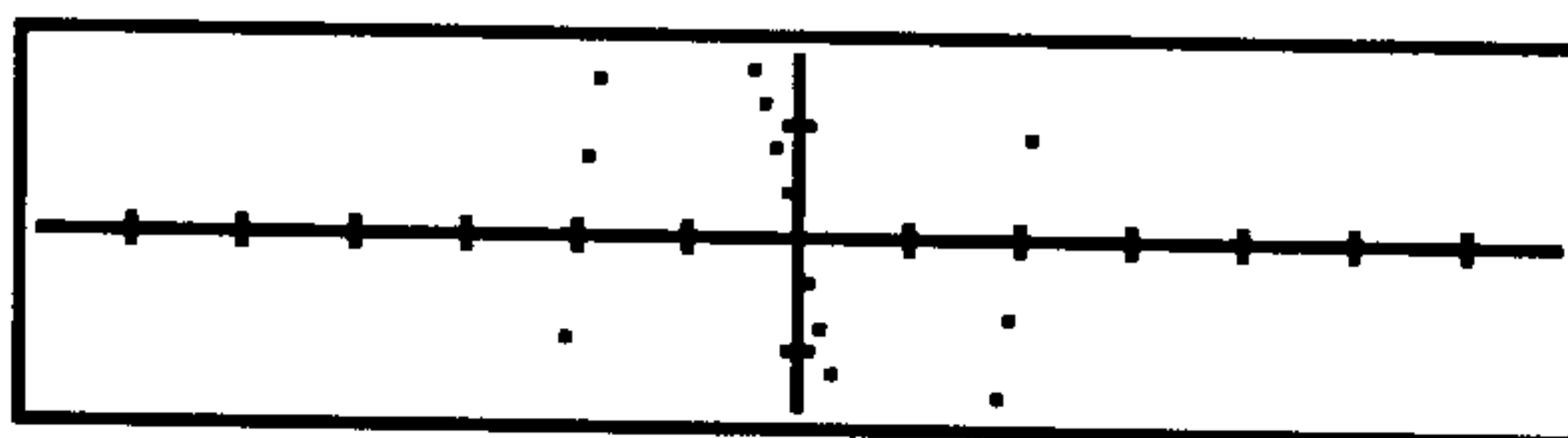
To use a plot to obtain first guesses, all you have to do is digitize points on the plot of the current equation which are near the curve intersection corresponding to the particular root that you want. For this purpose, you use the DRAW cursor and the **INS** key.

To obtain a *single* guess, move the cursor to the desired intersection, and press **INS**. Then press **ON** to return to the stack display. You will see the coordinates of the digitized point in level 1 as a complex number object. You can use this number directly as a first guess, since the Solver is designed to use only the first number (real part) in a complex number guess.

To complete the solving process, press **SOLV** **SOLVR** to activate the Solver menu (remember that DRAW and the Solver use the same current equation in EQ). Store the complex number from level 1 into the unknown variable by pressing the corresponding menu key. Then press the shifted menu key to compute the 12-digit root.

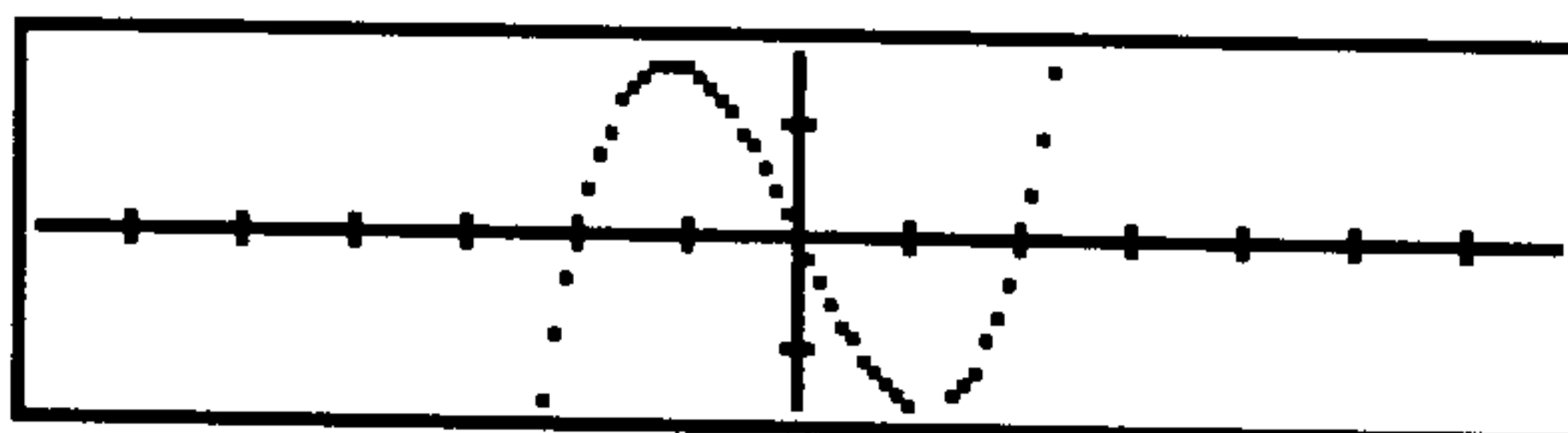
As an example, consider again the expression $x(x-2)(x+2)$ from section 7.5.1. To help you understand its properties, plot the expression.

1. Enter ' $X*(X-2)*(X+2)$ ' **PLOT** **STEQ**.
2. Reset the default plot parameters: **NEXT** **PPAR** **PURGE**.
3. Make the plot: **PREV** **DRAW**.



4. To get a better picture, try doubling the vertical range (see section 13.4.2.3):

ON **NEXT** 2 ***H** **PREV** **DRAW**.



From the plot you can see that the expression has roots at $x \approx 2$, $x \approx -2$, and $x \approx 0$, plus a local maximum at $x \approx -1$ and a local minimum at $x \approx 1$. To get a more precise value for the root near 2 (of course, you can see from the equation that the root is exactly $x = 2$, but usually roots aren't so obvious), move the cursor over to the rightmost intersection and press **INS**, then **ON**. You should see the complex number (2,0) in level 1. Now solve for x:

SOLV **SOLVR** **X** **X** 2.

The Zero message indicates that the expression evaluates to zero at $X = 2$.

7.6.2 Double Guesses

The procedure for obtaining a double guess is a simple extension of that for digitizing a single guess. That is, you digitize two points, one just on each side of the intersection representing the root. Then when you exit the plot, combine the two resulting complex numbers into a list by pressing 2 **LIST** **→LIST**. Next, as in the case of the single first guess, activate the Solver menu, store the list as the guess, and solve for the root.

Returning to the previous example,

1. Press **PLOT** **DRAW** to restore the plot.
2. Use **◀** to move the cursor to a point two dots to the left of the intersection at $X=2$, and press **INS**.
3. Use **▶** to move the cursor two dots to the right of the intersection, and press **INS** again.
4. Press **ON** to clear the plot. You should see the numbers (1.8,0) and (2.2,0) on the stack (or values close to these).
5. Enter 2 **LIST** **→LIST** to combine the numbers into the list {(1.8,0) (2.2,0)}.
6. Solve:

SOLV **SOLVR** **X** **X** 2.

7.6.3 Triple Guesses

To obtain a triple guess from a plot:

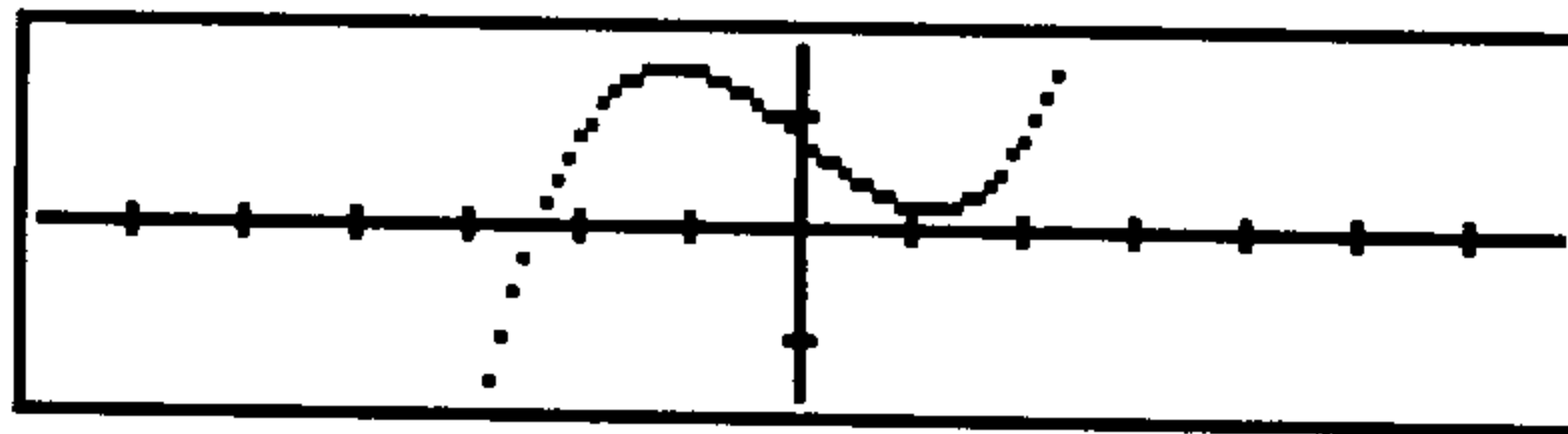
1. Move the cursor to the root intersection, and press **INS**.
2. Move the cursor just to the left of the root, and press **INS**.
3. Move the cursor just to the right of the root, and press **INS**.
4. Press **ON** to exit the plot, then 3 **LIST** **→LIST**.
5. Activate the Solver menu, store the guess, and solve for the unknown.

7.7 Finding Extrema

Although the HP-28 Solver is designed for finding roots of expressions, you can also use it to find *extrema*--local maxima and minima. In the process of searching for a point where an expression has the value zero, the Solver continually works to minimize the absolute value of the expression. To find a root, it keeps going until that absolute value reaches zero. But it can get "stuck" in a region where there is a minimum of the absolute value that is greater than zero. No matter whether the search proceeds to the left or to the right of the minimum, the absolute value of the expression increases, so the Solver quits and returns the message *Extremum*.

Consider, for example, the expression $'(X-2)*(X+2)*X+4'$. To understand its properties, make a plot:

```
'PPAR' PURGE 5 *H '(X-2)*(X+2)*X+4' STEQ DRAW
```



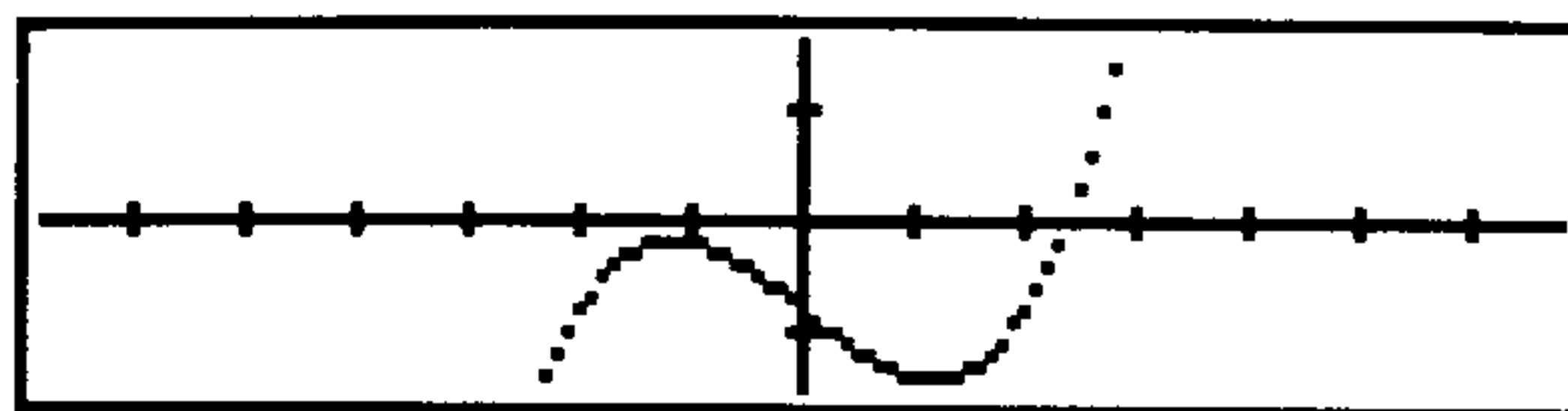
If you give an initial guess of 1 for X , the Solver returns the result 1.1547, which is the X -coordinate of the local minimum. Notice, however, that if the initial guess is -1 , the same result is returned; for an initial guess of -2 , the root near -2.38 is found. The Solver won't find the local *maximum* near $X = -1$, because it is not a minimum of absolute value.

You can find the coordinates of the local maximum in this case by the simple expedient of subtracting a number from the expression large enough to move the plotted curve down so that the maximum point is below the horizontal axis. To determine how much to subtract, you can digitize the coordinates of a point just above the maximum, then subtract the resulting vertical coordinate from the plotted expression.

To apply this technique to the current example, after executing **DRAW** do the following:

1. Press **■** **△** to move the cursor to the top of the screen, then press **INS** to record the coordinates.
2. Press **ON** to clear the plot. You should see the complex number (0,8). Execute **IM** to extract the vertical coordinate (imaginary part) from the complex number.

3. Subtract the coordinate value from the current equation: RCEQ  SWAP  STEQ .
4. Execute DRAW again.



Notice that the maximum is now below the axis.

- 5. Solve:**

SOLV **SOLVR** -1 **X** **X**  $-1.1547.$

The result is the X-coordinate of the maximum. To compute the vertical coordinate, execute the original expression using this value of X.

To summarize the process of finding extrema:

1. If the extremum is a local maximum, subtract an amount from the expression sufficient to move the maximum below the horizontal axis. If the extremum is a local minimum, add an amount to the expression sufficient to move the minimum above the horizontal axis.
2. Supply a first guess near the extremum, then solve for the unknown variable.
3. To obtain the value of the expression at the extremum, restore the original expression to EQ and press ≡EXPR≡.

7.7.1 Using the Derivative

There is another, more elegant way to find extrema in addition to the curve displacement method described above. Extrema are points at which the derivative of a function is zero--that is, the extrema of $f(x)$ are the roots of $df/dx=0$. Therefore, to find the extrema of an expression, you can use the ∂ function to differentiate the expression, store the result as the current equation, and use DRAW and the Solver to find the roots. 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).

To try this process, return again to the example expression $x(x-2)(x+2)$. Start by

plotting the function and its derivative:

1. Purge X if it exists: 'X' **PURGE**.
2. Enter the expression and differentiate:

'X*(X-2)*(X+2)' **ENTER** **ENTER** 'X' **d/dx**

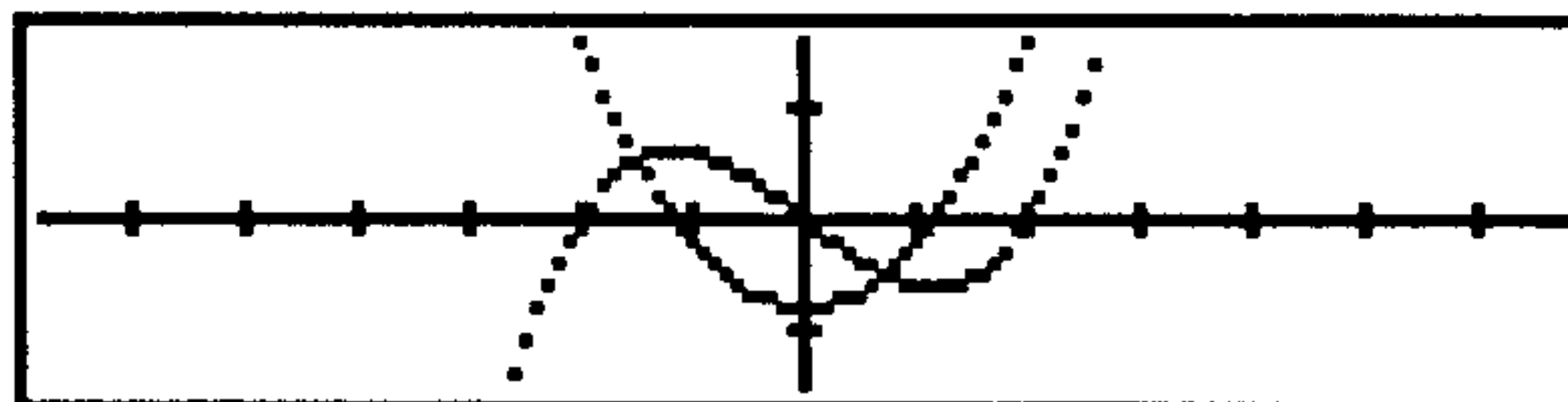
☞ 'X*(X-2)*(X+2)' '(X-2+X)*(X+2)+X*(X-2)'

3. Create the "equation" $df/dx = f(x)$:

ENTER **STACK** **ROT** **=** **ENTER**

☞ '(X-2+X)*(X+2)+X*(X-2)=X*(X-2)*(X+2)'

4. Make this the current equation: **PLOT** **STEQ**.
5. Reset the plot parameters: **NEXT** **'** **PPAR** **PURGE** 5 ***H**.
6. Make the plot: **PREV** **DRAW**.



The parabola is the graph of the derivative. Note that its roots are at the same X values as the extrema of the cubic curve. To obtain precise values for the roots of the derivative:

1. Clear the plot: **ON**.
2. Store the derivative expression in EQ: **SOLV** **STEQ** **SOLVR**.
3. Solve for the negative root (maximum of the original expression):

-1 **X** **X** ☞ -1.1547

4. Solve for the positive root (minimum):

1 **X** **X** ☞ 1.1547.

7.8 Evaluating Rather than Solving

Although the Solver is designed for the application of a numerical root-finder to expressions and equations, the Solver menu is convenient also for problems where no “solving” is necessary. For example, for an equation of the form ‘ $A=B+C+D$ ’, you might want simply to obtain values of A for various inputs B , C , and D . If you store this equation in EQ, you can use the Solver menu keys $\boxed{\boxed{B}}$, $\boxed{\boxed{C}}$, and $\boxed{\boxed{D}}$ as usual to store the inputs. However, using $\boxed{\boxed{A}}$ to solve for A is unnecessarily slow.

The Solver menu always includes either the menu key $\boxed{\boxed{\text{EXPR=}}}$ (when EQ contains an expression or program) or keys $\boxed{\boxed{\text{LEFT=}}}$ and $\boxed{\boxed{\text{RT=}}}$ (when EQ contains an equation). In addition to their uses for verifying solutions, you can use these keys to evaluate the current equation without solving. In the current example, after storing values of B , C , and D , you press $\boxed{\boxed{\text{RT=}}}$ to return the computed value of A . (You don’t press $\boxed{\boxed{\text{LEFT=}}}$ even though A is on the left side of the equation, because that returns the current value of A , not the computed value of $B+C+D$ as returned by $\boxed{\boxed{\text{RT=}}}$. Note, however, that $\boxed{\boxed{\text{LEFT=}}}$, $\boxed{\boxed{\text{RT=}}}$, and $\boxed{\boxed{\text{EXPR=}}}$ do not change any of the variables’ stored values; these operations return results only to the stack.

Each of these three operations *evaluates* an expression, in the same manner as the EVAL command. $\boxed{\boxed{\text{EXPR=}}}$ evaluates the expression or program stored in EQ; $\boxed{\boxed{\text{LEFT=}}}$ and $\boxed{\boxed{\text{RT=}}}$ evaluate the left and right side, respectively, of an equation stored in EQ. If the HP-28 is currently in symbolic evaluation mode (flag 36 is set--see section 9.2.2), the variables need not contain real or complex numbers, or exist at all, as they must when you use the root-finder. In the current example, B , C , and D can be matrices, in which case $\boxed{\boxed{\text{RT=}}}$ returns the matrix sum.

You can even use the Solver menu as the “front-end” of a program, to provide a convenient menu for entering initial data for the program (this is particularly useful on the HP-28C, which doesn’t have the custom menu feature of the HP-28S). All you do is store your program in EQ to make it the current equation. Then $\boxed{\boxed{\text{SOLVR}}}$ creates a menu with keys for all of the variables in the program. Use these keys to store values for the variables, in any order, then press $\boxed{\boxed{\text{EXPR=}}}$ to run the program.

In many cases, you may want only certain program variables to appear in the menu. You can “hide” any part of a program from the Solver menu system by putting it inside a list. For example, here’s a program that adds A , B , C , and D :

```
<<  A  B  +  C  +  D  +  >>
```

If this program is the current equation, A , B , C , and D will all show up in the Solver menu. But if you write it like this:


```
<< A B + { << C + D + >> } LIST→ DROP EVAL >>
```

the result of executing the program is the same as before, but only A and B appear in the Solver menu.

7.8.1 Using ISOL with the Solver

Since EXPR= is faster than numerical solving, it is often advantageous to attempt to solve the current equation symbolically as a preliminary to using the Solver. This is particularly true in cases where you only intend to solve for one variable in the current equation. By using ISOL once to solve the equation, you can reduce the problem from a root-finding task to one of straightforward evaluation using EXPR=.

Consider, for example, the simple travel cost problem from section 6.1:

$$'COST = DIST * PPG / MPG'$$

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 should first solve the equation for DIST symbolically:

```
{ COST DIST PPG MPG } PURGE
```

makes the variables formal, for a fully symbolic solution. Then

```
'COST = DIST * PPG / MPG' 'DIST' ISOL → 'COST * MPG / PPG'
```

If you store the result expression 'COST * MPG / PPG' as the current equation (use STEQ), the resulting Solver menu will contain menu keys for COST, PPG, and MPG, plus EXPR=. EXPR= now represents the expression for distance, so for every set of values you enter for COST, PPG, and MPG, just press EXPR= to compute the corresponding distance.

Note that the result of EXPR= is returned only to the stack. The variable that you solved for with ISOL (DIST in the example) no longer appears in the current equation, nor in the Solver menu, so the Solver computations will not affect its value.

7.8.1.1 Limitations of ISOL

It is useful at this point to review the limitations of ISOL that affect its use as a preliminary to the Solver (ISOL is described in more detail in section 9.6.1):

- ISOL isolates only the first (reading left to right) occurrence of a variable in an algebraic object. If the variable occurs more than once, ISOL does not fail, but returns a result that does not represent a *solution* for the variable. For example, ' $X+Y+X=Z$ ' 'X' ISOL returns ' $Z-X-Y$ '. If your unknown variable appears more than once in the current equation, you will have to use the numerical Solver, unless you can rearrange the equation with COLCT, FORM, or other algebra tools (section 9.8) to combine all occurrences of the unknown into one.
- ISOL works only at the “top level” of an expression or equation; it does not evaluate any of the names. This means 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 ' $Z-Y$ '. But this result, when evaluated, returns ' $X-Y+Y$ ', showing that the “solution” is meaningless. To guard against this type of problem, it is wise to use SHOW with an equation prior to using ISOL. SHOW makes all references to a specified name explicit. In the example, ' $X+Y=Z$ ' 'X' SHOW returns ' $X+Y=X+Y$ ', making it obvious that the equation is meaningless. 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.

7.9 Secondary Results

The Solver is designed to work with a single equation. However, because the Solver menu variables, including the most recently solved value of an unknown variable, are available in the USER menu, it is easy to obtain secondary or derived results after a Solver solution. The general idea is as follows:

1. Create the current equation.
 2. Create additional named algebraic objects or programs that compute results from any or all of the Solver variables.
 3. Use the Solver to enter values for the known variables, and to solve for the unknown.
 4. Switch to the USER menu, and press the appropriate menu keys to compute the secondary results.
- *Example.* Compute the volume and the surface area of a right circular cylinder of radius R and height H. Evaluate for $R=3$ and $H=5$.

■ *Solution:***Keystrokes:****Results:**

'2* π *R*H' 'AREA'
[STO]

Formula for area
(secondary procedure).

'V= π *R²*H' [SOLV]
[STEQ] [SOLVR]

Formula for volume.

3 [R] 5 [H] [V]

1:

141.37 Volume

[USER] [AREA] [-NUM]

2:

141.37

1:

94.25 Surface Area

7.10 Differences between the HP-28 and other HP Solvers

Since the HP Solver was originally introduced on the HP-18C in 1986, it has appeared on several other calculators, including the HP-28C and HP-28S, and the HP-17B, HP-19B, and HP-27S. All of the calculators except the HP-28 have Solvers with a user interface modeled on that of the HP-18C. The HP-28 Solver differs from the HP-18C model in three important respects:

1. The HP-18C stores equations in a special equation file, and uses a keyboard-directed pointer to select the active equation. You can enter equations into the HP-18C only in a special Solver mode. The HP-28 stores equations in global variables in the USER memory along with all other named objects. The active equation for the Solver is the equation currently stored in a variable 'EQ'. Since HP-28 equations are just algebraic objects or programs, you can create them at any time, with any menu active.
2. The HP-18C attempts to find a symbolic solution before resorting to the numerical root-finder; the HP-28 does not.
3. In the HP-28, you select a variable for solving from the Solver menu by pressing the shift key followed by the menu key for that variable. The HP-18C solve is triggered by the second of two consecutive menu key presses.

The HP-18C implementation of the Solver may seem simpler than that of the HP-28. If you are used to one of the other calculators, or to the original TVM interface, you may be surprised by the necessity of pressing the shift key on the HP-28 in order to initiate a

solve. But the differences in the Solvers are generally consistent with the differences in overall design and use of the calculators.

For example, the HP-18C menu system has a definite hierarchy--every (main) menu is effectively a separate environment. In some cases, exiting a menu then returning to it even causes the loss of values stored in variables. The HP-18C doesn't have the equivalent of the HP-28 USER menu; instead, each menu has its own associated variables--lists in the SUM menu, appointments in the TIME menu, equations in the SOLVE menu, etc.

The HP-28, on the other hand, generally avoids special environments. You can change from any menu directly to any other. Changing menus never affects objects stored on the stack or in variables, or even calculations pending in the command line. USER memory is "global"--you can access any variable at any time, even by spelling out its name if you don't want to change to the USER menu. There are no special Solver-only equations. The Solver will work with any procedure object (programs must satisfy algebraic syntax). To select a particular procedure, you just store it (or its name if it is already stored in a variable) in a variable named 'EQ'. The idea is not to make the Solver more complicated, but to make all of the other resources of the calculator also available for analysis of the procedure. In particular, the plotting capabilities available with DRAW are an invaluable adjunct to the Solver, and can often make the difference between a successful and an unsuccessful root search.

The HP-28 Solver is also "deeper" than the HP-18C version. Each HP-18C equation is an isolated entity: the variables in the equation can represent only (real) numbers. In the HP-28, Solver equations are ordinary algebraic objects or programs, in which the variables can contain any object, including other algebraic expressions or equations. When the HP-28 Solver builds its variables menu, it tests the contents of each variable before adding its name to the menu. If a variable contains a procedure, the variable's name is not included in the menu; and the Solver looks for additional names in the named procedure. This process is the reason why the HP-28 Solver does not attempt a symbolic solve. Symbolic solving as performed by ISOL is hard enough without the additional complication of trying to resolve all variables referenced in an expression. Trying to do this (without even considering the problem of multiple solutions) every time the Solver attempts a solution could waste a lot of time. In effect, the decision of whether to solve symbolically is left to you; if a symbolic solution is appropriate, you can use ISOL (and SHOW) to rearrange an expression before making it the current equation. Then you can just use EXPR= to evaluate it, rather than using the numeric root-finder.

The necessity for using the shift key to initiate a root search on the HP-28 follows from the nature of the HP-28 stack, and the greater flexibility of the Solver menu compared

to that of the HP-18C. The key question is "how do you force the Solver to solve?" On the HP-18C, you must press two consecutive menu keys, which in many cases is the normal flow of events--you enter a value for the a known variable with its menu key, then press the menu key for the unknown variable. However, if you interrupt this flow by performing any other operation before pressing the final menu key, you will store the value in the calculator line (or the X-register, in the case of the 12C) into the unknown variable, rather than solving for that variable. If you press the key again, solving will start, but now you may have compromised the solving process because the value you stored into the unknown variable is used by the Solver as a starting point for the search. Unless that value is a good first guess, the Solver may be unable to find the correct solution. (Furthermore, on the HP-28, the stack may be empty, or level 1 may contain an object that is not a real number and which is completely unsuitable as a first guess.)

Another similar problem arises when you have values for several of the known variables in order on the stack. You should be able to store them into the variables by pressing the menu keys in order. You could not do this if pressing two consecutive menu keys started the root-finder.

The HP-18C Solver is perhaps a little easier to use for simple problems than that of the HP-28. The HP-28 design allows you to use a more flexible and general approach to solving equations, at the cost of some minor added complexity in the process.

8. User-Defined Functions

The archetype of a small HP-28 program is one that takes a few arguments from the stack, combines them according to some mathematical expression, and returns the computed result to the stack. For example, the distance between two points (x_1, y_1) and (x_2, y_2) is given by

$$[(x_2 - x_1)^2 + (y_2 - y_1)^2]^{1/2}.$$

This program takes the coordinates of two points from the stack, and returns the distance between the two points:

```
<< ROT - SQ 3 ROLL - SQ + √ >>.
```

The program assumes that x_1, y_1, x_2 and y_2 have been entered onto the stack, in that order (x_1 in level 4). It removes the four values, and returns the computed distance to level 1.

This program is short and efficient, because you (the programmer) did the work of translating the mathematics into the HP-28's RPN logic. But writing a program this way has two shortcomings:

1. When you develop the program, you have to keep track of the stack positions of the various arguments as they are needed by the successive program commands.
2. After the program is written, it is difficult to decipher. Notice that the program objects together bear little obvious resemblance to the original distance formula.

These problems become more severe as the number of arguments and the complexity of the calculation increase. Imagine trying to alter the example program so that it works with 3-dimensional points (x, y, z) . Because the stack positions of all of the arguments are changed, you have to rethink all of the stack manipulations, and almost rewrite the program entirely.

The difficulty of managing stack objects is substantially reduced if your program stores the objects in named global variables, then recalls the values by name as they are needed. However, there are disadvantages to using global variables for temporary storage in a program:

- You have to choose variable names that don't conflict with those of other programs.
- The program has to purge the variables at the end to avoid leaving unneeded variables in the USER menu.

The problem of program legibility is reduced if you represent the calculations by algebraic objects. Despite the virtues of RPN for interactive calculations, by and large people are more adept at reading calculations in a form approximating conventional mathematical notation than in RPN form.

HP-28 *user-defined functions* are designed to provide a simple means of writing programs without the problems listed above. User-defined functions are defined by algebraic expressions for easy development and modification. Furthermore, user-defined functions employ *local variables*, which exist only as long as the functions are executing. The local variables are used to provide names for stack arguments, and to minimize the need to manipulate lots of objects on the stack.

A user-defined function is a global variable that contains a program object which has a special structure, designed to compute a single algebraic expression defined in terms of local names. User-defined functions are called *functions* because they act like built-in functions: you can use them like RPN commands to compute from explicit stack arguments, or as prefix functions within algebraic objects, taking arguments from within parentheses.

You can program the example at the start of this section as a user-defined function named DIST, as follows:

```
<< → x1 y1 x2 y2 '√(SQ(x2-x1)+SQ(y2-y1))' >> 'DIST' STO.
```

The first part of the program (→ x1 y1 x2 y2) takes four numbers from the stack and names them x1, y1, x2, and y2, by storing them in local variables with those names. The algebraic object that makes up the rest of the program computes the distance from the four stored values. It's very easy to modify this program for three dimensions. Just edit the program to add two more local names, and add a term for $(z_2 - z_1)^2$ to the algebraic expression:

```
<< → x1 y1 z1 x2 y2 z2 '√(SQ(x2-x1)+SQ(y2-y1)+SQ(z2-z1))' >>
```

8.1 User-Defined Function Structure

In general, to create a user-defined function you first enter a program object with the following structure:

```
<< → x1 x2 ⋯ xn 'f(x1, x2, ⋯ , xn)' >>.
```

Then you store the program in a global variable. The variable's name subsequently acts as a user-defined function. Let's look at the separate pieces of the general form, using

the example DIST for illustration.

1. The first object in the program is the symbol $\rightarrow$. In words, this symbol can be translated as "take arguments from the stack, and assign them the following names..." The $\rightarrow$ is always followed by a sequence of local names. The end of the sequence of names is indicated by the start of an algebraic object that must follow the names. $\rightarrow$ takes one object from the stack for each name in the sequence. In DIST, there are four names, x_1 , y_1 , x_2 , and y_2 , so DIST requires four input arguments. The objects that $\rightarrow$ takes from the stack are matched up with the names in the order in which they are entered. The first object entered onto the stack, which was in the highest numbered stack level (level 4 in DIST), is matched with the first name (x_1) in the sequence.
2. The names x_1 x_2 $\cdots$ x_n in the series are *local* names. The combination of a local name and an object taken from the stack is called a *local* variable. Local names and variables are described in detail in section 10.7; for now, the important thing to know is that the variables exist only as long as the procedure that follows the local name list is executing. Local variables are stored in special areas of memory separate from the global variable memory; they don't appear in the USER menu.
3. The final part of the user-defined function structure is the algebraic expression ' $f(x_1, x_2, \cdots, x_n)$ '. This expression is called the *defining expression*, and constitutes the mathematical definition of the function. In the example, the defining expression is ' $\sqrt{\text{SQ}(x_2 - x_1) + \text{SQ}(y_2 - y_1)}$ '. Within the definition of this algebraic, you can use the local names as many times as you want, just as you would global names.

When you execute the name of a global variable containing a user-defined function, the stored program is executed as follows:

1. Objects are removed from the stack and stored in local variables, one object for each variable name.
2. The defining expression in the user-defined function is evaluated.
3. The local variables are purged.

To illustrate the function behavior of a user-defined function, consider a user-defined function SEC that returns the secant of a number:

$\ll \rightarrow x \text{ 'INV(COS(x))' } \gg \text{ 'SEC' STO.}$

You can execute SEC

- as an RPN command, e.g.

DEG 60 SEC  2.

- as an algebraic function, e.g.

'SEC(60)' EVAL  2.

Some other results:

'X' SEC		'INV(COS(X))'	<i>Symbolic arguments allowed.</i>
RAD 'SEC(X)' 'X' ∂		'SIN(X)/SQ(COS(X))'	<i>Differentiation works.</i>
'SEC(X)=Y' 'X' ISOL		Unable to Isolate	<i>Error!</i>

The last example shows that there is one important respect in which user-defined functions differ from built-in analytic functions. There is no inverse defined for a user-defined function, so ISOL can not solve for a name that is contained in the argument of the function.

One minor note: If the HP-28 is in algebraic entry mode (section 3.11.1), pressing the USER menu key corresponding to a user-defined function appends the function name to the command line, but does not add a trailing (.

8.2 User-defined Functions as Mathematical Functions

It is interesting to note the extent to which a HP-28 user-defined function is a realization of a mathematical function. That is, if you define a function such as $F(x) = 5x^2 + 2x$, you are stating that F is an operator that takes a single argument, and returns a single result that is computed from the argument. The function's definition has three parts:

1. The name F of the function.
2. A name x used to identify the function's argument. For the purpose of the definition, x does not have a value.
3. The expression in x that indicates how the result is computed from x .

When the function is applied to a specific argument, that argument is substituted for the name x in the defining expression, and the expression is evaluated. Thus

$$F(1) = 5 \cdot 1^2 + 2 \cdot 1 = 7$$

$$F(y^2) = 5(y^2)^2 + 2(y^2) = 5y^4 + 2y^2.$$

Each part of a function's definition has a corresponding representation in an HP-28 user-defined function:

1. The function's name is the name of the variable in which the user-defined function program is stored.
2. The argument name is the local name that follows the $\rightarrow$. A *local* name is appropriate because the name is not intended to have a value except when the function is actually being evaluated.
3. The expression defining the function is represented by the defining expression.

The example function $F(x) = 5x^2 + 2x$ is created in the HP-28 as:

```
<<  → x '5*x^2+2*x' >> 'F' STO.
```

Then

```
'F(1)' EVAL → 7,
```

and

```
'F(Y^2)' EVAL → '5*Y^2^2+2*Y^2'
```

In this example, we have considered a function of one variable. User-defined functions defined in terms of more than one local name naturally correspond to mathematical functions of more than one argument.

8.3 User-defined Functions Defined by Programs (HP-28S)

The HP-28S extends the definition of user defined functions to allow you to use a *defining program* in place of the defining expression, and still use the functions in algebraic expressions. The primary purpose of this facility is to allow you to include various RPN commands in algebraic calculations. For example, you can define an algebraic version of HMS+:

```
<<  → x y << x y HMS+ >> >> 'HMSP' STO
```

With this function, you can perform hours-minutes-seconds arithmetic within algebraic objects, e.g. `'5*HMSP(X,Y)'`.

Note, however, that you can not evaluate user-defined functions defined this way with *symbolic* arguments. When you evaluate an algebraic object containing a user-defined function defined by a program, its arguments must evaluate to numbers. For example, if you evaluate the algebraic '5*HMSP(X,Y)', both X and Y must have real-number values.

8.4 Beyond User-defined Functions

The user-defined function structure described in the preceding is a special case of the use of local variables. That is, you can use the $\rightarrow$ *names procedure* structure as a more general tool than just for creating user-defined functions. For example, if you want to use local variables in an RPN context, you can replace the defining expression in a user-defined function procedure with a program:

$$\ll \rightarrow \text{name}_1 \cdots \text{name}_n \ll \text{program} \gg \gg,$$

where $\ll \text{program} \gg$ is any program, which presumably uses the local variables. Furthermore, you can insert any number of objects into the outer program, ahead of the $\rightarrow$ or after the $\gg$ of the local variable procedure.

This more general use of local variables is described in detail in section 10.7. A key point to remember when you're working with user-defined functions is that a variable will not act as a user-defined *function* unless the program it contains has exactly the right structure--the first object in the program must be the $\rightarrow$, and the last must be the single algebraic object (or program in the HP-28S) immediately following the sequence of local names. If there is any variation from this prescription, you will not be able to use the variable name as a prefix function in an algebraic object definition.

8.5 Additional Examples

8.5.1 Permutations and Combinations

The HP-28S has built-in commands for computing permutations and combinations. On the HP-28C, you can easily program your own user-defined functions versions of these commands.

- The number of permutations of n objects taken m at a time is

$$P(n,m) = \frac{n!}{(n-m)!}$$

To create $P(n,m)$ as a user-defined function, enter:

`<< → n m 'FACT(n)/FACT(n-m)' >> 'P' STO.`

Then

`5 [ENTER] 2 [USER] [P] [F] 20;
'P(20,5)' [EVAL] [F] 1860480.`

- The number of combinations of n objects taken m at a time is

$$C(n,m) = \frac{n!}{m!(n-m)!}$$

The corresponding user-defined function is

`<< → n m 'FACT(n)/(FACT(m)*FACT(n-m))' >> 'C' STO.`

With this definition,

`5 [ENTER] 2 [USER] [C] [F] 10;
'C(20,5)' [EVAL] [F] 15504.`

- The n th *Catalan number* $CN(n)$ is equal to the number of paths between opposite corners of an $n \times n$ grid that may touch without crossing a straight line between the corners:

$$CN(n) = \frac{(2n)!}{(n+1)(n!)(n!)}$$

The corresponding user-defined function is

`<< → n 'FACT(2*n)/((n+1)*FACT(n)^2)' >> 'CN' STO`

8.5.2 Geometric Formulae

- $VCYL(r,h)$ returns the volume of a right-circular cylinder of radius r and height h :

`<< → r h 'π*SQ(r)*h' >> 'VCYL' STO,`

from the formula $V = \pi r^2 h$.

- $SCONE(r,h)$ returns the curved surface area of a right cone of altitude h and radius r .

`<< → r h 'π*r*√(SQ(r)+SQ(h))' >> 'SCONE' STO,`

from the formula $A = \pi r(r^2 + h^2)^{\frac{1}{2}}$.

- **CSEG(*r*,*x*)** returns the area of a segment of a circle, where *r* is the radius, and *x* is the perpendicular distance of the chord from the center:

`<< → r x 'π*r^2/2-x*√(r^2-x^2)-r^2*ASIN(x/r)' >> 'CSEG' STO,`

from the formula

$$A = \frac{\pi r^2}{2} - x \sqrt{r^2 - x^2} - r^2 \sin^{-1}\left(\frac{x}{r}\right).$$

- **PPER(*n*,*r*)** computes the perimeter of an *n*-sided polygon inscribed in a circle of radius *r*:

`<< → n r '2*n*r*SIN(π/n)' >> 'PPER' STO,`

from the formula $perimeter = 2n r \sin \frac{\pi}{n}$.

These user-defined functions return symbolic results containing π , unless you clear either flag 35 or flag 36 (see section 9.2.2) to cause automatic numerical evaluation of π .

9. Symbolic Math

The HP-28 is unique among calculators in its ability to apply mathematical operations to “symbolic” quantities--objects for which no numerical value has been assigned. If you’re a student learning algebra or calculus, or using their techniques in other mathematical or scientific studies, this capability may be very exciting. However, if you’re not directly interested in algebra for its own sake, you might wonder why the HP-28’s symbolic capabilities are important to you.

Actually, if you use a programmable calculator at all for more than simple keyboard arithmetic, you are already 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 an algebraic 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 new contribution of the HP-28 is that it allows 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 the HP-41, the program is

```
01 RCL 00
02 2
03 *
04 END
```

Here the “variable” you’re multiplying is represented by register 00. In BASIC, you can name the variable 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 either of these two languages, you have no choice but to edit your program. On the HP-41, you turn on program mode, find the correct place in the program to enter the SIN, and key it in. In BASIC, you edit line 100, being sure to enter the SIN in the right place and to include the parentheses.

On the HP-28, the original “program” consists of the algebraic 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. We showed a practical example of this facility in section 7.7. There you needed to modify the Solver current equation to move a local maximum below the horizontal axis. This is achieved by executing the sequence

RCEQ *number* - STEQ

where *number* is the value you wish to subtract from the current equation.

Another way to see the value of the HP-28 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-28, 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 using another calculator. The symbolic solution that you find with the HP-28 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 symbolically for the unknowns, you can still use the Solver 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.	'L+2*W=100' ENTER	'L+2*W=100'
2. Solve for L.	'L' SOLV ISOL	'100-2*W'
3. Assign this value to L.	'L' STO	
4. The area of the field is L times W.	'L*W=AREA' ENTER	'L*W=AREA'
5. Substitute for L.	EVAL	'(100-2*W)*W=AREA'
6. To find the maximum area, differentiate the expression.	'W' ENTER d/dx	'-(2*W)+(100-2*W)=0'
7. Collect terms.	ALGEBRA COLCT	'100-4*W=0'
8. Solve for W.	'W' SOLV ISOL	25
9. Assign this value to W and evaluate L.	'W' STO L EVAL	50

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

You can use the HP-28 to formulate and solve the entire problem. With a conventional calculator, all you can do is evaluate the final answer, once you have worked it out on paper.

As another example, in section 11.13.3 we list a program **SIMEQ** that solves a set 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-28'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-28'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.

As you will see in the remainder of this chapter, the HP-28 takes a generally *conservative* approach to symbolic mathematics. This means that it does not attempt to make decisions for you, but allows you to guide a symbolic calculation at each step. When you enter or compute an expression, the HP-28 does not force the expression into any particular form, but provides expression manipulation commands (section 9.8) so that you can rearrange it if necessary. For example,

$$'A+B' \quad 'B' \quad + \quad \Rightarrow \quad 'A+B+B'$$

The HP-28 does not automatically collect terms to return $'A+2*B'$; if you want that result, you can execute COLCT (section 9.8.1).

Another example of the conservative design is in the symbolic equation solutions returned by ISOL and QUAD (section 9.6). These commands return expressions representing *all* solutions to the equations, 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-28 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 among many solutions 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 9.1.1). 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.

9.1 The Nature of Algebraic Objects

In the preceding section we described algebraic 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-28 program works like this, since by definition a program contains a progression of any HP-28 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.

In section 2.1, we showed how RPN logic is derived from the desire to convert a mathematical expression into a series of steps by which you can evaluate the expression by hand or using a machine. Looking at this from a different point of view, you can note that since any expression can be translated to RPN, any expression can be represented in a calculator by an RPN program. In fact, this is what the HP-28 does--an algebraic object is stored in calculator memory in an RPN program form just like that of an actual program object. As a convenience, to save you from having to convert expressions to and from RPN as you must on an HP-41, the HP-28 provides the algebraic object type, for which it does the conversions automatically.

The only difference between algebraic objects and program objects is that the two are "marked" differently, so that the HP-28 knows which to display in algebraic form and which to display in RPN. Also, functions that accept symbolic arguments can only accept algebraic objects, not programs, since algebraics are by definition valid mathematical expressions, whereas program objects are completely unrestricted in their content and may not be suitable arguments for a mathematical function.

To illustrate the program nature of algebraic objects, create this program B:

```
<< DUP 20 >> 'B' STO
```

Next, enter the algebraic object '5+5+B', and press **EVAL**. The algebraic object disappears, and the numbers 10 and 30 appear on the stack. You can understand this result by following the execution of the equivalent RPN sequence 5 5 + B +. When this sequence is executed, two 5's are entered, then summed to 10 by the first +. B executes next, which duplicates the 10 and enters 20. Then the final + executes, returning

30. You can break down any algebraic object execution into RPN steps this way. Knowing how algebraic evaluation works is the key to understanding some of the subtleties of symbolic operations on the HP-28 in general.

Picturing an algebraic object as a program will also help you understand why evaluation of the object causes variable substitution "one level at a time." Consider the object 'A+B', where A has the value 10, B has the value 'C+D', C has the value 20, and D has the value 30. Evaluating 'A+B' once does one level of substitution, returning '10+(C+D)', not the numerical result 60. To see why, remember that 'A+B' is represented by the sequence A B +. Evaluating 'A+B' therefore executes A, B, and + in sequence: A returns 10, then B returns 'C+D', so that + returns '10+(C+D)'. [Note that the latter in RPN is 10 C D + +, which is obtained from the original A B + by substituting the RPN sequence C D + for B.]

These considerations also explain why you might get unexpected objects on the stack when an error occurs during evaluation of an algebraic object. For example, if you execute EVAL on an algebraic object and an error occurs, you might expect that the original object would be returned to the stack. But evaluating an algebraic object is the same as executing a program, so that an error returns the arguments of whatever function (within the algebraic) caused the error, along with anything else that was on the stack at the time of the error. Again, you can predict the contents of the stack from the RPN sequence that is equivalent to the algebraic object.

For example, suppose you execute 'A+(B+C)' EVAL, where A and B are undefined, but C has a vector value [1 2]. The HP-28 will halt and show the Bad Argument Type error message, with the stack containing

```

3:      'A'
2:      'B'
1:      [ 1 2 ]

```

This configuration results because the RPN sequence A B C + + errored at the first +. A, B, and C had already executed, leaving their values on the stack as shown; the + errored because the combination of a name ('B') and a vector ([1 2]) is not valid for addition. These arguments of +, not the original argument of EVAL, are returned to the stack. (Note that if you execute EVAL by using the **EVAL** key, you can restore the original algebraic object by pressing **UNDO**.)

9.1.1 Expression Structure

One advantage of writing a mathematical expression in Polish notation (section 2.1) is that it makes explicit the organization of the expression into a hierarchy of

subexpressions. A subexpression is any portion of a mathematical expression that can stand alone; that is, it can be treated as a complete expression by itself. Specifically, a subexpression consists of a number, or a name, or a function and its arguments. A number--real or complex--is the simplest case; if you like, you can think of a number as a function that takes no arguments and always returns the same value.

For example, consider the expression $a + \sin(b - c)$. Rewriting this in Polish form, you obtain $+(a, \sin(-(b, c)))$. The "outermost" subexpression is the entire expression, consisting of the function $+$ and its arguments a and $\sin(-(b, c))$. Each of the two arguments is a subexpression--the first is just the name a , the second is the function $\sin$ and its argument $-(b, c)$. The latter in turn is a subexpression consisting of $-$ and its arguments b and c , and so on as you peel off the layers of parentheses. The *level* of a subexpression is a measure of how deep it is in the hierarchy. The level is defined as the number of pairs of parentheses that surround the subexpression. In the example, the full expression is level 0; the a and $\sin(-(b, c))$ are level 1 subexpressions, $-(b, c)$ is level 2, etc.

There are two reasons for you to keep these ideas of expression structure in mind as you work with the HP-28:

1. The structure of an expression determines the order of evaluation of its subexpressions. For example, in the evaluation of ' $A+B+C$ ', the A and B are added first, then the sum is added to C . You can alter this order by changing the expression to ' $A+(B+C)$ ', in which case the B and C are added first. This distinction is important in a floating-point calculator, even though the two forms are formally the same. To see this, assign the values 10^{50} to A , -10^{50} to B , and 1 to C . If you evaluate ' $A+B+C$ ', you obtain 1, whereas if you evaluate ' $A+(B+C)$ ', you obtain 0.
2. Understanding the structure of an expression can help you follow the behavior of HP-28 symbolic manipulation commands. For example, EXPAN (section 9.8.2) is defined to work at one level of a subexpression at a time. ' $A*(B+C+D)$ ' EXPAN returns ' $A*(B+C)+A*D$ ' rather than ' $A*B+A*C+A*D$ ' as you might expect. This is more obvious if you think of the original expression as $*(A, +(+(B, C), D))$. When one of the arguments of $*$ is a sum, EXPAN multiplies the other argument by each of the two arguments of $+$, then adds the products. The fact that in this case the first argument of the (first) $+$ is also a sum is not considered--EXPAN only works one level at a time.

We can use these ideas to re-express the basic RPN calculator principle ("*any result can be an argument*") in "algebraic" terms by saying "*any expression can be a subexpression*." A subexpression is self-contained; it may or may not be embedded in a larger expression. The shortcoming of algebraic calculators is that they don't recognize this principle.

They are designed for evaluating an expression as a whole--“from the outside in,” so to speak. On the other hand, in an HP-41, you can only calculate an expression “from the inside out,” since you can only enter one number or function at a time. The HP-28 merges both approaches, by allowing you to enter any subexpression in its algebraic form. You can evaluate an entire expression at once, or you can divide it into subexpressions of any size, or you can work only with one object at a time.

As with most of the principles of HP-28 operation, the concept of evaluation embodied by EVAL is derived from a mathematical model. In ordinary terms, to “evaluate” means “to find the value.” For a mathematical expression, this translates to “perform the operations represented by the expression, to find its value.” Evaluation means to “activate” an expression, which in turn means to execute sequentially the objects that make up the expression.

As an example, consider the simple expression $1+2$. We showed in section 2.1 that an expression can be translated into an RPN form that represents a prescription for actually performing the operations of the expression--evaluating it. Thus the expression $1+2$ is the sequence $1\ 2\ +$ in RPN. This is a sequence of *objects*--remember (see section 3.2.1) that the $+$, as well as the 1 and the 2, can be considered as an object. When you write the expression, the objects are passive; but if you execute each object in turn--“enter the 1, enter the 2, do the $+$ ”--you obtain the value of the expression.

9.2 Function Execution

HP-28 functions have two important execution properties that are not shared by RPN commands. These are *automatic simplification*, and a choice of *symbolic* and *numerical* execution modes.

9.2.1 Automatic Simplification

When certain functions execute, they check their arguments for special cases in which ordinary calculation can be replaced by a mathematical simplification. For example, if you execute the sequence $1\ 'X'\ *$, you obtain $'X'$, not $'1*X'$. You can observe the same effect by executing $'1*X'$ EVAL. This simplification is a property of the $*$ function; when it is executed, $*$ explicitly looks for cases where one of its arguments is 1. In such cases, the subexpression consisting of the $*$ and its two arguments is automatically replaced by the non-1 argument. Other examples are the replacement of $\text{SIN}(\text{ASIN}(X))$ by X , and $\text{EXP}(\text{LN}(X+1))$ by $X+1$. Again, these simplifications are built into the functions SIN and EXP. A complete list of automatic simplifications is given in the HP-28 Reference Manual.

Note that not all cases of a function applied to its own inverse are simplified. For

example, $\text{ASIN}(\text{SIN}(X))$ does not automatically simplify to X , since there are infinitely many angles with the same sine as X . Similarly, since the HP-28 treats complex numbers uniformly with real numbers, $\text{LN}(\text{EXP}(X))$ does not reduce to X .

Automatic simplification is not the same as the simplification that results when a numerical expression is evaluated by COLCT. For example, although $'2/2'$ automatically simplifies to 1 when you evaluate it, $'2*X/2'$ does not automatically simplify to X . In order for the simplification to take place, the two 2's must be the arguments of the $/$, as in $'(2/2)*X'$. To simplify $'2*X/2'$, you can either use FORM to rearrange it to $'(2/2)*X'$, or use COLCT (section 9.8).

9.2.2 Symbolic and Numerical Evaluation; $\rightarrow\text{NUM}$

The key to the HP-28's ability to perform symbolic calculations is the fact that HP-28 functions used with symbolic arguments (names or algebraics) return symbolic results. Each time you evaluate an algebraic object, the names in the expression or equation are executed, so that those corresponding to existing variables are replaced by the objects stored in the variables. But the replacement objects are *not* evaluated, so that the final result may still be symbolic. If you want to evaluate a symbolic object all the way to a numerical value, you may have to use EVAL repeatedly until all of the names have been replaced by numbers.

In some circumstances, it is desirable to evaluate a symbolic object to its final numerical value in a single operation. For example, in the course of their execution, DRAW and the Solver both evaluate the current equation to numerical values. To deal with such cases, as well as the symbolic evaluation described already, the HP-28 provides you with the choice of *symbolic evaluation mode* or *numerical evaluation mode*. In symbolic evaluation mode, a function evaluated with symbolic arguments returns a symbolic result. In numerical evaluation mode, a function of symbolic arguments evaluates its arguments, repeatedly if necessary, until they are data objects (usually numbers). Then the function returns a numerical result. If any name is encountered during the evaluations that has no corresponding variable, the Undefined Name error is returned.

You can select numerical evaluation mode temporarily, for a single evaluation of a symbolic object, or for an indefinite period:

- To evaluate numerically a single object containing functions, use $\rightarrow\text{NUM}$ instead of EVAL. $\rightarrow\text{NUM}$ enables numerical evaluation mode, evaluates its argument in the same manner as EVAL, then restores the original evaluation mode. A common use of $\rightarrow\text{NUM}$ is to force the numerical evaluation of an algebraic object used as a test for a program branch. For example,

```
IF 'X==3 OR X>10'  $\rightarrow\text{NUM}$  THEN A ELSE B END
```


executes A if X has the value 3 or is greater than 10, or B otherwise (see section 10.4).

- To select numerical evaluation mode “permanently,” clear flag 36 (execute 36 CF). As long as flag 36 is clear, the evaluation of any functions will return a numerical result, or an error message if numerical evaluation fails. In this mode, EVAL and →NUM produce the same results.

To restore symbolic evaluation mode, set flag 36. Symbolic evaluation mode, with flag 36 set, is the default mode following a memory reset (section 11.3.4).

To illustrate these ideas, execute

30 'X' STO 'X'

to create a variable X with the value 30, and leave its name on the stack. Next select degrees mode by executing DEG (**MODE** **DEG**) if necessary. Now,

1. Select symbolic evaluation: 36 **TEST** **SF** .

Compute the sine:

TRIG **SIN** **→** 'SIN(X)'.

At this point, you still have a symbolic result. Find the numerical value:

EVAL **→** .5.

When 'SIN(X)' is evaluated, X is replaced by its value 30; then, since SIN has a numerical argument, a numerical result is returned.

2. Now try the calculation in numerical mode: **TEST** 36 **CF** .

Compute the sine:

'X' **TRIG** **SIN** **→** .5

This time, you immediately obtain the numerical result .5. This is because in numerical evaluation mode, SIN evaluates the symbolic argument 'X' to its value 30, then returns the numerical $\sin 30^\circ$.

9.3 Symbolic Constants

A frequently asked question about HP calculators is “why does the sequence π SIN (in radians mode) *not* return 0, when everybody knows that $\sin(\pi) = 0$?” On the HP-41, for example, π SIN returns $-4.1\text{E}-10$. The answer is that the π key does not return

mathematical π , but an approximation accurate to the numerical precision of the calculator, which is the 10 digit number 3.141592654 on the HP-41. When SIN uses this approximation as an argument, it treats it like any other floating-point number and computes its sine, again accurate to the calculator's precision. To understand the approximate value, consider that for small x , $\sin(\pi + x) = -x$. In this case, x is the difference between π and the calculator approximation: $\pi + x = 3.141592654$. Thus

$$x = 3.141592654 - 3.14159265359^+ \approx 4.1 \times 10^{-10},$$

and

$$\sin(\pi + x) = -4.1 \times 10^{-10},$$

which is just what the HP-41 returns. SIN is evidently returning an accurate result for its argument, but the argument is not π .

Could a calculator be designed to recognize the approximation as its best numerical representation of π and return zero for the sine of that number? Certainly it could, but HP calculators generally don't do this sort of thing, following the guideline that the limitations of fixed-precision calculations make it unwise to try to guess when a numerical value is supposed to be some special number. This sort of problem shows up in lots of cases: for example, should .142857142857 $\blacksquare$ $\boxed{1/X}$ return 7.00000000001, which is the most accurate 12-digit reciprocal of that argument, or 7.00000000000, on the chance that .142857142857 was obtained originally by computing the reciprocal of 7? This problem is a fundamental limitation of trying to represent arbitrary numbers with a finite number of digits.

The HP-28 provides a different approach to the problem of π from its predecessors. Assuming for the moment that flags 35 and 36 are set, executing π returns the *expression* ' π ' (note that this is an algebraic object, not a name--e.g. TYPE returns 9). If you execute $\pi \ 2 \ *$, you obtain ' $2*\pi$ '. As long as you don't force numerical evaluation by executing $\rightarrow\text{NUM}$, π retains its symbolic form through any number of operations. This has two immediate benefits:

- An expression containing the symbol π gives you more information about the nature and derivation of the expression. Once you convert it to a numerical form, no matter how accurate, the presence of π in the expression becomes obscured. The expression ' $\pi/4$ ' is more informative than the number 0.785398163398.
- Using symbolic π prevents errors arising from a finite precision numerical representation of π from accumulating in chained calculations. By delaying the substitution of a numerical value for π until a calculation is complete, you obtain maximum accuracy.

A symbolic π also permits a new resolution of the $\sin(\pi)$ issue. On the HP-28, if you

execute π SIN (with flags 35 and 36 set, and radians mode active), you obtain 0. This is an automatic simplification (section 9.2.1), not a numerical computation--when SIN is executed, it checks its argument to see if it is symbolic π . If so, the subexpression SIN(π) is replaced by 0. The following additional simplifications are also made, in the same spirit:

- SIN($\pi/2$) is replaced by 1;
- COS(π) is replaced by 0 (note: COS(3.14159265359) also returns 0);
- COS(π) is replaced by -1.
- TAN(π) is replaced by 0.

Only these four specific subexpressions are simplified. SIN($2*\pi$), for example, is not simplified, and returns 4.13523074713E-13 when evaluated numerically.

There are occasions when the symbolic form of π can be an impediment. One such case is when you want to use the numerical value of π as one of the limits of integration for a numerical integral. (This difficulty is reduced on the HP-28S, where you can actually include symbolic π in the list argument for the integral.) The limits are specified as two real numbers in a list containing two (implicit variable integration) or three (explicit variable integration) objects. If you enter the list directly from the keyboard using the { } delimiters, you must enter π by typing in its digits. It's usually easier to put the two or three list elements on the stack, using π where appropriate, and then combine the elements into a list with $\rightarrow$ LIST.

- Example: Integrate $\sin(x)$ from $x=0$ to $x=\pi$, with an accuracy of 10^{-5} .

Keystrokes:

'SIN(X)' [ENTER] 'X' [ENTER]

0 [ENTER] π [ENTER] $\rightarrow$ NUM

3 [ENTER] [LIST] $\rightarrow$ LIST

[EEX] [CHS] 5 [ENTER] [J]

Results:

2: 'SIN(X)'
1: 'X'

4: 'SIN(X)'
3: 'X'
2: 0
1: 3.14159265359

2: 'SIN(X)'
1: { X 0 3.14159265359 }

2: 2.00000003046
1: 2.00195785979E-5

9.3.1 Other Symbolic Constants

In addition to π , the HP-28 provides four other symbolic constants: e (numerical value 2.71828182846), i (value (0,1)), MAXR (value 9.99999999999E499), and MINR (value 1E-499). There are no special simplifications associated with e , MINR or MAXR, but the symbolic forms allow you to track the associated constants through calculations. i has these simplifications:

Subexpression	Replacement
SQ(i)	-1
$i*i$	-1
i^2	-1
$i^{(2,0)}$	-1
RE(i)	0
IM(i)	1
CONJ(i)	-i

You can use i to enter complex numbers in the form $a + bi$ rather than the standard object format (a,b) . For example, $1+2i$ can be entered as ' $1+2*i$ '. You can perform arithmetic with such expressions, using EXPAN and COLCT where appropriate to simplify a multi-term expression into the form $a + bi$.

9.3.2 Evaluation of Symbolic Constants

Symbolic and numerical evaluation modes affect the way all built-in HP-28 functions evaluate symbolic arguments. The five symbolic constants π , e , i , MAXR and MINR behave as functions of zero arguments--and as functions they are sensitive to the evaluation mode. When flag 36 is set, evaluation of any of these constants returns a symbolic result, which is just the constant itself unchanged. When flag 36 is clear, evaluation of a symbolic constant replaces it with its numerical value.

It is possible by means of flag 35 to select a restricted form of numerical evaluation mode that affects only these constants. When symbolic evaluation mode is active (flag 36 set), clearing flag 35 causes symbolic constants to evaluate numerically, without affecting the evaluation of other functions. This permits, for example, replacement of symbolic constants with numerical values in expressions that contain formal variables (undefined names). To see this, enter 'X' PURGE, then enter the expression ' $2*\pi*X$ ' into level 1. Then,

35 SF 36 SF EVAL  ' $2*\pi*X$ '

and

→NUM Undefined Name error.

But if you clear flag 35:

'2* π *X' 35 CF EVAL '6.28318530718*X'.

π evaluates to its numerical value, while with flag 36 set * still returns a symbolic product.

9.4 General Problem Solving with Symbolics

At the start of this chapter, we outlined a five-step general problem solving process. Now we will review those steps, and see how they are realized on the HP-28. 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 variables that you assign to the known quantities are called *known* variables, or *independent* variables since their values can be set arbitrarily. The *unknown* variables are the quantities you are going to calculate.

In the example, the independent 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, or independent, 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-28 proceeds as follows:

Verbal Statement	Keystrokes		Stack
(Purge existing variables.)	{ D T N } ■ 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 * = ENTER	1:	'D+T=(S+T)*N'

The equation in level 1 is the relationship you need.

4. *Solve the relationships for the unknowns.*

This is a process of algebra, where you apply standard rules for expression rearrangement to isolate the unknown variable as a single quantity on one side of an equation. The HP-28 command **ISOL** (section 9.6.1) will do this for you automatically if the variable appears just once in the expression or equation. An automatic solution is also possible for quadratic equations, for which **QUAD** (section 9.6.2) will find both solutions regardless of the specific form of the equation.

If the unknown variable appears more than once in an expression that is not a simple quadratic, you must try to find a rearrangement of the expression using standard rules of algebra to combine the occurrences of the unknown into a single one. The HP-28 lets you perform many such operations right on the calculator. To illustrate, return to the example. At the last step, you had obtained the equation $'D+T=(S+T)*N'$. The unknown T appears twice, so you can't use **ISOL** yet. Instead, you can follow the same steps on the HP-28 as you might on paper:

Verbal Statement	Keystrokes	Stack
(Result of previous steps.)		1: 'D+T=(S+T)*N'
Expand the product.	ALGEBRA EXPAN	1: 'D+T=S*N+T*N'
Subtract T from both sides.	'T' -	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' - 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 FORM to merge the two terms:

Keystrokes:

Display:

FORM

((**-** (S*N) + D) = ((T*N) - T))

[←] (11 times)

NEXT ***1**

((- (S*N) + D) = ((T*N) - **T**))
 ((- (S*N) + D) = ((T*N) - (T ***** 1
)))

ENTER **[←]** **[←]**

((- (S*N) + D) = ((T*N) **-** (T*1
)))

NEXT **-M**

((- (S*N) + D) = (T ***** (N - 1)))

ON

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

(The use of FORM is explained in section 9.8.4.) Now you have achieved the desired goal of a single appearance of the unknown T. At this point, you can execute either

'T' ISOL **[→]** '(-(S*N)+D)/(N-1)',

or

'(N-1)' **÷** **COLCT** **[→]** '(-(S*N)+D)/(-1+N)=T'

The second choice produces an equation rather than an expression, which has the

advantage that it keeps T in the solution in case you want to solve later for a different variable. With either method, you now have a symbolic solution to the problem, and you can proceed to substitute numbers for specific solutions.

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 the Solver 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 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-28:

- 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 assign values to D , S , and N from the original problem:

40 'D' **STO** 10 'S' **STO** 2 'N' **STO**

Then, with the solution ' $(-(S*N)+D)/(-1+N)=T$ ' on the stack,

EVAL $\Rightarrow$ '20=T'

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 the Solver (Chapter 7). Make your solution object the current equation by pressing **SOLV** **STEQ**, and press **SOLVR** to activate the Solver menu. Then,

40 **D** 10 **S** 2 **N** **T** $\Rightarrow$ 20.

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

3 **N** **T** $\Rightarrow$ 5.

9.5 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 the Solver for purely numerical answers, it’s 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-28 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, that approximate the steps you carry out in pencil-and-paper calculations. Included among these is FORM, for detailed expression manipulation, EXPAN (expand), for distributing multiplication and powers, and COLCT (collect), for combining like terms. 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) in the unknown. For equations that don’t fit either of these criteria, the typical HP-28 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.

9.6 Automated Symbolic Solutions: QUAD and ISOL

The commands QUAD and ISOL automatically carry out several steps in 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 expression representing the solution--the value of the unknown variable that satisfies the original expression or equation. For example, in section 9.8 we use ISOL to solve the equation

$$'((A-B)*X) = (- (A*Y) + C)'$$

for X, obtaining the *expression*

$$'(- (A*Y) + C) / (A - B)',$$

in which X does not appear. You can understand the result expression as the right side of the equation $'X = (- (A*Y) + C) / (A - B)'$. ISOL returns only the expression, so that you can store it in the unknown variable and use that value for the unknown in subsequent calculations.

If you would rather have your answer as an equation, you can use the following program as an alternate form of ISOL:

```
<< SWAP OVER ISOL = >> 'ISOLE' STO
```

ISOLE works just like ISOL, except that it returns an equation. You can do the same for QUAD:

```
<< SWAP OVER QUAD = >> 'QUADE' STO
```

9.6.1 ISOL

ISOL solves an equation for an unknown much the same way you would with pencil and paper. That is, ISOL finds the (first) term containing the unknown, and moves it to the left side of the equation. It moves 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. ISOL then returns the expression on the right side of the final equation as its result. For example,

and

'2*X+8=0' 'X' ISOL $\Rightarrow$ -4,

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

[In the original Version 1BB HP-28C, ISOL evaluates the solution expression before returning it to the stack. Therefore, when you want a purely symbolic ISOL solution on that calculator, you must purge the variables that appear in the original equation. ISOL was modified in HP-28C Version 1CC and the HP-28S to omit the final evaluation.]

A result returned by ISOL is only a proper solution to the original equation if three conditions are met:

1. the unknown variable name appears just once in the equation; and
2. the unknown appears only in the arguments of HP-28 *analytic* functions.
3. no variables in the equation contain algebraic objects or programs that have the unknown in their definitions.

Condition 1 is easy to understand. If the unknown appears more than once, ISOL still returns a result, but it is of limited value because the unknown will still be present in the result expression. ISOL finds the first (from left-to-right) occurrence of the unknown and solves for that, treating remaining occurrences as if they were different variables:

'X+X=Y' 'X' ISOL $\Rightarrow$ 'Y-X'

The "solution" you get in this case is not really a solution at all, but just a rearrangement of the original expression. Your challenge, when you have an expression with multiple occurrences of the unknown, is to use the various algebra tools provided by the HP-28 to rewrite the expression so that it contains only one instance of the unknown (this is just what you would do with pencil and paper). This process was illustrated in section 9.4, and is described in detail in sections 9.8.

Condition 2 is really part of a circular definition--ISOL will only work with analytic functions, but part of the definition of an HP-28 analytic function is that the HP-28 "knows" its inverse and hence can isolate its argument. The HP-28 term *analytic* is derived from the mathematical definition of an analytic function as one that is continuous and differentiable. If you inspect the HP-28 function set, it is usually easy to figure out why a particular function is analytic or not 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.

Condition 3 means that you must be aware of any occurrences of the unknown in the objects stored in any of the other variables in the expression. ISOL only works at the "surface level" of an expression. 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 ' $C-A$ '. But if A has the value ' $B-C$ ', then the result ISOL returns does not represent a proper solution. In that case, you need to use SHOW to make all references to B explicit: ' $A+B=C$ ' 'B' SHOW returns ' $B-C+B=C$ ', which you can then proceed to solve for B.

When ISOL returns the Unable to Isolate error message, it means that its argument equation does not satisfy condition 2 (or does not contain the specified variable at all). There is no error generated when condition 1 is not met. This has the disadvantage that when you use ISOL in a program, the program will continue even if the result returned by ISOL is meaningless in some cases. To protect against this situation, you can substitute the following program ISOLCK for ISOL. ISOLCK tries to isolate the variable twice--once from the original expression, and once from the result. If the second ISOL does not error, then the variable occurred at least twice in the expression and the program will abort. ISOLCK is not foolproof, however, because the error could be caused by the unknown appearing (the second time) in a non-analytic function.

ISOLCK		Isolate and Check	
	level 2	level 1	level 1
	'algebraic'	'name'	'solution'
<pre> << → x << 31 SF x ISOL IFERR DUP x ISOL THEN DROP2 ELSE DROP "Multiple" ABORT END >> >> </pre>		Activate LAST. First isolation. An error means a good solution. Discard the extra x and solution. Return a message and quit.	

ISOLCK sets flag 31.

9.6.2 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 quadratic 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 algebraic argument is evaluated. Therefore, if you want to prevent substitution for the names in the algebraic, you must purge the corresponding variables before executing QUAD.

In keeping with the HP-28'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

⌞ '(-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.

9.7 Multiple Roots

A fundamental principle 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`. `s1` represents a $\pm$; it is called 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.

Similarly, ISOL returns a single expression representing all possible solutions for its algebraic 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 in a solution. These names represent *arbitrary integers*. The arbitrary integers may each take *any* integer value `0`, `$\pm 1$` , `$\pm 2$` , ..., for each of which ISOL's result will evaluate to a specific solution to the original expression. For example,

```
RAD 'SIN(X^2)=Y' 'X' ISOL ␣ '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 $x^2 = y$. 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 esthetic 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 in the Solver, which only returns a single answer. In cases with multiple roots, the Solver 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 7.4), 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 Solver uses zero as an initial guess--which may or may not be a good choice for the problem at hand.

The HP-28 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 34 set), 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:

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

The complex result is one of the three cube roots of -1 , but it's not the one you're after. Now try solving again, with principal value mode off:

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

```
(EXP(2*PI*I*N1/3)*(.500000000001,.866025403784)).
```

Translated to common notation, this result is

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

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 repeat the same roots. The following table lists the values returned by the HP-28 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-28 Result	Exact Value
0	(.5000000000001,.866025403784)	$(\frac{1}{2}, \frac{\sqrt{3}}{2})$
1	(-1,4.465E-12)	$\frac{-1}{2}$
2	(.5000000000001,-.866025403784)	$(-\frac{1}{2}, \frac{\sqrt{3}}{2})$

Unless you force it by setting principal value mode, the HP-28 does not attempt to choose one possible root over any other when you use ISOL. There's really no mathematical grounds on which it 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 good 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-28 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 USER menu, the Solver, etc.) to select values for the constants.

9.7.1 Using the Solver to Select Roots

The Solver 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 **EXPR=** key. Then you can use the menu keys to select values for the variables, and press **EXPR=** to obtain the evaluated expression.

To illustrate, return to the example $x^3 = -1$:

Keystrokes:**Results:**

■ **TEST** 35 **≡SF≡** 36 **≡SF≡**
 34 **≡CF≡**
 ■ **MODE** 3 **≡FIX≡**

Select symbolic evaluation.
 Principal value mode off.
 3 decimal places.

'X³ = -1' **ENTER**
 'X' **SOLV** **≡ISOL≡**

1: 'EXP(2*π*i*n1/3)*
 (0.500,0.866)'

≡STEQ≡ **≡SOLVR≡**

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:**Results:**

0 **≡N1≡** **≡EXPR=≡**

1: (.500,.866)

1 **≡N1≡** **≡EXPR=≡**

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

■ **~NUM**

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

2 **≡N1≡** **≡EXPR=≡** ■ **~NUM**

3: (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.

9.8 Expression Manipulations

The most common operations you perform when solving algebraic equations are these:

- *Reordering terms.* A frequent step in equation solving is moving all of the terms containing the unknown variable together so that multiple occurrences of the unknown

can be merged into a single occurrence. This is achieved in the HP-28 by means of various operations in the FORM menu, especially 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-28 is represented by EXPAN, and by the D→ and ←D operations in the FORM menu.
- *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 merge operations M→ and ←M in FORM.

As an example of HP-28 algebra, consider solving the equation $a(x+y) = bx + c$ for x :

1. Represent the equation with the algebraic object ' $A*(X+Y)=B*X+C$ '.
2. Distribute the term $A*(X+Y)$:

EXPAN  'A*X+A*Y=B*X+C'.

3. Move the term $A*Y$ to the right side, by subtracting it from both sides:

DUP 6 EXGET - COLCT 'A*X = -(A*Y) + B*X + C'

The sequence `DUP 6 EXGET` extracts a copy of the term `A*Y`. You can also just type `'A*Y'`, but `EXGET` can save you some effort if the term is large.

4. Similarly, move $B \cdot X$ to the left:

DUP 11 EXGET - COLCT 'A*X-B*X=- (A*Y)+C'

5. Now merge the terms that contain X. To achieve this, use FORM:

FORM '(((**A** *X) - (B*X)) = (- (A*Y) + C))'

Move the **FORM** cursor to select the **-**:

 '(((A*X) - (B*X)) = (- (A*Y) + C))'

6. X is a common right factor of the two terms that are the arguments for $-$. Therefore, you can merge the terms:

NEXT **M** $\rightarrow$ '(((A-B) * X) = (- (A*Y) + C))'

7. Press **ON** to end FORM:

ON $\rightarrow$ '(A-B)*X = - (A*Y) + C'

8. Now there is only one occurrence of X, so the equation can be solved by ISOL:

'X' ISOL $\rightarrow$ '(- (A*Y) + C) / (A-B)'

This example illustrates the use of the three primary symbolic manipulation commands, EXPAN, COLCT, and FORM. Each of these allows you to alter the form of an expression without changing its net value. EXPAN and COLCT are "broad brush" symbolic manipulation commands that perform wholesale rearrangements. They both have the shortcoming of trying to do many operations at once, which you can't control individually. This means that in many cases you may be surprised or dissatisfied with the results returned by these commands, because they don't match some particular form that you desire. Consider, for example, the expansion of x^3 . All of these forms are formally equivalent: xxx , x^2x , and xx^2 . Which form should EXPAN return? There is no "right" answer, so the HP-28 makes one choice, namely xx^2 :

'X^3' EXPAN $\rightarrow$ 'X*X^2'.

You can obtain the other choices by using FORM to commute the arguments of the * (switching 'X*X^2' to 'X^2*X') or using EXPAN again (to obtain 'X*(X*X)').

FORM, 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 one careful step at a time. The path of individual operations you need to follow to change an expression from one form to another may not be obvious, particularly since you have to work through a maze of parentheses to identify the particular subexpressions or functions you want to work on. FORM requires a good deal of skill and practice for any but the most straightforward rearrangements; for complicated situations you may just want to resort to EDIT.

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 FORM to rearrange parts of the expression, until you obtain the desired final version. For complicated expressions, you can extract subexpressions using EXGET, rearrange the subexpressions using the same methods, and substitute the subexpressions back into the main expression using EXSUB.

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.

9.8.1 COLCT

The purpose of COLCT is to simplify an expression by combining “like terms.” A “term” to the HP-28 is an argument of + or -; for example, in the expression $2\sin x + y$, the terms are $2\sin x$ and y . “Like terms” are 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 to combine 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, $(a \cdot b)c$ has the same value as $a(b \cdot c)$; 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-28 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.

9.8.2 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:

$$'A*(B+C)' \text{ EXPAN } \Rightarrow 'A*B+A*C'$$

$$'(B+C)*A' \text{ EXPAN } \Rightarrow 'B*A+C*A'$$

$$'(B+C)/A' \text{ EXPAN } \Rightarrow 'B/A+C/A'$$

$$'A^{(B+C)}' \text{ EXPAN } \Rightarrow 'A^B*A^C'$$

(You can also substitute $-$ for $+$ in the above examples.)

When both arguments of a product are sums, only the second sum is distributed:

$$'(A+B)*(C+D)' \text{ EXPAN } \Rightarrow '(A+B)*C+(A+B)*D'$$

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 → 'A*A^4'.

- Squares of sums are expanded from $(a + b)^2$ to $a^2 + 2ab + b^2$:

'(A+B)^2' EXPAN → 'A^2+2*A*B+B^2'

'SQ(A+B)' EXPAN → '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 → 'LN(A)+LN(B)'

'LN(A/B)' EXPAN → 'LN(A)-LN(B)'

EXPAN distributes the antilogs of sums and differences as follows:

'EXP(A+B)' EXPAN → 'EXP(A)*EXP(B)'

'EXP(A-B)' EXPAN → '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 → 'A*(B*(C+D))+A*(E*(F+G)).'

To understand this example, it's useful to write the expression in Polish notation:

*(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

$$'A*(B*C+B*D)+A*(E*F+E*G)'$$

9.8.3 Simplifying Polynomials

A very convenient method of rearranging an expression into a polynomial in a specific variable is provided by the TAYLR command (section 9.9.2). TAYLR approximates an arbitrary expression with a finite-order MacLaurin polynomial (expansion around zero); if the expression is already a polynomial of order n , the approximation is exact if carried out at least to degree n .

For example, to expand the expression $'(X+Y)^4'$ into a standard polynomial, you need to execute EXPAN and COLCT several times. It's hard to tell in advance how many EXPAN's are needed. However, you can achieve the rearrangement in a more straightforward manner using TAYLR:

$$'(X+Y)^4' \quad 'X' \quad 4 \quad \text{TAYLR} \quad \text{COLCT}$$

$$\quad \quad \quad '6*X^2*Y^2+4*X*Y^3+4*X^3*Y+X^4+Y^4'.$$

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.

9.8.4 FORM

FORM is the ultimate "conservative" HP-28 symbolic manipulation command. It provides an extensive menu of one-step operations that allow you to rearrange an expression into almost any form you want. It won't make any changes that you don't specify; but this means that a substantial rearrangement can be a formidable process.

FORM shares with EXPAN and COLCT the fundamental property that it is an *identity*

operation--it never changes the formal value of an expression. You can use it to rearrange an expression into a form more suitable for further calculations, with the confidence that FORM won't let you make mistakes that alter the symbolic value of the calculation that the expression represents. This differs significantly from the ordinary syntax checking that an editor like EDIT or VISIT performs.

The basic use of FORM centers around subexpressions defined by functions and their arguments. Associated with each function is a menu of operations that correspond to various rules of mathematics that apply to the function. A function's arguments in a subexpression determine which of those operations is meaningful in each particular case. For example, the FORM operations defined for $*$ are commutation, association, distribution, merging, double negation, double inversion, 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 ' $INV(INV(A)/B)$ ').

The mechanics of using FORM are covered adequately in the HP-28 manuals. The applications of FORM in the examples in sections 9.4 and 9.8 are typical of its most common use, which is to restructure an expression for use with ISOL. Our discussion here is limited to some observations on the individual FORM operations.

9.8.4.1 Commutation: $\leftrightarrow$

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 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, e.g. reordering $ax + by + cx$ to $ax + cx + by$, by commuting the arguments of the second $+$.

9.8.4.2 Association: $\leftarrow A$ and $A \rightarrow$

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:

$$a + (b + c) \text{ is } a \ b \ c \ + \ +$$

$$(a + b) + c \text{ is } a \ b \ + \ c \ +$$

There are two functions involved in any association (the two $+$'s in the example); in FORM, you need to 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. $\equiv \leftarrow A \equiv$ (*associate left*) works when the selected operator is to the left of the second operator, moving the parentheses to the left. For example,

$$(A + B) \boxed{+} (C + D) \equiv \leftarrow A \equiv \Rightarrow ((A + B) + C) \boxed{+} D.$$

Similarly, $\equiv A \rightarrow \equiv$ 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, use the opposite operation.

Notice that the selected function changes after $A \rightarrow$ or $\leftarrow A$. The selected object after any FORM operation is always the object that defines the new subexpression that contains all of the objects from the original subexpression.

9.8.4.3 Distribution: $\leftarrow D$ and $D \rightarrow$

The *distribution* operations in FORM 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 ' $A*(B+C)$ ', you select the $*$, then press $\equiv \leftarrow D \equiv$. This returns ' $A*B+A*C$ ', in which the $+$ is highlighted, since it is the defining object for the new subexpression.

Two distribution operations are necessary because of the ambiguity of expressions like $(a + b)*(c + d)$. $D \rightarrow$ distributes the sum on the right:

$$(A+B) \boxed{*} (C+D) \equiv \boxed{D\rightarrow} \Rightarrow ((A+B)*C) \boxed{+} ((A+B)*D).$$

$\leftarrow D$ distributes the sum on the left:

$$(A+B) \boxed{*} (C+D) \equiv \boxed{\leftarrow D} \Rightarrow ((A*(C+D)) \boxed{+} (B*(C+D)).$$

For logarithms and antilogarithms, which are functions of one argument, there is no ambiguity, and only $D\rightarrow$ is allowed. Thus,

$$\boxed{LN} (A*B) \equiv \boxed{D\rightarrow} \Rightarrow LN(A) \boxed{*} LN(B),$$

but you can not apply $\leftarrow D$.

9.8.4.4 Merging: $\leftarrow M$ and $M\rightarrow$

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:

$$A*B \boxed{+} A*B \equiv \boxed{M\rightarrow} \Rightarrow (A+A) \boxed{*} B,$$

and

$$A*B \boxed{+} A*B \equiv \boxed{\leftarrow M} \Rightarrow A \boxed{*} (B+B).$$

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

$$\text{EXP}(A) \boxed{*} \text{EXP}(B) \equiv \boxed{\leftarrow M} \Rightarrow \text{EXP}(A+B).$$

9.8.4.5 Prefix Operations: $\rightarrow()$, $\rightarrow()$, $1/()$, DNEG and DINV

This group of operations is organized around the "prefix operators" $\rightarrow$ and INV , which also happen to be their own inverses: $a = -(-a)$ and $a = INV(INV(a))$. The basic operation is *distribute-prefix-operator*, $\rightarrow()$, which "pushes" one of these operators "into" its parentheses by altering the argument. For example,

$$\boxed{-} (A+B) \equiv \boxed{\rightarrow()} \Rightarrow -(A) \boxed{-} B$$

The inverse of $\rightarrow()$ depends on the operator that is distributed.

- For negation, the effect of $\rightarrow()$ is reversed by $\rightarrow()$. The latter is called *double negate and distribute*, since it is equivalent to double negation (see below) followed by a distribution of a $\rightarrow$ prefix operator. Example:

$$\boxed{-} (A * B) \quad \boxed{\boxed{-}(\boxed{})} \quad \Rightarrow \quad -(A) \boxed{*} B.$$

and

$$-(A) \boxed{*} B \quad \boxed{\boxed{-}(\boxed{})} \quad \Rightarrow \quad \boxed{-} (A * B).$$

- *Inverse of power or inverse of inverse-product:*

$$\boxed{\text{INV}} (A^B) \quad \boxed{\boxed{-}(\boxed{})} \quad \Rightarrow \quad A \boxed{\wedge} -B.$$

$$\boxed{\text{INV}} (\text{INV}(A)/B) \quad \boxed{\boxed{-}(\boxed{})} \quad \Rightarrow \quad A \boxed{*} B.$$

To return either of these subexpressions to its original form, you must use $\boxed{\boxed{1/(\boxed{})}}$, which inverts a subexpression by inverting its arguments. Another example of $1/(\)$ is

$$\boxed{\text{EXP}} (A) \quad \boxed{\boxed{1/(\boxed{})}} \quad \Rightarrow \quad \boxed{\text{INV}} \text{EXP}(- (A)).$$

- *Double inverse:*

$$\boxed{\text{INV}} (\text{INV}(A)) \quad \boxed{\boxed{-}(\boxed{})} \quad \Rightarrow \quad \boxed{A}.$$

The reverse operation is DINV (double-inversion) which takes any subexpression A and changes it into $\text{INV}(\text{INV}(A))$.

- *Double negative:*

$$\boxed{-} (- (A)) \quad \boxed{\boxed{-}(\boxed{})} \quad \Rightarrow \quad \boxed{A}.$$

The reverse operation is DNEG (double-negation), which takes any subexpression A and changes it into $-(- (A))$.

9.8.4.6 Identities: *1, /1, ^1, and +1-1

These four simple identity operations 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 ' $A*B+A$ ' into ' $A*(B+1)$ ', 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:

$$(A*B) + \boxed{A} \quad \boxed{\boxed{*1}} \quad \Rightarrow \quad (A*B) + (A \boxed{*} 1)$$

$$\boxed{\text{ENTER}} \quad \boxed{\boxed{-}} \quad \boxed{\boxed{-}} \quad \Rightarrow \quad (A*B) \boxed{+} (A*1)$$

$$\boxed{\text{NEXT}} \quad \boxed{\boxed{-M}} \quad \Rightarrow \quad A*(B+1).$$

Another example:

$(A^B) * [A] \equiv [^1] \Rightarrow (A^B) * (A^1)$

$[ENTER] \equiv [-] \equiv [-] \Rightarrow (A^B) * (A^1)$

$[NEXT] [NEXT] \equiv [-M] \Rightarrow A^{(B+1)}.$

/1 and +1-1 are included for completeness, but they have little general purpose use.

9.8.4.7 Adding Fractions: AF

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

$(A/B) [+] (C/D) \equiv [AF] \Rightarrow ((A*D) + (B*C)) / (B*D)$

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*C+B)/C', for example.
- If the two subexpressions have the same denominator, use M→ rather than AF, so that the final denominator is the same as the original.

9.8.4.8 Logarithms: L* and L()

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 LN(A^B) into the product LN(A)*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 *; if you have the form B*LN(A), you will have to commute the arguments with ↔ before applying L().

9.8.4.9 Exponentials: E^ and E()

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 EXP(A*B) into (EXP(A))^B, for example. E() is the right-to-left operation. Either works with / in place of *, and with ALOG instead of EXP.

9.8.5 Subexpression Substitution

EXPAN, COLCT, and FORM let you rearrange an expression while preserving its value. It is also possible to substitute alternate objects or subexpressions into an expression, using OBSUB and EXSUB, respectively. These commands' counterparts, OBGET and

EXGET, reverse these processes, enabling you to extract an object or subexpression from an algebraic object.

The use of the four commands requires an index to “point” to an object or subexpression, analogous to the index that identifies a particular element in an array or list. The index of an *object* in an algebraic is just the position of the object in the expression or equation, counting objects from left-to-right (ignoring parentheses). For example, in the expression ‘A+B*C’, A is index 1, + is 2, B is 3, * is 4, and C is 5. The index of a *subexpression* is the index of the object that defines the subexpression: in the example, index 1 refers to A, 2 to A+B*C, 3 to B, 4 to B*C, and 5 to C.

To illustrate a simple form of substitution, suppose that you decide to replace the B in the expression ‘A+B*C’ with the number 5. You could store 5 in the variable B, then evaluate the expression, but that would also replace A and C with their values. Instead, use this sequence:

```
‘A+B*C’ 3 5 EXSUB → ‘A+5*C’
```

EXSUB requires the target algebraic object to be in level 3, the index in level 2, and the subexpression-to-substitute in level 1. The latter can be a number, a name, or another algebraic object. You can substitute an expression just as easily:

```
‘A+B*C’ 3 ‘SIN(D)’ EXSUB → ‘A+SIN(D)*C’.
```

EXSUB replaces the entire subexpression indicated by the index. For the index 3, B is the whole subexpression. But if you change the index to 4, which points to the *:

```
‘A+B*C’ 4 ‘SIN(D)’ EXSUB → ‘A+SIN(D)’.
```

The subexpression B*C is replaced by the substituted SIN(D). You can even replace the entire subexpression by indicating the “outermost” object, in this case the +:

```
‘A+B*C’ 2 ‘SIN(D)’ EXSUB → ‘SIN(D)’.
```

EXGET is the reverse of EXSUB, allowing you to extract the indexed subexpression from the original algebraic. For EXGET, the target algebraic must be in level 2, and the index in level 1:

```
‘A+B*C’ 3 EXGET → ‘B’.
```

```
‘A+B*C’ 4 EXGET → ‘B*C’.
```


By using OBSUB, you can replace a single object, including a function, with another. As with EXSUB, the target algebraic must be in level 3, the index in level 2, and the substitute object in level 1. However, the substitute object must be contained in a (one-element) list:

`'A+B*C' 4 { + } OBSUB` $\Rightarrow$ `'A+(B+C)'`.

By putting the object in a list, you can include built-in objects, which can't be entered directly onto the stack. OBSUB requires the substitute object to have the same number of arguments as the object that it is replacing, to preserve the structure of the target expression:

`'A+B*C' 4 { SIN } OBSUB` $\Rightarrow$ `Bad Argument Value`

Notice that for names and numbers, OBSUB and EXSUB are equivalent. However, it's generally easier to use EXSUB for substituting these object types since EXSUB doesn't require them to be in lists.

OBGET lets you pull one object from an algebraic, returning the object in a list:

`'A+B*C' 2 OBGET` $\Rightarrow$ `{ + }`.

Contrast this result with that of EXGET, which returns the entire expression:

`'A+B*C' 2 EXGET` $\Rightarrow$ `'A+B*C'`

A special form of EXGET is included in the first-level FORM menu. Pressing EXGET while the FORM display is active terminates the FORM operation, and returns the (entire) current FORM object to level 3, the index of the highlighted object to level 2, and the selected subexpression to level 1. This provides a simple method for making a substitution:

1. Execute FORM.
2. Use the FORM cursor to point to the subexpression you want to replace.
3. Press EXGET.
4. Drop the old subexpression from level 1.
5. Enter the new subexpression.
6. Press EXSUB.

9.9 Calculus

The HP-28's symbolic mathematical capabilities extend into the realm of the calculus. In particular, the derivative function ∂ can symbolically differentiate algebraic expressions containing almost any combination of HP-28 *analytic* functions (section 3.1). Based on the differentiation command, there is also TAYLR for computing Taylor's polynomials; based in turn on TAYLR, there is a limited symbolic integration facility. The integral command $\int$ can symbolically integrate expressions that are polynomials in the variable of integration. If the integrand is not a polynomial, $\int$ will invoke TAYLR to make a polynomial approximation to the integrand, then integrate the polynomial. This form of symbolic integration augments a general purpose numerical integrator.

9.9.1 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 ∂ in an algebraic expression. This choice of RPN or algebraic format also determines whether the differentiation is performed in a single operation (RPN) or one step at a time (algebraic).

To use ∂ as a stack command, you must 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 algebraic form of a derivative does not quite follow the normal HP-28 conversion from a stack command to a function, where $A \ B \ \partial$ should become ' $\partial(A,B)$ '. Instead, the algebraic form of ∂ is modeled after the standard written form $D_B A$ (derivative of A with respect to B): this derivative is expressed in HP-28 syntax as ' $\partial B(A)$ '. The parentheses are necessary to separate the expression A from the name B , and from the remainder of the algebraic object, if any. The symbol ∂ is used 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-28 rules for naming variables, ' dA/dB ' could also mean dA divided by dB , so this form is not used.

∂ also differs from other HP-28 functions in that it executes differently as a stack command than it does when it is part of an algebraic object. In the RPN case, the

derivative is repeatedly executed until the ∂ function is no longer present. When, however, an algebraic containing ∂ is evaluated, the derivative is only carried out with a single application of the so-called chain rule of differentiation. This rule states that

$$\frac{d}{dx} f(g(x)) = \frac{df}{dg} \frac{dg}{dx}.$$

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-28, 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 algebraic expression is consistent with the general flavor of HP-28 algebraic evaluation (section 9.1) in which the substitution of a variable's value for its name is carried out one "level" each time the expression is evaluated. This type of differentiation is quite useful as a teaching tool,

with which you can watch the successive application of the chain rule.

Two additional notes on HP-28 differentiation:

- The names of the variable of differentiation and of other variables in the expression are executed during the final stages of execution of ∂ . Therefore, if you want a completely symbolic derivative, you should purge the appropriate variables before executing ∂ .
- If you differentiate expressions containing trigonometric functions while the HP-28 is in degrees mode, factors of $\pi/180$ will appear in the result. This is perfectly sensible mathematically, but it may surprise you when you take a derivative without thinking about the angle mode.

9.9.2 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-.166666666667*X^3+8.33333333333E-3*X^5'
```

In this result, the even-degree terms are absent, because they are all proportional to

$\sin^n 0 = 0$. 0.166666666667 and 8.33333333333E-3 are floating-point approximations to $1/3!$ and $1/5!$, respectively.

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 .

TAYLRX0					
Taylor's Polynomial at x_0					
level 4	level 3	level 2	level 1		level 1
'expression'	'name'	$x_0^\dagger$	degree	x	'polynomial'
<pre> << 31 SF 1 CF 3 PICK IFERR RCL THEN 1 SF END → x x0 d xv << x SHOW 'XPRIM' DUP x0 + x STO d TAYLR xv x0 - 'XPRIM' STO EVAL xv IF 1 FC?C THEN x STO ELSE PURGE END 'XPRIM' PURGE >> >> </pre>				Activate LAST. Initialize flag 1. Get the expansion variable name x. Set flag 1 if the variable has no value. Save the name, x_0, degree, and the variable's value. Make all instances of x explicit. Substitute $x' + x_0$ for x. Make the expansion. Substitute $x - x_0$ for x'. Did x have a value? Then restore it. Otherwise, purge x.	

$^\dagger x_0$ may be a name, expression, or number.

- TAYLRX0 uses and purges global variable XPRIM.
- TAYLRX0 clears flag 1 and sets flag 31.

■ *Example.* Compute the 3rd degree Taylor's polynomial for $\sin x$ at the point $x = \frac{\pi}{2}$.

```
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 $\sin x = \cos(x - \pi/2)$.

You can only apply TAYLR meaningfully to functions for which the function itself and its derivatives up to the Nth-order are defined at $x = 0$. For example, you can not compute a polynomial for $x^{1/2}$, since its first derivative is proportional to $x^{-1/2}$ and thus is infinite at $x = 0$.

In addition to its nominal use to compute Taylor's polynomials, TAYLR also is a convenient means of simplifying polynomial expressions. This application is discussed in section 9.8.3.

9.9.3 Integration

Unfortunately, there is no analog of the chain rule of differentiation that allows you to compute the integral of an arbitrary expression once you know the antiderivatives of the functions in the expression. The problem of general symbolic integration is one of pattern matching, and given that HP-28 algebraic objects are not constrained into any special structure, this task is beyond the memory resources of the calculator. Therefore, the integration of arbitrary expressions is limited to numerical methods.

Nevertheless, the HP-28 can compute an approximate symbolic integral of an expression. For expressions which are sufficiently well-behaved, the HP-28 will compute a MacLaurin polynomial in the variable of integration, then integrate the resulting polynomial. You can specify the degree of the polynomial, which determines the accuracy of the integral. The higher the degree, the more accurate the approximation--but also the longer the time and the greater amount of free memory required to compute the polynomial. As mentioned in section 9.9.2, "well-behaved" in this context means that you can compute a meaningful polynomial--the expression must be differentiable at the origin. Of course, for an integrand that is already a polynomial, these conditions are met automatically, and the integral is exact (as long as you specify a degree equal to or greater than the degree of the polynomial).

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 being able to determine a reasonable view of its behavior over some

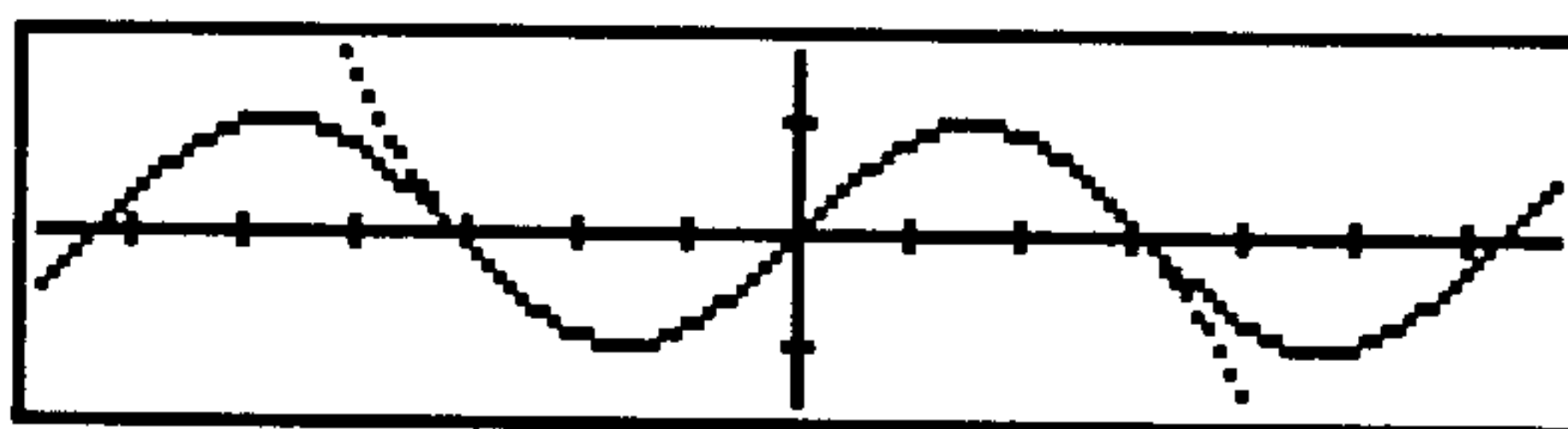
region of interest. For this purpose, the HP-28 approximate symbolic integral is an improvement over numerical integrals, since it is much faster to recompute its value as you vary some parameter in the expression (for example, if you plot the integral) than to recompute the integral numerically at many points.

To illustrate the idea of an approximate integral valid in a region, we will compare an HP-28 symbolic integral of $\cos(x)$ with the known form of the integral, $\sin(x)$ (here we are considering an indefinite integral, and ignoring the constant of integration). To obtain the HP-28 integral, execute

RAD 'COS(X)' 'X' 6 $\int$

where the 6 indicates a 6th degree polynomial. This also returns a 6th degree polynomial in X (the 7th order term is zero). To compare this with $\sin(x)$, execute

'SIN(X)' = STEQ DRAW



In the plot you can see that in the region $-\pi \leq X \leq \pi$, the approximation is good enough that the two curves fall on the same pixels and are indistinguishable. Outside of this region, the curve representing the series approximation diverges rapidly away from the ideal sine curve.

Now press **ON** to clear the screen, and **SOLV** **SOLVR** to activate the Solver menu. Store .5 in X, then press

LEFT=		.479425533235
RT=		.479425538604
-		-.000000005369

Here you have computed numerical values of the integral from the approximation (the left side) and the true value (right side), showing that the approximation is accurate to about one part in 10^{-8} . It takes the HP-28S about 4.2 seconds to compute the symbolic integral once, then about 0.08 seconds to evaluate it at a specific point. You can compare these times with that needed for computing the numerical integral

$$\int_0^{.5} dx \cos(x)$$

to an accuracy of .00000001, about 6.5 seconds on the HP-28S. The first evaluation of the integral is not much different for the two methods, but a *second* evaluation at a different point takes only another 0.08 seconds using the symbolic integral compared with the full 6.5 seconds of a numerical integral. The speed advantage of the symbolic method can be quite dramatic when you're attempting to use DRAW or the Solver with an expression that contains an integral, since either operation may involve evaluating the expression hundreds of times.

The mechanics of computing an integral on the HP-28 are straightforward. The symbolic case uses the same stack set-up as TAYLR, on which it is based. You need to specify the integrand, the variable of integration, and a number to specify the degree of the polynomial approximation. In standard HP-28 fashion, these three items go on the stack:

3:	<i>integrand</i>	(an algebraic)
2:	<i>variable name</i>	(a name)
1:	<i>degree</i>	(a real integer)

$\int$ takes these three arguments from the stack, and returns the integrated expression to level 1.

To obtain a numerical definite integral, you must set up the stack in a similar manner. The integrand again can be an arbitrary algebraic object, but for the numerical case you can also use a program that is equivalent to an algebraic expression (takes no arguments, and returns one number). The integration variable is represented by a list containing the variable name (first element), and the lower (second) and upper (third) limits of integration. A real number in level 1 again specifies the accuracy of the integral, but instead of being the degree of a Taylor's polynomial, this number is an *accuracy factor* that specifies the maximum fractional error you will accept in the computation of the integral. Typically, this accuracy factor is a negative power of 10 (1×10^{-n}), where $1 \leq n \leq 12$. Larger values of n produce more accurate integrals, but also make the computation time longer. You should always specify the smallest n that is consistent with the accuracy you need for a particular problem.

For a numerical integral, the stack set-up is like this:

3:	<i>integrand</i>	(a procedure)
2:	{ <i>name upper lower</i> }	(a list)
1:	<i>accuracy factor</i>	(a real number)

[On the HP-28S, the *upper* and *lower* limits can be represented in the level 2 list by real numbers, or by names, programs, or algebraics that evaluate ($\rightarrow$ NUM) to real numbers.]

For example, to compute the integral mentioned above,

$$\int_0^{.5} dx \cos(x),$$

enter (in radians mode)

'COS(X)' { X 0 .5 } .00000001 $\int$ $\Rightarrow$.479425538604 4.79427988979E-9

In the numerical case, $\int$ returns *two* real number results. Level 2 contains the computed value of the integral (.479425538604 in the example); level 1 has a number representing an upper bound on the (absolute value) of the error in the integral (4.79427988979E-9). The error value should be roughly the product of the integral value in level 2 times the original accuracy factor. If this is not true, the integral value is suspect. In particular, if a value -1 is returned for the error, it indicates that the integration failed to converge to a reliable answer.

You might wonder why the stack arrangement for numerical integration requires you to combine the variable name and the integration limits into a list rather than just using five separate arguments. The answer lies in the argument count uniformity that the HP-28 imposes on its commands. Most commands allow a variety of argument types, but the number of arguments for any command is always the same (you can check this by using the USE option in the command CATALOG; each entry for any single command always shows the same number of arguments). This uniformity makes possible a very compact and fast branching process whereby a command can select the execution logic suitable for the particular argument combination current on the stack. In the case of $\int$, the type of argument in level 2 signals the method of integration: a name by itself indicates symbolic integration, and a list indicates numerical integration. If the two methods were to use different numbers of stack arguments, each would have had to be represented by a different command name.

If you have trouble remembering how to set up the stack for $\int$, you can write a program that allows you to enter the arguments as individual entries in some mnemonic order, such as the order in which you might read a written integral. For example, the integral

$\int_a^b f(x)dx$ can be read as “the integral from a to b of $f(x)$ with respect to x .” The program IGL lets you enter the arguments in the same order as they are named in the quoted clause.

IGL					<i>Integral</i>	
level 4	level 3	level 2	level 1		level 1	
$a^\dagger$	$b^\dagger$	$f(x)^\dagger$	'x'		integral	
<pre><< 4 ROLL ->NUM 4 ROLL ->NUM 3 ->LIST ACC ∫ IF 0 < THEN "Bad Integral" 1 DISP END >></pre>					$\ddagger$ Set up the arguments. Compute the integral. Check the error. If it's negative, display a message.	

† May be a name, number, or procedure.

$\ddagger$ The $\rightarrow$ NUM's are not necessary on the HP-28S.

IGL requires that you keep an accuracy factor in a global variable ACC. (This is so you don't have to keep reentering the accuracy for different problems if the same accuracy is appropriate.) Also, it returns only the integral value; the error is discarded. If the error is negative, IGL warns you with a Bad Integral message. You can enter the integration limits as names, algebraics or programs that evaluate to numbers.

■ *Example.* Compute $\int_0^{2\pi} \cos^2(x)dx$, with an accuracy of 10^{-5} .

RAD 1E-5 'ACC' STO 0 '2*π' 'COS(X)^2' 'X' IGL  3.1415927

9.9.3.1 Integration with an Implicit Variable

The method of numerical integration described in the preceding paragraphs is called *explicit variable integration*, since the variable of integration appears explicitly by name in the integrand procedure and in the level 2 list. This form of integration is straightforward, since it corresponds more or less to the way you normally write integrals as functions of named variables. However, the HP-28 does offer a second method of numerical integration, in which the variable of integration is not explicitly named, but instead is represented by a series of values in stack level 1. This method is called *implicit variable integration*; its principal virtue is that it is faster than the explicit method. It saves time by not having to find the value of the integration variable in USER memory every time

the integrand is evaluated; the value is always available on the stack. (This method is similar to that used by the HP-15C integral function and by the integration program in the HP-41 Advantage Pac.)

To use implicit variable integration, you set up another version of the three $\int$ arguments:

3:	<i>integrand</i>	(a program)
2:	{ <i>upper lower</i> }	(a list)
1:	<i>accuracy factor</i>	(a real number)

This time, the list in level 2 contains only two real numbers, representing the lower and upper limits of integration. The level 3 integrand must be a program (algebraics can't take arguments from the stack). The program must have the logical form

$\ll \rightarrow x \text{ 'f(x)' } \gg$

where 'f(x)' represents the integrand in algebraic form. That is, the program should take one number from the stack as the current value of the variable of integration, then return to the stack the corresponding value of the integrand. You shouldn't actually write the program as above, since the use of a local variable defeats the intent of not naming the variable. Instead, you should keep the input number as a stack entry.

To recompute the example $\int_0^{.5} dx \cos(x)$ using the explicit method:

$\ll \text{COS} \gg \{ 0 \ .5 \} .00000001 \int$

$\llcorner .479425538604 \quad 4.79427988979E-9$

The program $\ll \text{COS} \gg$ follows the implicit variable prescription, taking a number from the stack and returning the value of the integrand, which is the cosine of that number. On the HP-28S, this method takes 5.2 seconds to return exactly the same results as the explicit variable method, which takes 6.5 seconds. The time difference is not dramatic in this case, but a 20% time savings for an integral requiring many minutes can be valuable.

Actually, you can still obtain an intermediate time savings over explicit variable integration by using the implicit variable method even when the integrand program does save the variable value in a local variable. It is generally faster for the HP-28 to find the value of a local variable than that of a global variable. For example, executing


```
<< → x 'COS(x)' >> { 0 .5 } .00000001 ∫  
⌞ .479425538604 4.79427988979E-9
```

takes 5.9 seconds on the HP-28S.

9.9.4 Double Integrals

A double integral is an integral in which the integrand itself is an integral, the limits of which may be functions of the original variable of integration. This is a special case of *multiple* integration, where the integrals are nested two or more deep. We will demonstrate a method of double integration; you can extrapolate the process to multiple integration by applying the same principles repeatedly.

Since $\int$ is not a function and hence can not appear in algebraic objects, the integrand in a double integral must be expressed as a program object. In section 9.9.3, we presented the program IGL that streamlines the entry of arguments for a single integral. The program IGL2 uses a similar approach for double integrals, and calls IGL to perform the actual integration. These two programs make a good illustration of HP-28 *structured programming* (section 10.1.3), in which complicated programs are built up from simpler ones.

IGL2								Double Integral	
level 7	6	5	4	3	2	1		level 1	
a†	b†	'x'	c†	d†	f(x)†	'y'	⌞	integral	
<pre><< → c d f y << << c d f y IGL >> SWAP IGL >> >></pre>					Store the arguments of the "inner" integral.				
					Program to compute $\int_a^b f dy$				
					Integral over x.				

†May be a name, number, or procedure.

IGL2 takes stack arguments in the order

$a \ b \ x \ c \ d \ f \ y.$

x and y must be names, and a , b , c , d , and f can be numbers, names or procedures. These objects and their order as stack arguments are derived from the general form of a double integral:

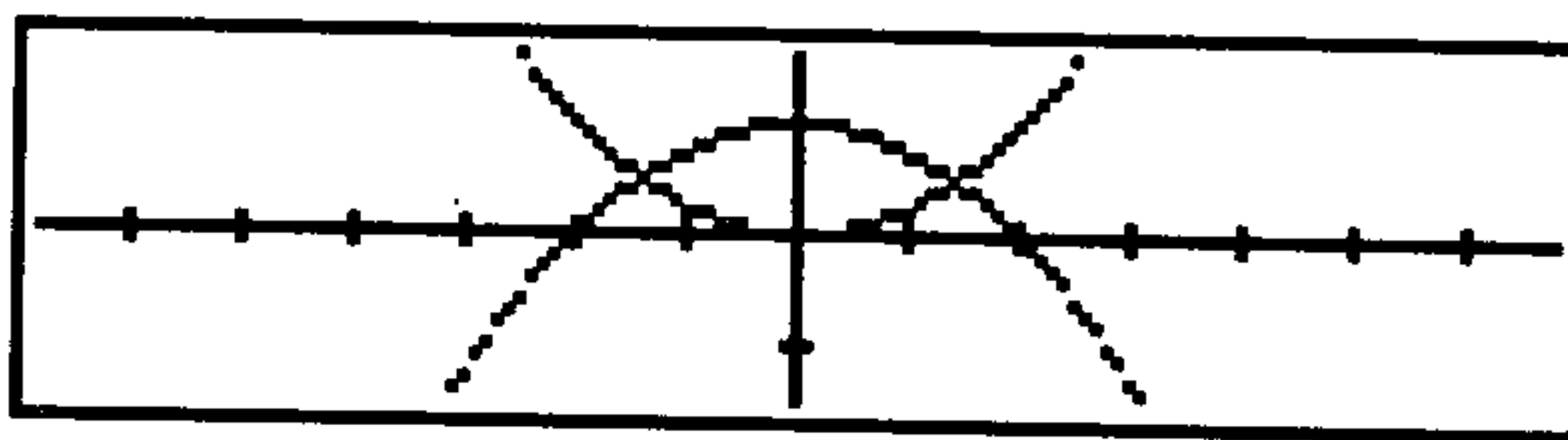
$$\int_a^b dx \int_c^d f(x,y) dy,$$

which you can read as “the integral from a to b with respect to x , of the integral from c to d of $f(x,y)$ with respect to y .” With respect to the integration variables, f can be a function of x and or y ; c , and d can be functions of x . Any other variables that appear in the arguments must evaluate ($\rightarrow$ NUM) to real numbers. Like IGL, IGL2 assumes that you have stored an accuracy factor in the variable ACC.

■ *Example.* Find the area of a region bounded by the parabolas $y = x^2$ and $y = 4 - x^2$.

To help in setting up the problem, start by plotting the two curves:

```
'PPAR' PURGE 4 *H 'X^2=4-X^2' STEQ DRAW
```



To find the points where the curves intersect, execute

```
RCEQ 'X^2' + COLCT 'X' ISOL ⌘ 's1*1.41421356237'
```

from which you can observe that the intersections are at $x = \pm\sqrt{2}$. The area is therefore given by the integral

$$\int_{-\sqrt{2}}^{\sqrt{2}} dx \int_{x^2}^{4-x^2} dy$$

In this case, the integrand is just a constant $f(x,y) = 1$. Choosing an accuracy factor .0001 by executing .0001 'ACC' STO, you obtain the integral value by executing

```
'-√2' '√2' 'X' 'X^2' '4-X^2' 1 'Y' IGL2,
```


which returns 7.54247233265. This differs from the exact answer $\frac{16}{3}\sqrt{2}$ only in the twelfth decimal place.

10. Programming

In preceding sections, we have reviewed the use of the Solver, symbolic math operations, and user-defined functions, as specialized programming methods that replace much of the programming done on conventional calculators. Each of these formula-oriented methods, however, has its limitations, and for many types of problems there is no substitute for ordinary programming. "Ordinary programming" on the HP-28 means the creation of *program objects*, which have no limits on the number or type of arguments and results, and can use all of the various resources of the calculator.

Creating a program object consists of entering a sequence of objects that are to be automatically executed together in order, enclosing the sequence in << >> delimiters to prevent immediate execution. When you name a program object by storing it in a user variable, you effectively extend the calculator's command set. You can use the variable name just as you would a built-in command. Imagine, for example, that you have created two program objects named DOTHIS and DOTTHAT. Then if you want to create a program that performs both of the tasks done by DOTHIS and DOTTHAT, you just enter << DOTHIS DOTTHAT >>, perhaps naming it DOBOTH. This process is unlimited--you can use DOBOTH as an element of another program. DOTHIS and DOTTHAT themselves may be combinations of other program names. As a matter of fact, the HP-28 commands that you use in your programs are themselves programs written the same way, stored in the calculator's permanent memory (ROM).

We have been using the term *sequence* to mean a series of objects that are executed in order. Remember (see section 3.2.1) that *objects* include commands--built-in program objects--as well as data, name, and procedure objects that you create. However, we are about to introduce the concept of a *program structure*, which uses command line and program entries that are not objects, so we need to extend our definition of *sequence*.

The non-object "entries" are *program structure words*, such as FOR, DO, →, END, etc. These are not objects, because you can't put them on the stack or execute them individually. You can only use them in certain specific combinations, like FOR...NEXT, or IF...THEN...END. A complete combination, including the objects between the program structure words, is called a *program structure*.

The more complete definition of *sequence*, then, is any series of objects *and* program structures that can "stand alone," and can constitute a program if the series is surrounded by << >> delimiters. A sequence can be an entire program, or part of a program. For example, in

```
<< 1 2 IF A THEN B C END D >>
```


1 2 is a sequence, B C is a sequence, and 1 2 IF A THEN B C END D is a sequence. IF, IF A, and IF A THEN are not sequences, because the program structure is not complete--you can not enter these by themselves without obtaining a Syntax Error message.

10.1 Program Basics

The basic structure of an HP-28 program is very simple:

`<< program body >>.`

The `<<` and `>>` are the program object delimiters that serve to identify this object as a program. *Program body* is the sequence of objects and program structures that make up the logical and computational definition of the program.

10.1.1 The `<<` `>>` Delimiters

The `<<` and `>>` that surround HP-28 programs serve a dual purpose. First, they are the delimiters that identify an object as a program. When you enter a program into the command line, the `<<` tells the HP-28 to create a program object from all of the objects, commands, names, etc., that follow, up to the next matching `>>`. Then, when the HP-28 displays a program object after it has been created, the `<<` and `>>` identify the object to you as a program.

The second role of these delimiters is to serve as logical “quotes” (see section 3.8) that postpone execution of a program sequence. When `<<` is encountered in program or command line execution, it is interpreted by the HP-28 to mean “put the following program object on the stack.” This behavior of `<<` allows you to include programs within other programs:

`<< object >> EVAL`

executes *object*, but

`<< << object >> >> EVAL`

leaves the program `<< object >>` on the stack. For every `<<`, there is always a `>>`. The `>>` ends the definition of the program started by the preceding `<<`.

The `<<` and `>>` keys are the closest analog the HP-28 has to the more traditional program mode keys you find on other calculators (`PRGM` on the HP-41). On the HP-28, instead of pressing `PRGM` to start program entry, you press `<<`. This activates alpha-

entry mode (section 3.11.1), which we have also called “program mode” since command names accumulate in the command line instead of executing. Then, after you have entered the program objects, you press **>>** to terminate program entry. You can even think of **<<** and **>>** as *programmable* program keys, since you can include those delimiters in programs.

10.1.2 The Program Body

The “body” of an HP-28 program, that is, everything between the **<<** and the **>>**, can consist of any combination of objects and program structures:

- Data objects;
- Quoted names and procedures, which go on the stack like data;
- Commands--RPN commands and functions;
- Unquoted names--which act like user-defined commands;
- Program structures--loops, conditionals, and local variable structures.

In general, when a program is executed, all of the items from the above list that constitute the program body are executed sequentially. The nominal order of execution is start-to-finish, or “left-to-right” in the command line order in which the program was entered originally. Within a program structure, there may be repetitive loops or conditional jumps. Of course, there’s nothing remarkable about this program flow--any programming language exhibits similar orderly execution.

The simplest programs are those which contain no program structures. Such programs only contain objects to be executed one after the other, starting with the first object after the **<<**, and ending with the last object just before the **>>**. Simple programs are very easy to create. All you do is

1. Press the **<<** key;
2. Press the keys for, or spell out, the objects you want the program to execute, in the same order used when you perform the calculation by pressing keys; then
3. End the program entry by pressing **ENTER**, or press **>>** to end the program and continue with additional command line entries.
4. To name a program, enter a name (quoted) and press **STO**. You can consider the resulting variable as a named program.

To “run” a program, you execute the program’s name, either by pressing the appropriate **USER** or **HP-28S CUSTOM** menu key, or by typing the name into the command line and pressing **ENTER**. If the program itself is in level 1, you can run it by executing

EVAL .**Examples:**

1. `<< 1 2 3 >> 'P123' STO` creates a program named P123 that enters the numbers 1, 2, and 3 onto the stack.
2. `<< 2 / SIN >> 'HSIN' STO` creates a program named HSIN, that returns the sine of 1/2 times the number in level 1.
3. `<< + + SQ >> 'SUMSQ' STO` creates SUMSQ, which adds three numbers from the stack and squares the result.

You can alter the basic start-to-finish execution flow of programs by adding program structures that define branches and loops. *Branches* are forward jumps in a program, that cause program sequences to be skipped. *Loops* contain backward jumps, which cause program sequences to be repeated one or more times. These structures are described later in this chapter.

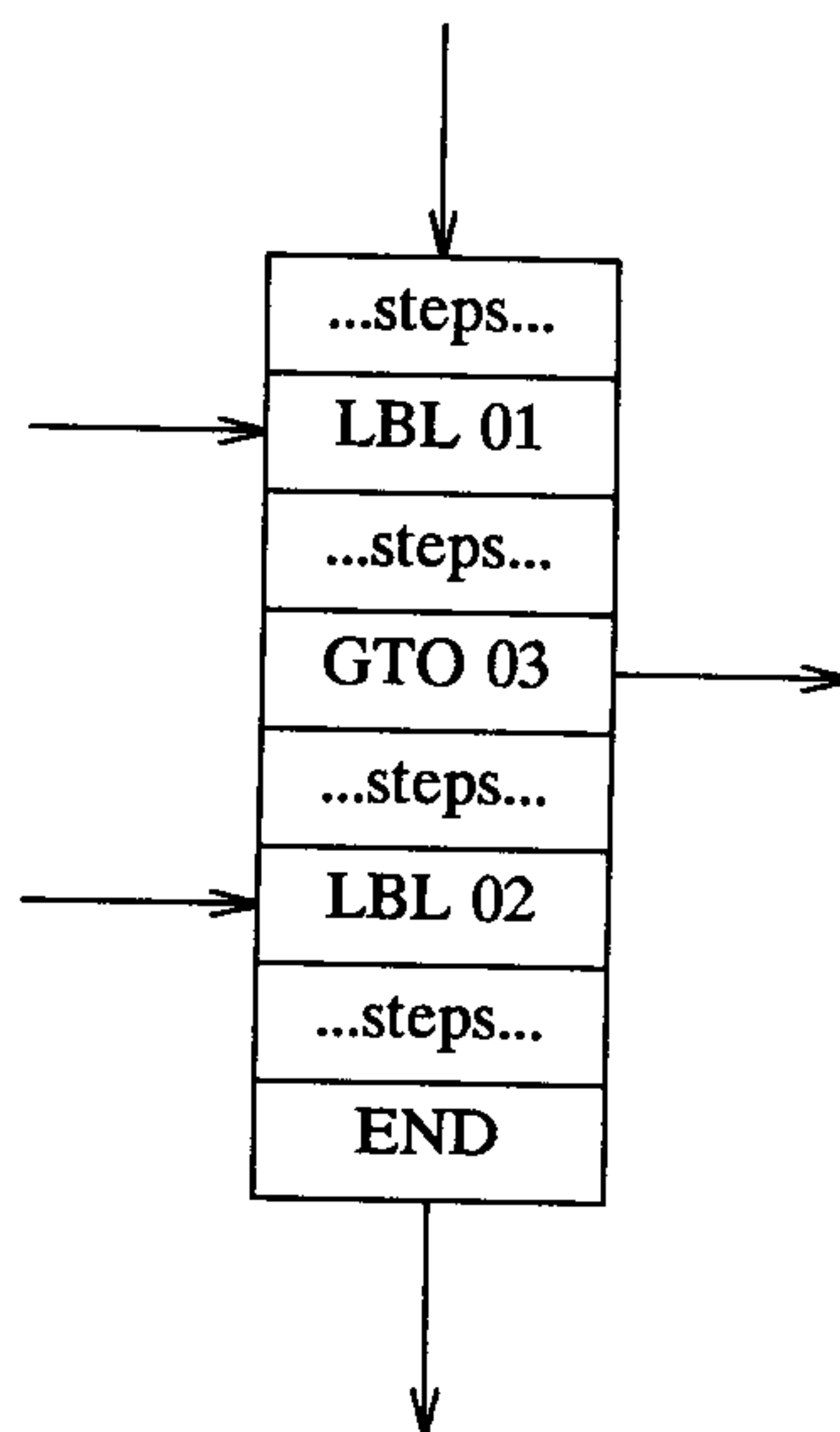
Notice that HP-28 ENTER never appears in a program, unlike HP-41 ENTER[†]. As we discussed in section 3.11, ENTER means “execute the command line,” which has no meaning in a program. When you enter consecutive numbers into a program, just use a space or the non-radix character (comma or period) to separate them. For other types of objects, their delimiter characters serve to separate the objects--no space is necessary before, after, or between delimiters.

10.1.3 Structured Programming

A property of HP-28 programs that is common among many computer languages, but may be unfamiliar to HP-41 and other calculator language programmers, is their well-determined “entrance” and “exit.” That is, in any program there is only one point--the start--where execution can begin. Similarly, there is only one exit, or point at which a program completes execution. A diagram to represent the execution flow in and out of an HP-28 program is very simple:



Contrast this diagram with one that illustrates a possible program flow in an HP-41 program:



There is no limit on the number of entrances and exits in an HP-41 program. The principal program constructs that make this possible are labels and GTO commands. A GTO (go to) is an unconditional jump, with no return, to a label (or line number in some calculators and in BASIC). Using labels and GTO's, program execution can jump around from program to program, in and out of portions of programs, or round and round within a single program. At first glance (and more, if you're used to programming this way), this capability seems like an advantage. You may wonder why the HP-28 does not provide the same capability.

The answer is that the HP-28 is designed for *structured programming*. Structured programming consists of writing small programs as building blocks, or modules, from which bigger programs are assembled as series of subroutine executions. A *subroutine* is a program that is executed, or *called*, from within another program, and which returns to the original calling program when it is finished. Bigger programs themselves may become subroutines for even bigger programs, and so on. Each program, at every level, has a single entrance and exit; there is no jumping in and out of programs at intermediate points. Structured programming has the following advantages:

- Programs are easy to write. Each program can be designed to fulfill a single task, and can thus consist of relatively few steps. If a program gets too long, you just divide it into smaller programs.

- Programs are easy to decipher. By choosing meaningful names for subprograms, you can read a program almost as text. For example, a program might look like this:

```
<< GETINPUT  DOMATH
    IF  BIG
    THEN IGNORE
    ELSE SAVE
    END
>>.
```

It is easy to understand what this program does. It gets input (GETINPUT), then does some calculations (DOMATH) on that input. Next, it checks a result to see if it's too large (IF BIG); if so, it discards the result (THEN IGNORE), otherwise saves it (ELSE SAVE). At this level, you can see the overall structure of the program. To see more detail, you can examine the individual subroutines. For example, BIG must be a program that tests the results returned by DOMATH, and returns a *true* flag (see section 10.3) if the results are too big according to some criterion. BIG might be something like this:

```
<< DUP2  +  LIMIT  >  >>.
```

This program makes copies of two numbers in levels 1 and 2, then adds them and tests to see if the sum is greater than the value of LIMIT (which might be a number, or another calculation to perform, etc.).

- Programs are easy to alter. In the above example, you can completely change the internal definition of BIG, without worrying about the main program. All you have to do is ensure that BIG works the same from an external point of view--it must take the right number of objects from the stack, and return the right number, etc. Similarly, you can change the value of LIMIT from a specific number to a program that computes a result, without any change in the design of BIG.

In a programming language that permits GTO's into the middle of a program, any modification of a program must ensure that the correct entry conditions are met at any point at which execution can start. This is especially difficult to manage in languages like BASIC, where a GTO can jump to any line in a program, with no label or other indication to remind the programmer that execution may start at that line.

- Programs can be written without any regard to the internal behavior of programs that call them, or programs that they may call. All that matters about a program is its input and output, not the steps that it uses in its execution.

The last point is a key concept in HP-28 structured programming. A program is defined externally only in terms of its input and output:

1. The number and type of objects it takes from the stack;
2. The number and type of objects it returns to the stack;
3. The variables that it uses;
4. Flags that are tested or changed.

From the point of view of one program calling another as a subroutine, the first program doesn't have to care *at all* about how many stack levels or additional subroutine returns are needed by the subroutine. It just has to be sure to provide the correct inputs for the subroutine, and know where to find the results returned by the subroutine (usually on the stack). The calling program also can depend on having program execution return to it after the subroutine is finished, no matter how many other subroutines are called by the subroutine.

10.1.4 Comparing HP-28 and HP-41 Programs

HP-41 programs have this general form:

```
01 LBL A
02 program line
03 program line
...
nn RTN (or END)
```

Strictly speaking, HP-41 programs don't have to start with a label, but it is usually most convenient to start execution at the beginning of a program, by means of a label there. The programs always finish with a RTN or an END. A single program may contain multiple RTN's--different programs are separated by END's.

If you match up the parts of HP-41 and HP-28 programs, you may observe that:

- The HP-41 line 01, and the label that marks the start of a program, are replaced in the HP-28 by the << delimiter.
- The program steps or lines that make up an HP-41 program body are replaced by the objects that define the HP-28 program.
- The END, or last line of an HP-41 program, which acts as a subroutine return when the program is called as a subroutine, is replaced by the HP-28 >> delimiter.

HP-41 program lines or steps are always numbered. The line numbers help show the program flow unambiguously, and are useful in moving a program counter to specific points in a program for editing or single-stepping. The line numbers are artificial in the sense that they are not stored as part of a program (they are created as part of the program mode display), and have nothing to do with program execution while it is running. All program branching is accomplished by GTO's or XEQ's (GSB's) to labels that are explicit program steps.

HP-28 programs have no line numbers. Whether this is an advantage or a disadvantage is a matter of taste. Since line numbers have no purpose during execution, showing them as part of a program can be considered as a superfluous complication. On the other hand, in a large program, line numbers can help you keep track of where you are looking in a program.

To some extent, you can write HP-41 programs in structured form, if you can resist the temptation to use GTO's. Of course, you can't eliminate GTO's entirely, because the HP-41 doesn't provide any program structures in the HP-28 sense. But you can preserve a structured form by avoiding intertwined branches and loops. For example, to treat the program sequence

```
01 LBL 01
02
...
98 ISG 00
99 GTO 01
```

as a "structure," you must make sure that there are no labels between the LBL 01 and the ISG 00, so that program flow can not jump into the middle of the sequence.

The fixed size stacks available on the HP-41 makes truly structured programming in the style of the HP-28 rather difficult. The HP-41 has only a 6-level subroutine return stack, so that any time you write one program that calls another, you must verify that the subroutine does not itself call other routines, and so on, to such an extent that the return stack overflows and execution never returns to the original program. The four-register data stack produces a similar limitation: a program can't arbitrarily leave data on the stack when calling a subroutine, in case the subroutine needs so much of the stack that the calling program's data gets pushed off the stack. After writing a program, if you later decide to modify one of its subroutines, you may also have to change the calling program if the new version of the subroutine uses an additional stack register.

Structured programming is not just a matter of programmer style in the HP-28--you

have no option. The HP-28 won't let you write an unstructured program. There is no GTO command, and all branching and looping is accomplished by means of well-defined structures. This requirement is perhaps the single thing that HP-41 programmers will have the most trouble getting used to on the HP-28.

Table 10.1 matches various programming concepts in the HP-41 with their analogs in HP-28 programs. All of these topics are discussed in subsequent sections.

Table 10.1. HP-41 and HP-28 Programming Analogs

HP-41	HP-28
Program mode	Alpha-entry mode, command line
Program file	Program object
Global label	Program name
Local label	None
Line number	None
GTO <i>label</i>	None
Subroutines	Named programs
XEQ <i>label</i>	Execute by name or EVAL
RTN	>>
END	>>
ISG, DSE	DO...UNTIL...END
	WHILE...REPEAT...END
	FOR...NEXT
	FOR...STEP
	START...NEXT
	START...STEP
Flag 25 error handling	IFERR...THEN...ELSE...END
Test and Skip	IF...THEN...ELSE...END
STOP	HALT
R/S	CONT

10.2 Program Structures

A simple program consisting of a sequence of objects can be broken into two or more programs at any point in the sequence. For example, the program

<< 5 * 6 + 10 - >>

is equivalent to the two programs


```
<< 5 * >> << 6 + 10 - >>
```

executed consecutively.

A *program structure* is a program segment that can not be broken into stand-alone sections. A user-defined function (Chapter 8) is an example of a program structure; for example, this program

```
<< → x '2*x+3' >>
```

can *not* be divided like this:

```
<< → x >> << '2*x+3' >>.
```

The first part would give a Syntax Error message if you entered it. Similarly, you can't break

```
<< 1 5 FOR n n SQ NEXT >>
```

into

```
<< 1 5 FOR >> << n n SQ NEXT >>.
```

The FOR and the NEXT must be in the same program.

Program structures are defined by *program structure words*. These words are special command line words that do *not* represent objects, but cause other objects to be combined into structures. The structure words always appear in specific combinations that define complete structures. Table 10.2 lists all of the HP-28 program structures and their uses.

Table 10.2. HP-28 Program Structures

Structure	Type	Typical Use
IF...THEN...ELSE...END	Conditional	Program Decisions and Cases.
START...NEXT/STEP	Definite Loop	Execute a sequence a specified number of times.
FOR <i>index</i> ... NEXT/STEP	Indexed Definite Loop	Execute a sequence once for each value of an index.

DO...UNTIL...END	Indefinite Loop	Repeat a sequence until a condition is satisfied.
WHILE...REPEAT...END	Indefinite Loop	While a condition is satisfied, repeat a sequence.
→ ...names... procedure	Local Variable Structure	User-defined functions. Creating local variables.
IFERR...THEN...ELSE...END	Error trap	Handling expected and unexpected command errors.

Before studying the various program structures, we need to describe HP-28 test commands and flags, which are key concepts in understanding the execution of program structures.

10.3 Tests and Flags

A calculator program “asks a question” by executing a *test* command. A test command is any command that in effect returns “true” or “false” as a result, which then can be used to choose a particular program branch to execute. The HP-28 differs from the HP-41 in that tests return stack results called *flags*, whereas HP-41 test commands include immediate branching based on the test result.

In HP-28 terminology, the word *flag* has a dual meaning. The first meaning is the traditional one inherited from the HP-41, where a flag is one of a group of numbered memory locations that are used to store logical *true* or *false* values. A “memory location” in this context is just a binary bit; if the bit is 1, the flag is *true*; if it is 0, the flag is *false*. A *user flag* is one that can be set (made *true*), cleared (made *false*), or tested by by means of commands. A *system flag* is one reserved for use by the calculator operating system, and which can only be tested by the user, not set or cleared. The HP-41 has 56 flags, of which those numbered above 29 are system flags. The HP-28 has 64 flags; all are user flags. In both the HP-41 and the HP-28, some of the flags represent modes, such as the angle mode and the beeper enable/disable. Changing the state of one of these flags changes the corresponding calculator mode, and vice-versa.

The HP-28 introduces a second meaning to *flag* that has no equivalent in the HP-41. A flag can be any real number, so that flags can be represented on the stack and used as arguments or returned as results by commands. The HP-28 conventions for real number values used as flags are:

- As arguments to commands, 0 means *false*; any non-zero real number means *true*.
- When a command returns a flag result, 0 again means *false*; the value 1 means *true*.

With these ideas in mind, we can make the following definitions:

<i>Test:</i>	A command that returns a flag to the stack. Examples: SAME, =, FS?
<i>Logical operator:</i>	A function that makes a logical combination of two flags (AND, OR, XOR), or inverts a flag (NOT), and returns a new flag.
<i>Conditional:</i>	A program structure that includes a structure word that uses a flag as an argument, and causes a program branch according to the flag value. The conditionals are IF...THEN...(ELSE...)...END, DO...UNTIL...END, and WHILE...REPEAT...END.

In the HP-41, all test commands combine a test and a branching operation. If the test is *true*, one choice of branch is made; if *false*, another choice is made. For example, when a test such as FS? (flag set?) or X=Y? is *true*, the program line immediately following the test is executed. If the test is *false*, that next line is skipped.

In the HP-28, a test and a corresponding conditional branch are separate operations. To permit this separation, a test command returns its result in the form of a (real-number) flag on the stack, which can then be manipulated like any other stack object. Consider a typical test command, >. > compares real numbers in levels 1 and 2: if the number in level 2 is greater than that in level 1, > returns 1 (*true*); it returns 0 (*false*) if the level 2 number is equal or smaller. For example, to compare the values of X and Y in a program, you use the sequence

X Y >.

This returns 1 (*true*) if X is greater than Y, or 0 (*false*) otherwise.

In a conditional structure, one particular structure word actually makes the branch decision, taking a flag from the stack for this purpose:

- the THEN in IF...THEN...(ELSE...) END (section 10.4).
- the END in DO...UNTIL...END (section 10.5.2.1).
- the REPEAT in WHILE...REPEAT...END (section 10.5.2.2).

But note that you can include any number of intervening objects and commands between the point at which the flag is put on the stack, and the structure word that uses

the flag for a branch decision. This separation of tests and decisions makes possible the use of logical operators to combine flags. For example, the logical operator AND takes two flags from the stack and returns a true flag if both of the original flags are *true*, and a false flag otherwise. The sequence

X Y > Y Z > AND

returns 1 only if X is greater than Y, *and* Y is greater than Z. Furthermore, since the logical operators and most tests (except SAME) are functions, you can rewrite the above sequence in a more legible manner:

'X>Y AND Y>Z' →NUM.

The →NUM converts the algebraic expression into a real number suitable for use as a flag.

Suppose you want to write a program that returns the sum of two numbers if they are both greater than 1, and otherwise returns the difference. In HP-41 language, the program might look like this:

01 LBL "SUMDIFF"	
02 1	
03 X<=Y?	Is the first number ≤ 1?
04 GTO 01	If so, go to LBL 01.
05 RDN	If not, check the other number.
06 X<>Y	
07 R↑	
08 X<=Y?	Is the second number ≤ 1?
09 GTO 02	If so, go to LBL 01.
10 RDN	Drop the 1.
11 +	Add the two numbers.
12 RTN	
13 LBL 02	The second number is ≤ 1.
14 RDN	
15 X<>Y	Restore the original order.
16 R↑	
17 LBL 01	One or both numbers are ≤ 1.
18 RDN	Drop the 1.
19 -	Compute the difference.
20 RTN	

In this program, there are separate tests and separate branches for each of the two

numbers. The two branches have to be to different destinations (LBL 01 and LBL 02) because the stack is in a different configuration at the time of the tests. In a HP-28 program, the tests are logically combined before the branching decision is made:

<pre> << DUP2 IF 1 > SWAP 1 > AND THEN + ELSE - END >> </pre>	Make a copy of the two numbers. Test the first number. Test the second number. Are both tests true? ...then add. ...otherwise subtract.

Notice how easy it is to read the HP-28 program compared to the HP-41 version. The two tests are right next to each other between the IF and the THEN. The two possible branches then follow immediately after the AND that combines the tests. You can also write this program as a user-defined function (Chapter 8), using the IFTE function (section 10.4.2):

```
<< → x y 'IFTE (x>1 AND y>1, x+y, x-y)' >>
```

You can think of user flags as a kind of variable: the flag number is the variable name, the number 1 or 0 is the value. FS? plays the role of RCL for a user flag--it transfers the flag value to the stack. You use SF and CF to store the values 1 and 0, respectively, into a user flag. There's no single command to store a stack flag directly into a user flag, but the sequence

```
IF SWAP THEN SF ELSE CF END
```

will accomplish that, where the flag number is in level 1 and the new flag value is in level 2.

One by-product of using real numbers as flags for conditionals is that it's easy to test a real number to see if it's zero. In the sequence

```
IF X 0 ≠ THEN A ELSE B END,
```

the $0 \neq$ is superfluous. You can rewrite the sequence as

```
IF X THEN A ELSE B END.
```


10.3.1 HP-28 Test Commands

The HP-28 test command set is comparable to that found in the HP-41 and most other languages: numerical and string comparisons for equality, inequality, and order, and user flag tests. It is worth emphasizing, however, the difference in the argument order for the HP-28 tests compared with their HP-41 counterparts.

The order of test arguments in the HP-28 is chosen to be consistent with the argument order for all other HP-28 functions: the arguments are entered onto the stack in the same order as they appear in algebraic expressions. For example, consider the "greater-than" operator $>$. In an algebraic expression, "is A greater than B?" is written as $A > B$. A is the first argument, reading left-to-right; B is the second. The comparison is true if the first argument is greater than the second. If you rewrite the infix operator $>$ in Polish notation, the expression becomes ' $>(A,B)$ '. Converting to RPN, this becomes $A B >$, which indicates that A should be entered into the stack before B. When $>$ executes, A should be in level 2, and B in level 1.

This is the reverse of the order of HP-41 tests. $X > Y?$ in the HP-41 means "is X (level 1) greater than Y (level 2)?" Therefore, when translating HP-41 programs to the HP-28, you must be careful to use the opposite tests in cases where the order of arguments is important.

Table 10.3. HP-28 and HP-41 Test Commands

HP-28 Test	Meaning	HP-41 Equivalent
$<$	Less than?	$X > Y?$
$\leq$	Less than or equal to?	$X > = Y?$
$>$	Greater than?	$X < Y?$
$\geq$	Greater than or equal to?	$X < = Y?$
$= =$	Equal to?	$X = Y?$
$\neq$	Not equal to?	$X \neq Y?$
SAME	Object same?	$X = Y?$
FS?	User flag set?	FS?
FC?	User flag clear?	FC?
FS?C	User flag set?--clear	FS?C
FC?C	User flag clear?--clear	FC?C

10.3.2 SAME, ==, and =

It is important to distinguish carefully between the three commands SAME, ==, and =, which may appear to have similar meanings. The first point to note is that = is *not* a

test command, so it is fundamentally different from the other two commands, which are tests. $=$ is a *function* that creates an equation from two expressions. Its execution does not return a flag; in symbolic evaluation mode, it does nothing other than evaluate its arguments. In numeric evaluation mode (including using $\rightarrow$ NUM) it acts the same as $-$, returning the numerical difference of the two sides of the equation.

$==$, on the other hand, is a *test*, and always returns a flag when executed. $==$ is primarily intended for ordinary numerical equality comparisons. You can use $==$ in algebraic expressions as an infix operator, just like $<$, $>$, etc. $==$ and $=$ must have different names to distinguish their quite different meanings, and to prevent ambiguity within algebraic expressions. Note that $A=B$ is an “assertion,” whereas $A==B$ is a “question.”

SAME is very similar to $==$; in many cases you can use them interchangeably. Other than the fact that SAME is an RPN command that is not allowed in algebraic objects, the two commands differ only in the manner in which they deal with algebraic and binary integer objects:

- $==$ operates on algebraic objects like any other function, returning a symbolic result when appropriate. SAME compares the original objects themselves, always returning a flag. Thus, $'1+2' 3 ==$ returns the *expression* $'1+2==3'$ (which evaluates to a *true* flag), whereas $'1+2' 3$ SAME returns a *false* flag.
- When comparing binary integers, $==$ ignores leading zeros and compares only the numerical values, so that the relative wordsize of the two integers does not matter. For SAME to return a true flag, the two integers must have the same wordsize as well as the same value. [This distinction is not present in the first HP-28's. In the original 1BB version, $==$ does compare wordsizes, and works like SAME for binary integers.]

10.4 Conditional Branches

Many programming problems require a program to make simple decisions: “If this is true, do that--otherwise do something else.” To deal with program decisions like this, the HP-28 provides the *IF structure*, a program structure that has the general form:

IF *test-sequence* THEN *then-sequence* ELSE *else-sequence* END

You can read this structure as “if *test-sequence* is *true* (returns a true flag), then execute *then-sequence* and jump past the END. If *false*, skip the *then-sequence* and execute *else-sequence*.”

The ELSE *else-sequence* portion of the structure is optional; for cases where “do

something else" is just "do nothing," you can use:

IF *test-sequence* THEN *then-sequence* END,

which translates to "If *test-sequence* is *true*, execute *then-sequence*; otherwise, skip past the END."

- *Example.* Test a user flag specified by its number in level 1, and display YES if the flag is set, or NO if it is clear:

IF FS? THEN "YES" ELSE "NO" END 1 DISP

- *Example.* Order two numbers so that the smaller one is returned in level 1, the greater in level 2.

DUP2 IF < THEN SWAP END	Copy the two numbers. Test if the first is less than the second. If so, switch the numbers.

- *Example.* RTOP is an alternate form of R→P that works on either two real numbers or a single complex number, returning the results in the corresponding forms.

RTOP <i>Rectangular to Polar</i>				
<i>level 2</i>	<i>level 1</i>		<i>level 1</i>	
<i>x</i>	<i>y</i>		<i>r</i>	<i>θ</i>
	(<i>x,y</i>)		(<i>r,θ</i>)	
<< DUP IF TYPE THEN R→P ELSE R→C R→P C→R END >>		Test if the argument is not real. Transform and leave in complex form. Convert to complex, transform, convert to real.		

Because it is THEN that actually removes a flag from the stack and makes the branch decision, the position of the IF in the sequence that precedes THEN is unimportant:


```

1 2 IF > THEN ..., and
1 2 > IF THEN ..., and
IF 1 2 > THEN ...,

```

all produce the same result. You can choose to position the IF wherever you want to make a program the most readable. (The most memory-efficient form has a single object between the IF and the THEN. Thus of the three forms above, the first uses the least memory. See section 11.6.)

10.4.1 Nested IF Structures

Although the HP-28 does not provide a multi-case test structure, you can create multi-branch programs by “nesting” IF structures inside other IF structures.

■ *Example.* The program COUNT4 is a simple four “bin” counting routine.

COUNT4		Count in 4 Ranges	
level 1		level 1	
x		x	
<pre> << IF DUP 0 < THEN 1 ELSE IF DUP 0 == THEN 2 ELSE IF DUP 1 ≤ THEN 3 ELSE 4 END END END END 1 →LIST 'COUNTS' SWAP DUP2 GET 1 + PUT DROP >> </pre>		Test $x < 0$. Range 1. Test $x = 0$. Range 2. Test $x \leq 1$. Range 3. Other tests failed, so x must be greater than 1 (range 4). Convert the bin number to a vector index. Make two copies of the vector name and the index. Get the element, add 1, put it back. Discard the leftover x.	

COUNT4 tests an argument x to see in which of four ranges its value lies. The total in each range is stored in the four-element vector COUNTS. The elements of the vector

represent these ranges:

Element	Range
1	$x < 0$
2	$x = 0$
3	$0 < x \leq 1$
4	$x > 1$

Another way to make a multi-case choice is to create a list of programs, then select one of the programs from the list according to an index. For example, this sequence takes a real number from the stack, and executes a name corresponding to the number:

{ ONE TWO THREE FOUR FIVE }	List of name choices.
SWAP	Put the index in level 1.
GET	Get the indexed choice.
EVAL	Execute the selected name.

10.4.2 Command Forms

An alternate means of achieving IF structure branching is provided by the IFTE and IFT commands. For these commands, the various sequences included in an IF structure are entered as stack arguments, either as single objects or as programs. That is,

test-sequence << *then-sequence* >> << *else-sequence* >> IFTE

is equivalent to

IF *test-sequence* THEN *then-sequence* ELSE *else-sequence* END.

Similarly,

test-sequence << *then-sequence* >> IFT

is equivalent to

IF *test-sequence* THEN *then-sequence* END.

To use IFTE, you put a flag in level 3, a program representing the *then-sequence* in level 2, and a program representing the *else-sequence* in level 1. IFTE tests the flag; if the flag is *true* (non-zero), the *else-sequence* is dropped, and the *then-sequence* is executed. If the flag is *false* (zero), the *then-sequence* is dropped, and the *else-sequence* is executed. IFT works much the same way: the flag must be in level 2, and a *then-sequence* in

level 1. If the flag is *true*, the *then-sequence* is executed, otherwise it is dropped.

For both IFTE and IFT, if any of the then- or else- sequences consist of a single object, you don't need to make a program out of it--just put the object itself in the appropriate level.

■ *Example.* This version of RTOP (section 10.4.1) uses IFTE:

RTOP2 <i>Rectangular to Polar</i>				
<i>level 2</i>	<i>level 1</i>		<i>level 1</i>	
<i>x</i>	<i>y</i>	→	<i>r</i>	<i>θ</i>
	(<i>x,y</i>)	→	(<i>r,θ</i>)	
<< DUP TYPE << R→P >> << R→C R→P C→R >> IFTE >>			Get the input type. Complex case (type ≠ 0). Real case (type 0). Execute appropriate procedure.	

There is no particular advantage within a single program to using IFT or IFTE rather than the corresponding IF structure, so which form you use is mostly a matter of taste. However, the RPN command forms have one advantage for more sophisticated programming: their use allows you to place the *test-sequence*, the *then-sequence*, and the *else-sequence* in separate programs or program structures. If you use an IF structure, all must be contained in the same program.

IFTE is also a function, which means you can use it in algebraic objects as well as in programs. It is a prefix function of three arguments:

$$\text{IFTE}(\text{test-expression}, \text{then-expression}, \text{else-expression})$$

Notice that the arguments are in the same order as the stack arguments when IFTE is executed as an RPN command. All three arguments are ordinary expressions. *Test-expression* is evaluated, and its value is interpreted as a flag. If the flag is *true*, *then-expression* is evaluated; if the flag is *false*, *else-expression* is evaluated. Typically, the *test-expression* contains a comparison operator, so that evaluation automatically returns a flag.

■ *Example.* 'IFTE($X > 0, X, 1 - X$)' returns X if $X > 0$, and $1 - X$ otherwise.

IFT has no algebraic form. This is because algebraic objects must return a result when

evaluated--an algebraic conditional can't "do nothing" if the test flag is false.

10.5 Loops

A *loop* is a program structure containing a sequence that is executed more than once. In a *definite loop*, the number of repeats is known in advance. In an *indefinite loop*, execution repeats until some specified condition is met, then exits the loop and continues with the rest of the program.

10.5.1 Definite Loops

The most common form of definite loop structure is the FOR...NEXT loop. This kind of loop is appropriate when you want a program sequence to repeat several times, making use of an *index* that is incremented by 1 at each iteration of the sequence. The general form of a FOR...NEXT loop is:

start stop FOR name sequence NEXT,

where

- *start* is the (real number) initial value of the index.
- *stop* is the (real number) final value of the index.
- FOR identifies the start of the structure; it removes the start and stop values from the stack.
- *name* is the name of the (local) variable that contains the index.
- *sequence* is any program sequence, which can contain any number of uses of *name*.
- NEXT is the structure word that identifies the end of the sequence. It increments the index by one, then tests its value against the stop value to determine whether to repeat the sequence.

You can read a FOR...NEXT loop as "For each value from *start* through *stop* of an index named *name*, execute the sequence that ends with NEXT."

- *Example.* Sum the squares of the integers from 1 through 100.

0	Initialize the sum.
1 100	Start and stop values.
FOR n	Create a local variable n, with initial value 1.
n SQ +	Square the current index and add to sum.
NEXT	Increment n by 1. If $n \leq 100$, loop again.

Executing this sequence returns the answer 338350.

A few observations:

- *Start* and *stop* as shown above are *not* part of the FOR...NEXT program structure. FOR expects to take two real numbers from the stack, but those numbers can be entered or computed at any time in advance of the FOR, as long as they are in levels 1 and 2 when the FOR executes.
- The *start* and *stop* values are removed from the stack by FOR. They are not accessible afterwards; if a program needs their values for other purposes, it should copy them or store them in variables before executing the FOR.
- The *index* is kept in a local variable identified by the name that immediately follows FOR. You can return the current value of the index by executing its name. You can also change the value of the index after the loop has started, by storing a real number into the local variable. The naming and use of the index variable are subject to the same restrictions as local variables created by $\rightarrow$ (section 10.7). After the loop is finished, the index variable is automatically purged.
- The name following a FOR is *not* part of the sequence that is repeated. For example,

```
1 10 FOR n n NEXT
```

puts integers 1 through 10 on the stack, but

```
1 10 FOR n NEXT
```

does nothing.

- The sequence between FOR *name* and NEXT always executes at least once, even if the specified *stop* value is less than the *start* value.
- The *start* and *stop* values don't have to be integers. NEXT always increments the index by 1; the loop will repeat as long as the index is less than or equal to the stop value.

```
.5 .6 FOR n sequence NEXT
```

executes *sequence* once, with $n = .5$.

- The combination FOR *name* acts like a single operation when you single-step the FOR.

10.5.1.1 Varying the Step Size

The FOR...STEP program structure is a variation of FOR...NEXT, that allows you to increment the loop index by amounts other than one, including negative values. A FOR...STEP structure looks like this:

start stop FOR name sequence STEP.

Start, stop, name, and sequence play the same roles as in FOR...NEXT loops. The structure word STEP plays a similar role to NEXT, but allows you to control the amount by which the index is incremented (or decremented). STEP takes a real number from level 1, and adds it to the current value of the index. Then:

- If the *step* value is *positive*, the loop repeats if the index is less (more negative) than or equal to the stop value.
- If the *step* value is *negative*, the loop repeats if the index is greater (more positive) than or equal to the stop value.

Note that since STEP takes a number from the stack, *sequence* must end with the step value on the stack (the step value doesn't have to be the same each time).

■ *Example.* The program DFACT computes the double factorial $n!! \equiv n(n-2)(n-4)\dots 1$, where n is an integer.

DFACT	
Double Factorial	
level 1	level 1
n	$n!!$
<pre><< 1 SWAP 2 FOR m m * -2 STEP >></pre>	Initialize the product. Loop from n down to 2. m is the index. Multiply the product by m. Decrement m by 2. Repeat if $m \geq 2$.

10.5.1.2 Looping with No Index

In some circumstances, there is no need for an index when a program sequence is to be repeated a fixed number of times. In such cases, you can use START in place of FOR. START...NEXT and START...STEP are the same as FOR...NEXT and FOR...STEP, respectively, except that the loop index is not accessible. The index name that must follow FOR is not used with START (if a name does follow START, it is just treated as part of the loop sequence, and has nothing to do with the loop index).

■ *Example.* The program VSUM sums the n elements of a vector.

VSUM <i>Sum Vector Elements</i>	
<i>level 1</i>	<i>level 1</i>
[<i>vector</i>]	<i>sum</i>
<< ARRAY→ LIST→ SWAP OVER - START + NEXT >>	Put the elements on the stack, with the number of elements in level 2, and a 1 in level 1. Loop start and stop values for $n-1$ additions. Execute + $n-1$ times.

You should use **START** instead of **FOR** when possible

- to save the memory used by the index variable, and
- because a **START...NEXT/STEP** loop executes faster than the corresponding **FOR...NEXT/STEP** loop.

10.5.1.3 Exiting from a Definite Loop

Definite loop structures are designed to repeat a predetermined number of times. There is no “exit” command that can cause program execution to jump out of a loop before it has completed the specified number of iterations. Ordinarily, you should use an indefinite loop (section 10.5.2) for calculations where you don’t know in advance how many iterations are needed. However, the indefinite loops don’t provide an automatic index like that in **FOR...NEXT/STEP** loops, so for some problems you may find it more convenient to use a definite loop with a contrived exit rather than an indefinite loop where you have to provide your own index.

All you have to do to cause a loop to exit before the prescribed number of iterations is to store a number greater than or equal to the stop index value into the index variable. In loops with a positive *step* size, an obvious choice for an exit value is **MAXR**, the largest number that the HP-28 can represent, although you have to be sure to convert the symbolic constant into a real number. For loops with a negative *step*, you can use **-MAXR**.

Typically, the exit from a definite loop is taken as the result of a test. The general form of such a loop is as follows:


```

      start  stop
    FOR  n  sequence
      IF  test
      THEN MAXR →NUM 'n' STO
      END
    NEXT

```

This structure executes *sequence* for every value of *n* starting with *start*, and ends when either *n* is greater than *stop*, or *test* returns a true flag.

- *Example.* Determine the value of *N* for which $\sum_{n=1}^N n^2 \geq 1000$.

<pre> 0 1 10000 FOR n n SQ + IF DUP 1000 > THEN n MAXR →NUM 'n' STO END NEXT </pre>	Initial value of sum; <i>start</i> and <i>stop</i> values. Loop index is <i>n</i>. Increment the sum. Is the sum ≥ 1000? The current value of the index is <i>N</i>. Set the index past the stop value.

Executing this sequence returns the sum 1015, and the value 14 for *N*.

10.5.1.4 Comparison with the HP-41

HP-28 program loop structures replace the HP-41 loops created with the ISG (“increment and skip if greater”) and DSE (“decrement and skip if equal”) commands. Those commands combine the *start*, *stop*, *index* and *step* values into a single decimal “control number” of the form *iiii.fffcc*. The correspondence between the digits of the HP-41 control number and the HP-28 loop parameters is as follows:

- *iiii* is the *index*, which is an integer of up to five digits. Its initial value is the *start* value.
- *fff* is the three-digit integer *stop* value.
- *cc* is the two-digit integer *step* size. If it is omitted, the default step size is one.

The control number is stored in a stack or memory register. This scheme has an advantage over the HP-28 approach in that all three of the loop parameters are available as long as the register is undisturbed, although it takes some calculation to extract the various parts of the control number (and to create the control number in the first place).

The following examples show equivalent HP-41 and HP-28 sequences:

Task	HP-41 Sequence	HP-28 Sequence
Execute a sequence ten times.	1.010 STO 01 LBL 01 <i>sequence</i> ISG 01 GTO 01	1 10 START <i>sequence</i> NEXT
Sum the integers between the values stored in variables (registers) R05 and R06.	RCL 06 1000 / RCL 05 + STO 01 0 LBL 01 RCL 01 INT + ISG 01 GTO 01	0 R05 R06 FOR n n + NEXT

10.5.2 Indefinite Loops

An *indefinite loop* is a loop where the number of iterations is not determined in advance. Instead, the loop repeats indefinitely until some exit condition is satisfied. The HP-28 provides two program structures for indefinite looping, the *DO loop* and the *WHILE loop*. The primary difference between the two structures is the relative order of the test and the loop sequence. In a DO loop, the sequence is performed first, then the test; in a WHILE loop, the test is performed first.

10.5.2.1 DO Loops

The basic form of a DO loop structure is:

DO *loop-sequence* UNTIL *test-sequence* END.

Loop-sequence is any program sequence. *Test-sequence* is a second program sequence, which must end with a flag on the stack. END removes the flag; if the flag is *true* (non-zero), execution jumps back to the start of *loop-sequence*. If the flag is *false* (0),

execution proceeds with the remainder of the program after the END. You can read a DO loop as:

“Do *loop-sequence* repeatedly, until *test-sequence* is true.”

■ *Example.* Compute $\sum_{n=1}^{\infty} \frac{1}{n^5}$.

■ *Solution:* The sequence below sums terms of the form n^{-5} , until two consecutive sums are equal. Executing the sequence returns 1.03692775496, after 184 iterations.

1 'N' STO	Initialize a variable N as a counter.
0	Initialize the sum.
DO	Start of loop.
DUP	Copy the old sum.
N -5 ^ +	Add n^{-5} .
1 'N' STO+	Increment the counter.
SWAP	New sum in level 2, old in level 1.
UNTIL	Start test-sequence.
OVER ==	True if old sum = new sum (leaves only new sum in level 1)
END	Repeat if test was <i>true</i> , otherwise done.

The position of the UNTIL between DO and END is unimportant. That is, the division of the program steps into *loop-sequence* and *test-sequence* is only a matter of program legibility. Both *loop-sequence* and *test-sequence* are executed at each iteration of the loop, so it doesn't matter where you put the UNTIL. We recommend that you use the UNTIL to isolate that portion of the program that constitutes the logical test--the program steps which produce the flag that determines whether or not to repeat. The portion that precedes the UNTIL should be the part of the loop that computes the results used by the remainder of the program after the END.

To reverse the sense of the test, that is, to make a loop that repeats until a test is *false*, you can either substitute an opposite test command (> for <, FC? for FS?, etc.), or insert a NOT immediately before the END:

DO *loop-sequence* UNTIL *test-sequence* NOT END.

10.5.2.2 WHILE Loops

In a WHILE loop, a test sequence is defined in the first part of the structure:

WHILE *test-sequence* REPEAT *loop-sequence* END.

Here again *loop-sequence* is any program sequence, and *test-sequence* is any sequence that returns a flag. REPEAT removes the flag; if the flag is *true*, the program executes *loop-sequence*, then loops back to *test*. If the flag is *false*, *loop-sequence* is skipped, and execution proceeds with the remainder of the program after the END. You can read a WHILE loop like this:

“As long as *test-sequence* is *true*, keep repeating *loop-sequence*.”

- *Example.* The program GCD finds the greatest common divisor (GCD) of two integers n and m . GCD repeatedly computes $r = m - n \text{ IP}(m/n)$; if each successive r is greater than zero, it replaces n with r , m with n , and repeats. When r is finally zero, the value of n is the GCD.

GCD			
<i>Greatest Common Divisor</i>			
level 2	level 1	level 1	
n	m	GCD(n, m)	
<pre><< WHILE DUP2 DUP2 / IP * - DUP 0 > REPEAT ROT DROP END ROT DROP2 >></pre>		Beginning of test-sequence. Make 2 copies of m and n. Compute $r = m - n \text{ IP}(m/n)$. Test $r > 0$. If true, do the following: Replace m and n by new values. Loop back and repeat the test-sequence. Leave n in level 1.	

To reverse the sense of the test, that is, to make a loop that repeats while a test is *false*, you can either substitute an opposite test ($>$ for $<$, FC? for FS?, etc.), or insert a NOT immediately before the REPEAT:

WHILE *test-sequence* NOT REPEAT *loop-sequence* END.

10.5.2.3 DO vs. WHILE

DO loops and WHILE loops are very similar in purpose, and often you can use either form for a programming problem. Here is a summary of the differences between the two structures:

- In a DO loop, the test for looping is made *after* the *loop-sequence* is executed. In a WHILE loop, the test is made before the *loop-sequence*.
- In a DO loop, the *loop-sequence* is executed at least once, and again at every iteration. In a WHILE loop, the *loop-sequence* may not be executed at all. In general the WHILE loop *loop-sequence* is executed one time fewer than the *test-sequence*.
- The position of UNTIL between DO and END is arbitrary, and has no effect on results. The position of REPEAT between WHILE and END is significant.

10.6 Error Handling

Any condition that produces an error beep and an error message display in the HP-28 will also cause any current procedure evaluation to stop. By means of the *IFERR structure*, programs can intercept all errors (except Out of Memory) and continue execution. The IFERR structure has the following general form:

IFERR *error-sequence* THEN *then-sequence* ELSE *normal-sequence* END,

where the three sequences are arbitrary program sequences. You can read an IFERR structure as:

“If any error occurs during the execution of *error sequence*, then execute *then-sequence* and continue execution after the END. If no error occurs, skip *then-sequence* and execute *normal-sequence*, and continue on after the END.”

There does not have to be a *normal-sequence*--the ELSE *normal-sequence* is optional.

IFERR *error sequence* THEN *then-sequence* END

executes *then-sequence* if an error occurs during *error sequence*, but does nothing special otherwise.

■ *Example.* Compute $\sin x/x$, where x is a stack argument, using an IFERR structure to handle the undefined result error condition at $x=0$.

DUP SIN SWAP IFERR / THEN DROP2 1 END

This sequence returns 1 for an argument of zero.

The position of the IF structure word in the sequence preceding THEN in an IF structure is unimportant because it is THEN that actually makes the branch decision. However, the position of IFERR in an IFERR structure is significant; the IFERR and the succeeding THEN define the extent of the sequence for which errors are trapped. IFERR A B THEN intercepts errors in A and B, whereas A IFERR B THEN traps errors occurring only in B. The jump to the *then-sequence* happens immediately upon the error; any remaining steps preceding the THEN are skipped. Thus if an error occurs in A in the structure IFERR A B C THEN D END, B and C are not executed--execution jumps from the point in A where the error occurred directly to D.

Because the reaction to an error is usually specific to a particular error, it is generally a good idea to keep the *error-sequence* short, containing as few as one object if possible. Then there is no ambiguity about which object caused the error, and no part of the sequence that will be skipped. Of course, even a single object may cause different types of errors. To sort out such possibilities, you can use the ERRN command to return the error number of the most recent error, and ERRM to return the text of the error message. For example, suppose that a program adds two arguments. The addition can fail either because the stack is empty, or because the arguments are of the wrong type. The following IFERR structure can deal with either problem:

<pre> IFERR + THEN ERRN IF #513 == THEN GETMORE ELSE ERRM ABORT END END </pre>	Get the error number. Is it error 513 (Too Few Arguments)? Use GETMORE to get more arguments. If the arguments are the wrong type, return the error message as a string.

10.6.1 The Effect of LAST

The design of an IFERR structure must take into account whether last arguments recovery is active at the time an error occurs. If LAST is enabled, the arguments of the command that errors are restored to the stack. If LAST is disabled, the arguments are discarded. The *sinx/x* example in the preceding section assumes that LAST is enabled. The DROP2 in the *then-sequence* is intended to discard the two zeros that cause the division error, and which are restored by the error system. If LAST is disabled, the DROP2 is inappropriate because the two zeros are not returned after the error.

Since in most cases it is preferable for programs to work correctly regardless of the existing state of LAST, IFERR structures should include steps to determine whether LAST is active or not and to act accordingly. Flag 31 provides programmable control of LAST. A program can test the flag to determine if LAST is enabled or disabled, or it

can set or clear the flag itself. These options allow two general approaches for designing error traps:

1. Set or clear flag 31 in the program before the error trap, then write the IFERR structure accordingly. Returning to the `sinx/x` example, either

```
31 SF DUP SIN SWAP IFERR / THEN DROP2 1 END
```

or

```
31 CF DUP SIN SWAP IFERR / THEN 1 END
```

will work. This method has the disadvantage that it may alter the state of flag 31 and thus affect other programs that may depend on the flag. As a rule, any program that does depend on flag 31 or any other flag should itself set the flag the way it wants, so this should not be a major limitation. (You can preserve the state of the LAST flag, and all other flags, by executing RCLF, keeping the resulting binary integer on the stack or in a local variable, then using it with STOF at the end of a sequence that alters the flags.)

2. Include a conditional in the *then-sequence* that can react to the current state of flag 31 without altering it. For example,

```
DUP SIN SWAP
IFERR /
THEN
  IF 31 FS?
  THEN DROP2
  END
  1
END
```

10.6.2 Comparison with HP-41 Error Handling

The HP-41 provides a simple error trapping mechanism centered around flag 25, the *error-ignore* flag. If this flag is set, the HP-41 will abort any command causing an error, clear the flag, and proceed with program execution with the step following the erring command. The HP-41 analog to an HP-28 IFERR structure is a program sequence like this:

SF 25	
<i>command</i>	<i>error “sequence”.</i>
FS? 25	“IFERR”
GTO 01	
<i>normal-sequence</i>	“ELSE...”
GTO 02	
LBL 01	“THEN...”
<i>then-sequence</i>	
LBL 02	“END”
...	

The advantages of the HP-28 system are flexibility, in that the *error-sequence* can contain one or more objects; and legibility--when you read a program it is easy to identify the various sequences because they are set off by the IFERR/THEN/ELSE/END structure words.

An HP-28 IFERR structure is similar in some respects to the ON ERROR GOTO capability in BASIC. The *error-sequence* in BASIC is the sequence between the ON ERROR GOTO and the subsequent OFF ERROR statement; the *then-sequence* is the program starting at the line identified by the GOTO.

10.6.3 Exceptions

A mathematical *exception*, in HP-28 terminology, is a condition encountered in the execution of certain functions for which you are given a choice of how subsequent execution should proceed. You can treat an exception as an execution-halting error, or have the calculator supply a default result and continue normally. You make your choice by means of the various exception action flags (57, 58, and 59). If one of these flags is set, the corresponding exception results in an error. If it is clear, a default result is returned with no error.

A typical exception is division by zero. The behavior of / when the divisor is zero is controlled by flag 59, the *infinite result action* flag. If flag 59 is set, division by zero is treated as an error, causing the Infinite Result error. However, if flag 59 is clear, no error is reported, and one of the values ±9.9999999999E499 (±MAXR) is returned, which are the HP-28's best representations of ±∞. The sign of the result is determined by the sign of the dividend.

The choice to error or to supply a default generally depends on whether the exceptional condition is expected. For example, if you don't anticipate that a program might cause a division by zero, it is better to set flag 59 so that the program will halt and report the error. On the other hand, if you know that the divide-by-zero situation can happen, and that $\pm \text{MAXR}$ is a good approximate result that lets a calculation proceed to meaningful

results, then clearing flag 59 is a good choice.

A program can detect when an exception occurs even if the action flag is cleared to prevent an execution halt. Flags 61 through 64 act as *signal* flags--when an exception occurs, the corresponding signal flag is set automatically. For example, flag 64 is set by an infinite result exception. Therefore, a program can clear flag 64, carry out a calculation with flag 59 clear, and still determine if a division by zero occurred by testing flag 64.

In addition to the infinite result exception, the HP-28 also recognizes two other exceptions:

- *Overflow* (action flag 58, signal flag 63). Overflow occurs when a function returns a result that is finite, but larger than the HP-28 can represent, such as $\text{FACT}(2000)$. With flag 58 clear (the default setting), overflowing functions return $\pm 9.999999999999999\text{E}499$. An overflow is *not* the same as an infinite result, for which the correct value is $\pm\infty$ rather than a too-large finite number.
- *Underflow* (action flag 57, signal flags 61 and 62). Underflow occurs when a function returns a result that is not zero but smaller in absolute value than $1\text{E}-499$ (MINR), the smallest non-zero number that the HP-28 can represent. If flag 57 is clear (the default setting), any underflowing function returns zero as its default result. Since zero has no sign, two signal flags are used: flag 61 is set to indicate that the function underflowed from the negative side of zero; flag 62 set indicates underflow to a small positive number.

Note that $0 \div 0$ is *not* an exception. That quantity is mathematically undefined--it is neither an overflow nor an infinite result. There is no appropriate default result to supply, so the HP-28 always reports the Undefined Result error and halts execution. You can, of course, create your own exception handling by using an IFERR structure to trap this error.

The HP-41 *range-error-ignore* flag (flag 24), is the HP-41's closest equivalent to HP-28 mathematical exception handling. If that flag is set, functions that return results larger than the HP-41 maximum $9.999999999999999\text{E}99$ return the maximum value; if the flag is clear, the OUT OF RANGE error occurs. The HP-41 does not distinguish between infinite and finite-but-too-large results. For example, $\text{TAN}(90^\circ)$ and $\text{EXP}(2000)$ are both treated as the same out-of-range exception. Moreover, division by zero always returns DATA ERROR--this exception can only be trapped by a flag 25 error branch. In converting HP-41 programs to the HP-28, therefore, you can't just use HP-28 flag 58 or 59 to act as HP-41 flag 24. If, for example, you set HP-28 flag 59 to match HP-41 flag 24 in preventing $\text{TAN}(90^\circ)$ from producing an error, the HP-28 then does *not* report an error for division by zero, which the HP-41 always does.

10.7 Local Variables

The HP-28 variables and their associated names that you see in the USER menu are referred to as *global* variables and names. The term *global* implies that these variables can be accessed by any program, or from the command line. The HP-28 also provides variables that are associated only with individual procedures. The use of these *local* variables and the corresponding *local name objects* is a very useful and powerful programming technique.

It is possible, with the “unlimited” stack provided by the HP-28, to carry out an arbitrarily complicated calculation on the stack without any use of variables to store inputs, intermediate results, or final outputs. The fastest and most efficient computation is usually achieved in this manner. The HP-41, which has a four-level stack, lets you keep a few numbers “active” on the stack, but for any but the simplest calculations, you must continually move numbers back and forth between the stack and the data registers. A language like BASIC, which has no stack at all, requires that all input, output, and intermediate results must be stored in variables. This makes individual BASIC statements easy to read, but not particularly efficient.

The popularity of BASIC suggests that it is not always program execution efficiency that is paramount, but rather the overall “throughput” of the problem solving process. If a calculator is easy to program, you can usually get a result in less total time even if the program itself may execute more slowly than if you developed a solution in an efficient but arcane language. Thus while you *can* write a HP-28 program that is a marvel of structure and efficiency by using only stack objects, the time and skill required for you to keep track of everything on the stack during program development may be too high a price for the result. In short, there is often a compelling advantage to assigning names to objects to simplify the programming process.

At first glance this seems to imply the use of global variables that are created by STO, are available at any time, and appear automatically in the USER menu. However, while global variables are fine for “permanent” data and procedures, they are not as attractive for storing intermediate results. They stay around indefinitely, so that you have to remember to purge them to avoid cluttering up the USER menu and to conserve memory. Furthermore, you have to be careful when you create a variable in one program to avoid using the same name as that used by another program, unless you deliberately intend the two programs to share a common variable.

HP-28 *local* variables are a means for saving intermediate data and results that is intermediate between using the stack exclusively and using global variables. Local variables exist only in a context defined by the program structure that creates them; therefore there is no question of name conflicts with global variables or other procedures’ local

variables. Also, when the defining structure has completed evaluation, all of its local variables are automatically purged.

There are two methods by which you can create local variables. The primary method is by means of *local variable structures*, which use the program structure word $\rightarrow$ to create local variables. In addition, the FOR...NEXT/STEP loops described in section 10.5.1 use a local variable to store the current value of the loop index. Although the index variable is used for a special purpose, it is otherwise the same as a local variable created by $\rightarrow$, with the same applicable commands and restrictions. In the remainder of this section, we will concentrate on local variable structures.

A *local variable structure* starts with the structure word $\rightarrow$ (called "arrow," or just "to") followed by one or more local names, and then by a program or an algebraic object referred to as the *defining procedure*. The closing delimiter (' or $\gg$) that ends the defining procedure also marks the end of the structure:

$$\rightarrow \text{ name}_1 \text{ name}_2 \cdots \text{ name}_n \ll \text{program} \gg, \quad \text{or}$$

$$\rightarrow \text{ name}_1 \text{ name}_2 \cdots \text{ name}_n \text{ 'algebraic' }.$$

The user-defined functions described in Chapter 8 are a special case of local variable structures. A user-defined function is a program containing one local variable structure, and no additional objects before the $\rightarrow$ or after the defining procedure. In the HP-28C, the defining procedure must be an algebraic object; in the HP-28S, it may be either an algebraic or a program.

The primary purpose of local variables is to provide a means of manipulating by name the stack arguments used by a procedure. You can think of the $\rightarrow$ as meaning "take objects from the stack and give them the following names; then evaluate a procedure defined using the names." Note that the procedure *is* evaluated, even though it is entered between quote delimiters ' ' or $\ll \gg$.

$\rightarrow$ takes objects from the stack and matches them each with one of the names that follows the $\rightarrow$. The number of objects taken is determined by the number of names that are specified. The end of the series of names is marked by the delimiter ' or $\ll$ that starts the defining procedure. The objects are matched in the order in which they appear in the stack; the object in the highest stack level goes with the first name; the object in level 1 is matched with the last name. A local variable is created for each of the names, with the local name as its variable name, and the matching object as its value. For example,

1 2 3 4 $\rightarrow$ a b c d

creates the local variables *a* with the value 1, *b* with value 2, *c* with value 3, and *d* with value 4.

■ *Example.* Compute the five integer powers x through x^5 of a number x in level 1. This first method does not use any variables except a loop index:

<pre><< 2 5 FOR n n 1 - PICK n ^ NEXT >></pre>	Powers 2 through 5. Loop with index n. Get a copy of the number. Raise to the nth power.

This is not a very complicated program. It is fast and efficient, because it uses only stack operations to obtain copies of the input number. The sequence `n 1 - PICK` is needed to return a new copy each time around because when the index is n , the original number has been pushed to level $n - 1$ by the growing stack of computed powers.

The program looks easy to write, but you do need a little thought to figure out where the input number will be on the stack at each iteration, and what stack operations are required to return a copy of the number. You can avoid the mental gymnastics by writing the program to remove the number from the stack at the outset, and name it with a local name:

<pre><< → x << x 2 5 FOR n x n ^ .NEXT >> >></pre>	Store the number as x. Powers 1 through 5. Loop with index n. Compute x^n. Repeat.

The latter program is slightly longer than the previous version, but the time it takes you to write it should be less because there is no effort required to keep track of the input number on the stack. Any time the program needs the number, it just executes the local name. The lesson of this simple example becomes more important as the complexity of the programmed calculation increases, to the point where using local variables can make the difference between success and failure in the development of a program.

You can use local variable structures at any point in a program, not just at the

beginning as in the case of user-defined functions. The program CINT illustrates the use of a local variable to name an *intermediate* result. CINT computes the radius of a circle inscribed in a triangle, where the lengths of the sides of the triangle are specified on the stack. The formula is:

$$r = \frac{[s(s-a)(s-b)(s-c)]^{1/2}}{s}$$

where a , b , and c are the lengths of the sides, and $s = \frac{1}{2}(a + b + c)$.

CINT				
Circle in a Triangle				
level 3	level 2	level 1		level 1
a	b	c	$\sqrt{}$	r
<pre><< - a b c << '(a+b+c)/2' EVAL -> s '√(s*(s-a)*(s-b)*(s-c))/s' >> >></pre>				Name the lengths of the sides. Compute and save s . Compute r . End of local variable structure.

There are numerous additional examples of the use of local variables in programs throughout this book. In the remainder of this section, we will review some of the idiosyncrasies of local names and variables, and local variable structures.

10.7.1 Comparison of Local and Global Variables and Names

Local names and variables are very similar to ordinary names and variables, but there are some important differences:

- *Global* variables are stored in a permanently established portion of memory we call the **USER** memory. *Local* variables are stored in dynamically created “local memories,” each of which is a segment of memory that acts like an independent **USER** memory assigned to a particular procedure. When the procedure has finished evaluation, the associated local memory is deleted, including all of its local variables.
- *Local* names are a different object type (7) from *global* names (6). This is how the HP-28 system knows whether to find the variable corresponding to the name in **USER** memory (global variables) or in a temporary local memory. When the HP-28 attempts to find a local variable, it searches the most recently created local memory first, then previous local memories in reverse chronological order, until it finds a variable matching the specified name.

- Executing a local name recalls to level 1 the object stored in the corresponding local variable, *without* executing the object. This means that if you store a program in a local variable, to execute that program you must execute the variable name followed by EVAL (or $\rightarrow$ NUM). The EVAL is not necessary for programs stored in *global* variables, since execution of a *global* name does execute the stored object.
- Most commands that can work with *formal* global variables (names with no associated variables) do *not* accept local names as arguments: ∂ , $\int$, TAYLR, DRAW, ISOL, QUAD, ROOT. OBSUB, EXSUB, and SHOW do allow local names as arguments. INDEP accepts a local name on the HP-28C, but DRAW will error if the independent variable entry in PPAR is a local name.
- The only command that can alter the object stored in a local variable after it is created is STO. CON, IDN, PUT, PUTI, RDM, SCONJ, SINV, SNEG, STO+, STO-, STO*, and STO/ do not accept local names as arguments.
- PRVAR works only with global variables.
- You can not delete a local variable with PURGE.
- *Local* names can be the same as HP-28 command names (except for single-character algebraic operator names like +, -, *, etc.). Notice that you can have local names *i* and *e*, but you should be careful not to use these names when you also want to use the symbolic constants *i* and *e*.

These differences between local and global names arise from the design of local variables as a means of simplifying stack manipulations. The idea is to take one or more objects from the stack that you expect to use repeatedly as command arguments, assign names to them, then recall their values by name when they are needed by the commands in a calculation. Local variables are not intended for storing objects that you expect to change--for that purpose global variables are better because of the various storage operations available (section 5.3.1). You can alter the value of a local variable by storing a new object in the variable with STO, but you can't use any of the commands that alter part of an object "in place," like STO+, PUT, or IDN.

Occasionally you may encounter a local name for which the associated local variable no longer exists. For example, a defining procedure may leave the name of a local variable on the stack after it completes evaluation.

```
<< 1  $\rightarrow$  x << 'x' >> >>
```

leaves the local name 'x' on the stack after evaluation, but the corresponding local variable *x* that was given the value 1 is gone. You can not successfully execute this "formal local variable"--EVAL returns the Undefined Local Name error. You should try to avoid leaving left-over local names on the stack or in algebraic objects that result from

symbolic calculations, to avoid confusion later.

10.8 Local Name Resolution

When ENTER processes a name in the command line, it normally interprets the name as a global name. However, if the name follows a FOR or an $\rightarrow$, then ENTER treats the name as a local name while it is handling the rest of the structure that follows. After the subsequent $\gg$, $'$, or NEXT that terminates the structure, further instances of the same names are again interpreted as global names. Thus in

```
<<  → X << X >> X >> ,
```

the X in the inner program ($\ll X \gg$) is a *local* name, but the final X is a *global* name. To help you keep track of which names are which type, we recommend that you adopt a naming convention, such as using lower-case letters for local names, and upper-case letters for global names. The above program then looks like this:

```
<<  → x << x >> X >> ,
```

making it clear that the global X is not to be confused with the two local x's. We will follow this convention in this book, except in certain examples in this section where we are illustrating possible confusions between global and local names.

The resolution of names as global or local can be complicated when you nest local variable structures. "Inner" structures can access the local variables of the "outer" structures that contain them, but not vice-versa. For example,

```
1 → x << 2 → y << x y + >> x + y + >>
```

returns '4+y' (not 6), as follows:

1 → x	Store 1 in local variable x.
<<	Start of program in which x is recognized.
2 → y	Store 2 in local variable y.
<<	Start of program in which y is recognized.
x y +	Add x from "outer" program to y from "inner" program, returning 3.
>>	End of inner program where y is recognized.
x +	Add x to 3, returning 4
y +	This y is <i>not</i> a local name, because it is outside of the program where y is local. It therefore names a global variable, which we are here assuming to have no current value. The sum is therefore '4+y'.
>>	End of outer program where x is a local variable.

If you rewrite the above sequence as


```
1 → x << 2 → y << x y + x + y + >> >> ,
```

moving the final *y* back inside the program where the local variable *y* is defined, the sequence then returns the value 6.

When two nested local variable structures define local variables with the same name, two separate local variables are created. Any use of the name refers to the most recently created local variable. The fact that there is another local variable with the same name in a previously created local memory does not matter. Thus

```
1 → x << 2 → x << x >> >>
```

returns 2, whereas

```
1 → x << 2 → x << >> x >>
```

returns 1.

It is important to note that a procedure represented by a *name* (rather than the procedure itself) within a local variable structure can not access the local variables defined by that structure (unless the procedure is created while the structure is evaluating or suspended; see below). For example, if you create the program A:

```
<< x y + >> 'A' STO,
```

and invoke it in another program like this:

```
<< 1 2 → x y << A >> >> ,
```

then executing the latter program returns '*x+y*' (global *x* and *y*), not 3. When you enter the program A, *x* and *y* are created as global names. The search for their values when A is executed in the second program therefore is made in USER memory, even though there are identically named local variables at the time of the search.

This property of local variables, which makes it possible for each program to define its own variables without name conflicts with those of other programs, has the disadvantage that you can't always easily break a program containing a local variable structure into smaller programs. For example, you can't rewrite

```
<< → x y << sequence1 sequence2 >> >>
```

as two programs


```

    << sequence1 >> 'SEQ1' STO
    << → x y << SEQ1 sequence2 >> >>,

```

if *sequence₁* contains either of the names *x* or *y*. There are several approaches that you can use instead:

- Use global variables. Rewrite the second program as

```

    << 'y' STO 'x' STO SEQ1 sequence2 { x y } PURGE >>.

```

In this case you might keep the lower-case names *x* and *y* for the global variables, to avoid editing *sequence₁* and *sequence₂*.

- Use the stack to pass the values from one program to the other. Rewrite the programs as:

```

    << → x y << sequence1 >> >> 'SEQ1' STO
    << → x y << x y SEQ1 sequence2 >> >>

```

The latter program puts the values of *x* and *y* back on the stack, where *SEQ1* can store them in its own local variables *x* and *y*. This approach requires no change to *sequence₁*.

- Force *x* and *y* in *SEQ1* to be created as local variables. You can achieve this by entering the *SEQ1* program while there is an existing local memory containing local variables *x* and *y*.

1. Type

```

    0 0 → x y << HALT >> ENTER

```

You will see the suspended program annunciator turn on. Because the local variable structure is executing when the program halts, the local memory containing local variables *x* and *y* is still present.

2. Enter the program *SEQ1*:

```

    << sequence1 >> 'SEQ1' STO.

```

All instances of *x* and *y* in *sequence₁* are treated as local names.

3. Now, when you execute the main program

```

    << → x y << SEQ1 sequence2 >> >>,

```


execution of the names `x` and `y` in `SEQ1` returns the values stored at the start of the main program.

This method, although it solves the problem with no rewriting, is dangerous because if you later edit `SEQ1`, you must remember to create again the halted program local memory. Otherwise, the command line reentry converts `x` and `y` back into global names. Also, you won't be able to use `SEQ1` as a subroutine for other programs unless those programs also define local variables `x` and `y`.

11. Program Development

The process of transforming a problem into a calculator program is seldom straightforward. There are elements of art, and of personal preferences and style, that preclude a single prescription for programming. It is not even easy to define what distinguishes a good program from a bad one. For example, one program might require less memory, or run faster, or have fewer steps than another. But perhaps you can develop the less efficient program and use it to obtain results in less time than it takes just to design the other; which, then, is the "better" program?

In this chapter, we will study some general-purpose topics in HP-28 program development, with examples to illustrate each topic. In some cases, we'll show several versions of a program, to illustrate how "first tries" can evolve into more refined or capable versions. From these examples, you should be able to see how various HP-28 programming tools and techniques can be combined. You can remember those methods that appeal to you, and through practice, develop your own methodology.

11.1 Program Documentation

Throughout this book we have been using a particular format for listing programs. We recommend that you adopt a system like this one for developing and recording your own programs. The following sample listing illustrates the various features of the format:

SAMPLE <i>Sample Program Listing</i>				
<i>level 3</i>	<i>level 2</i>	<i>level 1</i>		<i>level 1</i>
<i>"string"</i>	<i>[matrix]</i>	<i>n</i>		<i>[matrix']</i>
<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 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. Following 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 $\rightarrow$, 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 $\rightarrow$ 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.

3. 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.
4. Most program objects and program structures start on a new line. If the program or structure is a short one, the entire structure may be shown on one line. More frequently, each program 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 or print a program on the HP-28S, the program display follows these same conventions, within the limitation of the 23-character screen or printer width.
5. The comments at the right of the listing describe the purpose or results of the program lines at the left. 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.

11.2 Program Editing

To make any alteration to an existing program in order to correct an error, optimize execution, or add features, you must *edit* the program. Because HP-28 programs are objects, you edit a program the same way you edit any other object. That is, you use EDIT or VISIT to create a text version of the program in the command line, use the facilities of the command line to make the alterations you desire, then execute ENTER to replace the old copy of the program with the new one. Re-entering the entire program this way ensures that objects and program structures are entered correctly.

When an object is copied into the command line by EDIT or VISIT, any numbers in the object are shown to their full precision, regardless of the current number display mode. That is, floating-point numbers are shown in STD format, and binary integers with a wordsize of 64 bits. This prevents the accidental changing of numbers during editing. On the HP-28C, there is one pitfall to watch out for, however. If you edit a binary integer, or a list or program containing binary integers, changing the binary integer base while the object is still in the command line will change the interpretation of the number when you press **ENTER**. For example, if you select HEX mode, then edit the number #FF, change the base to BIN, and press **ENTER**, you will get a Syntax Error message because the digit F is not allowed in BIN mode. Worse, if you select BIN mode first, edit the number (which is #11111111 in binary), then press **HEX** **ENTER**, the number will be reentered as *hexadecimal* #11111111, which is quite different from the original value. Thus when you are working with binary numbers on the HP-28C, keep these potential effects in mind when you change bases during an edit. On the HP-28S, binary integers are shown with an identifying character (b, d, h, or c), so that reentering a binary integer will not change its base regardless of the current mode.

The advantages of the HP-28 program editing approach are:

- The same editing methods apply to all HP-28 object types, so that you don't have to learn special techniques for each object type.
- No changes you make during an edit are "final" until you press **ENTER**. If you change your mind while you are editing a program, you can just press **ON** to cancel the edit and leave the program intact.

On the other hand, there are two important disadvantages:

- For a large program, it can take a substantial amount of time for the HP-28 to translate the entire program object into its text form, and, when you're done editing, to build the new program from the command line text.

- During the execution of ENTER, there must be memory available for as many as three versions of the program (the original, the command line text, and the new version) simultaneously. This restricts the size of the program that can be edited.

The latter disadvantage is the most serious, because it can happen that there isn't enough memory to permit any changes to an existing program, even if the changes don't increase the final size of the program. (This is particularly serious on the HP-28C, which has only 2 Kbytes of user memory, restricting its programming primarily to small programs.) Both disadvantages dictate that you keep programs small, typically less than a few dozen objects. If a program starts to get too big as you develop it, break it up into smaller subprograms that are executed by a short main program. Even though this costs a little more memory for the subprogram names and variables, the smaller programs will be editable when a big single program is not.

11.2.1 Low Memory Editing Strategies

When you run out of memory (No Room to ENTER) trying to enter an edited program (or any other object), you can use the following steps to increase the available memory:

1. Remove any unwanted objects--clear the stack, kill any suspended programs (section 11.3), and purge unneeded variables from user memory.
2. Turn off LAST and UNDO: press **MODE** **NEXT**, then **-UND** **-LST** on the HP-28C, or **UND** **LAST** on the HP-28S.
3. Recall the object you want to edit to level 1. If the object is stored in a variable, purge the variable to save the memory used for the variable.
4. Press **EDIT**.
5. Press **-CMD** then **+CMD** (HP-28C) or **CMD** **CMD** (HP-28S). This empties the command stack, but leaves COMMAND active.
6. Make your changes, and press **ENTER**. If you still get the No Room to ENTER error, press **COMMAND** to return the object to the command line, **-CMD** (HP-28C) or **CMD** (HP-28S) to disable and clear the command stack, then **ENTER**. This step is unpleasant, because if there is still not enough room, you will have lost the edited version of the object.

If the preceding steps fail, you can take the more drastic step of purging the object you are trying to edit. That is,

1. With the object in level 1, press **-CMD** **+CMD** (HP-28C) or **CMD** **CMD** (HP-28S).

2. Press **■** **EDIT** to copy the object to the command line; make your changes.
3. Press **ENTER**. This will presumably fail, with the No Room to ENTER message.
4. Press **DROP** to discard the object.
5. Press **■** **COMMAND** to recover the command line with the altered text version of the object.
6. Try **ENTER** again. If there is no error message, you're finished. But if ENTER fails again, then...
7. Press **■** **COMMAND** to retrieve the command line one more time. Now press **≡-CMD≡** (HP-28C) or **≡CMD□≡** (HP-28S) to disable the command stack. Press **ENTER**. If this fails, you're out of options, and out of luck--all copies of the object are gone. Generally, however, this process will succeed unless you are making major additions to the edited object.

11.3 Starting and Stopping

As we have discussed in previous sections, HP-28 programs are highly structured, and each has only a single entrance and exit. This fact makes starting and stopping an HP-28 program a different proposition from the simple run/stop capability of other calculators. On the HP-41, for example, you can stop a running program at any time by pressing **R/S**. When that key is pressed, the program halts after the currently executing step, and returns control to you. You can use **GTO** to move the program counter to another line or label, or run another program, etc. When you press **R/S** again, program execution resumes from wherever the program counter happens to be.

In the HP-28, if a program is to stop and be able to be restarted, it must include a HALT command in its definition. You can stop any program by pressing **ON**, but as you will see below, that abandons all pending execution in the currently executing program and cancels pending returns to any other programs that may have called that program. (In more precise terms, the return stack is cleared, and the normal stack display and keyboard are reactivated.)

When HALT is executed, the program containing the HALT is *suspended*. The suspended program annunciator (the octagon) turns on to remind you that there is a program awaiting completion. The keyboard is activated, and all calculator operations work normally. The HP-28 can maintain this state indefinitely--it behaves as if you had started up another calculator "inside" the halted program. This suspended calculator *environment* even has its own UNDO stack that is independent of the usual UNDO stack that was present before the suspended program was started. The calculator operates in the suspended environment until you press **■** **CONT** (*continue*), whereupon the

suspended program resumes execution at the object following the HALT. (HALT is the HP-28 version of HP-41 STOP, and CONT corresponds to RUN--except that CONT has no effect unless there is a program suspended. In the HP-41, RUN just means "go from here," from wherever the program counter happens to be.)

You can "nest" suspended program environments one within another without limit (other than available memory). While one program is halted, you can run another program that itself halts and sets up another calculator environment with its own UNDO stack, and so on. When you press **CONT**, the latest suspended environment is deleted, including the UNDO stack created for that environment. If you press **UNDO** immediately after a program completes execution, the UNDO stack that was saved by the ENTER that started the program is restored. To illustrate, enter the following program and name it A:

```
<< CLEAR 1 2 HALT 3 4 >> 'A' STO
```

Then:

Keystrokes:

Results:

CLEAR 'X' 'Y'
ENTER

2: 'X'
1: 'Y'

A **ENTER**

2: 1
1: 2

Suspend Annunciator is on.
The program has put 1 and 2 on the stack, and halted.

CLEAR

2:
1:

UNDO

2: 1
1: 2

UNDO restores the stack cleared by the last CLEAR.

CONT

4: 1
3: 2
2: 3

	1:	4	The program A resumes, pushes 3 and 4 onto the stack, and is finished.
■ UNDO	2:	'X'	
	1:	'Y'	Back to the original environment; the last ENTER in this environment was the one that started the program A. Thus UNDO restores the stack as it was before that ENTER.

Since the command line itself is a program, you can include a HALT in any command line even if the HALT is not explicitly contained in a program object delimited by << >>. When you press **ENTER**, the command line is executed up to the HALT. Then you can perform any normal operations; when you finally press ■ **CONT**, the rest of the suspended command line is executed. This suggests an easy method for saving the current stack while you carry out some temporary calculation. With an empty command line, execute HALT **ENTER**. After any series of calculations you can restore the original stack by pressing ■ **CONT** ■ **UNDO**.

Keep in mind when you're working with a suspended program that *local* variables created by the program may exist (see section 10.8). For example, if a program halts while a local variable A that it created still exists, then executing the name A from the command line returns the value of that local variable, not the value of a global variable A that might also exist. (Pressing the **≡ A ≡** key in the USER menu always executes the global name A regardless of any local variables that might exist.)

11.3.1 WAIT

The WAIT command causes a simple pause in program execution. *x* WAIT produces a pause of *x* seconds. The WAIT pause is not interactive like HP-41 PSE; you can't use it as a means to enter data without halting a program. The primary application of WAIT is for displaying messages during program execution. If your program shows a series of messages, you can put a WAIT after one or more of the DISP commands to ensure that the message remains visible long enough to be read conveniently.

11.3.2 KEY

When you press an HP-28 key, a code representing that key is entered into a special memory location called the *key buffer*. When the HP-28 is otherwise idle, it checks the

key buffer to see if any key codes have been entered. If so, it removes the codes one at a time (in the same order in which they were pressed), then performs whatever operations are associated with the keys. This two-stage key processing is responsible for the HP-28's "type-ahead" capability, whereby up to 15 keystrokes can be stored in the buffer while the busy annunciator is on (section 3.11.3).

Programs can check and act on the contents of the key buffer by executing KEY. KEY attempts to remove the oldest keycode from the key buffer. If it succeeds, it returns a string object representing the key name to level 2, and a true flag (1) to level 1. If there are no entries in the key buffer, KEY returns only a false flag (0) to level 1, and no key string. By using KEY, programs can accept keyboard input, on a key-by-key basis, without actually halting execution.

The program PSE (listed on the next page) uses KEY to provide a number entry method similar to that of the HP-41 command PSE. PSE waits for up to 1 second; if a key is pressed during that interval, the 1 second wait is started over. If the key is a numeric key (digits **0** through **9**, **EEX**, **.**, or **-**), the key value is appended to a number string; all other keys (except **ON**) are ignored. When no further keys are pressed, PSE converts the current string to a number and is finished.

11.3.3 The ON key, ABORT and KILL

As the ATTN (*Attention!*) letters below the **ON** key suggest, this key is your means for getting the "attention" of the calculator. When you press **ON**, you tell the calculator to stop what it is doing: stop all operations, procedures, etc., clear any special displays, reactivate the keyboard, and show the standard stack display. This is a "gentle" interruption--USER memory is unaffected, the stack is preserved, and the UNDO stack, COMMAND stack, and LAST arguments are left intact. The procedure return stack is cleared, meaning that programs interrupted by **ON** can't be continued.

When a program is suspended, **ON** acts as above, but preserves the suspended environment. That is, any suspended programs (and all of the associated UNDO stacks) are unaffected by **ON**. This is a reassertion of the statement that all ordinary calculator operations can be carried out while a program is suspended without affecting the suspended program--**ON** is considered "ordinary" in this sense.

ABORT is a programmable form of **ON**. Executing ABORT in a program (or in the command line) acts as though the **ON** key were pressed at the point in the program where the ABORT appears. The program stops, and all pending returns to procedures that called that program are cleared. Like **ON**, ABORT works in the current suspended program environment--if there are any suspended programs, they are unaffected by ABORT. You can use ABORT in a program to terminate program execution

PSE	HP-41-like PSE
☐	
<pre> << "" WHILE 0 DO 1 + UNTIL IF KEY THEN 1 ELSE DUP 31/44† == END END DUP TYPE 2 == REPEAT SWAP DROP IF DUP "EEX" == THEN DROP "E" + ELSE IF DUP NUM 47 OVER < OVER 58 < AND OVER 46 == OR SWAP 45 == OR THEN + ELSE DROP END END DUP 1 DISP END DROP STR- CLMF >> </pre>	Start with an empty string. Keep going as long as keys are pressed. Start a counter. Increment the counter. Increment the counter until... ...a key is pressed ...or the counter reaches 31 or 44 (1 second). Repeat if KEY returned a string (type 2). Discard the counter. Now process the key. Append an "E" if the key is EEX. Get the character number of the key. Is it a digit key? Or a "."? Or "-"? Then append that character. Otherwise, discard it. Show the current string. Go get another key. Discard the counter. Convert the string to a number.

†Use 31 for the HP-28C, or 44 for the HP-28S.

early, when some situation is encountered that makes further execution pointless. Usually this is done with an IF structure, such as

IF *situation-is-hopeless* THEN ABORT END.

Note that ABORT, like ON, clears special displays. If you want a program that executes ABORT to return a message, the program must put the message on the stack as a

string object prior to executing ABORT.

The only command that does affect suspended programs is KILL. KILL is an extended form of ABORT that not only terminates the current program, but also clears *all* suspended programs and turns off the suspended program annunciator. All of the temporary UNDO stacks associated with the suspended program environments are deleted. You can use KILL in a program, but that is a rather drastic thing to do, since in general a program doesn't "know" what programs are suspended when it is executed. It's better to use ABORT in a program, then execute KILL manually if needed. Most likely, your most frequent use of KILL will be to clear some half-finished program that you have been single-stepping, once you have found the problem you have been seeking.

11.3.4 System Halt and Memory Reset

A *system halt*, obtained by pressing **ON** **Δ** together, is a means of stopping execution that is more drastic than pressing **ON**. A system halt clears the stack, the return stack, all suspended program environments, LAST arguments, the UNDO stack, and the COMMAND stack (plus the CUSTOM menu in an HP-28S). System halts also stop the endless loops (section 3.6) caused by executing the name of a variable that contains its own name, which you can't stop with **ON**.

You can also perform a *memory reset*, by pressing **ON** **INS** **▷** all together. A memory reset is a complete calculator reset, deleting all global variables and resetting all flags to their default values, as well as performing a system halt and displaying Memory Lost. [If you see this message when you turn the calculator on, or at any other time when you have not deliberately performed a memory reset, it indicates that the calculator has detected a corruption of memory contents such that it can not continue normal operation without a memory reset. This corruption can be caused by a hardware fault, including the effects of static electricity, or by the execution of SYSEVAL (section 3.10) with an incorrect system address.]

11.3.5 Single-Stepping

The SST (single-step) operation is a combination of CONT and HALT that lets you step through a program one object at a time. Single-stepping is an important debugging tool (section 11.4), because it lets you follow the execution of a program step-by-step and discover where its calculations go awry.

To understand the mechanics of SST, picture it as the equivalent of pressing **■** **CONT** with a HALT temporarily inserted immediately after the next object in the program. This model implies that in order to single-step a program, it must be suspended. Thus any program that you want to single step through must contain a HALT at the point where you want to start stepping. Then when you execute the program, it will suspend

execution after the HALT and you can proceed with single steps. At each SST press, the HP-28 executes the next object in the suspended program, then halts and suspends the program again. To help you keep track of where you are in the program, each object is momentarily displayed in inverse characters in display line 1 just before it is executed. If you single-step the >> that ends the suspended program, the program completes execution and the suspended program environment is cleared.

Another consequence of the behavior of SST as a one-step CONT is that each SST clears the current suspended program environment, then creates a new one after the step. This means that you can't cancel any stack effects of the object that was single-stepped by pressing UNDO --the UNDO stack present before the SST is deleted by the SST.

Some additional notes about SST:

- An IFERR structure is treated as a single object by SST. That is, when you press SST at an IFERR, the entire IFERR...THEN...ELSE...END structure is executed. If an error occurs between IFERR and THEN, the *then-sequence* between THEN and ELSE is executed; otherwise the *else-sequence* (if it is present) between ELSE and END is executed. The next SST will single step whatever object follows the END. If you want to step through individual parts of the IFERR structure, you must insert HALT(s) within the structure.
- If a single-stepped object causes an error, the error is reported normally, but the single step execution does not advance. If you press SST again, the HP-28 will attempt to execute the same object again. This gives you a chance to fix whatever it is that causes the error, such as a missing stack argument, then proceed with single-stepping.
- At any time while you are single-stepping a program, you can return to normal execution of the remainder of the program by pressing CONT.

11.4 Debugging

Debugging is the art of finding and removing programming errors--"bugs." The process ranges from simple visual inspection of a program to look for obvious errors, through careful single-stepping of parts of a program to watch for incorrect results at each stage.

Programming errors usually manifest themselves in two ways when you execute a program: either the program halts due to an error, or the program completes execution but returns incorrect results (which may be due to an incorrect algorithm, rather than a program defect). In either case, you know something is amiss--the trick is to find out where things go wrong in the program.

A good debugging technique for any programming language is to write the program correctly in the first place. This sounds facetious, but chances are, if you take extra time in designing a program before entering it into the calculator, you will save time in the long run by reducing the amount of debugging time. For HP-28 programs, a good approach is to write out a program of any complexity on paper, using the program formatting conventions discussed in section 11.1. Most importantly, as you add steps to a program, include simple stack contents listings at least every few steps. This will help you get the program right in the first place; failing that, the stack listings will be your most valuable tool for debugging.

When a program fails, the first step in finding errors is to view the program to verify that you have entered it correctly--that it matches your program listing. You can recall the program to level 1 and use **VIEW** and **VIEW**, or **EDIT** to scan through the program. Usually, the program is stored in a variable, so you can use **VISIT**. If you have a printer, use PRVAR to print out a complete listing of the program. If the program matches the original listing, there must be a logical error in the program design.

Before resorting to single-stepping, you may be able to apply the HP-28's symbolic capabilities to find an error. That is, even when a program is designed for purely numerical calculation, you can execute the program with symbolic arguments, then compare the symbolic results with the intended program algorithms (this is a good thing to do to verify any numerical program, not just when you're explicitly looking for an error).

For example, in section 11.5 we develop a program that finds the two roots of a quadratic equation $ax^2 + bx + c = 0$, where the three coefficients a , b and c are specified. The final version of the program is:

QU Quadratic Root Finder					
level 3		level 2	level 1	level 2	level 1
a		b	c	x_1	x_2
<pre> << 3 PICK / SWAP ROT 2 * / NEG DUP SQ ROT - √ DUP2 + 3 ROLL - >> </pre>				<pre> a b c a b c/a c/a -b/2a c/a -b/2a b^2/4a^2 -b/2a √[(b/2a)^2 - c/a] -b/2a √[(b/2a)^2 - c/a] x_1 x_1 x_2 </pre>	

Because this program involves a lot of stack manipulations, it's easy to lose track of the program flow as you develop it. Suppose that when writing the program, you

miscounted the number of stack objects, and entered SWAP in place of the 3 ROLLD at the end. If you execute the program with numerical values for the coefficients, you obtain incorrect results--but no indication that they are wrong. To guard against this, you can verify the program by executing it with symbolic arguments 'A', 'B', and 'C' (purging those variables first, if necessary, to ensure symbolic calculations). With these arguments, the bad version of the program returns

```

2:          ' - (B/(A*2))'
1:  ' - (B/(A*2)) + √(SQ(- (B
    / (A*2))) - C/A) - √(SQ(-
    (B/(A*2))) - C/A)'

```

By inspecting the level 1 result, you can see that the program correctly added the radical '√(SQ(- (B/(A*2))) - C/A)' to ' - B/(A*2)', but then subtracted the same radical from the sum in level 1 rather than from the other ' - B/(A*2)' in level 2. This suggests that the error is a stack error near the end of the program, and it is then a simple matter to figure out that the SWAP should have been 3 ROLLD.

The final resort in debugging is to single-step the program, from the beginning if necessary, until you discover an incorrect step. As described in section 11.3.5, in order to use SST, you must include a HALT in the program at the point where you want to single step. To single step from the beginning of a program, the HALT must follow the initial << as the first object in the program. Using SST for debugging is a three-step process:

1. Edit the program ( **EDIT** or  **VISIT**), insert HALT at the appropriate point, then press **ENTER** .
2. Execute the program. When the program stops (the suspended program annunciator turns on), switch to the CONTRL menu (CTRL on the HP-28C) and begin pressing **SST** to single step through the program.
3. When you have identified the incorrect step, press **KILL** to clear the suspended program. (At this point, if you press  **UNDO** , you will recover the program's arguments that were on the stack when you began execution. This will save reentering them when you try the program again.) Now edit the program to correct the problem.

If you are sure you have the solution, remember to remove the HALT as you edit the program. Otherwise, you can leave it in until after you verify the new version. When the program halts, press  **CONT** to resume.

■ *Example.* Find the error in the following program MINL. The program is designed to return the minimum value from a list of numbers. Starting with an initial value of MAXR, the program successively replaces the current value with the minimum (MIN) of

the current value and the next number from the list.

MINL	<i>Minimum in a List (Bad version)</i>	
	<i>level 1</i>	<i>level 1</i>
	{ <i>numbers</i> }	<i>minimum</i>
<pre> << MAXR -NUM SWAP DUP SIZE 1 DUP ROT START GETI 4 ROLL MIN 4 ROLL NEXT DROP2 >> </pre>		

If you execute this program with a list of numbers, the program aborts with the Too Few Arguments message (on the HP-28S, the guilty ROLL command is also identified, which simplifies the search for the program error). You can observe that the first command in the program that can error in this manner is the SWAP in the first line. So, you should edit the program to insert a HALT immediately before the SWAP. Then (start with an empty stack):

Keystrokes:**Results:**

{ 1 2 3 } <u>USER</u> <u>MINL</u>	2:	{ 1 2 3 }	The argument list.
	1:	9.999999999999E499	Initial "minimum" value.
<u>SST</u> (SWAP) <u>SST</u>			
(DUP) <u>SST</u> (SIZE)	3:	9.999999999999E499	
	2:	{ 1 2 3 }	
	1:	3	Number of elements in the list.
<u>SST</u> (1) <u>SST</u> (DUP)			
<u>SST</u> (ROT)	5:	9.999999999999E499	
	4:	{ 1 2 3 }	
	3:	1	Start value for GETI index.
	2:	1	Start value for START.
	1:	3	End value for START.

<u>SST</u> (START)	3:	9.999999999999E499	Current minimum.
	2:	{ 1 2 3 }	
	1:	1	Current GETI index.
<u>SST</u> (GETI)	4:	9.999999999999E499	Current minimum.
	3:	{ 1 2 3 }	
	2:	2	New GETI index.
	1:	1	First list element.
<u>SST</u> (4) <u>SST</u> (ROLL)	4:	{ 1 2 3 }	
	3:	2	GETI index.
	2:	1	List element.
	1:	9.999999999999E499	Current minimum.
<u>SST</u> (MIN)	3:	{ 1 2 3 }	
	2:	2	GETI index.
	1:	1	New minimum.
<u>SST</u> (4) <u>SST</u> (ROLLD)	Too Few Arguments.		

Here you can see exactly what is wrong. The program tries to execute 4 ROLLD with only three objects on the stack (attempting to put the objects back in the correct positions for the next iteration of the loop). The solution is to change the 4 ROLLD to 3 ROLLD. Here's the correct program listing:

MINL <i>Minimum of a List (Good Version)</i>	
level 1	level 1
{ numbers }	x_{min}
<< MAXR -NUM SWAP DUP SIZE 1 DUP ROT START GETI 4 ROLL MIN 3 ROLLD NEXT DROP2 >>	maxr { x_i } n Initialize m (list index). Loop from 1 to n. x_{min} { x_i } m x_m x_{min} { x_i } m

You can verify that this version works correctly by using a symbolic input. For example,

`{ A B C } MINL ⏏ 'MIN(C,MIN(B,MIN(A,9.999999999999E499)))'`.

11.5 Program Optimization

The fastest, most compact, and most memory efficient HP-28 programs are usually those that carry out all of their calculations on the stack, using no local or global variables, and only fine-tuned RPN sequences for mathematics. These programs are also the hardest to write, since you have to keep track of the stack positions of everything, and spend time thinking about efficient ways to write the programs.

In this section, we will illustrate the process of *program optimization*, the process of revising working programs so that they execute faster or more efficiently. In general, program optimization involves

- a. writing a first version of the program;
- b. replacing pieces of the program with more efficient ones;
- c. knowing when to stop optimizing and use the current version.

There is no fixed prescription for HP-28 program optimization. There are two general purpose approaches that apply in most situations:

- Reduce the use of variables by keeping more objects on the stack.
- Replace long algebraic objects with RPN sequences that allow you to reuse intermediate results.

We will illustrate the application and effect of these two ideas in an extended program development example. Other methods and tricks will be apparent in the program examples in this chapter and elsewhere in the book.

■ *Example.* Develop and optimize a program QU that computes both roots x of the quadratic equation $ax^2 + bx + c = 0$, where the (numerical) coefficients a , b , and c are supplied as stack arguments. The mathematical algorithm is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Using local variables and algebraic objects, it is easy to translate the algorithm into a first version of the program. This version uses 157.5 bytes (including the name QU), and takes .28 seconds (on the HP-28S) to execute:

Version 1:

<pre><< → a b c << '(-b+√(b^2-4*a*c))/(2*a)' EVAL '(-b-√(b^2-4*a*c))/(2*a)' EVAL >> >></pre>	<pre> a b c Name the arguments. x₁ x₂</pre>

To optimize this program, the first thing you might notice is that the solution algorithm can be written more compactly as

$$x = -b' \pm \sqrt{b'^2 - c'},$$

where $b' = b/2a$ and $c' = c/a$. You can incorporate this revised form into a new version of the program:

Version 2:

<pre><< → a b c << 'b/(2*a)' EVAL 'c/a' EVAL → c b << '-b+√(b^2-c)' EVAL '-b-√(b^2-c)' EVAL >> >> >></pre>	<pre> a b c Store c' and b'. x₁ x₂</pre>

Version 2 takes .23 seconds to execute, so compacting the algorithm has yielded a modest speed improvement. However, version 2 is 169 bytes, 12.5 bytes larger than version 1--the extra local variable structure has cost more in program size than the algorithm compaction saved. As the next step in optimization, you can eliminate that extra structure by computing b' and c' directly from the original stack arguments:

Version 3:

<pre><< 3 PICK / SWAP ROT 2 * / → c b << '-b+√(b^2-c)' EVAL '-b-√(b^2-c)' EVAL >> >></pre>	<pre> a b c a b c/a c/a b/2a Store c' and b'. x₁ x₂</pre>

Version 3 occupies 125 bytes of RAM, which is 31.5 bytes smaller than version 1. It is also slightly faster (.21 seconds) than version 2. To improve on this version, you can

observe that the two algebraic objects in the program are very similar, which means that the program performs some arithmetic twice. You should therefore be able to improve matters by breaking up the algebraic objects into smaller parts that are common to both expressions.

Version 4:

<pre><< 3 PICK / SWAP ROT 2 * / → c b << '-b' '√(b^2-c)' DUP2 + EVAL SWAP ROT - EVAL >> >></pre>	<table><tr><td> a b c </td></tr><tr><td> a b c/a </td></tr><tr><td> c/a b/2a </td></tr><tr><td>Store c' and b'.</td></tr><tr><td>Make 2 copies of the partial results.</td></tr><tr><td> -b √(b²-c) x₁ </td></tr><tr><td>x₂</td></tr></table>	a b c	a b c/a	c/a b/2a	Store c' and b'.	Make 2 copies of the partial results.	-b √(b ² -c) x ₁	x ₂
a b c								
a b c/a								
c/a b/2a								
Store c' and b'.								
Make 2 copies of the partial results.								
-b √(b ² -c) x ₁								
x ₂								

Version 4 has shrunk the program size to 106.5 bytes, but execution has slowed to .29 seconds. The slowdown has resulted from a subtle cause: the final + and - that combine the partial results are acting on symbolic arguments, returning symbolic results (which are then evaluated into the final numeric results using EVAL). Symbolic addition and subtraction are intrinsically slower than numeric arithmetic. You can fix this problem with a simple rearrangement so that the partial results '-b' and '√(b^2-c)' are evaluated *before* they are added or subtracted:

Version 5:

<pre><< 3 PICK / SWAP ROT 2 * / → c · b << '-b' EVAL '√(b^2-c)' EVAL DUP2 + SWAP ROT - >> >></pre>	<table><tr><td> a b c </td></tr><tr><td> a b c/a </td></tr><tr><td> c/a b/2a </td></tr></table> Store c' and b'. Make 2 copies of the partial results. <table><tr><td> -b √(b²-c) x₁ </td></tr></table> x₂	a b c	a b c/a	c/a b/2a	-b √(b ² -c) x ₁
a b c					
a b c/a					
c/a b/2a					
-b √(b ² -c) x ₁					

Version 5 is the same size as version 4, but it executes in .13 seconds, which is the fastest time yet.

The progress made so far in optimizing this program suggests completing the process of converting the algebraic expressions into pure stack arithmetic, eliminating the use of variables.

Version 6 (final version):

QU <i>Quadratic Root Finder</i>					
level 3	level 2	level 1		level 2	level 1
<i>a</i>	<i>b</i>	<i>c</i>		<i>z</i> ₁	<i>z</i> ₂
<< 3 PICK / SWAP ROT 2 * / NEG DUP SQ ROT - √ DUP2 + 3 ROLL - >>				<i>a</i> <i>b</i> <i>c</i> <i>a</i> <i>b</i> <i>c/a</i> <i>c/a</i> $-b/2a$ <i>c/a</i> $-b/2a$ $b^2/4a^2$ $-b/2a$ $\sqrt{[(b/2a)^2 - c/a]}$ $-b/2a$ $\sqrt{[(b/2a)^2 - c/a]}$ <i>x</i> ₁ <i>x</i> ₁ <i>x</i> ₂	

Version 6 requires only 64 bytes, and executes in .11 seconds. This represents a 60% reduction in program size, and a 2.5× speed improvement over version 1.

The lesson here is not that algebraic objects evaluate numerically more slowly than their RPN sequence equivalents. The execution time difference between, for example, '1+2+3+4+5+6+7' EVAL and 1 2 + 3 + 4 + 5 + 6 + 7 +, is only a few milliseconds--which can be accounted for by the time required to put the algebraic object on the stack. Instead, the point is that RPN lets you avoid repeating mathematical operations by breaking calculations into unique elements, and then duplicating and reusing the results. Furthermore, it is always faster for a program to leave results on the stack rather than storing them in variables, and similarly faster to retrieve arguments from the stack than to recall them from variables.

11.6 Memory Use

To help you in optimizing programs for minimum memory size, Table 11.1 lists the memory size of various objects included in a program. There are exceptions to the sizes listed in the table, since the HP-28 has built in certain commonly used objects, to save memory. For example, the real number 1 uses only 2.5 bytes, instead of the 10.5 bytes normally used by a real number. Similarly, each of the following built-in objects uses 2.5 bytes:

- Real integers from -9 through +9.
- The real constants 3.14159265359 (π), 2.71828182846 (*e*), 1E-499 (MINR), and 9.99999999999E499 (MAXR).

- The complex constant $(0,1)$ (i).
- The null string "".
- On the HP-28C, the single character global names A, B, C, ... Z.

Table 11.2 shows the memory occupied by program structures, not counting the objects that are entered between the structure words.

When an object is stored in a variable, the variable requires memory for the object itself, plus an additional amount for the variable structure. Specifically, the variable "overhead" is 4.5 bytes plus one byte for each character in the variable name. The memory used by a stored object is the same as the amount listed in the above table, with one exception. Programs require 10 bytes (plus the included objects) rather than the 12.5 bytes listed in the table for programs in programs.

Table 11.1. Object Sizes

<i>Object Type</i>	<i>Size (bytes)</i>
Real number	10.5
Complex number	18.5
Binary Integer	13
String	$5 + \text{number of characters}$
Vector	$12.5 + 8 \times \text{number of elements}$
Complex vector	$12.5 + 16 \times \text{number of elements}$
Matrix	$15 + 8 \times \text{number of elements}$
Complex matrix	$15 + 16 \times \text{number of elements}$
List	$5 + \text{included objects}$
Command	2.5
Unquoted name	$3.5 + \text{number of characters}$
Quoted name	$8.5 + \text{number of characters}$
Algebraic	$5 + \text{included objects}$
Program	$12.5 + \text{included objects}$

Table 11.2. Program Structure Sizes

Structure	Size (bytes)
IF/IFERR ...* THEN ...* END	17.5†
IF/IFERR ...* THEN ...* ELSE ...* END	25†
DO ... UNTIL ... END	7.5†
WHILE ... REPEAT ...* END	12.5†
START/FOR ... NEXT/STEP	5
→ ... << ... >>	7.5
→ ... ' ... '	7.5

†A program savings of 5 bytes in each instance is obtained whenever any of the structure sequences marked with an asterisk (...*) consist of one object.

11.7 Input and Output

In programs, as well as in keyboard calculation, the stack is the basic input/output mechanism. You can enter all the data a program needs as stack objects, execute the program, then read its results from the stack. This works fine under two conditions: first, you know in advance what objects to enter at the start, and second, there are not so many inputs or outputs that you lose track of which is which among the stack objects. We will describe two methods for improving on this bare-bones approach:

- Using prompted input and labeled output.
- Using variables and/or menu keys for input and output.

11.7.1 Prompts and Labels

11.7.1.1 Prompted Input

The HP-28 STRING menu provides commands for building display messages suitable for input prompting and output labeling. The central command is DISP, which displays a string in any of the four lines of the LCD. The other commands in the menu are for manipulating strings and characters to make strings for display.

The program PROMPT listed below is a utility program that you can use for general purpose input prompting. The HP-41 PROMPT command serves as the model for this program. That command halts execution and displays the contents of the alpha register to prompt you for data entry. The HP-28 program PROMPT takes a string from the stack to name the entry, appends that string to "Enter ", displays the result string at the

top of the display, then halts for input. After entering the requested data, you press **■** **CONT**, whereupon PROMPT ends, presumably to return to another program that called it.

PROMPT <i>Input Prompt Utility</i>	
<i>level 1</i>	
<i>"string"</i> ☐ <i>object</i>	
<pre><< "Enter " SWAP + CLLCD 1 DISP HALT >></pre>	Take the input string, and append it to "Enter ". Clear the display, show the prompt string in line 2. Stop for input.

■ *Example.* The following sequence prompts for Length, Width, and Height:

"Length" PROMPT "Width" PROMPT "Height" PROMPT.

Upon execution, the sequence halts and displays "Enter Length". At this point, you enter a value for the length, and press **■** **CONT**. Then the display shows "Enter Width", and so on.

There are various ways you can extend PROMPT. Imagine a program that works only for real numbers between 0 and 10. If you enter a number out of this range, you would like the program to return a message and prompt you for another entry, rather than halting with an error.

To achieve this, you can modify PROMPT as follows:

<pre><< "Enter " SWAP + WHILE CLLCD DUP 1 DISP HALT DUP 0 < OVER 10 > OR REPEAT DROP "Out of Range" 1 DISP .5 WAIT END SWAP DROP >></pre>	Build the prompt. Prompt for entry. Is the entry out of range? Discard the entry. Show a message. Wait .5 sec. Go back and try again. Discard the prompt.

If a program requires several inputs, each of which requires a different test than the one used here, it needs a more flexible version of PROMPT. We will show in section 11.10

how to modify PROMPT to handle more general tests, as part of a discussion of using programs as arguments.

You can create similar general purpose output labeling routines. The following program LABEL makes a labeled display of an object in level 2 by converting the object into a string, and appending it (with an "=") to a string provided in level 1. A copy of the object is left in level 1.

LABEL			
<i>Output Labeling Utility</i>			
	<i>level 2</i>	<i>level 1</i>	<i>level 1</i>
	<i>object</i>	<i>"label"</i>	<i>object</i>
<< " = " + OVER →STR + CLLCD 1 DISP >>		Append " = " to the label. Get the object and make a string. Append the object string to the label string. Clear the LCD and display the string.	

The next example program BOX computes the volume of a box, illustrating the use of PROMPT and LABEL:

BOX	
<i>Box Volume</i>	
	<i>level 1</i>
	<i>Volume</i>
<< "Length" PROMPT "Width" PROMPT "Height" PROMPT * * "Volume" LABEL >>	Enter <i>length</i> . Enter <i>width</i> . Enter <i>height</i> . Compute the volume. Label the result.

The above version of LABEL uses the current display mode (via →STR) to determine the appearance of numbers in the displayed string. In some cases, the string combining the label and the result may be too long to fit on one line of the display, so part of the result may not be shown. An alternate version of LABEL displays the label in line 1, and the result object starting in line 2:

LABELV		Output Labeling Utility, Variation	
level 2	level 1		level 1
object	"label"	⊠	object
<pre><< " =" + CLLCD 1 DISP 2 DISP >></pre>		<pre> object "label" Append " =" to the label. Display the label. Display the object.</pre>	

For programs that return more than one result, you have two choices: either show one result at a time, or, if the results are short numbers, show up to four at once. The next program, **MULTLABEL**, takes a number (from 1 to 4) from level 1 to determine how many results to display. Then it takes that many label strings and that many numbers, and combines them into a single display. The numbers (which will remain on the stack) and the labels must be in the same order.

MULTLABEL		Label Multiple Outputs	
level $2n+1$...level $n+2$	level $n+1$...level 2	level 1	level n ...level 1
$object_1 \cdots object_n$	"string ₁ "..."string _n "	n	⊠ $object_1 \cdots object_n$
<pre><< CLLCD → n << n →LIST n 1 + ROLLD n →LIST 1 n FOR m OVER m GET " =" + OVER m GET →STR + m DISP NEXT SWAP DROP LIST→ DROP >> >></pre>		<pre>Save the number of results as n. Combine the labels into a list. Move the label list behind the numbers. Combine the numbers into a list. For $m = 1$ to n: Get the mth label. Append " = ". Get the mth number. Append to the mth label string. Display in line m. Repeat. Discard the label list. Unpack the result list.</pre>	

11.8 Using The USER Menu for Input and Output

The USER menu provides a straightforward means for visually prompted input and output of several quantities in parallel. For example, when a program completes execution, instead of returning results to the stack, it can store them in variables. Then you can press the corresponding USER menu keys to see any of the results, in any order.

You can also use the USER menu for input, either by:

- using STO to enter values directly into the displayed variables (`[ ] [name] [STO]`); or
- including in the USER menu short programs, of the form `<< 'name' STO >>` that let you store a value you have entered with a single keystroke.

USER menu output is even simpler than input, since all you have to do is press a menu key to see the value of any output variable. You may want to include ORDER at the end of a program to insure that menu keys for its result variables are visible at the start of the USER menu when the program finishes.

On the HP-28C, where you can't activate a menu from a program, these methods presume that the USER menu is active when the program is started (which is not unreasonable, since the easiest way to start a program is to press the appropriate USER menu key), or that the program user knows to activate the menu when the program is complete. On the HP-28S, a program can activate the USER menu by executing 23 MENU.

11.9 The HP-28S CUSTOM Menu

The HP-28S CUSTOM menu is expressly designed for the input and output of data and results by means of the menu keys. Furthermore, you can create custom menus in which you can mix variable names and built-in commands. Here are some of the ways you can use this feature:

- By building a menu of the commands that you use most frequently, you can substantially reduce the number of keystrokes needed for subsequent operations.
- You can have menu keys for any of the variables in the current path (section 5.7.1), even when they are not in the current directory. This is particularly useful for variables containing utility programs that you have stored in a parent directory of the current directory.
- A program can halt and present its own special menus to facilitate data entry and results output.

There are two types of custom menus, which we will call *input* and *output* menus. Both are created by the MENU command (in the MEMORY menu); the type of custom menu

created is determined by the list of objects that MENU uses as its argument. When you execute MENU, the custom menu that you specify is activated on the menu keys immediately. If you change menus, you can reactivate the custom menu from the keyboard by pressing  **CUSTOM**, or in a program by executing 19 MENU (see section 11.9.5).

11.9.1 Output Menus

An *output menu*, which might also be called an *execute menu*, consists of menu keys that work like those in the USER menu. That is, when you press a menu key (in immediate entry mode--see section 3.11.1), the object that appears in the corresponding menu key is executed ("output").

You create an ordinary custom output menu by executing MENU with a list of names and/or built-in commands in any order. For example, to create a custom menu with menu keys for DUP, CON, TRN, and DET, plus variables ONE, TWO, ZIP, and ZAP, execute

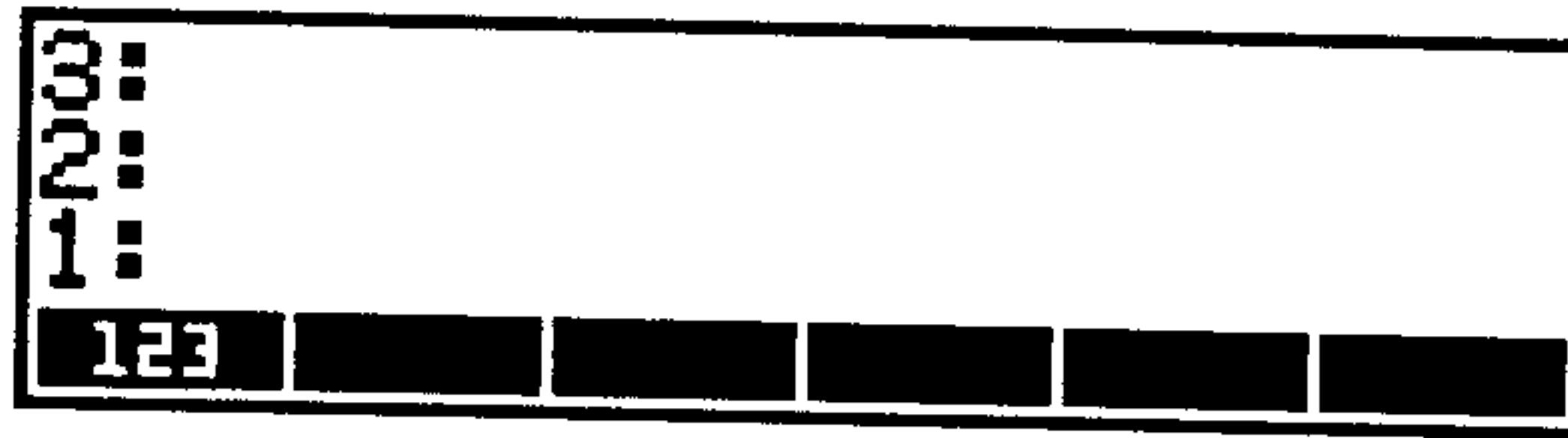
```
{ DUP CON TRN DET ONE TWO ZIP ZAP } MENU
```

The first six commands/names appear immediately as menu key labels. The remaining two appear when you press **NEXT**. There's no limit on the number of objects in the list; just like any other menu, the labels appear in groups of six as you step forward or backward through the menu using **NEXT** and  **PREV**.

Custom output menu keys corresponding to names and commands generally behave the same as they do in the USER menu or in a built-in menu. That is, in immediate entry mode (section 3.11.1), pressing a menu key *executes* the corresponding object. In alpha entry mode, the text characters for the name or command are put into the command line. In algebraic entry mode, functions and names are put into the command line, but RPN commands are executed. These rules are *always* true for custom menus, even for commands for which built-in menu keys have special behavior. For example, mode selection keys such as HEX in the BINARY menu execute their respective commands in immediate entry mode *without* performing ENTER (section 3.11.3). But if you put HEX in a custom menu, the CUSTOM menu HEX key is *not* special--pressing it does cause ENTER in immediate entry mode.

A shortcoming of "ordinary" custom output menus as described so far is that you can't include program structure words such as IF, UNTIL, or → in a custom menu list, because they are not objects. Attempting to enter a list containing any of these words returns Syntax Error. However, you *can* include structure words in a custom menu by entering the words as string objects in the menu list. In fact, you can put *any* type of object in

the list, not just names and commands. A custom menu contains keys for each object in a menu list; pressing a CUSTOM menu key executes the object “displayed” in the key label. For example, if you execute { 123 } MENU, you will see



Notice that $\equiv 123 \equiv$ appears as a menu label. When you press $\equiv 123 \equiv$, the number 123 is executed, and appears in level 1. If you press $\boxed{\alpha} \equiv 123 \equiv$, the characters 123 are entered into the command line.

In contrast to the USER menu, in which you execute stored objects by means of menu keys corresponding to global names, and to built-in menus, which contain the names of commands (built-in objects), custom output menus can include directly executable objects. Among other things, this means that the menu labels may be rather cryptic--not only is the label restricted to five or fewer characters, but objects other than real numbers and names are displayed with their characteristic delimiters, which take up space in the labels so that even fewer characters are left for “showing” the object. Furthermore, there are no menu label characters available for {, }, <<, or >>, which appear in menu labels as small box characters.

The rules for the behavior of menu keys are modified for the case of custom output menus containing arbitrary objects. The following summarizes custom output menu key actions, and brings out some additional strengths and weaknesses of the custom menus.

When you press a custom output menu key:

- In immediate entry mode, the current command line is entered (implicit ENTER), then the object shown in the key label is executed.
- In algebraic entry mode, if the menu key object is a real or complex number, a global or local name, or a function name, the text form of the object is added to the command line with the automatic addition of spaces (functions also have a “(” appended). For other object types, the current command line is entered (implicit ENTER), and the object is executed.
- In alpha entry mode, the text form of the menu key object is added into the command line. Spaces are added if necessary to separate the new text from the existing text. There are, however, certain limitations:

- a. The text added for numbers is determined by the the current decimal number display format and binary integer base and wordsize.
- b. String objects are “stripped” of their double-quote delimiters when added to the command line.
- c. Only the *first* line of text in the command line form of an object (as produced by EDIT) is added to the command line. Thus for example, if you press a menu key assigned to the program << X >>, only the two characters << X are actually added to the command line in alpha entry mode. If the program starts with << IF ..., only the delimiter << will go into the command line, since the EDIT form of the program puts the IF on a new line after the <<.

The fact that menu key string objects are entered without their quotes allows you to include the program structure words in a custom menu by entering them as strings in the MENU list. Since these words are normally only entered in alpha entry mode, the CUSTOM menu keys will enter the correct text surrounded by spaces, just like the BRANCH menu keys. This feature also provides a means for entering into the command line any of the HP-28 display or print characters that are not available on the keyboard. For example, to create a menu with the characters @, &, :, ;, and !, execute

```
64 CHR 38 CHR 58 CHR 59 CHR 33 CHR 5 →LIST MENU
```

When you press one of the resulting menu keys in alpha entry mode, the corresponding character is added to the command line. Spaces are automatically added around the character, but you can use the cursor menu keys to remove the spaces if you want.

11.9.2 Input Menus

The Solver menu is a convenient means for storing data into the variables contained in the current equation. In the HP-28S, the Solver menu key labels in the display are shown with black letters on a white background to distinguish them from other menus, particularly from the USER menu (white letters on black). When you press a Solver menu key, a stack object is *stored* into a variable. When you press a USER menu key, the reverse happens: the contents of a variable are *executed* (“recalled”). Most menus are output menus, but the Solver menu is an *input* menu.

The MENU command allows you to create custom input menus with which you can store objects into any variables. To do so, you enter a list of names, the same as for a custom output menu, except that the first element in the list must be STO. The STO signals MENU to create an input menu; if it's absent, MENU creates an output menu. All of the objects in the list other than the STO must be names.

For example, to create a custom input menu for variables named A, B, and C, execute

{ STO A B C } MENU

Notice that a custom input menu has key labels displayed in black letters on a white background, just like the Solver menu. Another feature adopted from the Solver menu is that when you press a custom input menu key, the top display line will show *name: object*, where *name* is the key label and *object* is the object that you stored from level 1. However, unlike the Solver menu, nothing special happens when you press the shift key followed by a menu key. The input menu keys work the same whether shifted or unshifted.

An important purpose of custom input menus is to let programs prompt you via menus for data entry while the programs are halted. The advantages of using an input menu for this purpose are:

- They provide an interface similar to that of the Solver, so that if you have learned how to use the Solver, it is an easy matter to use a custom menu.
- The input menu prompts you for several inputs at once.
- You can enter the individual data items in any order.
- You can use the menu keys as typing aids for recalling the contents of any of the menu variables (1 name RCL).
- While a program is halted, you can use any of the resources of the HP-28S to compute the data required by the program. If you change menus, you have only to press CUSTOM to restore the input menu.

A typical program use of a custom input menu is illustrated by the following sequence:

{ STO A B C } MENU HALT

When a program containing this sequence encounters the HALT, execution stops with the custom menu labels visible in the display. At this point you use the menu keys to enter data for any or all of A, B, and C, then press CONT. The program resumes execution, and may use any of the new values of A, B, and C in its calculations.

11.9.3 Local Names in Custom Menus

You can include local names as well as global names in the list used by MENU to create a custom menu. This means, for example, that a program can use a custom input menu to prompt you for values that it will store in local variables and discard when it is finished. This is appropriate when the data needed by the program is for temporary use,

and has no importance to you or to other programs after the program is completed. Similarly, a program might use a custom output menu to show you some intermediate results computed during its execution that are of no interest later.

The use of local names in MENU lists is subject to the usual restrictions associated with local names:

- In order for a name in a MENU list to be entered as a local name, it must be entered within the program structure that creates the associated local variable. For example, a program can prompt with an input menu for local variables a, b, and c, with a sequence like this:

→ a b c << { STO a b c } MENU HALT ... >>.

Notice that this requires that the local variables exist, i.e. have values, before the menu list is created. It may be necessary in some cases to create the local variables with “dummy” values (a real number 0 is a typical choice) that serve as place holders until the desired values are entered by means of the custom menu.

- The menu keys in the CUSTOM menu corresponding to local names are only useful as long as the corresponding variables exist. If you run a program that creates a CUSTOM menu containing local names, the menu will still be the CUSTOM menu after the program is finished, but the variables will no longer exist. Pressing any of the local name menu keys then returns an Undefined Local Name error.
- It is convenient in some calculations for a program to halt and offer you a custom output menu of other programs to execute during the halt. However, if the original program creates the secondary programs and stores them in local variables, pressing the menu keys labeled with the names of those variables won't run the corresponding programs. This is because executing a local name just *recalls* the object stored in the local variable without executing the object. To run one of the menu programs, you have to press the menu key and then **EVAL**. Use of global variables is usually better for this purpose, since pressing a global name menu key executes a stored program, without the distraction and delay of seeing the program returned to the stack and having to press **EVAL**.

11.9.4 Saving Custom Menus

The HP-28S stores the current CUSTOM menu list in a temporary location in memory. A temporary location is used rather than a reserved-name variable, for example, so that the menu list can be found by the system regardless of the current directory. The list is preserved until it is replaced by a later use of MENU, or until you perform a system halt (**ON** - **Δ**).

When you create a custom menu, it is a good idea to take the extra step of saving the

menu list in a variable before using it as an argument for MENU. That is, instead of executing a sequence like

```
{ DUP ZIP →ARRY OVER ZAP ROT } MENU,
```

execute

```
{ DUP ZIP →ARRY OVER ZAP ROT } DUP 'MENU1' STO MENU.
```

This doesn't take any extra memory (except for a few bytes for the variable name), but it does have these advantages:

- You can restore this custom menu after changing to another menu, or after a system halt, by executing `MENU1 MENU`.
- You can edit or add to the list without having to rebuild it.
- You can use the list for other purposes.

The easiest way to reconstruct a custom menu list, if you've executed MENU without saving the argument list, is to press `[ ] [α]`, then each of the menu keys in the CUSTOM menu, then `[ENTER]`.

11.9.5 Programmable Built-in Menu Selection

Besides creating CUSTOM menus as described in the preceding sections, you can use MENU with a real number argument to activate any of the 24 built-in menus. The menu numbers are determined by the order in which the menus appear on the HP-28S keyboard, starting with 1 for the ARRAY menu and continuing left-to-right, top-to-bottom down the left keyboard to TEST, menu number 15. Menus 16 (MODE) through 23 (USER) are those on the right keyboard. The SOLVR menu, which does not appear on the keyboard, is number 24. This use of MENU is primarily intended for programmable menu selection. It has little use as a keyboard operation, since it is easier to press one of the permanent menu selection keys.

11.10 Programs as Arguments

An unusual and powerful feature of the HP-28 is its ability to use *procedures* as arguments for commands and other procedures. This capability is clearly illustrated in HP-28 symbolic algebra, where algebraic objects can be the arguments for functions. In this section, we will demonstrate the use of *programs* as arguments. The fact that HP-28 programs are objects, and that therefore you can put an *unexecuted* program on the stack, means that one program can transfer procedural information to another program as easily as it can transfer data.

For example, here's a simple modification to the program PROMPT developed in

section 11.7.1.1, that allows a program that calls it to specify a test for the suitability of the input:

TESTPROMPT <i>Input Prompt Utility</i>		
level 2	level 1	
<< program >> "string" 		
<< CLLCD "Enter " SWAP + → t p << WHILE p 1 DISP HALT t EVAL NOT REPEAT DROP "Out of Range" 1 DISP .5 WAIT END >> >>		Save the test and prompt. Get the entry. Discard the entry Show a message. Wait .5 seconds. Go back and try again.

The program used as an argument should return the input, and a *true* flag if the input is acceptable or a *false* flag otherwise. For example, a program that wants you to enter a number "Length" in the range 0 through 10, can include the sequence

<< DUP DUP 0 ≥ SWAP 10 ≤ AND >> "Length" TESTPROMPT.

As a more ambitious illustration of the use of programs as arguments, we will develop a program INFSUM to compute the sum

$$\sum_{n=n_0}^{\infty} f(n),$$

where $f(n)$ is an argument for INFSUM, not part of the program. That is, to use INFSUM, you enter, as stack arguments, n_0 and a program representing $f(n)$.

The following is an example of program development, where you start with a single-purpose program, and expand it in stages to a more general case. The first step is a program SUM4 that computes the specific sum

$$\sum_{n=1}^{\infty} \frac{1}{n^4}$$

SUM4 accumulates terms until successive sums are equal, i.e. additional terms are less than 10^{-12} of the current total. It returns the result 1.08232323295.

SUM4	Sum $1/n^4$
	level 1
	sum
<pre> << 0 1 DO DUP -4 ^ SWAP 1 + ROT ROT OVER + DUP 4 ROLLD UNTIL == END DROP >> </pre>	Initialize sum. Starting value of n. $sum(n)$ n $sum(n)$ n n^{-4} Increment n. $n+1$ $sum(n)$ $sum(n+1)$ $sum(n+1)$ n $sum(n)$ $sum(n+1)$ Keep going until $sum(n+1) = sum(n)$. Drop n.

In reviewing SUM4, you can observe that the sequence $-4 \wedge$ is the only part of SUM4 that is specific to the particular sum $\sum n^{-4}$. The rest of the program just handles the mechanics of adding successive terms and deciding when to stop. You can make the program work for any sum $\sum f(n)$ by replacing $-4 \wedge$ in the fourth line of the program with the name TERM. The variable TERM should contain a program that computes $f(n)$, where n is provided in level 1. The summation program becomes:

SUMTERM	Compute an Infinite Sum from TERM
	level 1
	sum
<pre> << 0 1 DO DUP TERM SWAP 1 + ROT ROT OVER + DUP 4 ROLLD UNTIL == END DROP >> </pre>	Initialize sum. Starting value of n. $sum(n)$ n $sum(n)$ n $f(n)$ Increment n. $n+1$ $sum(n)$ $sum(n+1)$ $sum(n+1)$ n $sum(n)$ $sum(n+1)$ Keep going until $sum(n+1) = sum(n)$. Drop n.

To compute $\sum n^{-4}$ with SUMTERM:

```
<< -4 ^ >> 'TERM' STO SUMTERM 1.08232323295.
```


Actually, the use of the variable TERM is an unnecessary contrivance. The idea is to supply SUMTERM with the information of how to compute $f(n)$ --but that information, which is represented by the program `<< -4 ^ >>`, can just as well be supplied as a stack argument. To see how, omit the 'TERM' STO from the preceding sequence. Then, at the point where TERM is about to be executed in SUMTERM, the stack looks like this:

```

4:      << -4 ^ >>
3:      sum(n)
2:      n
1:      n

```

Thus the effect of executing TERM (evaluating $f(n)$) can be achieved by the sequence 4 PICK EVAL. The program INFSUM (listed on the next page) makes that replacement, and to generalize further, makes the initial index n_0 an input argument as well.

■ *Example.* Use INFSUM to compute the sum $\sum_{n=1}^{\infty} \frac{n^2}{2^n}$.

In this case, the program argument is `<< → n 'n^2/(2^n)' >>`, and $n_0 = 1$. So the sum can be obtained with

```
<< → n 'n^2/2^n' >> 1 INFSUM ⌂ 5.99999999999
```

or

```
<< DUP SQ 2 ROT ^ / >> 1 INFSUM ⌂ 5.99999999999
```

(The second version is faster.) INFSUM may run for a considerable amount of time if the sum converges slowly. For $f(n) = n^{-4}$, it takes 670 terms to compute the result 1.08232323295, which is accurate to the tenth decimal place (the correct value is 1.08232323371). The program will take correspondingly longer for sums that converge more slowly than this. We therefore list a second version INFSUMM, that you can use instead of INFSUM when you want to monitor the sum as it accumulates.

Additional variations of INFSUM are discussed in section 11.12.4.

INFSUM <i>Compute an Infinite Sum</i>	
level 2	level 1 level 1
<< term >>	n_0  sum
<pre> << 0 SWAP DO DUP 4 PICK EVAL SWAP 1 + ROT ROT OVER + DUP 4 ROLLD UNTIL == END ROT DROP2 >> </pre>	Initialize sum. <pre> proc. sum(n) n proc. sum(n) n f(n) </pre> Increment n. <pre> proc. n+1 sum(n) sum(n+1) proc. sum(n+1) n sum(n) sum(n+1) </pre> Keep going until $sum(n+1) = sum(n)$. Discard n and procedure.

The argument << term >> must have the logical form << $\rightarrow n$ 'term(n)' >>.

INFSUMM <i>Compute an Infinite Sum (Monitor)</i>	
level 2	level 1 level 1
<< term >>	n_0  sum
<pre> << 0 SWAP DO DUP 4 PICK EVAL SWAP 1 + ROT ROT OVER + DUP 1 DISP DUP 4 ROLLD UNTIL == END ROT DROP2 >> </pre>	Initialize sum. <pre> proc. sum(n) n proc. sum(n) n f(n) </pre> Increment n. <pre> proc. n+1 sum(n) sum(n+1) </pre> Display the running sum. <pre> proc. sum(n+1) n sum(n) sum(n+1) </pre> Keep going until $sum(n+1) = sum(n)$. Discard n and procedure.

The argument << term >> must have the logical form << $\rightarrow n$ 'term(n)' >>.

11.11 Timing Execution

Minimizing execution time is an important aspect of program development and optimization. The HP-28 contains a system clock that it ordinarily uses for scheduling internal events such as blinking the cursor or turning off the calculator after ten minutes of non-use. The program TIMED illustrates the use of the system clock to time the execution of any object. The object may either be in level 1 or stored in a variable specified by a name in level 1 (that is, if the level 1 object is a name, it is replaced by the contents of the corresponding variable). TIMED was used to determine the various execution times listed in this book. Since its argument can be any kind of object, TIMED also provides another example of the use of procedures as arguments.

TIMED			
<i>Timed Execution</i>			
<i>level 1</i>			<i>level 1</i>
<i>object</i>	☒		<i>time</i>
<i>name</i>	☒		<i>time</i>

Note: Read section 3.10 before entering this program!

<pre> << DUP TYPE IF 6 == THEN RCL END MEM → t << #123E/#1266/#11CAh† SYSEVAL 't' STO EVAL #123E/#1266/#11CAh† SYSEVAL t - B→R 8192 / .017 - >> >> </pre>	Determine object type. If it's a name, replace with the named object. Force memory packing, and create a local variable t. Get the start time. Store start time in t. Execute the timed object. Get the finish time. Compute the net time. Convert to decimal seconds. Subtract time for 't' STO.

†Choose #123E for HP-28C Version 1BB, #1266 for HP-28C Version 1CC, or #11CAh for HP-28S Version 2BB. Be sure the HP-28C is in HEX mode before entering this program.

TIMED makes use of a system program that is accessible by means of the SYSEVAL command (section 3.10). The system program returns the current time from the system clock, which counts time in units of 1/8192 second. The address of the system clock program depends on the HP-28 version:

HP-28C Version 1BB: #123E SYSEVAL
HP-28C Version 1CC: #1266 SYSEVAL
HP-28S Version 2BB: #11CA SYSEVAL

All of these integers are expressed in hexadecimal. The correct sequence returns a binary integer representing the system time.

- *Example.* How long does it take the HP-28C to invert a 4×4 identity matrix?

4 IDN << INV >> TIMED  2.07.

The answer is 2.07 seconds.

11.11.1 Erratic Execution

You have probably noticed that HP-28 execution, in everything from keystroke entry to user program execution, does not always proceed smoothly but is frequently interrupted by momentary pauses. This is quite noticeable in plotting, for example, where the orderly plotting of points is broken by periodic pauses as if the calculator were “catching its breath.” This erratic execution is normal behavior for the HP-28, and should not concern you except to keep it in mind when you are timing program execution--two consecutive identical operations may take quite different times to execute.

During the course of operations, the HP-28 creates dozens or hundreds of “temporary objects.” These are the objects that you put on the stack, that are not also stored in a global variable (or in another object in a variable). Between the times when the stack display is updated, the system itself may also create many temporary objects that you never see. When you or the system removes a temporary object from the stack, either by using it as an argument, or storing it in a variable, or just dropping it, the memory used for the temporary object is not recovered right away. Eventually, memory fills up with temporary objects, and the HP-28 must perform some “memory packing” in order to continue. This packing consists of reviewing all of the temporary objects, discarding those that are no longer needed, then packing together the remaining objects into the minimum amount of memory. It is this memory packing that is taking place during the execution pauses that you observe.

Ordinarily, the execution pauses caused by packing are so short that they have little effect on your use of the calculator. However, there are some circumstances in which the packing can be very time consuming, effectively paralyzing the HP-28 for many seconds or even minutes. For example, if you enter 1000 numbers onto the HP-28S stack, executing MEM takes about five seconds (MEM always performs a memory pack). The worst situation, which you should be careful to avoid, involves the creation of large

temporary lists, and the extraction of the objects within the lists. After this sequence on the HP-28S,

```
1 1000 FOR x x NEXT 1000 →LIST LIST→
```

MEM takes about 10 minutes to execute, during which the keyboard does not respond (type-ahead still works, however). You can only interrupt the packing with a system halt (section 11.3.4), which also clears the stack.

If you find it necessary to work with large lists, you can avoid the delays due to memory packing by storing the lists in global variables before you take them apart. A similar warning applies to stack programs that enter a large number of objects onto the stack during their execution.

[The problem of memory packing is never serious on an unmodified HP-28C, in which the available memory is too small to cause lengthy packing delays. However, if you have added additional memory to your HP-28C, you will encounter delays, although under different circumstances than on the HP-28S. In particular, an HP-28C stack of 1000 objects requires much longer packing times than the same stack on the HP-28S, whereas the problem of large lists is much less severe on the HP-28C.]

The program TIMED listed in the preceding section executes MEM before starting the actual object timing. This minimizes the chance that packing will take place during the timing. However if the object's execution uses a lot of temporary objects, packing may take place one or more times anyway.

11.12 Recursive Programming

The unlimited depth of the HP-28 subroutine return stack provides that programs can not only call other programs without limit, but they can even call themselves any number of times. This feature permits so-called *recursive programming*, in which a repetitive calculation can be achieved by a compact program that iterates by calling itself.

A classic example of recursion is the calculation of a factorial $n! = n(n-1) \cdots 2 \cdot 1$. This definition can be restated in a recursive form:

If $n \leq 1$ then $n! = 1$; otherwise $n! = n(n-1)!$.

The following user-defined function embodies the recursive definition:

```
<< → n 'IFTE(n≤1,1,n*FCT(n-1))' >> 'FCT' STO
```


The function is defined in terms of itself, so that the name of the variable in which it is stored must match the name used within the defining procedure.

Recursion is not always the fastest or most memory efficient method of computing a result. For the factorial (ignoring the built-in FACT function), a FOR...STEP loop is better than the recursive version:

```
<< 1 SWAP OVER FOR n n * -1 STEP >>.
```

The looping done by FOR...STEP is faster than a program calling itself, and the program structure also takes care of incrementing n . However, in cases involving nested data structures, recursion may provide the only solutions.

The program RMINL listed in section 11.4 finds the minimum in a list of real numbers. Using recursion, it is a simple matter to extend that program so that any element of the input list can itself be a list containing numbers or additional lists, and so on. Here's the revised version:

RMINL <i>Recursive Minimum of a List</i>	
level 1	level 1
$\{x_1 \cdots x_n\}$	$x_{\min}$
<pre><< MAXR -NUM SWAP DUP SIZE 1 DUP ROT START GETI DUP TYPE IF 5 == THEN RMINL END 4 ROLL MIN 3 ROLLD NEXT DROP2 >></pre>	<pre> MAXR { x_i } n Initialize m (list index). Loop from 1 to n. x_{min} { x_i } m x_m Determine the type of object x_m. Lists are type 5. If it's a list, find its minimum. x_{min} { x_i } m </pre>

This program provides another illustration of the power of the unlimited stack. At the point in the program where RMINL calls itself, there is a list in level 1, which is the required argument. It doesn't matter that previous parts of the program have put other objects on the stack--they will still be in the right place when RMINL returns (to the rest of itself). RMINL returns one number to level 1, which is appropriate for the remainder

of the program. The initial list can be a list of lists of lists ..., nested indefinitely. For example:

```
{ 1 { 2 3 } { 4 { 5 { 6 7 8 } 9 0 } { 11 } } 12 } RMINL 0.
```

An additional example of simple recursive programming is provided by the Purge and Clusr programs listed in section 5.9.3 for use with HP-28S directories. More complicated examples of recursive programming are described in Chapter 12, where we discuss list objects, which play an important role in recursive programming. Examples of recursive programs using lists are given by the programs SORT and GSORT, listed in section 12.3.3. Lists also figure prominently in the recursive system of programs used for computing the determinants of symbolic matrices, described in section 12.6.3, and in the HP-28S program FIND, listed in section 5.7.3. The latter program features a self-recursive program created within a program and stored in a local variable.

A final note on recursive programs. Remember that if you change the name (variable) of a program that calls itself, you have to edit the program to replace all incidences of the old name with the new.

11.13 Additional Program Examples

11.13.1 Random Number Generators

The HP-28 command RAND generates uniformly distributed pseudo-random numbers x_i , where an x_i is equally likely to have any value in the range $0 < x < 1$. Using a uniform distribution generator, it is possible to generate random numbers with various other distributions.

11.13.1.1 Poisson Distribution

Assume x is a random variable with a uniform distribution $0 < x < 1$. If k is the smallest integer for which

$$\prod_{n=1}^{k-1} x_n \leq e^{-N}$$

is satisfied, then k is a random variable from a population conforming to the Poisson distribution with mean N . This distribution is defined as

$$P(k) = \frac{N^k}{k!} e^{-N},$$

where $P(k)$ is the probability of obtaining k events in an interval where the mean number of events is N .

The program POIS uses this algorithm to return one random value k , where the mean N is entered as a stack argument.

■ *Example.* Generate 100 random numbers from a Poisson distribution with mean 10, and compute the mean and standard deviation of the 100 numbers.

■ *Solution.* Use $\Sigma+$ to accumulate the random numbers into ΣDAT , then use MEAN and SDEV.

```
.54321 RDZ CLΣ 1 100 START 10 POIS Σ+ NEXT
```

generates the numbers (include the sequence .54321 RDZ if you want to check your results against those shown below). After executing the sequence, you can compute the sample statistics:

MEAN 9.7400

SDEV 3.0867

The nominal standard deviation of a Poisson distribution is $\sqrt{N}$, which is ≈ 3.1623 for $N = 10$.

POIS <i>Poisson Generator</i>	
<i>level 1</i>	<i>level 1</i>
N	k
<pre><< NEG EXP -1 1 DO SWAP 1 + SWAP RAND * UNTIL DUP 4 PICK ≤ END DROP SWAP DROP >></pre>	<pre>exp(-N) Start k at -1; the product at 1. Increment k. Multiply by the next x. Keep going until the product is ≤ exp(-N). Return k.</pre>

11.13.1.2 Normal Distribution

Assume x is a random variable with a uniform distribution $0 < x < 1$. With a definition of y as

$$y = \sqrt{-2\ln x_i} \cos(2\pi x_j),$$

where x_i and x_j are randomly drawn from the population of x , y is a random variable from a population conforming to the normal (Gaussian) distribution with mean 0 and standard deviation 1. The normal distribution for a variable with mean $\bar{y}$ and standard deviation σ is

$$P(y) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y-\bar{y})^2}{2\sigma^2}\right),$$

where $P(y) dy$ is the probability of obtaining a value in the range between y and $y + dy$.

NORM <i>Normal Distribution Generator</i>	
	level 1
	y_i
<pre><< RAND LN -2 * √ RAND 2 * π →NUM * RAD COS * >></pre>	x_i $\sqrt{-2\ln x_i}$ x_j $\cos(2\pi x_j)$ y

NORM leaves radians mode active.

You can obtain random numbers y'_i from a normal distribution with mean $\bar{y}$ and standard deviation σ by multiplying the values y_i obtained with NORM by σ and adding $\bar{y}$. The program MNORM returns such random numbers y'_i , where the mean and standard deviation are specified on the stack:

MNORM <i>Modified Normal Distribution Generator</i>			
level 2	level 1		level 1
$\bar{y}$	σ	$\rightarrow$	y'_i
<pre><< NORM * + >></pre>	y'		

MNORM leaves radians mode active.

■ *Example.* Create a Σ DAT matrix that contains points $[x_i, y_i]$ representing a “noisy” straight line:

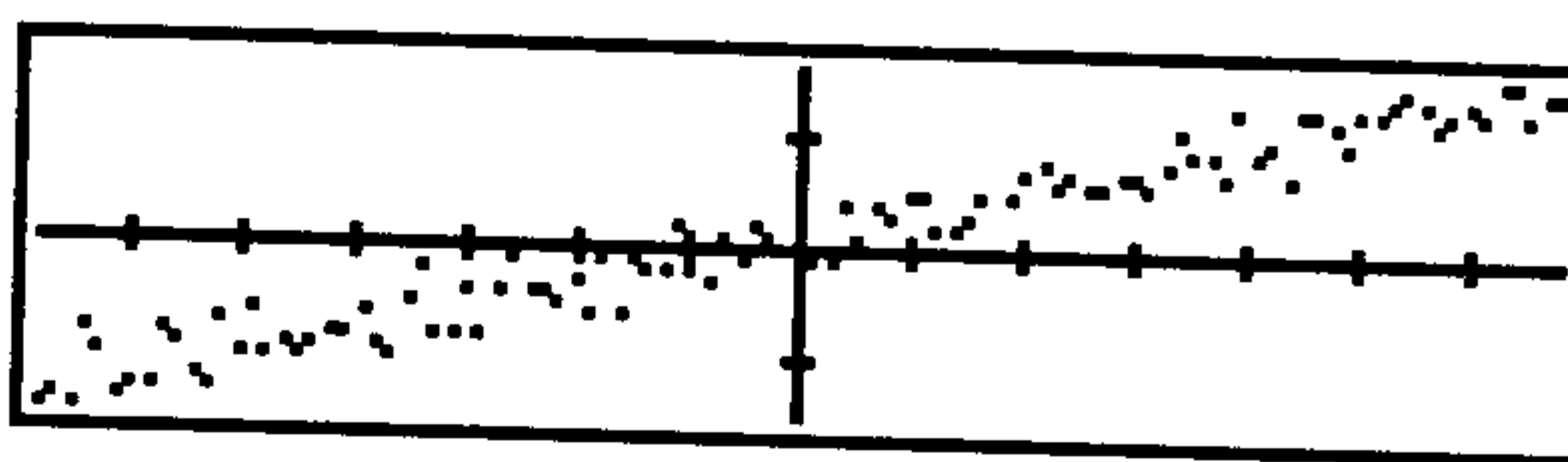
$$y_i = 0.5x_i + b_i,$$

where b_i is a normally distributed random variable with mean 1 and standard deviation 3, and the x_i are the integers -50 through $+50$.

■ *Solution:*

<pre>.54321 RDZ CLΣ -50 50 FOR x x 1 3 MNORM x 2 / + { 2 } →ARRY Σ+ NEXT</pre>	Random number seed. Initialize ΣDAT. x from -50 to $+50$. x_i y_i. Store the point.

You can create a plot of this data by executing `SCLΣ DRWΣ`:



11.13.2 Prime Numbers

The program PRIMES returns a list of the first n prime numbers (not counting 1), where n is specified on the stack. The program demonstrates the use of stack flags (section 10.3) to “remember” the results of tests, so that those results can be used for later decisions.

This program starts with a list of three prime numbers 2, 3, and 5, then successively tests integers m greater than these to see if they are prime by dividing each by all prime numbers n_i for which $n_i \leq \sqrt{m}$. If any quotient is an integer, m is not prime, and is discarded. If m is prime, it is appended to the current list of primes. The process continues until the list grows to the specified size.

You can obtain a significant economy in the execution of this process by observing that you don’t need to test every integer explicitly, but only those in the series 7, 11, 13, 17, 19, ..., obtained by alternately adding 2 and 4. All integers not in this series are divisible by 2 or 3, and so are not prime.

PRIMES <i>Find Prime Numbers</i>	
<i>level 1</i>	<i>level 1</i>
<i>n</i>	<i>{ primes }</i>
<pre> << 7 → x << { 2 3 5 } 1 SF DO 2 SF 3 DO GETI x OVER / UNTIL IF SWAP OVER > DUP THEN SWAP DROP ELSE SWAP IF FP NOT THEN 2 CF NOT END END END DROP IF 2 FS? THEN x 1 →LIST + END IF 1 FS?C THEN 4 ELSE 2 1 SF END x + 'x' STO UNTIL DUP2 SIZE ≤ END SWAP DROP >> >> </pre>	x is the next candidate number; start with 7 First three primes. Flag 1 determines increment size. Main loop to test x. Flag 2 set means x may be prime. Start with $n=3$ (3rd number in the list). Inner loop—divide x by primes $\leq \sqrt{x}$ { primes } n p_n x/p_n Keep going until $FP(x/p_n) = 0$ or $p_n > x/p_n$ {primes} n x/p_n flag If the test is true... ...then return true to stop the loop. Otherwise, check if evenly divisible. If the fractional part is zero... If x is prime... ... add it to the list If flag 1 is set... ...then add 4; ...else add 2 and set the flag. Increment x. Repeat until list is the desired length. Leave the list on the stack.

PRIMES uses flags 1 and 2.

The basic structure of PRIMES is as follows:

```

DO
  DO Divide a number by the next prime from the list.
  UNTIL (1) either a quotient is a non-integer.
    or
    (2) the prime is bigger than the number's square root.
  END
UNTIL enough primes are found.
END

```


The combination of tests (1) and (2) is complicated by the fact that there is no point in making test (2) if test (1) is true. In PRIMES, therefore, the (flag) result of test (1) is used twice, once by an IF structure than contains test (2), and again to determine whether to continue through the list of prime number divisors.

11.13.3 Simultaneous Equations

Consider the set of simultaneous linear equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= c_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= c_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= c_n, \end{aligned}$$

where there are n equations in n unknowns $x_1 \cdots x_n$. The a_{ij} are the coefficients of the unknowns, and the c_i are the constant terms.

These equations are straightforward to solve on the HP-28. Defining the coefficient matrix

$$\mathbf{A} = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \ddots & \\ & & & a_{nn} \end{vmatrix},$$

and the unknown and constant vectors

$$\vec{x} = \begin{vmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{vmatrix} \quad \vec{c} = \begin{vmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{vmatrix},$$

then the set of simultaneous equations can be represented as the matrix equation

$$\mathbf{A} \vec{x} = \vec{c}.$$

The solution can be found by premultiplying both sides of the equation by the inverse of $\mathbf{A}$:

$$\vec{x} = \mathbf{A}^{-1} \vec{c}.$$

On the HP-28, you can obtain this solution by entering the constant vector $\vec{c}$ into level 2 and the coefficient matrix $\mathbf{A}$ into level 1, then executing / (divide). This returns the unknown vector $\vec{x}$ to level 1.

This method is very simple, but has the drawback that it requires *you* to determine the coefficients and constants from the equations, and enter them in a very specific order. The program SIMEQ below does all of this work for you. SIMEQ expects to find a list of names in level 1, preceded in higher levels by as many equations as there are names in the list. The specified names indicate which of the variable names in the equations are the unknown variables--all other variables that appear in the equations must have numerical values (via $\rightarrow\text{NUM}$). The equations may appear in any order, and there are no restrictions on the form of the equations, except that they must be linear in the unknown variables.

SIMEQ determines the constant terms in the equations by setting all of the unknowns to zero, then evaluating the equations. It next subtracts the constants from the equations, and determines the coefficients by assigning the value 1 to one unknown variable at a time, and evaluating the equations. The coefficients are combined into a matrix, and the constants into a vector so that the vector of unknowns can be obtained by dividing. Finally, the values of the unknowns are stored in the corresponding variables.

- *Example.* Five packages are weighed in pairs, yielding the weights 90, 110, 120, 140, 120, 130, 150, 150, 170, and 180 pounds. What are the weights of the individual packages?
- *Solution.* Call the unknown weights A , B , C , D , and E , where A is the lightest weight package and E is the heaviest. Then the lightest combination is A and B , so

$$A + B = 90 \text{ lbs.}$$

The next lightest combination must be A and C :

$$A + C = 110 \text{ lbs.}$$

Similarly, the heaviest two combinations are

$$D + E = 180 \text{ lbs,}$$

and

$$C + E = 170 \text{ lbs.}$$

SIMEQ <i>Simultaneous Equations</i>	
<i>level n ... level 2</i>	<i>level 1</i>
<i>'equation₁' ... 'equation_n'</i>	<i>{ name₁ ... name_n }</i>
<pre> << DUP SIZE → v n << n →LIST → e << 1 n FOR x 0 v x GET STO NEXT e LIST→ 1 SWAP START n ROLL →NUM NEG NEXT n →LIST → c << 1 n FOR x 1 v x GET STO e LIST→ 1 SWAP FOR i n ROLL →NUM c i GET + NEXT 0 v x GET STO NEXT n DUP 2 →LIST →ARRY TRN c LIST→ 1 →LIST →ARRY SWAP / ARRY→ DROP n 1 FOR m v m GET STO -1 STEP >> >> >> </pre>	Save the list of names in <i>v</i>, and the number of names in <i>n</i>. Combine the equations into a list, and save in <i>e</i>. Store zero in each unknown variable. Put the equations on the stack. Compute each constant term. Combine the constants into a list, and save in <i>c</i>. For each variable... Assign the value 1 to the variable. Put the equations on the stack. For each equation... Evaluate the equation, and subtract the constant term, leaving the coefficient. Reset the variable to 0. Combine all the coefficients into a square matrix. Convert the constant list in a vector. Compute the unknown vector. Put the values on the stack. Store each value in its variable.

Finally, you can observe that the total weight of all the combinations must be four times the total weight of the packages:

$$4(A + B + C + D + E) = 1360 \text{ lbs.}$$

These are the five equations you need to solve the problem:

'A+B=90' **ENTER**

'A+C=110' **ENTER**

'D+E=180' **ENTER**

'C+E=170' **ENTER**

'4*(A+B+C+D+E)=1360' **ENTER**

puts the equations on the stack; then

{ A B C D E } **SIMEQ**

solves the equations: $A=40$ lbs, $B=50$ lbs, $C=70$ lbs, $D=80$ lbs, and $E=100$ lbs.

11.13.4 Infinite Sums

In section 11.10 we presented a program INFSUM that computes an infinite sum of terms defined by a separate program. For some sums, it is more accurate to compute each term T_n from the previous one T_{n-1} , rather than computing each term independently. The programs PTINFSUM and XPTINFSUM (listed in section 11.12.4.3) use this approach. The first program PTINFSUM is a variation of INFSUM, for which you supply a stack program that computes T_n as a function of n and T_{n-1} . PTINFSUM also requires you to specify the initial value n_0 of the index, and the value of the first term T_{n_0} .

■ *Example.* Compute $\sum_{n=1}^{\infty} \frac{n^3}{2^n}$.

■ *Solution:* In this case, $T_n = \frac{1}{2} \left(\frac{n}{(n-1)} \right)^3$, $n_0 = 1$, and $T_1 = 0.5$. Thus,

```
<< DUP 1 - / 3 ^ 2 / * >> .5 1 PTINFSUM 25.9999999997.
```

Many mathematical functions can be computed from an infinite sum for which the terms are functions of a variable as well as of the summation index. The program XPTINFSUM is a further variation of PTINFSUM, in which the value of a variable is also an input argument, in addition to the arguments required by PTINFSUM. The program that computes T_n from T_{n-1} and n can also be a function of the variable.

The programs SI and CI (listed in section 11.12.4.3), illustrate the use of XPTINFSUM to compute sine and cosine integrals, respectively. The series expansions for these integrals are taken from M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions* (National Bureau of Standards, 1964).

11.13.4.1 Sine Integral

The *sine integral* $Si(x)$ is defined as follows:

$$Si(x) = \int_0^x \frac{\sin t}{t} dt$$

The integral can be computed from the infinite series:

$$Si(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)(2n+1)!}$$

for $x > 0$, and $Si(x) = -Si(-x)$ for $x < 0$.

The program SI uses XPTINFSUM to compute this sum, with the assignments $n_0 = 0$, $T_0 = x$, and

$$T_n = T_{n-1} \left(-\frac{(n - 1/2)x^2}{4n(n + .5)^2} \right)$$

Since T_n is a function of x^2 , SI saves repeated computation of the square of x by using x^2 rather than x as the variable argument for XPTINFSUM.

Examples:

.5 SI  .493107418043

3 SI  1.848652528

11.13.4.2 Cosine Integral

The *cosine integral* $Ci(x)$ is defined by

$$Ci(x) = \gamma + \ln x + \int_0^x \frac{\cos t - 1}{t} dt,$$

where $\gamma = .5772156449$ (Euler's constant). $Ci(x)$ can be calculated from the infinite series

$$Ci(x) = \gamma + \ln x + \sum_{n=1}^{\infty} \frac{-1^n x^{2n}}{2n(2n)!}$$

for $x > 0$, and

$$Ci(x) = Ci(-x) - i\pi \text{ for } x < 0.$$

The parameters for XPTINFSUM are $n_0 = 1$, $T_1 = -x^2/4$, and

$$T_n = T_{n-1} \left(-\frac{x^2(n-1)}{2n^2(2n-1)} \right).$$

T_n is a function of $-x^2$, so Ci uses $-x^2$ rather than x as the variable argument for XPTINFSUM.

Examples:

$$0.5 \quad Ci \quad \rightarrow \quad -.177784078808$$

$$3 \quad Ci \quad \rightarrow \quad .11962978602$$

11.13.4.3 Sum Programs

PTINFSUM <i>Infinite Sum from Previous Term</i>				
level 3	level 2	level 1		level 1
<< term >>	T_{n_0}	n_0	☐	sum
<< ROT → term << OVER SWAP DO 1 + SWAP OVER term →NUM SWAP ROT 3 PICK OVER + DUP 5 ROLLD UNTIL == END DROP2 >>			Save << term >>. T_{n_0} T_{n_0} n sum(n) T_n n Increment n . sum($n-1$) n T_n n T_n sum($n-1$) sum(n) sum(n) T_n n sum($n-1$) sum(n) Repeat until the sum is unchanged.	

The argument << term >> must have the logical form << → t n 'term(t,n)' >>.

XPTINFSUM <i>Infinite Sum in x from Previous Term</i>					
level 4	level 3	level 2	level 1		level 1
<< term >>	T_{n_0}	n_0	x	☐	sum
<< 4 ROLL → x term << OVER SWAP DO 1 + SWAP OVER x term →NUM SWAP ROT 3 PICK OVER + DUP 5 ROLLD UNTIL == END DROP2 >>			Save << term >> and x . T_{n_0} T_{n_0} n_0 sum(n) T n Increment n . sum($n-1$) n T_{n-1} n x sum($n-1$) n T_n T_n n sum($n-1$) sum(n) sum(n) T_n n sum($n-1$) sum(n) Repeat until the sum is unchanged.		

The argument << term >> must have the logical form << → t n x 'term(t,n,x)' >>.

CI		Cosine Integral	
level 1		level 1	
x		Ci(x)	
<pre><< DUP ABS DUP LN SWAP SQ NEG DUP 4 / SWAP 1 SWAP << SWAP → n << 2 / n SQ / n 1 - * n 2 * 1 - / * >> >> 4 ROLLD XPTINFSUM + .5772156649 + SWAP IF 0 < THEN i π * - END >></pre>		$\left x \ln x - x^2 \right $ $\left x \ln x - x^2/4 \quad 1 \quad -x^2 \right $ Start of << term >>. End of << term >>. $\left x \ln x - \ll term \gg x T_0 n_0 x^2 \right $ $\left x \ln x - \sum \right $ Subtract $i \pi$ if $x < 0$. Ci(x).	

12. Arrays and Lists

The HP-28 *array* and *list* object types allow you to deal with collections of numbers or other objects as single units, as well as to access the individual objects in the collections. You are probably familiar with arrays--vectors and matrices--from mathematics. Arrays are one-dimensional (vectors) or two-dimensional (matrices) ordered sets of numbers that satisfy certain rules of arithmetic and transformation properties. However, you may find the idea of a *list* as a useful computational tool to be a new concept, since it has no obvious mathematical counterpart, and there is nothing similar in the HP-41 or other calculator languages. (Lists will be very familiar to you if you have studied LISP, or a similar computer language.)

12.1 Arrays

The particular contribution of the HP-28 to calculator array computation is its ability to manipulate arrays as self-contained units. This means, for example, that you can perform array arithmetic on the stack using the same steps and commands as you would for real number arithmetic. Programs can use arrays as input and return arrays as output; the arrays themselves contain all of the dimensional information that the programs need to deal with the data in the arrays. This is in distinct contrast to the HP-41, for example, in which an "array" is only a series of consecutive data registers, and the specification of the location and size of the array is stored and interpreted separately from the array data.

The mathematical operations that the HP-28 provides for matrices and vectors are not remarkable. The strength of the HP-28 is the ease with which you can apply the operations to the arrays. We will not dwell on the mathematical commands here, since they are described adequately in the owner's manuals. Instead we will focus on the array manipulation commands.

- To assemble a series of numbers on the stack into an array, use $\rightarrow$ ARRAY.

1 2 3 4 { 2 2 } $\rightarrow$ ARRAY $\Rightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

Note that the level 1 argument of $\rightarrow$ ARRAY (the list { 2 2 } in this example) determines how many numbers are taken from the stack to form the array, and the dimensions of the array. If the list is { n }, n numbers are used to form an n -element vector. If the list is { n m }, $n \cdot m$ numbers are combined into an $n \times m$ matrix (the HP-28S also permits a real number n --not in a list--to specify an n -element vector). The order in which the elements are placed on the stack is called *row-order*. This order has element 1 or 1-1 in the highest stack level, followed by the

elements of the first row in left-to-right order, then by the row 2 elements, if any, and so forth, ending with the last element in row n .

- To take an array apart, use **ARRAY→**. Reversing the previous example:

$$\left[\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \right] \text{ ARRAY→ } \Rightarrow 1 \ 2 \ 3 \ 4 \ \{2 \ 2\}$$

ARRAY→ returns the elements of an array as individual numbers in row order, and leaves the number of elements in level 1.

- To determine the dimensions of an array, use **SIZE**.

$$\left[\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \right] \text{ SIZE } \Rightarrow \{3 \ 2\}$$

- To extract individual numbers from an array, use **GET** or **GETI** (section 5.2).

$$\left[\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \right] \{2 \ 1\} \text{ GET } \Rightarrow 3.$$

- To substitute numbers into an array, use **PUT** or **PUTI** (section 5.3.2).

$$\left[\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \right] \{2 \ 1\} \ 8 \ \text{PUTI} \Rightarrow \left[\begin{bmatrix} 1 & 2 \\ 8 & 4 \end{bmatrix} \right] \{2 \ 2\}$$

the $\{2 \ 1\}$ element in the array is replaced with a new value 8, and the next index is returned.

PUTI provides a convenient means of entering numbers into an array for cases in which you need to compute the numbers instead of just entering them into the command line.

- *Example.* Enter the matrix

$$\begin{vmatrix} 1/3 & -1/6 \\ -1/3 & 1 \end{vmatrix}$$

Keystrokes:

■ **MODE** 2 **FIX**

{ 2 2 }

Stack:

Set number display mode.

Matrix dimensions.


```

3 █ 1/x
█ ARRAY NEXT █ CON █ 1:      [[ 0.33  0.33 ]   Create a starting matrix.
                               [ 0.33  0.33 ]]

{ 1 2 } 6 █ 1/x █ CHS
█ PREV █ PUTI █ 2:      [[ 0.33  -0.17 ]   Second element.
                               [ 0.33   0.33 ]]
                               1:      { 2 1 }

3 █ 1/x █ CHS █ PUTI █ 2:      [[ -0.33  -0.17 ]   Third element.
                               [ -0.33   0.33 ]]
                               1:      { 2 2 }

1 █ PUTI █ 1:      [[ -0.33  -0.17 ]   Done.
                               [ -0.33   1.00 ]]

```

The program MINOR (listed on the next page) illustrates the use of array manipulation commands. MINOR computes the *nm minor* of a matrix, which is defined as the original matrix with its *n*th row and *m*th column removed. Assuming that MINOR will start with the original matrix in level 3, the row number *r* in level 2, and the column number *c* in level 1, we can sketch a preliminary version of MINOR:

```
<< 3 ROLLD DELROW SWAP DELCOLUMN >>
```

DELROW must be a subroutine that removes the *r*th (level 1) row of a matrix (level 2); DELCOLUMN removes the *c*th row. However, you can observe that removing a column of a matrix is the same as removing a row of the transposed matrix, so that a program that removes a row can do both jobs if combined with the transpose command TRN. We choose to work with rows rather than columns because ARRY→ puts elements on the stack in row order, so that it is easier to delete the elements of a row.

The programs in section 12.6 contain several additional examples of the uses of array manipulation commands.

A note about vectors: Although HP-28 vectors are displayed with the elements laid out horizontally, the vectors actually have the properties of column (*contravariant*) vectors. This means, for example, that an *n*-element vector $\vec{v}$ is suitable for pre-multiplication ($A \cdot \vec{v}$) by an $m \times n$ matrix *A*. The vectors are displayed horizontally in order to show as many elements as possible on the display. You can represent row (*covariant*) vectors as $1 \times n$ matrices.

MINOR <i>Minor of a Determinant</i>				
level 3	level 2	level 1		level 1
[[matrix]]	r	c	✂	[[matrix']]
<< 3 ROLLD DELROW TRN SWAP DELROW TRN >>			c [[matrix]]	
			r Remove the rth row. Remove the c column. Transpose back again.	

DELROW					Delete a Matrix Row	
level 2		level 1		level 1		
[[matrix]]		n		[[matrix']]		
<pre><< → r << ARRAY→ LIST→ DROP → n m << n r - m * →LIST → s << m DROPN s LIST→ DROP >> n 1 - m 2 →LIST →ARRAY >> >> >></pre>			Store the row number. Put the array elements individually on the stack. Save the dimensions in r and c. Save the last $(n-r)m$ elements in a list s. Discard m elements. Recover the saved elements. The new array has dimensions $(n-1) \times m$. Make the result array.			

12.2 Lists

In this section, we'll review the general ideas of list objects, and study their application by means of examples.

A list is an object that is simply a collection of other objects (called the *elements* of the list). At first glance, a list resembles a program, in that it contains a sequence of objects. However, unlike the objects in a program, those in a list are *not* intended for execution while they remain in the list. Also, a list contains no structures (in fact, you can't even enter program structure words like IF or START into a list unless they are part of a program object). Whereas a program is intended to represent a calculation, a list is designed to hold objects that can be the input or output of a calculation.

Lists also resemble vectors, since they are both one-dimensional arrays of objects. You can create either a list or a vector out of a series of numbers (using `→LIST` or `→ARRAY`). The difference is that in a list, the numbers do not necessarily have any particular association, whereas in a vector, they may be considered as the coordinates of a geometrical point, and hence are subject to various arithmetic operations and transformation rules.

12.2.1 List Operations

The `LIST` menu contains several commands that enable you to manipulate lists and their elements. The commands are quite similar to those used for array operations.

- To assemble objects into a list, use `→LIST`.

```
1 (1,2) 'A+B' 3 →LIST → { 1 (1,2) 'A+B' }.
```

Note that the level 1 argument of `→LIST` (the 3 in this example) determines how many objects are taken from the stack to be combined into the list.

- To take a list apart, use `LIST→`.

```
{ 1 (1,2) 'A+B' } LIST→ → 1 (1,2) 'A+B' 3
```

`LIST→` returns the elements of the list as separate stack objects, and leaves the number of elements in level 1.

- To determine the number of elements in a list, use `SIZE`.

```
{ 1 (1,2) 'A+B' } SIZE → 3.
```

- To substitute objects into a list, use `PUT` or `PUTI`.

```
{ 1 (1,2) 'A+B' } 2 "ABC" PUT → { 1 "ABC" 'A+B' },
```

where the second element (1,2) in the initial list is replaced with the string "ABC". `PUTI` makes the substitution like `PUT`, but also leaves the index of the next element in level 1.

- To pull individual objects out of a list, use `GET` or `GETI`.

```
{ 1 (1,2) 'A+B' } 2 GET → (1,2).
```

- To combine (concatenate) lists, use `+`.

$$\{1\ 2\ 3\ 4\}\ \{5\ 6\ 7\ 8\} + \Rightarrow \{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\}.$$

- To add stack objects to a list, combine $\rightarrow$ LIST and $+$:

$$\{1\ 2\ 3\ 4\}\ 5\ 6\ 7\ 8\ 4\ \rightarrow\text{LIST} + \Rightarrow \{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\},$$

the same result as the previous example. On the HP-28S, you can add a single object (that is not a list itself) to a list by using $+$:

$$1\ \{2\ 3\ 4\} + \Rightarrow \{1\ 2\ 3\ 4\}$$

$$\{2\ 3\ 4\}\ 1 + \Rightarrow \{2\ 3\ 4\ 1\}$$

- Since a list is an object, you can include lists within other lists. Notice the distinction between

$$\{1\ 2\ 3\ 4\}\ \{5\ 6\ 7\ 8\} + \Rightarrow \{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\}$$

and

$$\{1\ 2\ 3\ 4\}\ \{5\ 6\ 7\ 8\}\ 1\ \rightarrow\text{LIST} + \Rightarrow \{1\ 2\ 3\ 4\ \{5\ 6\ 7\ 8\}\}.$$

- To extract sublists from a list, use SUB. For example, the sequence

$$2\ \text{OVER}\ \text{SIZE}\ \text{SUB}$$

takes a list from level 1 and returns a shorter list consisting of the original list minus its first element (like the LISP function CDR). Thus,

$$\{A\ B\ C\}\ 2\ \text{OVER}\ \text{SIZE}\ \text{SUB} \Rightarrow \{B\ C\}.$$

- On the HP-28S, you can find an object in a list by using POS:

$$\{A\ B\ C\}\ 'A'\ \text{POS} \Rightarrow 1.$$

POS can take a significant amount of time for large lists. Its execution time increases roughly with the square of the element number returned.

12.3 List Applications

The basic ideas of the use of the HP-28 object stack carry over into the principles and applications of list objects. A list is like an auxiliary stack, in which you can store and

retrieve an indefinite number of objects, with no restrictions on the order or type of objects in the list. To illustrate this point, try the following:

1. Enter several objects of any types onto the stack.
2. Press **■** **STACK** **NEXT** to activate the second row of the STACK menu.
3. Press **≡DEPTH≡**. This returns the number of objects currently on the stack.
4. Press **≡-LIST≡**. Now you have only one object on the stack, a list containing all of the previous stack objects. Note that the objects are present in the list in the same order in which they were originally entered into the stack. The object that was in the highest stack level is the first element in the list; the object that was in level 1 (prior to **≡DEPTH≡**) is the last element. The list thus preserves an image of the original stack.
5. Save the list: 'OLD' **STO**. The stack is now empty.
6. Imagine carrying out any number of calculations, which leave various objects on the stack. Discard these objects with **CLEAR**, then enter

OLD LIST→ DROP.

This restores the stack as it was after step 1.

The ability to "freeze" a copy of the stack, store it away, then retrieve it later, is a useful list application in itself. But the main point of the example is to bring out the similarities between the stack and a list object, which suggests how you might use lists. The stack provides a medium for the ordered presentation of objects as input arguments for procedures (built-in or user-created), and for receiving the result objects. Lists can be used for the same purposes, especially for cases where juggling mixtures of input, intermediate, and output objects during the course of a calculation can become complicated.

A list can also play a role like that of numbered registers in the HP-41. That is, you can create a list of any size, store it in **USER** memory to give it a name, then use 'name' *n* **PUT** to store an object in the *n*th "register," and 'name' *n* **GET** to recall it. An advantage of lists over HP-41 registers for this purpose is that you can have as many such lists as you want. Also, you can manipulate the group of objects stored in a list as a single entity, which is rather difficult on the HP-41 (that calculator provides certain register-block commands, but they lack the simplicity and power of HP-28 list commands).

To summarize, lists are a valuable programming tool for any situation in which the number of objects with which a program has to deal is not specified at the time the program is written. When a program works with a definite number of objects, it is appropriate to store those objects in variables, or to manipulate them on the stack as

individual objects. But when you don't know in advance how many objects are to be handled, the best approach by far is to manage the objects together in a list. We will give some examples of this concept in the next sections.

12.3.1 Input Lists

Certain HP-28 commands provide examples of the use of lists to combine several input objects into a single argument. There are two basic reasons for this approach:

1. *To provide flexibility along with uniformity.* For example, consider the command CON, which creates an array in which all elements have the same value. CON requires two pieces of information: 1) the common value for the elements, and 2) the dimensions of the array. The first is easy; the value is specified by a real or complex number in level 1. The second is a little more difficult, since an array can either be a one-dimensional vector, or a two-dimensional matrix. The use of a list as the level 2 argument for CON allows CON to handle both matrices and vectors. If the level 2 list contains one number, CON creates a vector; if the list contains two numbers, CON creates a matrix. If the dimensions were not combined into a list, there would have to be two versions of CON: one that takes two real numbers as arguments--the value and the vector dimension; and one that takes three numbers--the value and two matrix dimensions.
2. *To reduce the number of separate arguments.* $\int$, for example, uses three argument objects to specify as many as five inputs (section 9.9.3). All three forms of integration provided by $\int$ require an integrand procedure in level 3, and an accuracy specifier in level 1. The level 2 argument must be a name for symbolic integration, a list containing two limits of integration for implicit variable numerical integration, or a list with a name and two limits for explicit variable integration. By using a list for the latter two cases, $\int$ satisfies the HP-28 convention of using a maximum of three arguments, which allows the USE operation in CATALOG to display all of its arguments in three display lines, leaving the fourth for the menu key labels. The limit of three arguments also keeps the number of argument type combinations that a command must check within reasonable bounds.

Of these two reasons, the first is the only one of significance as a model for the use of lists as input arguments for user programs. That is, lists are ideal for situations where you have an indefinite number of inputs. An example of this is provided by the program MINL (section 11.4), that finds the minimum among a series of numbers in a list. The program is written for series of any length--it has only to execute SIZE on the input list to determine how many numbers it needs to compare. Furthermore, during its execution, the numbers remain in the list, except for when they are extracted one-by-one from the list for the comparisons. Keeping track of that single list, which could be stored in a global or local variable if necessary, is much simpler than trying to maintain the series

of numbers as separate stack objects. If you are not yet convinced of the utility of lists, try writing a version of MINL that uses no lists (or arrays). See also the recursive program RMINL, on page 258.

12.3.1.1 HP-28S Command List Arguments

On the HP-28S, any command that uses an argument list containing one or more real numbers allows you to substitute other types of objects for the numbers. The substitute objects must evaluate (by means of $\rightarrow\text{NUM}$) to real number values. In particular, this means you can use symbolic values (names or expressions), or even programs, rather than specific numerical values. For example, to compute

$$\int_{-\pi}^{\pi} \cos x \, dx$$

you can execute

`'COS(X)' { X '-π' π } .00001 ∫.`

Or you can extract an element from an array with the sequence `{ A B } GET`, where A and B name variables containing the element indices.

This capability can lead to some convoluted executions when argument lists contain (directly or indirectly) programs that manipulate the stack. You can predict the execution in such cases as follows:

1. Empty lists cause the Bad Argument Value error.
2. Lists containing only real numbers go directly on to the computation part of the command.
3. When a list contains elements other than real numbers:
 - a. The stack depth (less the list) is recorded.
 - b. Each non-real number list element is evaluated numerically ($\rightarrow\text{NUM}$). After each evaluation, if the resulting stack is empty, the error Too Few Arguments is reported. If the resulting level 1 object is not a real number, the Bad Argument Type error is reported.
 - c. If the stack depth has decreased, the Too Few Arguments is returned. Otherwise, the new objects, plus any excess, are combined back into a list.
 - d. The command execution is started over again with the new list.

Errors that occur during evaluation of user procedures within the argument list identify the guilty command and return its arguments, as usual on the HP-28S. However, other

errors that occur in step 2 do not identify any command.

If a non-numeric list is used as the index argument for GETI or PUTI, the incremented index list is returned with real number indices.

12.3.2 Output Lists

Just as you can use a list to combine an indefinite number of *input* objects into a single argument, you can use a list to receive the multiple-object *output* of a program. This approach makes it easy to manipulate a program's output--either to save it in a variable, or to use it as the input for another program.

■ *Example.* For any integer n , compute the first $n+1$ terms F_n of the Fibonacci series. This series is defined as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

FIB <i>Fibonacci Series Generator</i>	
<i>level 1</i>	<i>level 1</i>
n	$\{ 0 \ 1 \ \cdots \ f_n \}$
<pre><< { 0 1 } SWAP DUP 1 IF > THEN 0 1 3 4 ROLL 1 + START DUP ROT + ROT OVER 1 →LIST + 3 ROLL NEXT DROP2 ELSE DROP END >></pre>	Start the list with F_0 and F_1. If n is ≤ 2, quit. Initial values F_{n-2} and F_{n-1}. From 3 to n... $F_{n-2} + F_{n-1}$. Add F_n to the output list. { ... F_n } F_{n-2} F_{n-1}

12.3.3 Lists of Intermediate Results

When a program contains loop structures, or is written recursively, it is usually necessary to ensure that the stack has the same configuration at each iteration. A particularly convenient means of achieving this is to use a list as an auxiliary data stack, to hold an indefinite number of intermediate results in a constant position on the stack.

The program SORT illustrates the use of lists of intermediate results. SORT orders a list of numbers (or strings), so that the smallest (most negative, or alphabetically first) object is moved to the start of the list, and so on to the largest (alphabetically last) object as the last element. SORT uses a recursive algorithm that can be summarized as:

1. Remove the first object from the list and separate the remaining objects into two lists, one containing objects that are smaller than the first object, and the other containing objects larger than the first.
2. Sort the two lists using the same algorithm.
3. Combine the results back into a single list, with the sorted "smaller" objects first, followed by the original first object, then the sorted "larger" objects.

SORT	
Sort a List in Increasing Order	
level 1	level 1
{ list }	{ ordered list }
<pre> << IF DUP SIZE 1 > THEN LIST→ DUP 1 + ROLL 1 →LIST LIST→ DROP → x << {} {} ROT 1 SWAP 1 - START ROT DUP 1 →LIST SWAP x IF < THEN ROT + SWAP ELSE + END NEXT SORT SWAP SORT x 1 →LIST + SWAP + >> END >> </pre>	If the list has fewer than 2 elements, just return. Put the objects on the stack. Get the first object. Save the first element as x.[†] Initialize "less" and "greater" lists. Iterate for n-1 elements: Get the next element. { list } element x ‡ Element < x, so add to first list. Element ≥ x, so add to second list. Sort the first list. Sort the second list. ‡ Combine the lists.

[†]The sequence 1 →LIST LIST→ DROP saves memory by separating x from the original list. See section 12.4.

[‡]The sequence 1 →LIST is unnecessary on the HP-28S.

■ *Example.*

{ 5 1 2 4 3 } SORT → { 1 2 3 4 5 }.

The algorithm used by SORT is not specific to numerical ordering; you can rewrite SORT for other types of sorting by replacing the < comparison with any other test or sequence of tests. A more general approach is taken by the program GSORT, which sorts a list of objects according to a test defined by another program that is supplied as a second argument to GSORT.

GSORT <i>General-purpose Sort</i>	
level 2	level 1
{ list }	<< test >> → { list }
<pre> << → test << IF DUP SIZE 1 > THEN LIST→ DUP 1 + ROLL 1 →LIST LIST→ DROP → x << { } { } ROT 1 SWAP 1 - START ROT DUP 1 →LIST SWAP x IF test EVAL THEN ROT + SWAP ELSE + END NEXT test GSORT SWAP test GSORT x 1 →LIST + SWAP + >> END >> >> </pre>	Save test program as <i>test</i>. If the list has fewer than 2 elements, just return. Get the first element. Save the first element as x. Initialize "true" and "false" lists. Iterate for n-1 elements: Get the next element. † Make the test. Test is true, so add element to true-list. Test is false, so add element to false-list. Sort the true-list. Sort the false-list. † Combine the lists.

†The sequence 1 →LIST is unnecessary on the HP-28S.

To use GSORT, enter the unsorted list of objects, followed by a program *test-program* that represents a logical test. *Test-program* should work like this:

*object*₁ *object*₂ *test-program* → *flag*.

Flag should be *true* if *object*₁ is to precede *object*₂, or *false* otherwise. GSORT sorts the list so that the sequence

*object*_n *object*_{n+1} *test-program*

will return a *true* flag for any two consecutive objects *object*_n and *object*_{n+1} in the list (unless the order is ambiguous). For example, the numerical ordering performed by SORT is represented by the program << < >>; therefore << < >> GSORT is equivalent to SORT. Other examples:

- << > >> GSORT sorts numbers or strings in decreasing numerical or alphabetical order.
- << ABS SWAP ABS > >> GSORT sorts in order of increasing absolute value.
- << SIZE SWAP SIZE > >> GSORT sorts strings or lists in order of increasing length.
- To sort complex numbers in order of increasing polar angle from 0° to 360°:

```
<< << ARG DUP 0 IF < THEN -1 ACOS 2 * + END >>
ROT OVER EVAL ROT ROT EVAL < >> GSORT
```

12.4 Lists and Memory

There is a subtlety in the management of lists that you should keep in mind when programming with lists. When objects are pulled out of a list, with GET, GETI, or LIST→, the original list remains in memory as long as any of its component objects remains on the stack. If the list itself has been recalled from a global variable (or is part of a program or another list in a variable), this point is unimportant, since the memory used by the list is accounted for in the variable. However, if the list was created on the stack, the memory it uses will not be recovered until the list *and* any objects that have been extracted from it are removed from the stack. For the individual objects, “removed” means dropped, stored in a global variable (not a local variable), or combined into a vector or another list.

To see this effect, disable LAST, COMMAND, and UNDO so that no memory is used by those recovery systems, then execute

```
1 50 FOR n n NEXT 50 →LIST
```

to create a list of 50 numbers. Now execute 50 GET, so that the number 50 (from the list) is left on the stack. Next, execute MEM to determine how much memory is available. Use SWAP DROP to drop the 50, then execute MEM again. Notice that the difference is 448 bytes (447.5 on the HP-28S)--far more memory than you would expect

to be recovered by dropping the single real number 50. The large difference between the successive MEM's actually arises because the removal of the 50 allowed the HP-28 to delete the copy of the list that it had been preserving.

As mentioned above, you can "uncouple" an object from the list from which it came by either storing the object in a global variable, or by including it in another list (or an array, if the object is a number). The otherwise pointless sequence 1 →LIST LIST→DROP, executed with an object on the stack that came from a list, is a fast and reasonably efficient method of minimizing the memory used by an object. This method is used in the SORT and GSORT programs listed in the preceding section. If this sequence were omitted from the programs, the size of the lists that could be sorted in a given amount of free memory would be substantially reduced.

One additional note: if you are dealing only with a collection of numbers (all real or all complex), you can often use a vector (or a matrix, if you want a rows-and-columns type of organization) to store the numbers, instead of a list. For storing more than a few numbers, a vector is more memory-efficient than a list, and you can perform many of the same operations to assemble and disassemble vectors as you can with lists. The main disadvantage of using a vector in place of a list is that there is no built-in command for adding (concatenating) numbers to vectors, or combining two vectors into a longer one. The following program provides list-like concatenation for vectors:

ADDV <i>Concatenate Vectors</i>	
level 2	level 1 level 1
[vector ₁] [vector ₂]	→ [vector ₃]
<pre> << << DUP TYPE IF 1 ≤ THEN 1 ELSE ARRAY→ LIST→ DROP END DUP 2 + ROLL >> → s << SWAP s EVAL s EVAL + 1 →LIST →ARRAY >> >> </pre>	Program to apply to both vectors. Is the object a number? Then treat as a one-element vector. For a vector, put its elements on the stack. Get the object above the vector. Store the program as a subroutine s. Apply s to both vectors. Total number of elements. Combine the numbers into the result vector.

12.5 Indexed Variables on the HP-28S

The HP-28S has extended the operation of STO, and the syntax of algebraic objects, over the respective capabilities of the HP-28C, to provide a straightforward means of working with *indexed* variables. For example, if you wish to define indexed variables $X(1)$, $X(2)$, and $X(3)$, all you have to do is store a list or vector of three or more elements in a global variable X . Then

object 'X(*n*)' STO

stores *object* into the indexed variable $X(n)$. To recall the object:

'X(*n*)' EVAL  *object*.

If X contains a list, *object* may be of any type. If X contains a vector or a matrix, *object* must be a number. In the case of a matrix, X can have one or two indices;

25 'X(3)' STO

or

25 'X(2,1)' STO

stores 25 into the 2-1 element of a 2×2 matrix stored in X .

The program SUBCOL demonstrates a use of indexing. It replaces a matrix stored in the global variable MAT with a new version in which the elements in column i have been replaced by their original values minus the corresponding elements in column j .

SUBCOL		Subtract Columns
	level 2	level 1
	<i>i</i>	<i>j</i>
<pre> << - i j << 1 MAT SIZE 1 GET FOR n 'MAT(n,i)-MAT(n,j)' EVAL 'MAT(n,i)' STO NEXT >> >> </pre>		Store column numbers. Number of rows. Compute the difference. Replace the value.

12.6 Symbolic Arrays

HP-28 array objects are designed for the efficient storage of real and complex numbers, and can not contain symbolic elements. Nevertheless, it is possible to deal with symbolic arrays on the HP-28 by using the more flexible list objects to represent the arrays. In this section, we will present several programs for symbolic array calculations. These programs do not exhaust the subject, but serve as models from which you can develop additional programs.

All of the programs follow the convention that a symbolic array is represented by a list of lists. An $n \times m$ array is represented as a list containing n m -element lists. For example, the list $\{\{a\ b\}\{c\ d\}\{e\ f\}\}$ stands for the matrix

$$\begin{vmatrix} a & b \\ c & d \\ e & f \end{vmatrix}.$$

There is no special provision for vectors, which may be represented as $1 \times n$ or $n \times 1$ arrays in this system. Since all of the arrays are two-dimensional, we will always use two separate (i.e. not in a list) real numbers to specify elements or dimensions.

The programs do not check for the integrity of the lists you may enter--they presume that all of the inner lists in a particular symbolic array list have the same number of elements, that all of the elements are either names, numbers, or algebraic expressions, and that there are no extraneous elements in any of the lists. If the programs are applied to lists that violate any of these assumptions, they may error or return nonsensical results. If this is not satisfactory, you can easily revise the programs to include more argument testing.

12.6.1 Utilities

To start with, here are several utility programs for symbolic arrays that are analogous to various HP-28 array commands:

DIM	returns the dimensions n (rows) and m (columns) of a symbolic array.
SA→	unpacks a symbolic array into separate stack objects.
→SA	combines stack objects into a symbolic array.
N→S	converts an ordinary numerical array into a symbolic array. Vectors are converted into $n \times 1$ symbolic arrays.
S→N	attempts to evaluate all elements in a symbolic array into numbers. If successful, it then converts the symbolic array into a numeric array.
APLY1	applies a program to each element of a symbolic array.

N→S <i>Numeric to Symbolic</i>	
<i>level 1</i>	<i>level 1</i>
<code>[[array]]</code>	<code>{ { array } }</code>
<pre> << ARRAY→ LIST→ IF 1 == THEN 1 END →SA >> </pre>	Put elements on the stack. Is this a vector? Then add the other dimension. Combine into a symbolic array.
S→N <i>Symbolic to Numeric</i>	
<i>level 1</i>	<i>level 1</i>
<code>{ { array } }</code>	<code>[[array]]</code>
<pre> << SA→ DUP2 * → n m p << 1 SF 1 p START p ROLL IFERR DUP →NUM THEN DEPTH p - DROPN 1 CF ELSE SWAP DROP IF DUP TYPE THEN 1 CF END END NEXT n m IF 1 FC? THEN →SA ELSE 2 →LIST →ARRAY END >> </pre>	Put elements on the stack. Save dimensions and number of elements. Flag 1 clear will indicate a non-number. Get the next element. Convert it to a number. If →NUM fails, discard any partial results. Remember the failure. If the result is not a number... ...clear flag 1. Dimensions for result array. If there are non-numbers, return a symbolic array. Otherwise, return a numeric array.

S→N sets flag 1 to indicate a successful conversion, and clears it otherwise.

APLY1 <i>Apply Program to 1 Symbolic Array</i>		
level 2	level 1	level 1
<code>{{ array }}</code> <code><< program >></code>		<code>{{ array' }}</code>
<pre> << OVER DIM → a f n m << 1 n FOR i 1 m FOR j a i GET j GET f EVAL NEXT m →LIST NEXT n →LIST >> >> </pre>		Store the array, program and dimensions. Get the ij element. Apply the program. Pack up the ith row. Pack up the array.

APLY2 <i>Apply Program to 2 Symbolic Arrays</i>		
level 3	level 2	level 1
<code>{{array₁}}</code> <code>{{array₂}}</code> <code><< program >></code>		<code>{{array₂}}</code>
<pre> << ROT DUP DIM → a2 f a1 n m << 1 n FOR i 1 m FOR j a1 i GET j GET a2 i GET j GET f EVAL NEXT m →LIST NEXT n →LIST >> >> </pre>		Save the arrays, the program, and the dimensions. Get $a1_{ij}$. Get $a2_{ij}$. Execute the program. Pack up the ith row. Pack up the result array.

STRN <i>Transpose Symbolic Array</i>	
<i>level 1</i>	<i>level 1</i>
$\{\{ A_{ij} \} \}$	$\{\{ A_{ji} \} \}$
<pre><< DUP DIM → a n m << 1 m FOR j 1 n FOR i a i GET j GET NEXT NEXT m n >> -SA >></pre>	
Save array and dimensions. A_{ij} Elements are now in transposed order. Discard the original array. Pack up the new array.	

12.6.2 Symbolic Array Arithmetic

Using the APLY1 and APLY2 utilities listed in the preceding section, it is straightforward to create programs for simple symbolic array arithmetic.

- SADD adds two symbolic arrays.
- SSUB subtracts two symbolic arrays.
- SMS multiplies a symbolic array by a scalar (number, name, or algebraic).
- SMUL multiplies two symbolic arrays.

SADD <i>Add Symbolic Arrays</i>	
<i>level 2</i>	<i>level 1</i>
$\{\{ A_{ij} \} \}$	$\{\{ B_{ij} \} \}$
$\{\{ A_{ij} + B_{ij} \} \}$	
<pre><< << + COLCT >> APLY2 >></pre>	
SSUB <i>Subtract Symbolic Arrays</i>	
<i>level 2</i>	<i>level 1</i>
$\{\{ A_{ij} \} \}$	$\{\{ B_{ij} \} \}$
$\{\{ A_{ij} - B_{ij} \} \}$	
<pre><< << - COLCT >> APLY2 >></pre>	

You may wish to omit COLCT from SADD or SSUB, to speed up execution or to prevent an unwanted rearrangement. You can execute << COLCT >> APLY1 on an array to collect terms once after a series of calculations.

SMS <i>Scalar Multiply Symbolic Arrays</i>			
	level 2	level 1	level 1
	$\{\{A_{ij}\}\}$	$z^\dagger$	$\{\{z \cdot A_{ij}\}\}$
	$z^\dagger$	$\{\{A_{ij}\}\}$	$\{\{z \cdot A_{ij}\}\}$
<pre><< IF DUP TYPE 5 == THEN SWAP END → z << << z * >> APLY1 >> >></pre>		Put the array in level 2.	
		Save the scalar.	
		Program for APLY1.	

$^\dagger z$ can be a number, a name, or an algebraic expression.

SMUL <i>Multiply Symbolic Arrays</i>			
	level 2	level 1	level 1
	$\{\{A_{ij}\}\}$	$\{\{B_{ij}\}\}$	$\{\{(AB)_{ij}\}\}$
<pre><< DUP2 DIM ROT DIM → a1 a2 n2 m2 n1 m1 << 1 n1 FOR i 1 m2 FOR j 0 1 m1 FOR k a1 i GET k GET a2 k GET j GET * + NEXT NEXT m2 →LIST NEXT n1 →LIST >> >></pre>		Save the arrays and dimensions.	
		Compute $\sum_k A_{ik} B_{kj}$: A_{ik} A_{kj}	
		Pack up the i th row.	
		Pack up the result array.	

12.6.3 Determinants and Characteristic Equations

In this section, we develop a program DETM that computes the determinant of a symbolic matrix from the formula

$$\text{DETA} = \sum_{i=1}^n (-1)^{i+1} A_{i1} A_{i1}^C,$$

where A_{ij}^C is the ij cofactor (unsigned) of element A_{ij} , and n is the number of rows or columns in the (square) matrix. This is a recursive form of the definition of DET, since the cofactor of an element is the determinant of its minor:

$$A_{ij}^C = \text{DET } A_{ij}^M.$$

(The minor A_{ij}^M is defined in section 12.1. Note that some textbooks may give different definitions for the terms *minor* and *cofactor*.)

The programs to compute determinants of symbolic matrices, SDET (*symbolic determinant*), SCOF (*symbolic cofactor*), and SMINOR (*symbolic minor*), are straightforward realizations of the above definitions, including the recursion. They are presented in an order (SDET first, SMINOR last) that demonstrates a “top-down” programming approach, where you write a program before writing the subroutines that it calls. This kind of approach lets you concentrate on the essential main logic flow of a program, before worrying about the details. Also, when you come to write the subroutines (the “details”), you know exactly what the stack use of the subroutines should be. Note, however, that the opposite, “bottom-up” order is usually more convenient for actually entering the programs into the HP-28. By entering the subroutines first, you can then enter their names into other programs just by pressing the appropriate USER menu keys.

SDET computes the determinant of a matrix as a sum along the first column, of elements times their respective signed cofactors. (The sign -1^{n+1} is computed explicitly in this program, rather than as part of the cofactor program, so that the row and column numbers that determine the sign don't have to be passed along down through all of the levels of recursion.) The unsigned cofactor of a matrix element is the determinant of the corresponding minor; for a 1×1 matrix, the cofactor is 1. The program SCOF called by SDET embodies these points. At the point in SDET where SCOF is executed, the stack contains a matrix and the row and column number of the desired cofactor.

The two programs SDET and SCOF call each other back and forth--each is a subroutine of the other. The calculation proceeds the same way it would if you were computing the determinant by hand, where you use cofactors to compute the determinants and determinants to compute cofactors.

SDET <i>Symbolic Determinant of a Matrix</i>	
<i>level 1</i>	<i>level 1</i>
<i>{{ matrix }}</i>  <i>determinant</i>	
<pre> << DUP DIM DROP → a n << 0 1 n FOR i a i GET 1 GET a i 1 SCOF * -1 i 1 + ^ * + NEXT >> >> </pre>	Save the matrix (a) and its dimension. Initialize the sum. For each element in column 1... Get the element. Multiply by the (unsigned) cofactor. Multiply by $(-1)^{i+1}$ Add to the current sum.

SCOF <i>(Unsigned) Symbolic Cofactor</i>	
<i>level 3</i>	<i>level 2</i> <i>level 1</i> <i>level 1</i>
<i>{{ matrix }}</i>	<i>r</i> <i>c</i>  <i>cofactor</i>
<pre> << 3 PICK DIM DROP IF 1 == THEN 3 DROPN 1 ELSE SMINOR SDET END >> </pre>	Get the dimension of the matrix. If it's a 1×1 matrix... ...then just return 1. ...else, return the determinant of the cofactor.

SCOF uses a subprogram SMINOR to compute the nm minor of a symbolic matrix. It would be straightforward to modify the programs MINOR and DELROW from section 12.1 to work with symbolic matrices; however, because the structure we are using for symbolic arrays makes it easy to break an array into rows, we use a different approach and write SMINOR as a single program.

SMINOR <i>Minor of a Symbolic Matrix</i>				
level 3	level 2	level 1		level 1
{{ matrix }}	r	c	⌞	{{ minor }}
<pre> << → r c << LIST→ OVER SIZE OVER 1 - → m n << r - 1 + ROLL DROP 1 n START n ROLL IF c 1 - THEN DUP 1 c 1 - SUB SWAP c 1 + m SUB + ELSE 2 m SUB END NEXT n →LIST >> >> >> </pre>				Save the row and column number. Put the rows on the stack. Save the (final) dimensions. Discard the rth row. For each remaining row: Get the next row. $r=1$ is a special case. Elements in columns $< r$. Columns $> r$. New row. $r=1$ case. Pack up the result.

■ *Example.* Compute the determinant of the matrix $\begin{vmatrix} A & B \\ C & D \end{vmatrix}$.

■ *Solution.*

`{{ A B } { C D } } SDET ⌞ 'A*D-C*B'`

You might note that for purely numeric matrices, SDET can occasionally produce more accurate results than you obtain by applying the HP-28 command DET to the same matrix. For example, applying SDET to the matrix

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

returns 0, which is exactly correct, whereas using the command DET returns $2.1425999999999999\text{E}-10$. This happens because SDET actually carries out all of the matrix element multiplications explicitly, whereas, except for 2×2 matrices, DET does not. DET uses more advanced numerical methods to speed up calculation and minimize memory use for large matrices, and to insure a reliable answer even for matrices with

elements of widely varying values.

An excellent application of the symbolic array capabilities presented here is the computation of the characteristic equation of a matrix, which is used in the determination of eigenvalues. The characteristic equation of a matrix A is defined as

$$\text{DET}(A - xI) = 0,$$

where x is an eigenvalue, and I is the identity matrix. The program CEQN returns the characteristic equation of a symbolic or numeric matrix, where you specify the matrix in level 2, and the name to be used for the eigenvalue variable in level 1. [Note: the sequence $x \ n \ \text{TAYLR}$ is used in CEQN to simplify the result (see section 9.8.3). You can omit this sequence for faster execution of CEQN, which will then return an equivalent but longer form of the equation.]

CEQN		Characteristic Equation	
level 1	level 2		level 1
{{matrix}}	'name'		'equation'
[[matrix]]	'name'		'equation'
<pre><< IF OVER TYPE 5 ≠ THEN SWAP N→S SWAP END OVER SIZE → x n << n IDN N→S x SMS SSUB SDET x n TAYLR 0 = >> >></pre>		If it's a numeric matrix... ...make it symbolic. Save the name and dimension. Make an identity matrix. Make it symbolic. Multiply by X Subtract from the original matrix. Determinant Simplify the expression. Make into an equation.	

■ *Example.* Find the characteristic equation in X of $\begin{vmatrix} 1 & 0 & 2 \\ 0 & 1 & 4 \\ 0 & 1 & 2 \end{vmatrix}$.

■ *Solution:*

`[[ 1 0 2 ][ 0 1 4 ][ 0 1 2 ]] 'X' CEQN  '-2-X+4*X^2-X^3=0'.`

13. Plotting

The HP-28 provides a modest yet versatile plotting facility that enables you to display mathematical function curves, statistical data, and miscellaneous dot-pictures on its liquid-crystal display (*LCD*). A plot gives you a picture of the behavior of a program or mathematical expression, or a set of data, over an extended region. Such pictures are “worth a thousand words” when you are trying to find roots or extrema (see section 7.7), to study the distribution of statistical data, or to understand the result of a calculation. The 32 x 137 pixel screen is adequate for such purposes; it’s not intended for fancy high-resolution graphics.

There are three general methods for obtaining a plot on the HP-28, each associated with a particular command:

- DRAW** is for automated plotting of one or two mathematical expressions that are single-valued functions of an independent variable. The expressions can be represented by algebraic objects or programs.
- DRW Σ** is for automated creation of *scatter plots*, where data points from the statistics matrix Σ DATA are plotted as individual dots.
- PIXEL** lets you create plots in situations not suitable for either of the two automatic methods. Examples are plots in polar coordinates, multi-valued functions, and statistical data not stored in Σ DATA or requiring special processing.

The HP-28S adds an additional plotting method, via the commands $\rightarrow$ LCD and LCD $\rightarrow$. These commands convert strings to and from display pictures, allowing you to store, combine, and redisplay LCD images without having to reconstruct them pixel-by-pixel.

We will start the detailed discussion of plotting by studying function plots. Then we will use function plots to illustrate the techniques of digitizing, storing pictures, and controlling the plot ranges, topics which are common to statistical scatter plotting and general pixel plotting. The final sections of this chapter cover the latter two types of plotting, and the use of the HP 82240A Infrared Printer for plotting.

13.1 Function Plots

The most straightforward type of plotting on the HP-28 is so-called *mathematical function plotting*. In this type of plotting, performed by DRAW, a series of points is plotted, for which the vertical coordinate of each point is computed as a single-valued function of the horizontal coordinate. The function itself is specified by a procedure named EQ. As a very simple example, try the following:

Keystrokes:**Comments:**■ **MODE** **RAD**

Select radians mode.

■ **PLOT** **NEXT** **1** **PPAR**

Reset default plot parameters.

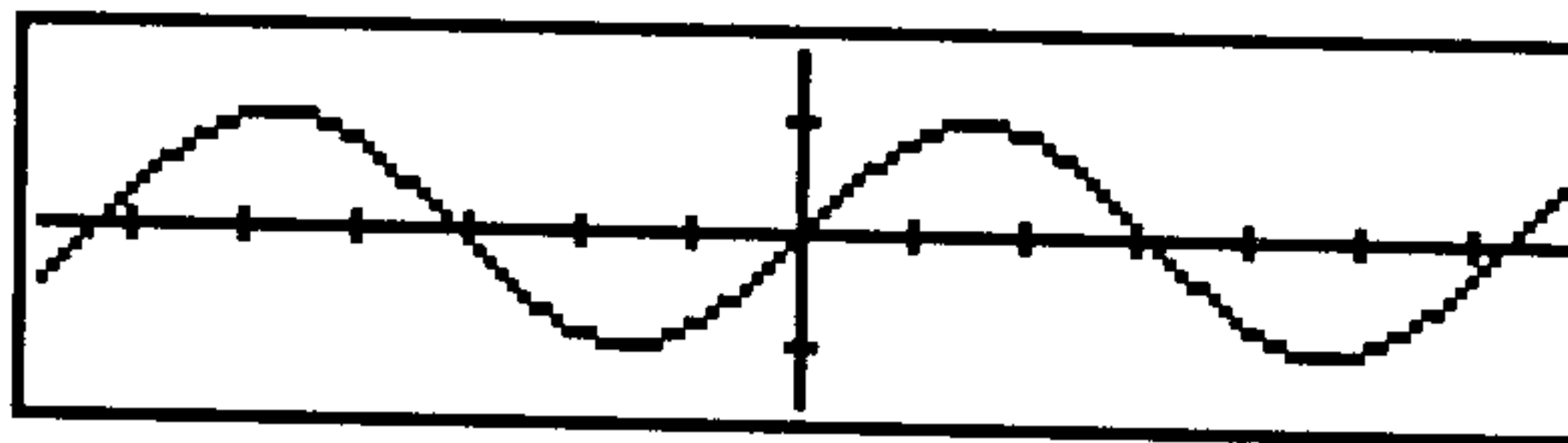
■ **PURGE**'SIN(X)' ■ **PREV** **STEQ**

Store the expression 'SIN(X)' in EQ.

DRAW

Make the plot.

You should see this picture:



This plot represents a graph of the values of $\sin x$ over the domain $-6.8 \leq x \leq +6.8$. One point is plotted for each of the 137 values of x , in steps of 0.1, corresponding to the 137 columns of pixels from left to right. The vertical scale also is 0.1/pixel, so that the screen runs from -1.5 at the bottom row of pixels, to $+1.6$ at the top row. Axes are drawn through the point $(0,0)$.

All of the parameters that determine the precise appearance of a plot are stored together in a user variable named **PPAR** (*Plot PARameters*). You can change any or all of these parameters to customize a plot, either by editing **PPAR**, or, more commonly, by using various **PLOT** menu commands.

If you execute **PPAR** after making the plot in the above example, you will obtain the following list:

$$\{ (-6.8, -1.5) \quad (6.8, 1.6) \quad X \quad 1 \quad (0,0) \}$$

In the list you can recognize the numbers that specify the horizontal and vertical ranges of the plot, the name of the independent variable, and the coordinates of the intersection of the axes. These are specific examples of the general form of **PPAR**:

$$\{ (X_{\min}, \psi_{\min}) \quad (X_{\max}, \psi_{\max}) \quad X \quad r \quad (X_{\text{axes}}, \psi_{\text{axes}}) \},$$

where we have used the following notation:

χ represents the name of the horizontal coordinate, and

ψ is the vertical coordinate.

$\chi_{\min}$ is the χ -coordinate of the leftmost column of pixels;

$\chi_{\max}$ is the χ -coordinate of the rightmost column of pixels;

$\psi_{\min}$ is the ψ -coordinate of the top row of pixels;

$\psi_{\max}$ is the ψ -coordinate of the bottom row of pixels.

χ_{axes} is the χ -coordinate of the vertical axis, and

ψ_{axes} is the ψ -coordinate of the horizontal axis.

We will use these symbols and elaborate on the meanings of the corresponding quantities in the remainder of this chapter.

13.1.1 Notation

In common discussions of graphing functions, we speak of graphing the “equation” $y = f(x)$. That is, we set up a two-dimensional coordinate system, where the horizontal direction represents the variable x , and the vertical direction represents y . Then the equation is plotted by drawing a line (or lines) through all of the points (x,y) for which the equation $y = f(x)$ is satisfied.

If you are drawing a function plot by hand, most likely you will do so by choosing a finite number of values of x , and evaluating the function $f(x)$ at each of these values. For each value of x , you place a dot on the paper at the point $(x, f(x))$. When you have enough points to show the structure of the curve, you can finish by connecting the dots to make a continuous line.

The HP-28 command DRAW carries out a process very much like this to create a function plot. Starting at the left edge of the screen, DRAW computes the value of the function for the horizontal position represented by each column of pixels, and turns on the corresponding pixel. However, DRAW does not use any special names or symbols for the ordinate or abscissa, which we called x and y in the preceding discussion. Thus it is important for you to understand how DRAW determines which function to plot, and which variable is represented by the plotting ordinate.

To avoid possible confusion with HP-28 variable names x and y , either of which may appear in the plotted function, we will use the following symbols instead of x and y :

- χ stands for the abscissa, also called the horizontal coordinate or the independent variable;
- ψ is the ordinate, also called the vertical coordinate.

Thus we will speak of plotting the function $f(\chi)$, between the limits from $\chi_{\min}$ to $\chi_{\max}$. The vertical range of the screen is from $\psi_{\min}$ to $\psi_{\max}$. Although we will use ψ to identify the ordinate, we will avoid referring to the plotted curve as $\psi = f(\chi)$, because of the distinction between "the equation $\psi = f(\chi)$ that specifies a curve," and the curves produced by DRAW for an equation object.

13.1.2 The Plot Procedure EQ (Current Equation)

The function-to-be-plotted, which we are calling $f(\chi)$, is represented in the HP-28 by a procedure stored in a global variable named EQ. Because this procedure is also the one designated for use with the Solver, it is called the "current equation," since we most often think in terms of "solving" equations. However, we will usually call the plot procedure "EQ" as a convenient shorthand, and to remind you to select a procedure for plotting by storing it in that variable. In most cases, it will not be necessary to distinguish between the variable EQ and its value--when we say "EQ" we mean "the procedure stored in EQ."

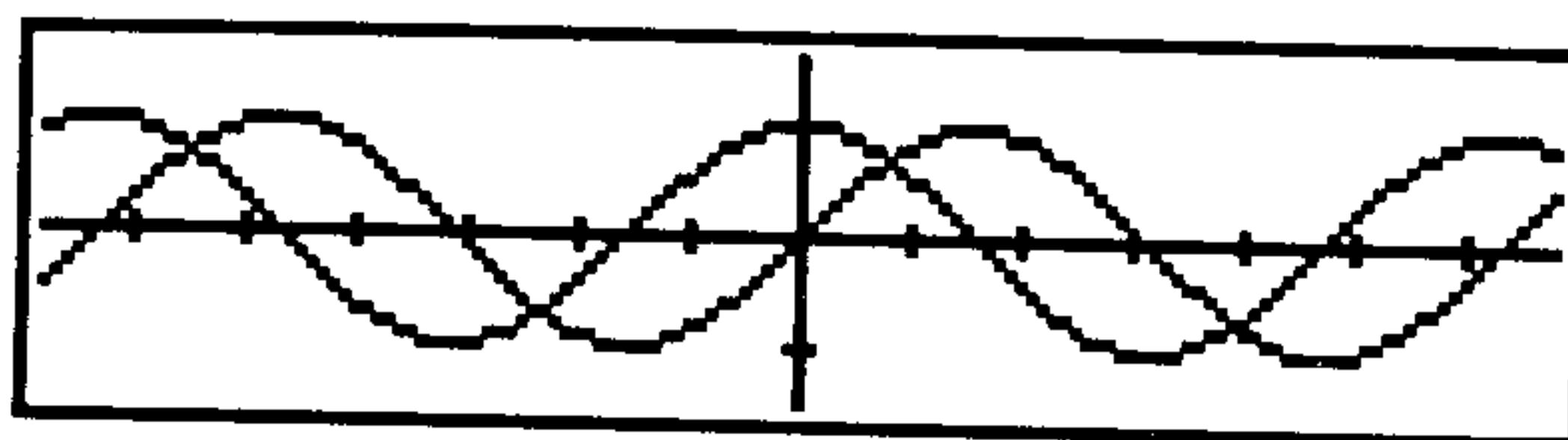
- STEQ stores an object in the variable EQ. It is equivalent to 'EQ' STO.
- RCEQ recalls the contents of EQ. It is equivalent to 'EQ' RCL, except that in the HP-28S, RCEQ only looks for EQ in the current directory (section 5.7.1).

The procedure stored in EQ can be an expression or a program as well as an equation:

- If EQ contains an *expression*, $f(\chi)$ is that expression, where χ is one of the variables in the expression. It is important to notice that the ordinate ψ does *not* correspond to any variable in the expression-- ψ is the numerical value obtained by evaluation of the expression.
- If EQ contains a *program*, it must behave like an algebraic expression, adding one number to the stack when it is executed. In this case $f(\chi)$ represents the expression equivalent to the program. For the actual production of an HP-28 plot it makes no difference whether you use a program or an algebraic object, but using programs does provide some additional flexibility (section 13.6.3).
- If EQ contains an *equation* (an algebraic object that includes an "="), the HP-28 draws *two* curves, one for each side of the equation. Each side is considered as an independent expression for plotting.

The treatment of an equation as two separate expressions has two advantages:

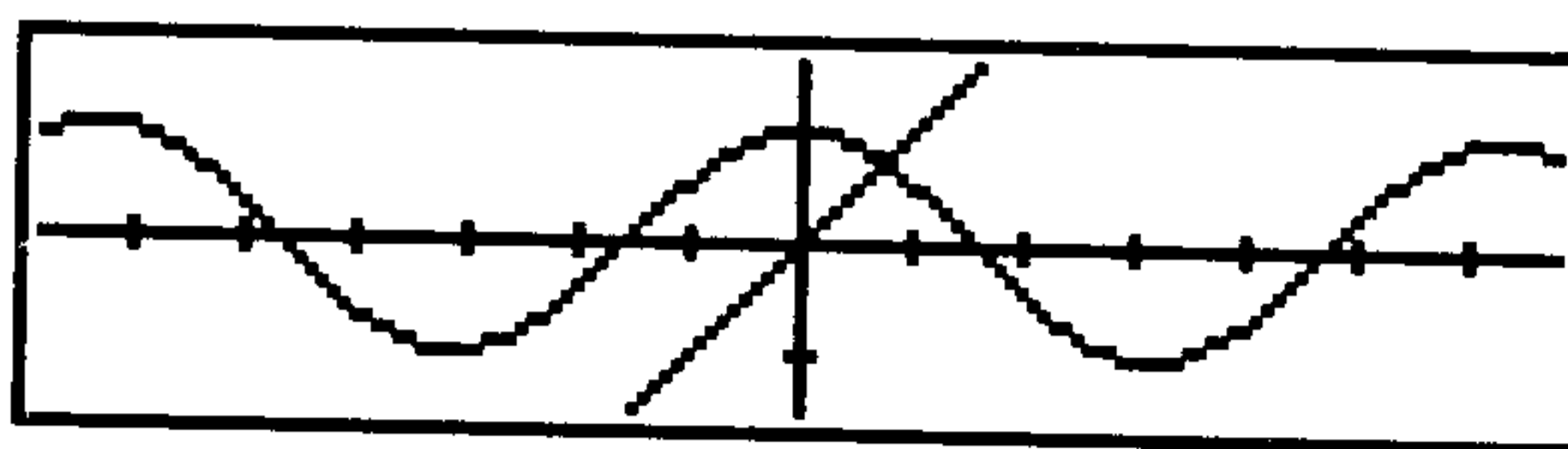
1. It provides a simple means of plotting two curves simultaneously. For this purpose, it doesn't matter whether the algebraic object makes sense as an equation--combining two unrelated expressions into an equation signals the HP-28 to draw two curves instead of one. For instance, if EQ is 'SIN(X)=COS(X)', DRAW plots the sine and cosine curves together:



2. It helps you find the numerical solutions to an equation. Any intersection of the two plotted curves occurs at a value of x that is a solution to the equation considered as a function of x . To find, for example, the values of x for which $\cos x = x$, you can plot:

RAD 'COS(X)=X' STEQ 'X' INDEP DRAW

In the picture you can see that the two curves intersect at one point. You can use the digitizing cursor to obtain approximate values for the X-coordinate of the point, then use the Solver to improve the accuracy of the estimate (section 7.6).



A disadvantage of the HP-28 treatment of plotting equations is that it may cause confusion over the meaning of the left-hand side of a plot equation and the ordinate of the plot. At the start of section 13.1.1, we discussed the common notion of the plot of an equation $\psi = f(x)$ as a prescription for producing a curve by finding all of the points (x, ψ) that satisfy the equation. But if you have an HP-28 equation of this form stored in EQ, DRAW produces *two* curves, one corresponding to $f(x)$ and the other a straight horizontal line. The latter comes from treating the left-hand side of the equation (the expression ψ) as an independent function of x --in this case, a constant represented by a horizontal line.

Notice that the HP-28 convention allows you to plot equations like 'SIN(X)+X=LN(X)'

as two expressions, whereas in the usual " $\psi = f(\chi)$ " mode of thinking, such an equation makes no sense for plotting. Another way to consider this is to note that the HP-28 plots *equations* (as a pair of expressions) rather than *solutions* to equations. The solutions to the HP-28 equations are indicated by the intersections of the two plotted curves.

In short, if you want to plot something of the form $\psi = f(\chi)$, omit the ψ and store the expression $f(\chi)$ in EQ. Then remember that the vertical direction corresponds to your variable ψ .

13.1.2.1 Evaluation of EQ

When DRAW makes a function plot, it creates a temporary copy of the independent variable χ in USER memory (in the HP-28S, the copy is created in the current directory). Then, starting with the leftmost column of pixels, it computes the value of χ that corresponds to that column, and stores it in the temporary variable. Next, it evaluates EQ and uses the resulting value (two values, if EQ contains an equation) as the vertical coordinate ψ of a point (χ, ψ) to plot on the screen. If ψ falls within the vertical range of the screen, the pixel at (χ, ψ) is turned on (made black). This process is repeated for each column of pixels on the screen (unless you have selected a resolution other than 1 to produce a faster plot with fewer dots--see section 13.1.4).

EQ is evaluated in the ordinary way, except that numerical evaluation mode is temporarily set if necessary to ensure that EQ evaluates to a number. This means that every variable (except the independent variable) in EQ must have a real number value, or contain a procedure or name that evaluates to a real number.

13.1.2.2 Missing Points and Errors

Three things can cause a plotted curve to "miss" part or all of the screen:

1. The ψ value for a column is out of range, being too large (positive) or too small (negative) to match one of the 31 pixels in a column.
2. EQ evaluates to an object of a type other than a real number. This can happen even when EQ contains an apparently simple expression, if it happens to evaluate to a complex number. For example, ' $\sqrt{X}$ ' returns a complex result for $X < 0$.
3. Evaluation of EQ produces any kind of error.

In the first version of the HP-28C (Version 1BB), the latter two problems are always fatal--DRAW aborts and displays an error message, clearing the partially completed plot. In HP-28C Version 1CC, and in the HP-28S, DRAW is enhanced to ignore many errors and continue plotting, bypassing the values of χ that cause errors. The newer DRAW ignores real number math errors (errors #301 - #305), and cases where EQ returns an

object that is not a real number. Other kinds of errors terminate the plotting.

You can make a version 1BB HP-28C show a similar tolerance as follows:

1. Store this program in EQ:

Select hexadecimal mode before entering this program.

<pre> << IFERR EQP ->NUM THEN IF ERRN #301 < ERRN #305 > OR THEN ABORT ELSE MAXR END ELSE IF DUP TYPE THEN DROP MAXR END END >> </pre>	If EQP errors... ...check the error type. If it's not a real number error... ...then quit; ...otherwise return MAXR. If EQP doesn't return a real number... ...then replace it with MAXR.

2. Store any program or algebraic object you want to plot in a variable named EQP. (EQP takes the place of EQ for this and subsequent plots.)
3. Execute DRAW.

This method actually plots a point in every column, but for any value of χ that causes an error, a point is plotted at $\psi = \text{MAXR}$, which is normally out of the vertical range of the screen.

13.1.3 The Independent Variable

So far we haven't specified how the HP-28 knows the name of the independent variable for plotting. Often EQ contains only one variable name, so there is no ambiguity--in such cases, the HP-28 automatically chooses that variable, and you need give it no further thought. In general, though, procedures used for function plotting can contain any number of variable names. For any one plot, of course, only one of the variables can be considered as the *independent* variable χ . The rest must either have numerical values or contain names or procedures that evaluate to numbers. But rather than require you to use a special name for the independent variable, the HP-28 provides methods for you to specify any name you want for the independent variable. That name is stored as the third object in the list stored in PPAR.

The command INDEP (*independent variable*) is the most straightforward means for choosing an independent variable. You enter the desired name into level 1, then execute INDEP. INDEP stores that name in PPAR, where it identifies the independent variable for subsequent uses of DRAW. INDEP is equivalent to 'PPAR' 3 ROT PUT, except:

- INDEP will error if the level 1 object is not a name. (The HP-28C allows INDEP to store local names as well as global, but DRAW can not use a local name for the independent variable. The HP-28S INDEP is modified to accept only global names.)
- INDEP will create PPAR if it does not already exist in the current directory.

To recall the current independent variable name, you can use 'PPAR' 3 GET. If you just want a visual reminder, 'PPAR'  VISIT will show you all of PPAR, where it's easy to pick out the independent variable name, the only name in the list.

If you don't specify an independent variable, the HP-28 will choose one for you. This is a great convenience, since in many cases there is only one variable in EQ, and you are saved the step of using INDEP to "specify the obvious." DRAW searches for independent variables in EQ in the same manner as the Solver builds its variables menu (section 7.3). The first independent variable encountered is selected as the default independent variable; the name of that variable is the first that appears in the SOLVR menu for the same EQ.

The search for independent variable names by DRAW or the Solver returns names found either directly or indirectly in EQ, for which the corresponding variable either contains a data object or does not exist. The name search proceeds left-to-right through the EQ procedure; when a name is encountered that specifies a variable containing another name or procedure, the latter is then searched for names, and so on. Thus if EQ contains 'A+B', and both A and B are undefined variables or contain just data objects, 'A' will be the default independent variable name for DRAW. But if you store the name 'X' in A, 'X' will become the default name. Or, if you store the program << 1 >> in A, 'B' will be the independent variable name.

DRAW actually uses the default independent variable name in these cases:

- No PPAR exists in the USER menu (in the current directory in the HP-28S). DRAW creates a PPAR with the default independent variable name determined from EQ (and also default values for the other plot parameters).
- The name specified in PPAR is not present in EQ, and DRAW is executed by means of the menu key DRAW. In this case, you will see the message

oldname Not In Equation Using *newname*

displayed briefly before the plotting starts. *Oldname* is the name that was stored in PPAR; *newname* is the new default name that replaces it. This replacement does not occur if DRAW is used in a program.

- PPAR was created by a command other than DRAW. PMAX, PMIN, RES, AXES, CENTR, *W, *H, DRWΣ, PIXEL, and DRAX each create a default PPAR if one does not already exist. The resulting PPAR contains the name constant. When DRAW sees this name, it replaces it with a default determined from EQ (unless constant is actually present in EQ), without displaying any message.

In summary, DRAW tries to use the independent variable name you have specified with INDEP. If you have not chosen a name, or your choice is not present in EQ, DRAW takes the first appropriate name found in EQ to be the independent variable.

13.1.4 Reducing the Number of Points Plotted

When EQ contains a complicated program or algebraic object that takes a significant amount of time to execute, you may wish to speed up the plotting process by reducing the number of points that are plotted. With the default plot parameters, a point (or two, for equations) is plotted in each of the 137 pixel columns. You can instruct the HP-28 to plot only every second column, or every third column, etc., by using the RES (*resolution*) command.

RES takes a real number r and stores it as the fourth element in PPAR. DRAW uses this number to determine the χ -coordinates χ_n of the points it plots:

$$\chi_n = \chi_{\min} + (\chi_{\max} - \chi_{\min})/r$$

Thus if r is 1 (the default) or less, DRAW plots a point in every column; if r is 2, it plots points in every second column, and so forth. If r is non-integer, the points will be irregularly spaced across the screen.

13.2 Digitizing

The purpose of a plot is to let you observe the behavior of a function or a set of data over an extended range. In some cases, the picture itself is the end result, but it is also possible to extract information in the form of point coordinates from a plot for use in subsequent calculations. The process of obtaining the coordinates is called *digitizing*; in the HP-28, to digitize a point means to return its coordinates (χ, ψ) to the stack.

You can digitize points any time the plot cursor keys are active, which is indicated by the presence of the plot cursor on the screen. In the HP-28C, the plot cursor keys are activated only by the menu keys DRAW and DRW Σ (executing the commands in the command line or in a program draws the plots but does not turn on the plot cursor). In the HP-28S, the command DGTIZ allows you to turn on the plot cursor at any time.

13.2.1 HP-28C Digitizing

Digitizing is achieved in the HP-28C plot cursor menu by means of the two leftmost menu keys, labeled INS and DEL (the names come from their ordinary command line editing uses). Both keys return the coordinates of the cursor. The **[INS]** key combines the two coordinates into a single complex number (χ, ψ). The **[DEL]** key splits the coordinates into separate real numbers, with χ returned to level 2 and ψ returned to level 1. The complex number form is most convenient when you are obtaining guesses for the Solver (see the examples in section 7.6), or rescaling the plot (section 13.4.2). The real number form is useful when you are just interested in one coordinate or the other.

It is possible to activate the plot cursor at any time in the HP-28C by means of one of the following programs:

DIGITIZE

For HP-28C Version 1BB

Note: Read section 3.1 before entering this program! Enter in hexadecimal mode.

<<

```
#A26B SYSEVAL
#1214C SYSEVAL
DROP
#32219 SYSEVAL
#4D9E SYSEVAL
#3F56D SYSEVAL
#12066 SYSEVAL
#A28B SYSEVAL
```

>>

DIGITIZE*For HP-28C Version 1CC*

Note: Read section 3.1 before entering this program! Enter in hexadecimal mode.

<<

#A236 SYSEVAL

#12183 SYSEVAL

DROP

#3230E SYSEVAL

#4D9E SYSEVAL

#3F68E SYSEVAL

#1209D SYSEVAL

#A256 SYSEVAL

>>

Both programs draw axes through the point specified in the current PPAR, then activate the plot cursor. They act nearly the same as the HP-28S command DGTIZ except that the latter does not draw axes.

13.2.2 HP-28S Digitizing

The HP-28S plot cursor menu is the same as that of the HP-28C except for the **DEL** key, which plays a different role. The **INS** key works like its HP-28C counterpart; pressing it stores the cursor coordinates in level 1 of the stack as a complex number (χ, ψ). The HP-28S **DEL** key, rather than capturing the cursor coordinates as on the HP-28C, converts the entire display picture into a 548-character string object entered in level 1. This feature is discussed in the next section.

The HP-28S also has two additional features not included in the HP-28C:

- When you press and hold , the cursor coordinates are displayed in the bottom line of the display. When you release the key, the full picture is restored.
- The DGTIZ command allows you to activate the plot cursor at any time. Thus, for example, the sequence CLLCD DRAW DGTIZ has the same effect as pressing the PLOT menu key **DRAW**. If DGTIZ is encountered during program execution, the program halts with the plot cursor active. When you press **ON** to clear the plot, the program resumes execution.

13.3 Storing Pictures (HP-28S Only)

The HP-28 LCD contains 32×127 pixels, each of which is represented by one bit in the display memory. At eight bits per byte, it requires 548 bytes of memory to store an image of the display. Since the HP-28C contains only about 1700 bytes of user memory,

it was not practical on that calculator to include facilities for storing and retrieving LCD pictures. (We are using the term *picture* here to refer to any HP-28 display image, not just those plotted by DRAW or DRWΣ.)

With 32 K-bytes of user memory, the HP-28S has sufficient memory to hold and manipulate several dozen stored pictures. Therefore, the HP-28S operation set was enhanced over that of the HP-28C in four ways related to plotting:

- The command LCD→ creates a 548-byte string object, which we'll call an *image string*. This string encodes a pixel-by-pixel image of the display as it appears at the moment the command is executed. As an ordinary string object, the image string can be manipulated on the stack or stored in a variable, and is subject to all HP-28 string commands.
- The command →LCD is the reverse of LCD→; it takes an image string from the stack and converts it to an LCD picture. →LCD sets the user message flag, so that the new picture persists until program execution halts and a key is struck. →LCD takes only a few hundredths of a second to execute, so it is obviously a far faster method of reproducing a picture than re-executing DRAW or any of the other plotting commands.
- In the plot cursor menu, the **DEL** key executes LCD→ without clearing the plot. When you do clear the plot with **ON**, the image string is left on the stack.
- The logical operators AND, NOT, OR, and XOR accept string arguments and return string results (the input strings must be of the same length). Of these commands, OR is the most useful, since it allows you to create superpositions of individual plots by combining the two image strings.

The primary purposes of these commands are to allow you to store individual pictures for reuse, and to combine pictures. The former is accomplished by executing LCD→ after a picture is created. If you create a picture by using one of the menu keys **DRAW** or **DRWΣ**, you can execute this command by pressing the **DEL** key while the picture is showing. If you create the picture by other means and intend to store the picture, you should include the command LCD→ at the end of the command sequence that creates the picture.

■ *Example.* Plot $\sin x$ using the default plot ranges, and store the resulting graph in a variable SINE.

■ *Solution:*

```
'PPAR' PURGE RAD 'SIN(X)' STEQ
```

```
CLLCD DRAW LCD→ 'SINE' STO CLMF
```

To recreate the plot:

```
SINE →LCD DGTIZ
```

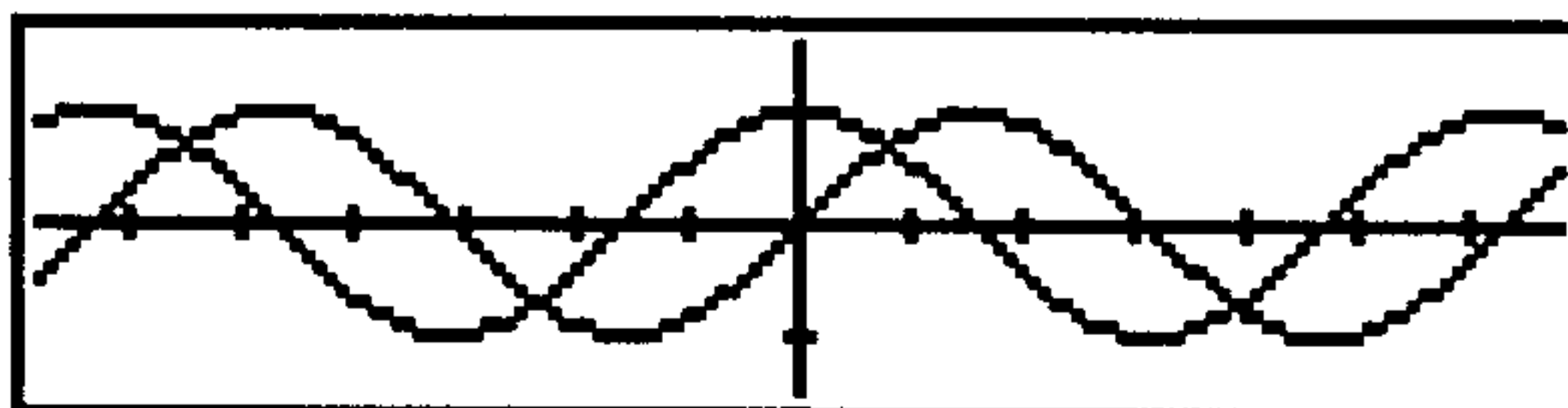
The command DGTIZ at the end is optional--include it in the command line with →LCD when you want to use the plot cursor keys on a reconstructed picture.

To combine two or more pictures, place the appropriate image strings on the stack and execute OR as many times, less one, as there are image strings.

■ *Example.* Plot $\cos x$, then combine that picture with the plot of $\sin x$ that was stored in SINE in the previous example.

■ *Solution:*

```
CLLCD 'COS(X)' STEQ DRAW LCD→ SINE OR →LCD
```



The other logical operators, AND, XOR, and NOT can also be applied to strings, but have less practical value.

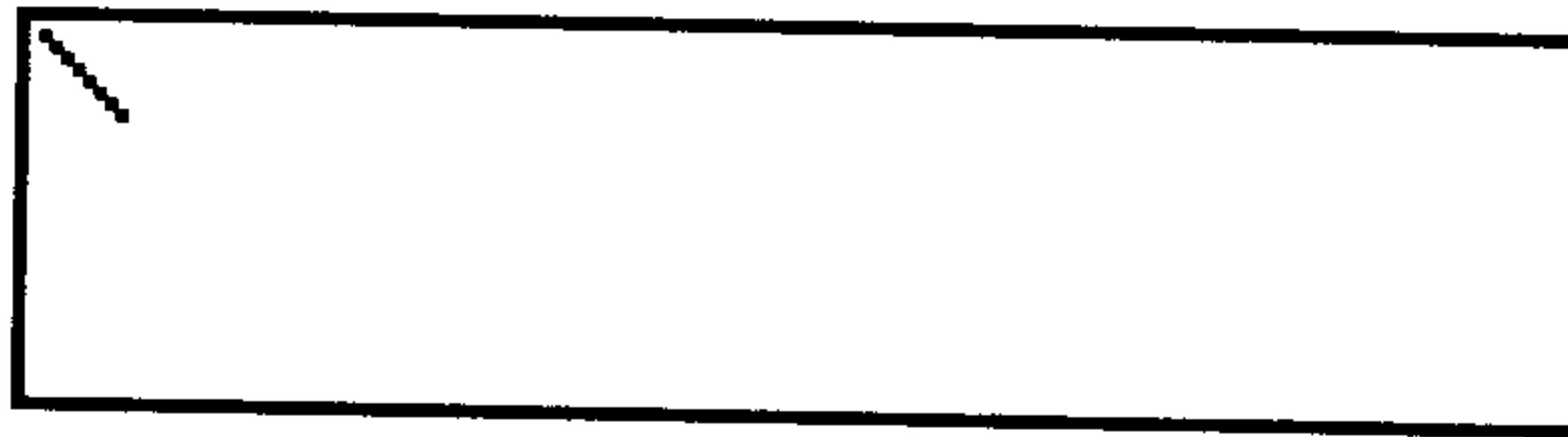
- NOT inverts all of the bits in a string. LCD→ NOT →LCD produces a picture that is a black/white reversed version of the current display.
- AND combines two image strings into a new image string that has pixels turned on only in those positions where *both* original pictures have pixels turned on.
- XOR combines two image strings into a new image string that is the superposition of the original pictures, except that “on” pixels common to both will *not* be on in the combined picture.

13.3.1 The Image String

The image string object produced by `LCD→` is 548 characters, or bytes, long. Each character represents a partial display column of eight pixels. The first character in the image string corresponds to the eight pixels along the left edge of display line 1 (upper left corner). The numerical value, or character number, of the character is equal to the binary value of the pixels read from bottom-to-top, where a black (on) pixel is 1 and a white pixel is 0. The second character in the image string matches the next column of pixels to the right, and so on left-to-right across line 1, followed by lines 2 through 4.

The following sequence demonstrates the numerical translation of image strings:

```
"" 0 7 FOR n 2 n ^ CHR + NEXT CLLCD →LCD
```



The sequence creates an eight-character string, where the first character is character number 1, the second number 2 (#10b), the third 4 (#100b), and so on up to the eighth, character number 128 (#10000000b). When `→LCD` is executed, the bit patterns of those characters are transferred to the screen starting in the upper left hand corner. The uppermost pixel of the first column is turned on, the second in the second column, etc.

This example also demonstrates that `→LCD` does not require a full 548-character image string. `→LCD` starts building the display in the upper left corner--if it runs out of characters in the image string, it leaves the remainder of the screen unchanged. If the image string is longer than 548 characters, `→LCD` ignores the excess.

Display pictures usually contain blank regions with no pixels turned on, so that most image strings contain one or more occurrences of character number zero. This means that you cannot edit image strings in the ordinary way with `EDIT` or `VISIT`, since character zero is not allowed in the command line (Can't Edit `CHR(0)` error). However, you can use string commands to alter the contents of an image string. The next two example programs produce amusing displays that demonstrate image string operations.

BARBERPOLE <i>Scroll the Picture</i>	
<pre><< LCD→ DO DUP 548 DUP SUB SWAP 1 547 SUB + DUP →LCD UNTIL KEY END DROP2 >></pre>	Capture the existing display. Take the last character... ... and move it to the front. Display the new string Repeat until a key is struck.
REVERSE <i>Invert a Picture</i>	
<pre><< LCD→ "" 548 1 FOR n OVER n n SUB + -1 STEP →LCD DROP >></pre>	Capture the current display. Start an empty image string. Get the nth character. Add it to the new string. Display the reversed picture.

13.4 Plot Ranges

When you make a plot, whether on paper or on a computer screen, you must start by specifying the *plot ranges*--the ranges of values of the χ - and ψ - coordinates that the rectangular plotting area represents. For a computer, this tells its plotting programs which point on the screen corresponds to the mathematical point (χ, ψ) .

The HP-28 LCD consists of an array of pixels, arranged in 137 columns of 32 pixels. The conversion between pixel column and row numbers and an χ - ψ coordinate system is determined by the χ and ψ coordinates that you assign to the two corner pixels $P_{\min}$ (lower left) and $P_{\max}$ (upper right). The coordinates of $P_{\min}$ and $P_{\max}$ are stored as two complex numbers $(\chi_{\min}, \psi_{\min})$ and $(\chi_{\max}, \psi_{\max})$, which occupy the first two positions, respectively, in the PPAR list.

Figure 13.1 illustrates the positions of the points and coordinates that determine the scaling of an HP-28 plot.

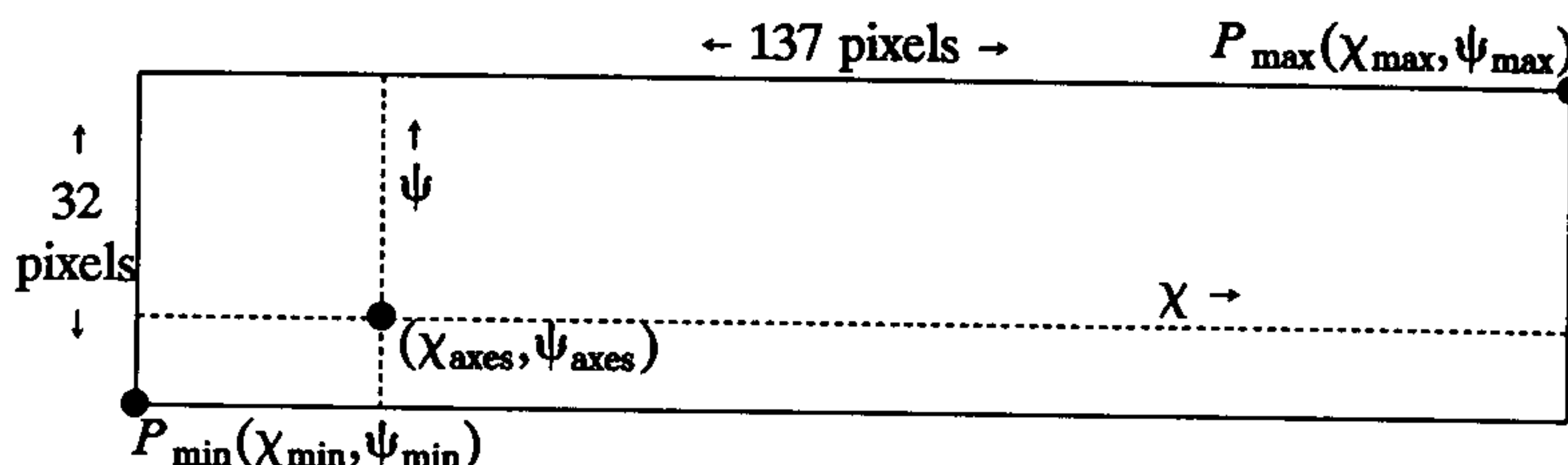


Figure 13.1. Plot Ranges

If you create a plot without having previously specified the plot ranges, the HP-28 uses defaults of $P_{\min} = (-6.8, -1.5)$, and $P_{\max} = (6.8, +1.6)$. These values are chosen so that:

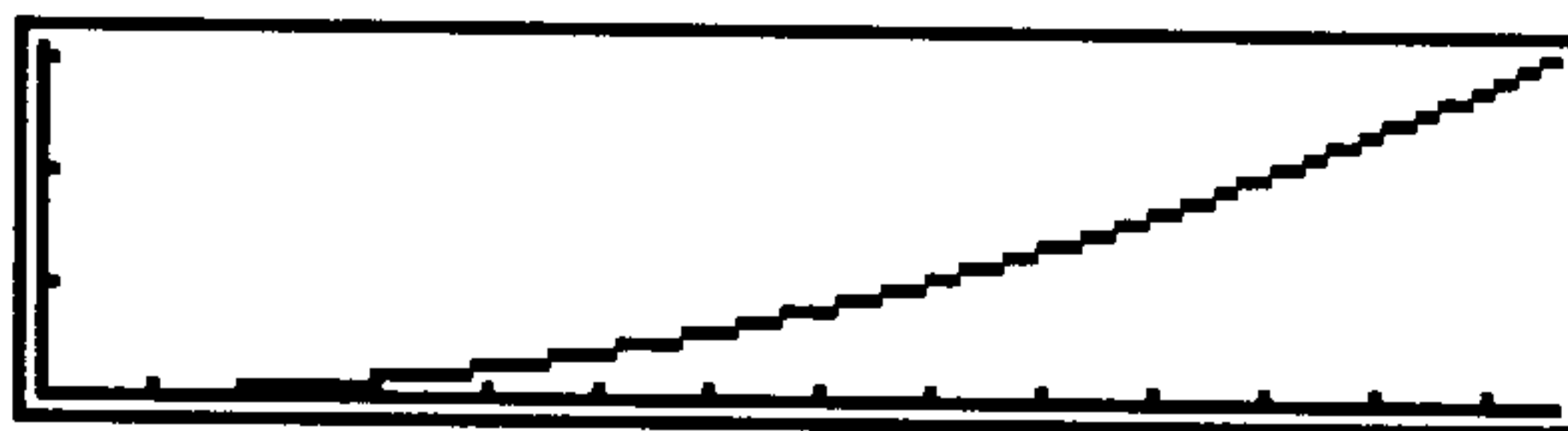
- The center pixel has the coordinates (0,0).
- The scale is the same horizontally and vertically.
- Moving by one pixel parallel to either axis corresponds to a coordinate change of 0.1.

The most straightforward method of choosing plot ranges other than the default values is to use the commands PMIN and PMAX. For either command, you enter the coordinates of the appropriate corner pixel (lower-left for PMIN or upper right for PMAX) as a complex number of the form (χ, ψ) . PMIN stores its argument as the first element in the PPAR list; PMAX stores the second element. (Either command will create PPAR if it does not exist.)

■ *Example.* Plot x^2 from $x = 0$ to 10, with the vertical range from 0 to 100.

■ *Solution:*

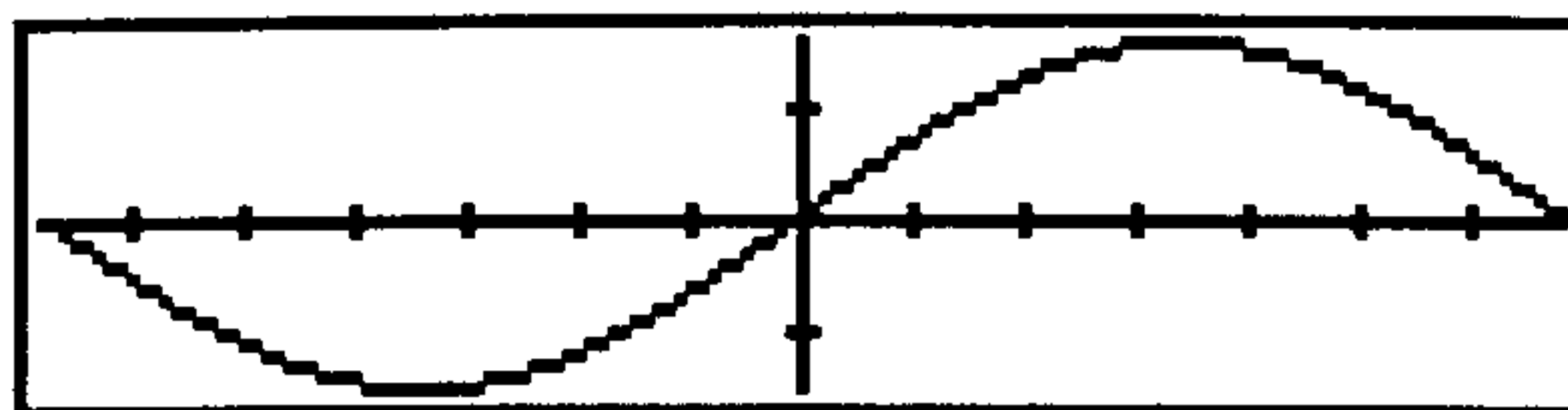
(0,0) PMIN (10,100) PMAX 'X^2' STEQ DRAW.



■ *Example.* Plot $\sin x$ from $x = -\pi$ to $+\pi$, vertical range -1 to $+1$.

■ *Solution:*

RAD π →NUM 1 R→C DUP NEG PMIN PMAX 'SIN(X)' STEQ DRAW.



You may find it more convenient to enter the χ - and ψ -coordinates separately, especially if you want to change, for example, the χ -coordinates of $P_{\min}$ and $P_{\max}$, without changing the ψ , or vice-versa. This can be achieved with the following programs:

NPPAR		New PPAR	
level 3	level 2	level 1	
x	<< procedure >>	n	☐

<< ROT PPAR IF DUP TYPE 6 == THEN 1 *H END 3 PICK GET 4 ROLL EVAL R-C 'PPAR' 3 ROLL PUT >>	<< proc >> n x $ppar$ Does PPAR not exist? Create a default PPAR. Old value. New value. Store new value.

XMIN		Store $\chi_{\min}$		
level 1		level 3	level 2	level 1
$\chi_{\min}$	☐	$\chi_{\min}$	<< IM >>	1

<< << IM >> 1 NPPAR >>

XMAX		Store $\chi_{\max}$		
level 1		level 3	level 2	level 1
$\chi_{\max}$	☐	$\chi_{\max}$	<< IM >>	2

<< << IM >> 2 NPPAR >>

YMIN				
Store $\psi_{\min}$				
level 1		level 3	level 2	level 1
$\psi_{\min}$	$\rightarrow$	$\psi_{\min}$	<< RE SWAP >>	1
<< << RE SWAP >> 1 NPPAR				
>>				

YMAX				
Store $\psi_{\max}$				
level 1		level 3	level 2	level 1
$\psi_{\max}$	$\rightarrow$	$\psi_{\max}$	<< RE SWAP >>	2
<< << RE SWAP >> 2 NPPAR				
>>				

13.4.1 Conversions Between Plot Ranges and Pixel Numbers

If the coordinates of $P_{\min}$ are $(\chi_{\min}, \psi_{\min})$, and the coordinates of $P_{\max}$ are $(\chi_{\max}, \psi_{\max})$, then the coordinates of the pixel P_{mn} (m th row, n th column) are

$$\chi_{mn} = \chi_{\min} + (n-1)(\chi_{\max} - \chi_{\min})/136$$

$$\psi_{mn} = \psi_{\min} + (m-1)(\psi_{\max} - \psi_{\min})/31$$

$P_{\min}$ is P_{1-1} , and $P_{\max}$ is $P_{137-137}$. In all cases, we are describing the coordinates of the center of a pixel; a pixel is actually a square 1/31 of the LCD height in width (including the narrow inactive areas between the pixels as part of the pixel area). Consequently, the coordinates (χ, ψ) fall within the nm -th pixel, where

$$n = \text{IP} \left[136 \frac{\chi - \chi_{\min}}{\chi_{\max} - \chi_{\min}} + 1.5 \right]$$

$$m = \text{IP} \left[31 \frac{\psi - \psi_{\min}}{\psi_{\max} - \psi_{\min}} + 1.5 \right]$$

Because the HP-28 LCD has an even number of pixel rows, the geometric center of the screen $(P_{\min} + P_{\max})/2$ lies between the sixteenth and seventeenth rows rather than within any pixel. However, to make digitizing more straightforward, the "center" of the screen is defined to be the pixel in row 16 (from the bottom) and column 69 (from the

left). With the default plot ranges (see the next section), the coordinates of this pixel are (0,0); when you use **CENTR** to move the center of the plot (section 13.4.1.1), this pixel is assigned the coordinates you enter for that command.

13.4.2 Adjusting the Plot Ranges

Often after making a plot, you find that the picture is not "quite right." Perhaps the interesting part of the curve is not on the screen, or perhaps the scale is too large or too small to show the features of interest clearly. In such cases, you need to adjust the plot parameters, then execute **DRAW** again to obtain a better picture.

Although **PMIN** and **PMAX** are the most precise means of specifying the plot ranges, they are not always the most convenient. When the plot scale is wrong, it may take some calculation to determine what the new $P_{\min}$ and $P_{\max}$ should be for a better picture. The HP-28 provides several tools to simplify this process.

13.4.2.1 Moving the Center of the Plot

A simple case of adjusting the plot parameters arises when the plot scale is satisfactory but a region of interest is off-screen. The **CENTR** command is designed for a simple translation of a plot without affecting the scale. You enter the coordinates of the point that you want to be at the center of the screen, then execute **CENTR**. Executing (χ, ψ) **CENTR** adjusts the plot parameters as follows (the primed coordinates are the results after **CENTR**):

$$\chi'_{\min} = \chi + \frac{1}{2}(\chi_{\max} - \chi_{\min})$$

$$\chi'_{\max} = \chi - \frac{1}{2}(\chi_{\max} - \chi_{\min})$$

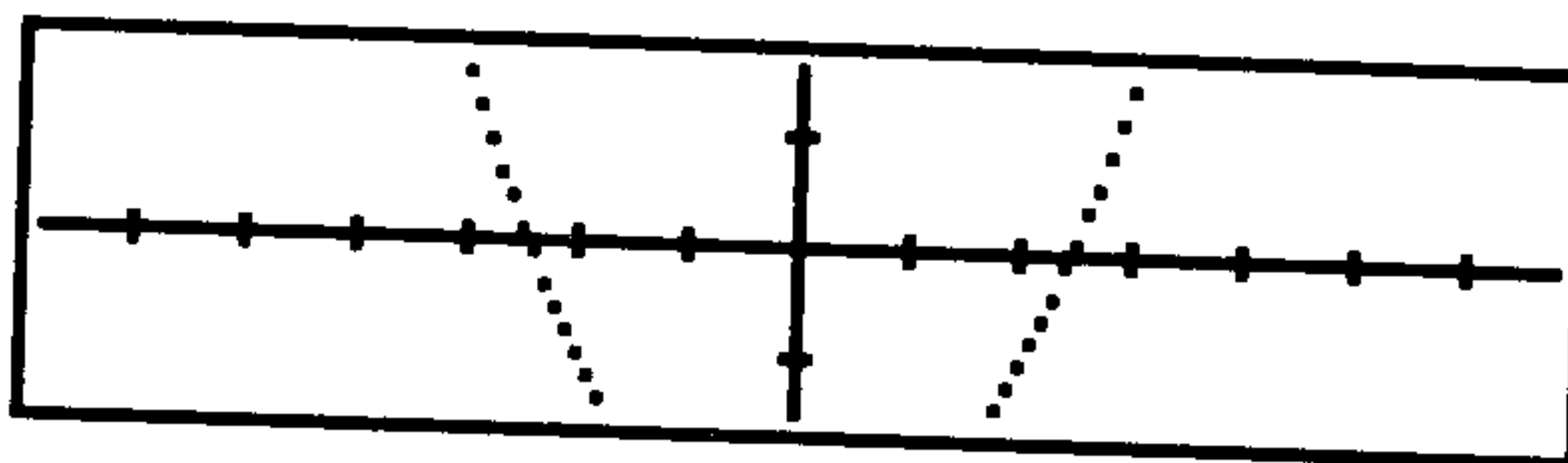
$$\psi'_{\min} = \psi + \frac{16}{31}(\psi_{\max} - \psi_{\min})$$

$$\psi'_{\max} = \psi - \frac{15}{31}(\psi_{\max} - \psi_{\min})$$

■ *Example.* Plot the expression $'.5*X^2-3'$, adjusting the plot ranges so that the curve minimum is visible.

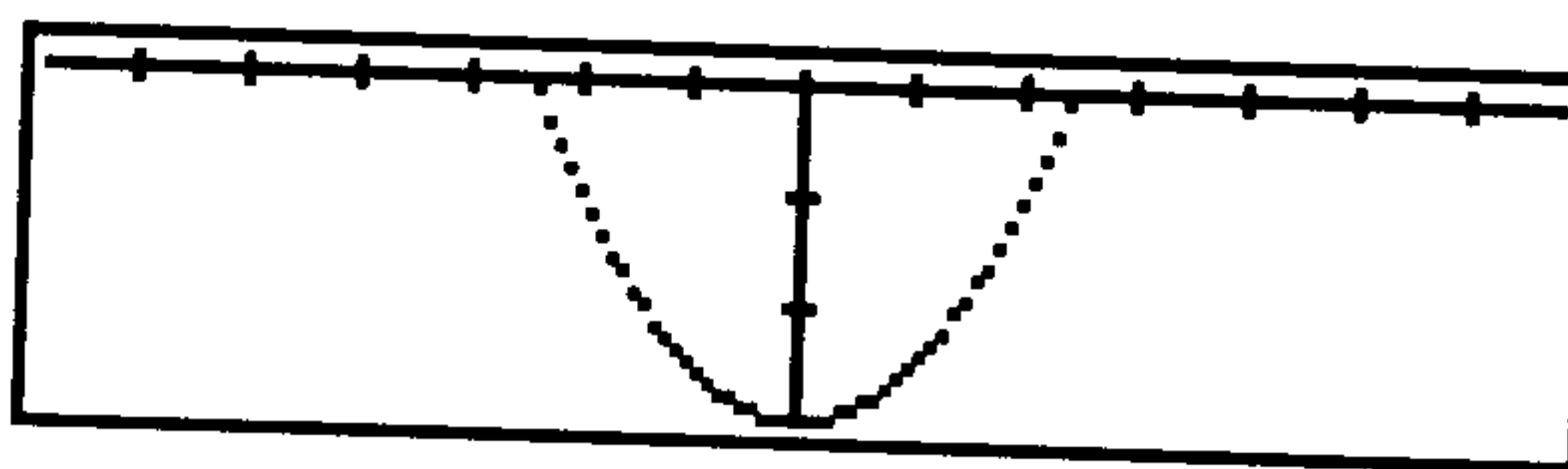
■ *Solution:* First, plot the expression using the default plot parameters:

```
'PPAR' PURGE '.5*X^2-3' STEQ DRAW
```

The minimum is below the edge of the screen. At $X=0$, the expression has the value -3 , so move the center halfway to that point:

(0, -1.5) CENTR DRAW



Now you can see the minimum, as desired.

13.4.2.2 Digitizing New $P_{\min}$ and $P_{\max}$

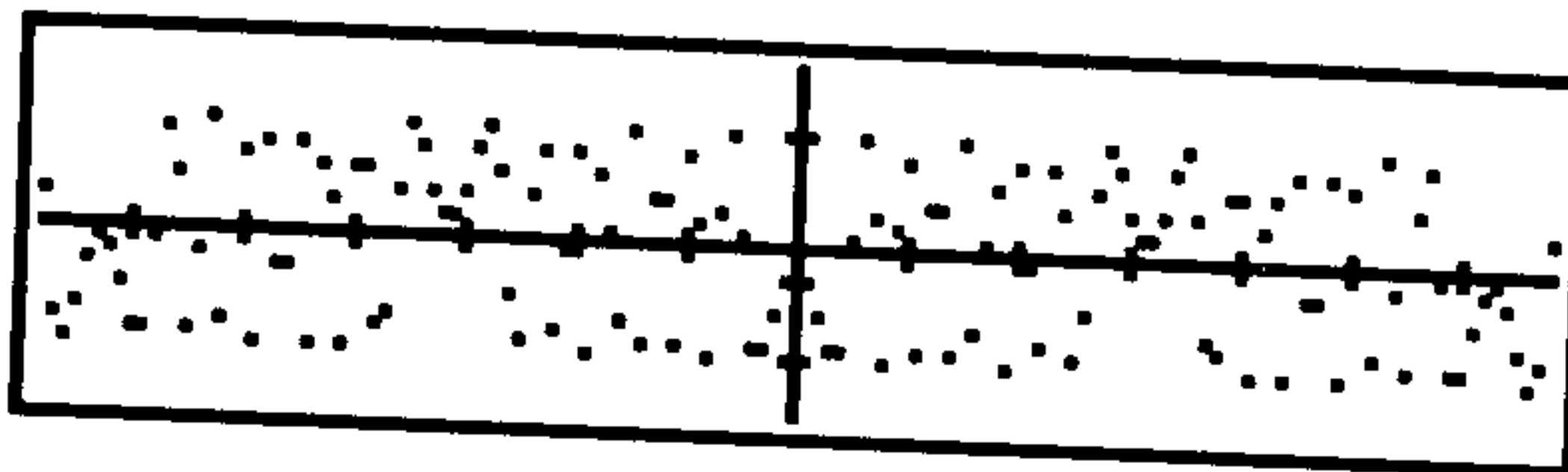
Once you've made a plot using DRAW, it's a simple matter to use the plot menu keys to specify a smaller plotting region so that you can "zoom in" on a feature of interest in the plot. Here's the basic process:

1. Create the plot. If you don't use **DRAW** or **DRWΣ**, be sure to include HP-28S command DGTIZ or the HP-28C program DIGITIZE (section 13.2.1) at the end of the plotting sequence.
2. Move the cursor to the upper-right corner of the area you want to expand. Press **INS**.
3. Move the cursor to the lower-left corner of the area. Press **INS**.
4. Press **ON** to exit the plot display.
5. Press **PMIN** **PMAX**. This sets $P_{\min}$ and $P_{\max}$ to the values you chose with the cursor. (Notice that since you digitized the new $P_{\min}$ last, its coordinates are in level 1 when you exit the plot. Thus you press **PMIN** first, then **PMAX** --the opposite order in which you digitized the points.

■ *Example.* Plot the expression $\sin(25\cos x)$.

■ *Solution:*

RAD 'PPAR' PURGE 'SIN(25*COS(X))' STEQ DRAW
produces the display



Because this expression varies so rapidly with x , the plot appears as an unintelligible scattering of dots. You need to reduce the plot scale by zooming in on the region around the origin:

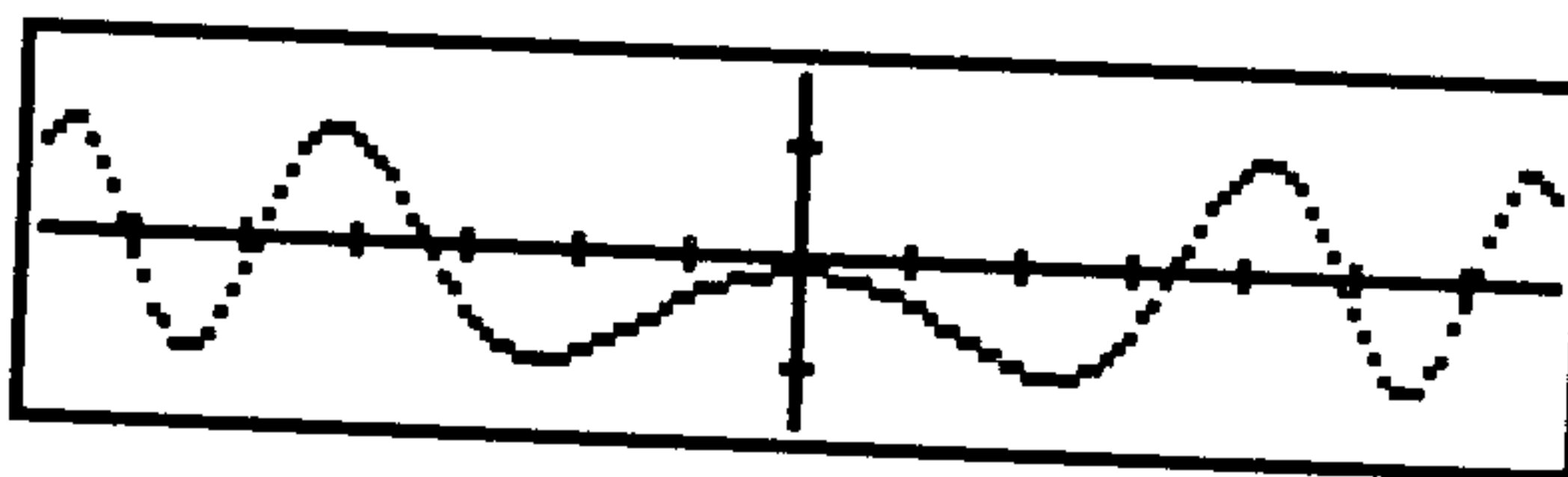
▶▶▶ ... (10 times) ■ ▲ INS

digitizes a new $P_{\max}$; then

◀◀◀ ... (20 times) ■ ▼ INS

digitizes a new $P_{\min}$. Press ON, and you should see the coordinates (1,1.6) in level 2, and (-1,-1.5) in level 1. Now press

PMIN PMAX DRAW



Now you have a meaningful curve, which you can use for further analysis.

Note that although this example expands the plot of a region around the origin, the method of digitizing new values for $P_{\min}$ and $P_{\max}$ works equally well for any portion of the plot. The region can be off-center in either or both directions.

13.4.2.3 Rescaling with *H and *W

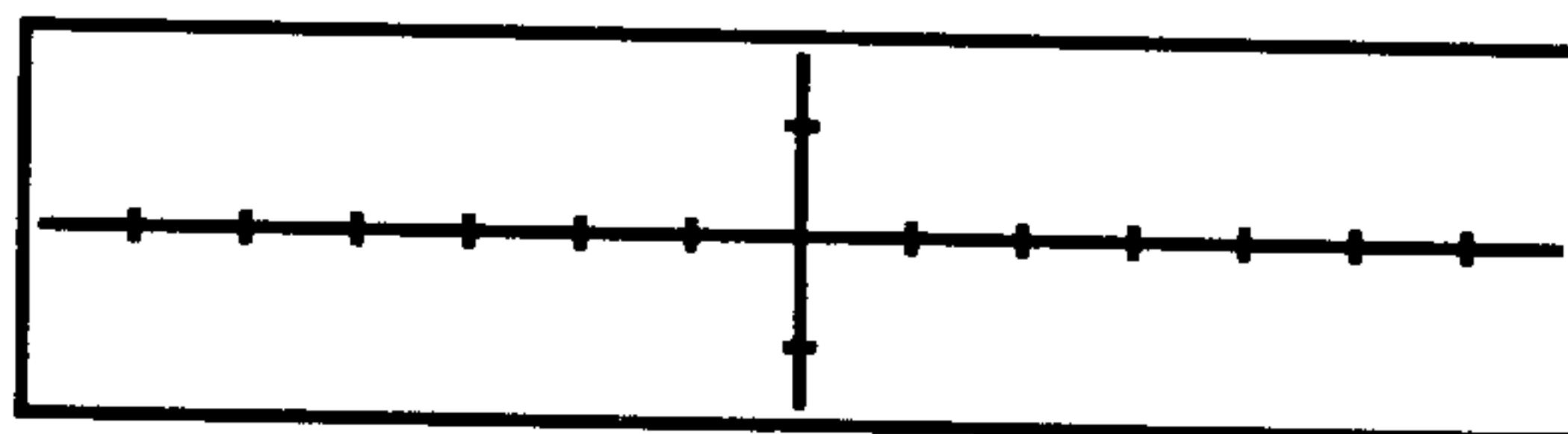
The method of digitizing new $P_{\min}$ and $P_{\max}$ described in the preceding section works well for focusing in on a subregion of a displayed plot. However, it won't serve for cases where the desired region is bigger than the current plot, or situated off-screen partly or completely. A common problem, for example, is that the vertical range of a plotted curve is larger than the display, so that important parts of the curve are not shown.

The commands *H and *W provide simple means for stretching or shrinking 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 enter 2 *H, then DRAW again. To make a plot cover twice the horizontal range of the previous plot (compressing the plotted curve sideways), enter 2 *W, then DRAW.

■ *Example.* Plot the expression $x(x-8)(x+8)$ to show its three roots. Start with the default plot parameters, to illustrate the rescaling process.

■ *Solution:*

```
'PPAR' PURGE 'X*(X-8)*(X+8)' STEQ DRAW
```

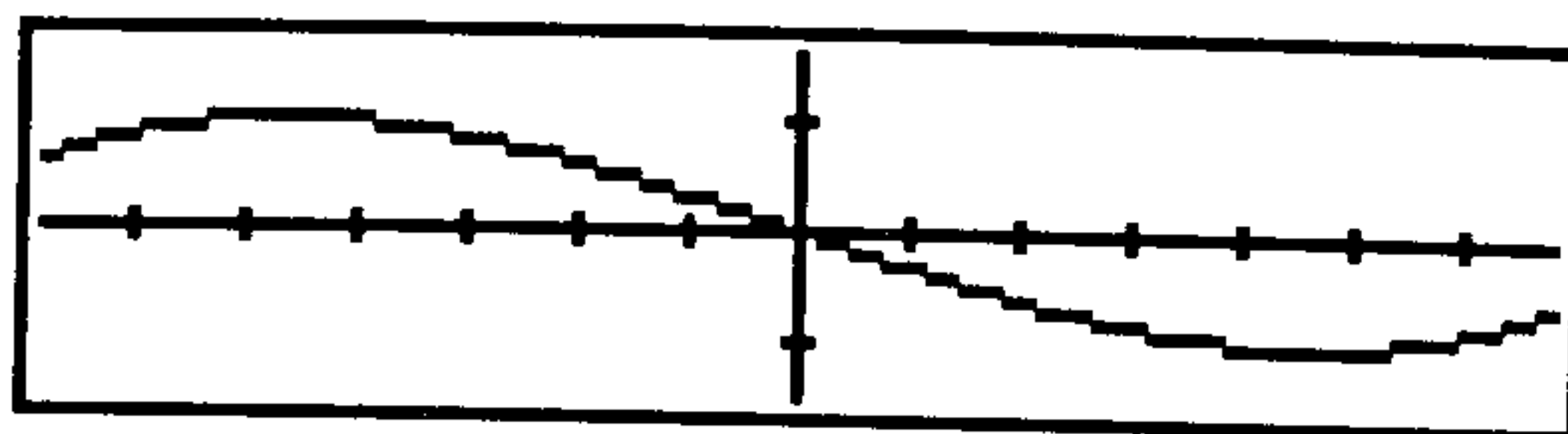


In this first attempt, you don't see the curve on the screen at all. This is because over the range of X shown, the curve is too positive or too negative to fall on the screen. It does cross the axis at $X=0$, but it's so steep there that any points that appear on the screen fall right on the axis. To improve matters you need to increase the vertical range of the plot.

The expression is a cubic in x , so you can estimate that its maximum value in the plot region is on the order of $x_{\max}^3 = 6.8^3 \approx 314$. The default $\psi_{\max}$ is 1.6, so you need to multiply the height of the plot by $314/1.6 \approx 200$. Thus

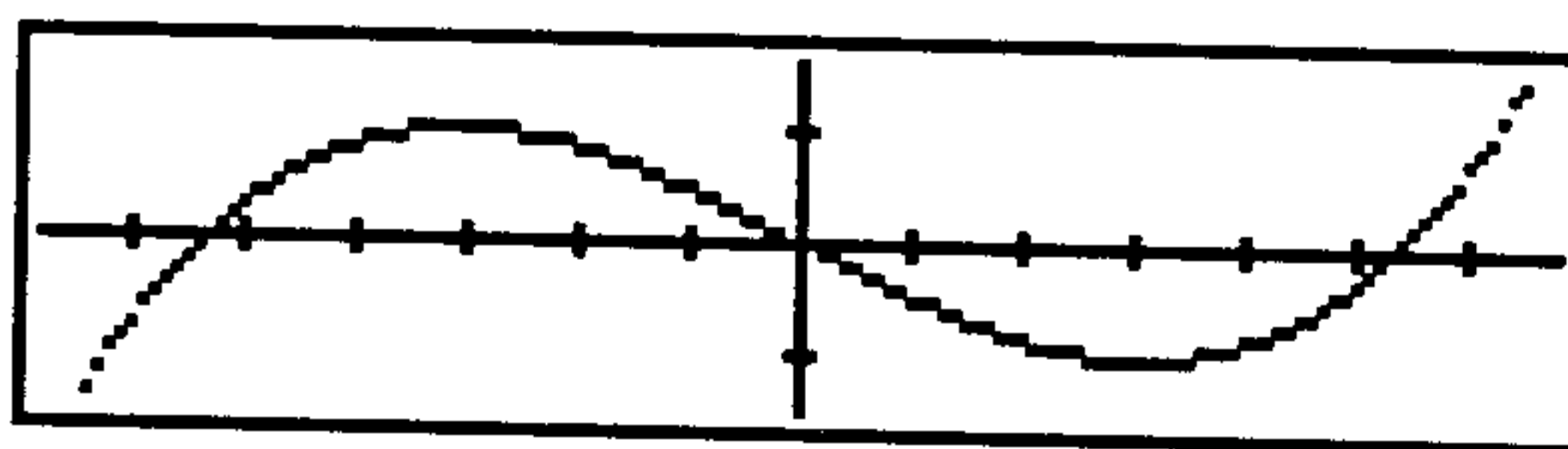
```
200 *H DRAW
```

yields



Now the curve fits nicely on the screen vertically, but the plot range is a bit too narrow to show the roots at +8 and -8. The next step is to stretch the horizontal range:

1.5 *W DRAW



This plot shows the three roots and the two extrema.

In the HP-28C, *H and *W work by simply multiplying the appropriate coordinates of $P_{\min}$ and $P_{\max}$ by the command arguments. Thus executing h *H produces the following new values for $P_{\min}$ and $P_{\max}$:

$$\chi'_{\min} = \chi_{\min}$$

$$\psi'_{\min} = h\psi_{\min}$$

$$\chi'_{\max} = \chi_{\max}$$

$$\psi'_{\max} = h\psi_{\max}$$

The primed quantities are the new values, and the unprimed quantities are the original coordinates. Similarly, w *W transforms the χ -coordinates:

$$\chi'_{\min} = \chi_{\min}$$

$$\psi'_{\min} = w\psi_{\min}$$

$$\chi'_{\max} = \chi_{\max}$$

$$\psi'_{\max} = w\psi_{\max}$$

This simple algorithm has the consequence that the coordinates of the plot center are multiplied by the same factor as the corner coordinates. Therefore, unless the center of the plot is originally at (0,0), the plot "moves" when it is redrawn after a rescaling by either of these two commands.

In the HP-28S, the algorithms for *H and *W are modified from the HP-28C versions. The HP-28S versions rescale the plot parameters by the specified factor, but keep the center of the plot fixed. Here are the HP-28 formulae:

*H

$$\psi'_{\min} = \frac{(16 + 15h)}{31} \psi_{\min} + \frac{(15 - 15h)}{31} \psi_{\max}$$

$$\psi'_{\max} = \frac{(16 - 16h)}{31} \psi_{\min} + \frac{(15 + 16h)}{31} \psi_{\max}$$

*W

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

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

You can duplicate HP-28S behavior on an HP-28C by substituting the following programs for *H and *W:

TH		*H Substitute	
level 1			level 1
h			
<< << *H >> RECENTER >>			
TW		*W Substitute	
level 1			level 1
w			
<< << *W >> RECENTER >>			

RECENTER	Rescale and Recenter
<pre> << 'PPAR' DUP 1 GET C-R ROT 2 GET C-R 15 * ROT 16 * + 31 / SWAP ROT + 2 / SWAP R-C 3 ROLLD EVAL CENTR >> </pre>	Get $P_{\min}$. Get $P_{\max}$. ψ_{cent} X_{cent} Execute *H or *W Restore the center

13.4.2.4 Positioning the Axes

Axes in a plot normally have the x and ψ axes drawn through the point (0,0). The HP-28 follows this convention as its default. However, in many cases you may be interested in a region sufficiently far from the origin that the axes don't appear on the screen. Then, if you still want to use axes to provide a visual reference, you can use the AXES command to cause subsequent plotting to draw axes through any point you choose. Just enter a complex number for the coordinates of the point at which you want the axes to intersect, then execute AXES. The selected coordinates are stored as the fifth and last element in the PPAR list.

The "tick" marks that are drawn on the axes are positioned at every tenth pixel, starting at the intersection of the axes. Thus the coordinate values corresponding to the tick marks vary according to the current plot scale, and the coordinates of the axes. With the default parameters, the tick marks are 1 unit apart (each pixel is 0.1). If you execute, for example, 2 *W, the ticks on the horizontal axes are then 2 units apart. In general, however, it is more trouble to figure out the positions of the ticks than the information is worth--use the cursor to determine the coordinates of points of interest.

13.5 Scatter Plots

A *scatter plot* is a plot of a series of individual 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 function plots, where the points usually follow each other in a regular progression that forms a smooth curve. In the HP-28, scatter plots are made by the DRW Σ (*Draw Statistics*) command. The $\equiv\text{DRW}\Sigma\equiv$ key in the PLOT menu works like $\equiv\text{DRAW}\equiv$, in that it not only executes DRW Σ to draw axes and plot the data points, but also activates the plot cursor for digitizing.

Most of the concepts that we have studied in this chapter for function plots carry over directly for scatter plots--digitizing, plot scaling and rescaling, etc. Three of the five

parameters stored in PPAR are used by scatter plots: $P_{\min}$, $P_{\max}$, and P_{axes} . The *resolution* (section 13.1.4) has no meaning for scatter plots, nor do scatter plots make any use of the independent variable name.

ΣDRW takes the data for a scatter plot from the statistics matrix variable ΣDAT , which plays a role for statistics analogous to EQ for function plotting and solving. Each row in ΣDAT represents the coordinates of a point in n dimensions, where n is the number of columns in the matrix. ΣDAT must have at least two columns for this purpose. To make a plot on a two-dimensional screen, you must select one column of the matrix to represent the horizontal coordinate (χ) and another column for the vertical (ψ). These coordinates are not independent and dependent in the sense of function plots, since the points are just coordinate pairs; but we'll call them χ and ψ as before.

You make the choice of ΣDAT columns for χ and ψ with the command $\text{COL}\Sigma$. $\text{COL}\Sigma$ takes two real numbers from the stack; the entry in level 2 specifies the χ column, and that in level 1 the ψ column. For example, if your ΣDAT looks like this,

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

then $1\ 2\ \text{COL}\Sigma$ selects column 1 as the horizontal coordinate, and column 2 as the vertical coordinate, so that $\text{DRW}\Sigma$ plots the points (11,12), (21,22), and (31, 32). $3\ 2\ \text{COL}\Sigma$ $\text{DRW}\Sigma$ plots the points (13,12), (23,22), and (33,32).

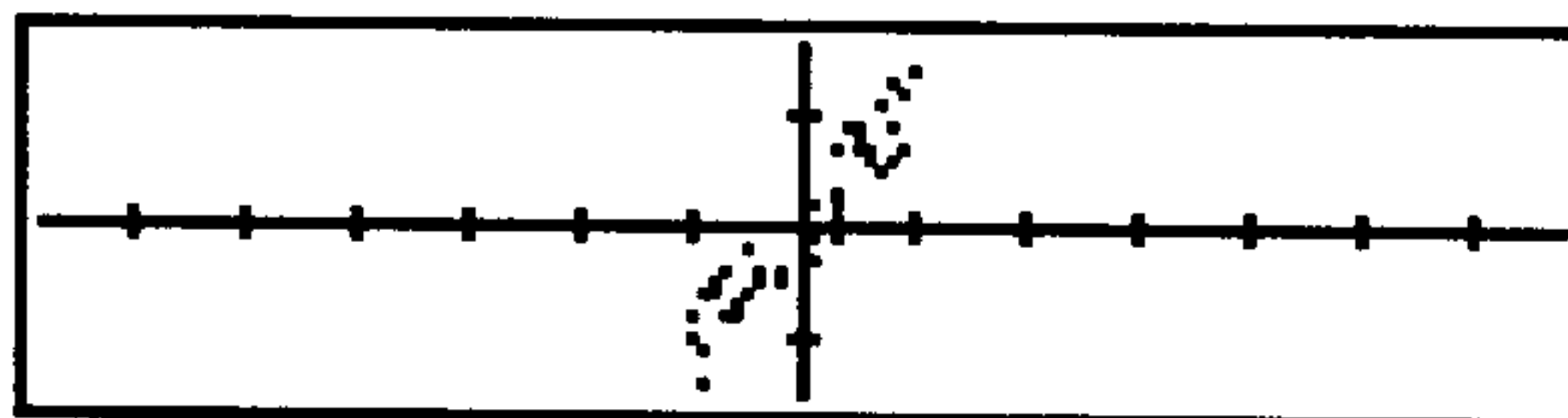
Because the column numbers identified by $\text{COL}\Sigma$ are used by other statistics functions, they are not stored in PPAR. Instead, the column numbers are stored as the first two elements in another reserved-name list variable ΣPAR (Statistics Parameters). If you don't specify column numbers, $\text{DRW}\Sigma$ (and the other commands that use the numbers) creates a default ΣPAR with horizontal coordinate column number 1 and vertical coordinate column number 2.

To illustrate the process of making a scatter plot, you can use the following program to create some sample data. $\text{MAKE}\Sigma$ creates a ΣDAT that contains 50 points scattered around the straight line $\psi = \chi$, between $\chi = -1$ and $\chi = +1$:

MAKEΣ <i>Make Statistics Data</i>	
<pre><< .12345 RDZ CLΣ 1 50 START RAND .5 - 2 * DUP RAND + .5 - { 2 } →ARRY Σ+ NEXT >></pre>	Initialize the random number generator. χ_n. ψ_n. Add to ΣDAT.

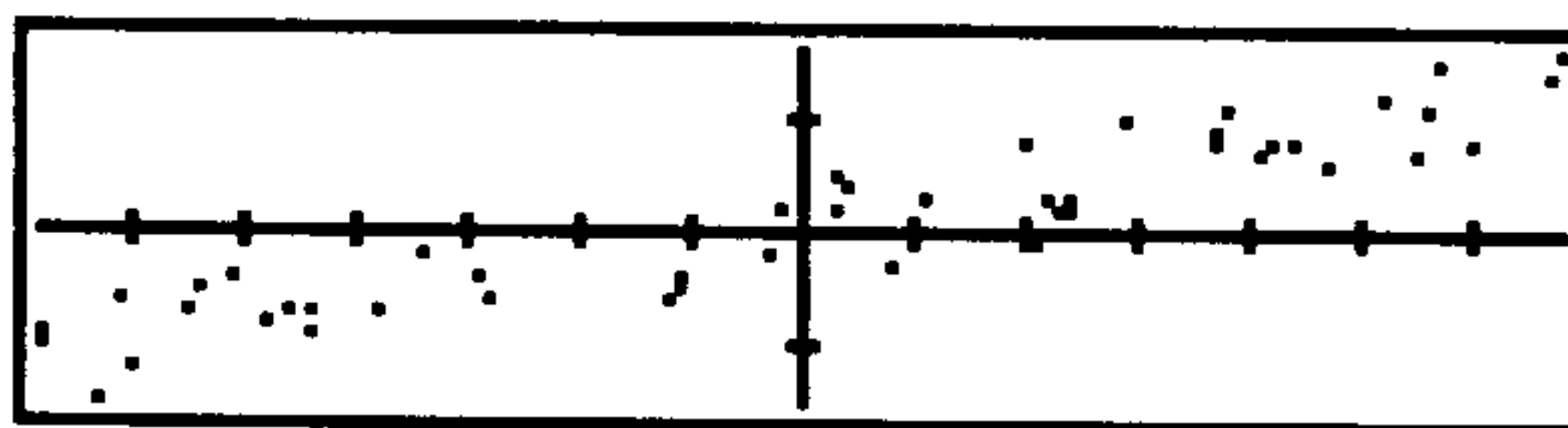
To plot this data, execute:

MAKEΣ 'PPAR' PURGE 1 2 COLΣ DRWΣ



The picture shows a cluster of points near the origin. You can use any of the methods of section 13.4.2 to adjust the plot scale to spread the points out over more of the screen. However, for a scatter plot the coordinates of all of the points are known in advance, so it is possible for the calculator automatically to compute plot parameters such that the points are spread in both directions from one end of the screen to the other, with no points off-screen. The command SCLΣ (*Scale Statistics*) does this rescaling. Try plotting the data again:

SCLΣ DRWΣ

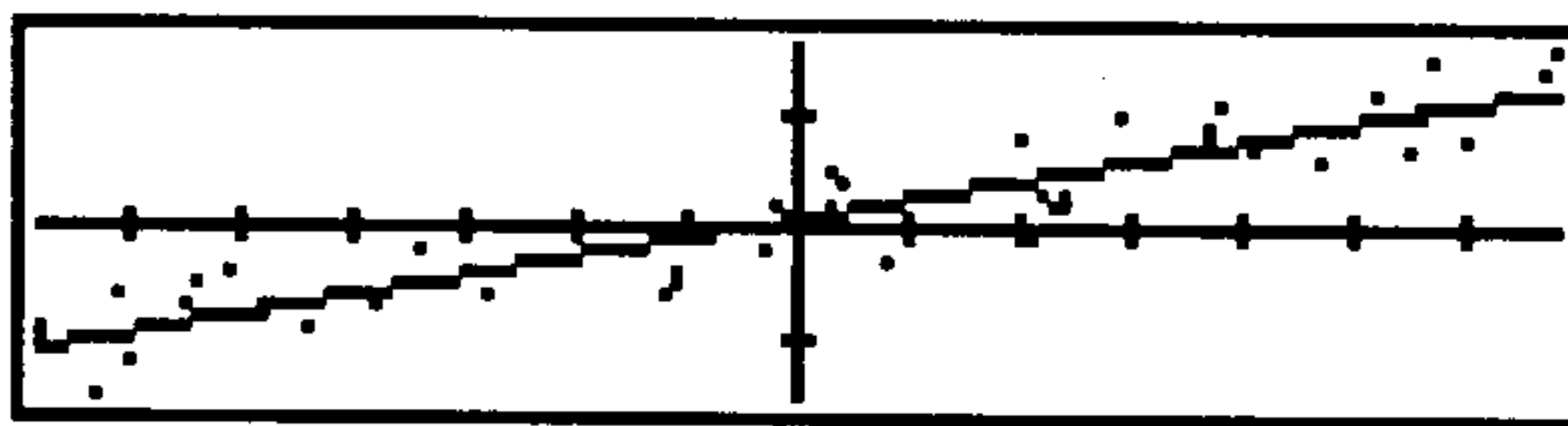


Notice that there is at least one point plotted on each edge of the screen. If you're curious, you can count and verify that all 50 points are visible (you'll have to replot the picture with the axes moved off-screen, since some of the points fall on the axes). If you take a look at PPAR, you'll see that $P_{\min}$ and $P_{\max}$ have been redefined from their

default values.

Since the sample data set is a “noisy” straight line, you ought to be able to find a straight line of the form $\psi = a\chi + b$ that fits the data. LR computes the coefficients a and b ; with the current data LR returns $b \approx .0421$ (level 2) and $a \approx 1.086$ (level 1). To see whether the fit is reasonable, plot the data and the regression line together:

```
LR << X PREDV >> STEQ CLLCD DRAW DRWΣ
```



The picture shows how the data is scattered around the straight line.

13.6 Generalized Plotting using PIXEL

DRAW and DRWΣ are very convenient methods for making certain common types of plots. However, there are many plotting problems that don't fit the requirements of these commands. Plotting in polar coordinates is an obvious example of a simple function plotting task that can't be achieved using DRAW.

To make any plot, all you need is 1) a program to determine the coordinates in two dimensions of each point that you want plotted, and 2) a command to turn on the display pixel that corresponds to the point coordinates. The command PIXEL satisfies the second of these requirements; the coordinates program, of course, you must write yourself using any of the HP-28's capabilities.

PIXEL takes a single complex number (χ, ψ) as its argument, interpreted as the χ - and ψ - coordinates of a point to plot. The actual pixel that corresponds to those coordinates is determined from the equations in section 13.4.1, using the plot parameters $P_{\min}$ and $P_{\max}$ stored in PPAR (like the other plotting commands, PIXEL will create a default PPAR if it doesn't already exist).

The program DRAWPIX is equivalent to DRAW (assuming that EQ contains an expression or a program, and that PPAR already exists). Although there's no particular reason to use DRAWPIX instead of DRAW, the program illustrates the use of PIXEL, and you can use it as a starting point for writing specialized forms of DRAW.

DRAWPIX	DRAW using PIXEL
<pre> << CLLCD DRAX PPAR 1 GET RE PPAR 2 GET RE DUP2 - 136 / PPAR 4 GET * PPAR 3 GET → step indep << indep RCL 3 ROLLD FOR x x indep STO EQ →NUM x SWAP R→C PIXEL step STEP indep STO >> >> </pre>	Prepare the screen. Get $x_{\min}$. Get $x_{\max}$. Compute the step size. Get the independent variable name x. Save the initial value of x. Loop from $x_{\min}$ to $x_{\max}$: Store the current value of x in the independent variable. Evaluate the current equation (ψ). Combine the coordinates into a complex number. Plot the point. Increment x and repeat. Restore the original value.

DRAWPIX shows the basic steps in creating a plot with PIXEL:

1. Clear the screen (unless you are adding to an existing picture).
2. Draw axes if desired.
3. Execute a loop that plots one or more points at each iteration.

You can consider the function plotting performed by DRAW as a special case of *parameterized* plotting. In the general case, *both* coordinates of each plotted point are computed as functions of a parameter ϕ , as ϕ is varied over a specified range. To specify a parameterized plot, you must determine:

- the range of the parameter $\phi_{\min} \leq \phi \leq \phi_{\max}$;
- the increment $\delta\phi$ between successive values of the parameter; and
- the functions $\chi(\phi)$ and $\psi(\phi)$ that determine the coordinates (χ, ψ) of each point.

For DRAW, the parameter is just the horizontal coordinate x itself, which is varied over the range $x_{\min}$ to $x_{\max}$. In the general case, each point in a parameterized plot is generated by a sequence like this:

$\chi(\phi) \quad \psi(\phi) \quad R \rightarrow C \quad \text{PIXEL},$

where $\chi(\phi)$ and $\psi(\phi)$ represent sequences that compute the horizontal and vertical coordinates, respectively, from the parameter ϕ .

13.6.1 Polar Plots

(All of the examples in this section assume the use of the default plot parameters, and degrees mode. If you want to follow the examples on your calculator, execute 'PPAR' PURGE DEG before starting.)

An excellent example of parameterized plotting is polar coordinate plotting. Since polar plots are usually defined by specifying the radius r as a function of the polar angle θ , the natural parameter to use is θ itself. You can compute pixel coordinates as complex numbers in polar form (r, θ) , then convert to the rectangular coordinates needed by PIXEL by applying P→R.

The range $\theta_{\min}$ to $\theta_{\max}$ depends on each individual problem; typically, θ might range from 0° to a multiple of 360° . Similarly, the increment $\delta\theta$ depends on how much detail you wish to show. The angular width of a pixel at the top middle of the screen is about 3.5° when viewed from the center of the screen; at the middle left or right edge, it is about 0.8° . Therefore, values of $\delta\theta$ from 1° to 5° will generally produce satisfactory plots.

The following program POLAR is a general-purpose polar-coordinate plotting routine.

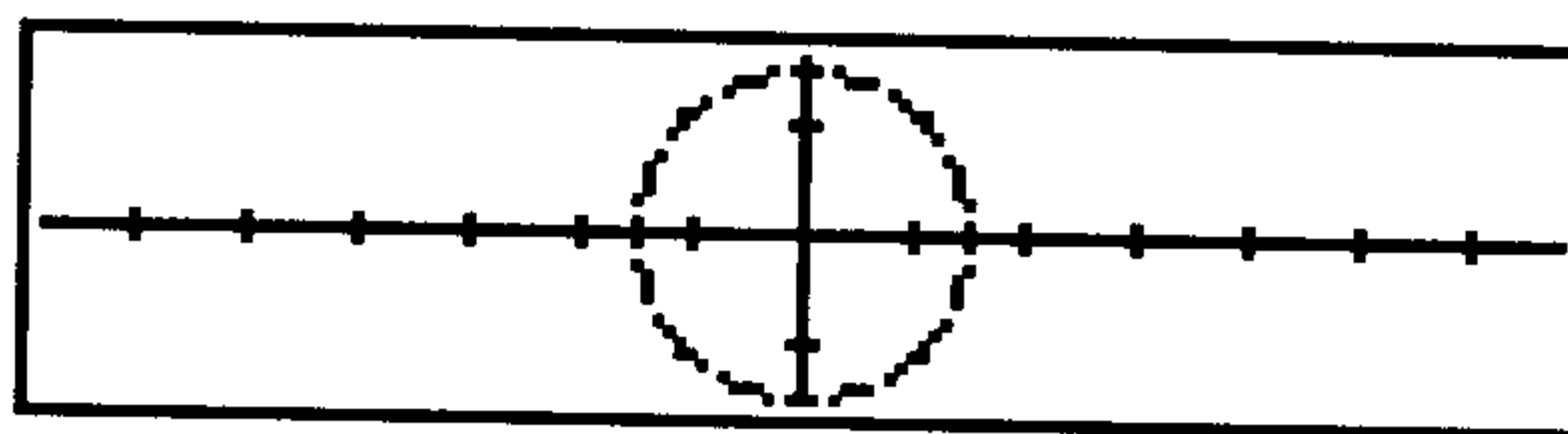
POLAR		Polar Draw			
		level 4	level 3	level 2	level 1
		procedure	$\theta_{\min}$	$\theta_{\max}$	$\delta\theta$
<pre> << CLLCD DRAX 4 ROLL → delta r << FOR theta theta r EVAL theta R→C P→R PIXEL delta STEP >> >> </pre>		Initialize the screen. Save the radius procedure and increment. Compute $r(\theta)$. Polar coordinates. Plot the point. Increment θ.			

POLAR takes four arguments:

1. A procedure that takes a value for θ and returns $r(\theta)$. The procedure should have the logical form `<< → t 'r(t)' >>`, where $r(t)$ is the expression that represents the plotting function.
2. The initial value $\theta_{\min}$.
3. The final value $\theta_{\max}$.
4. The increment $\delta\theta$.

■ *Example.* Plot a circle of radius 1.5.

```
<< → t '1.5' >> 0 360 5 POLAR
```



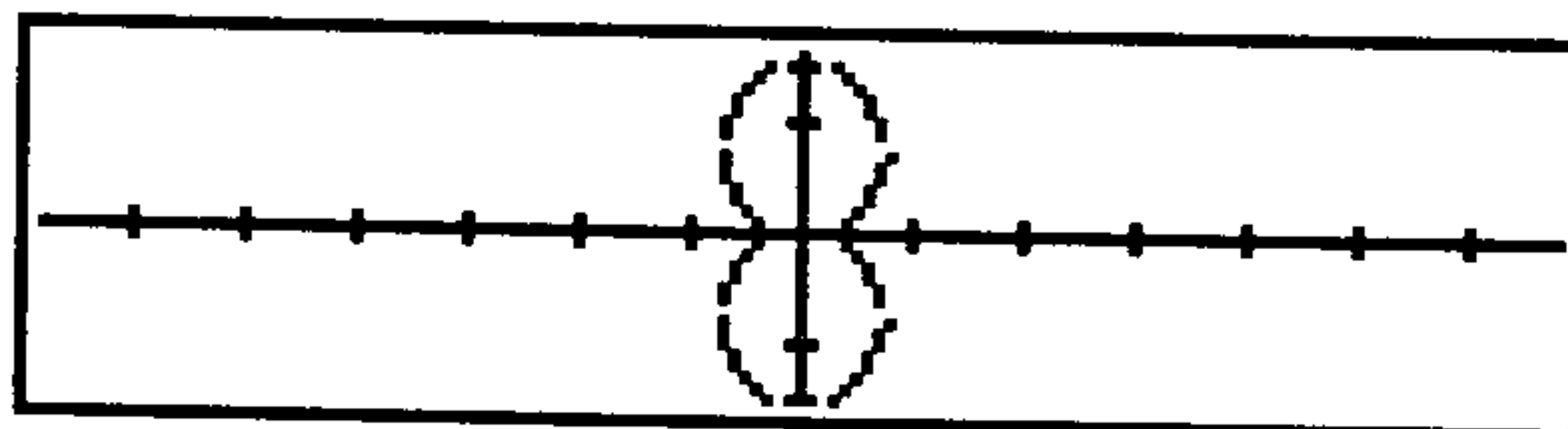
Notice that in this case r does not depend on θ , so the input procedure effectively discards the value of θ supplied by POLAR by storing it in the local variable t , which is never used. A more efficient version of the procedure is

```
<< DROP 1.5 >>.
```

Here are some examples where r is a function of θ :

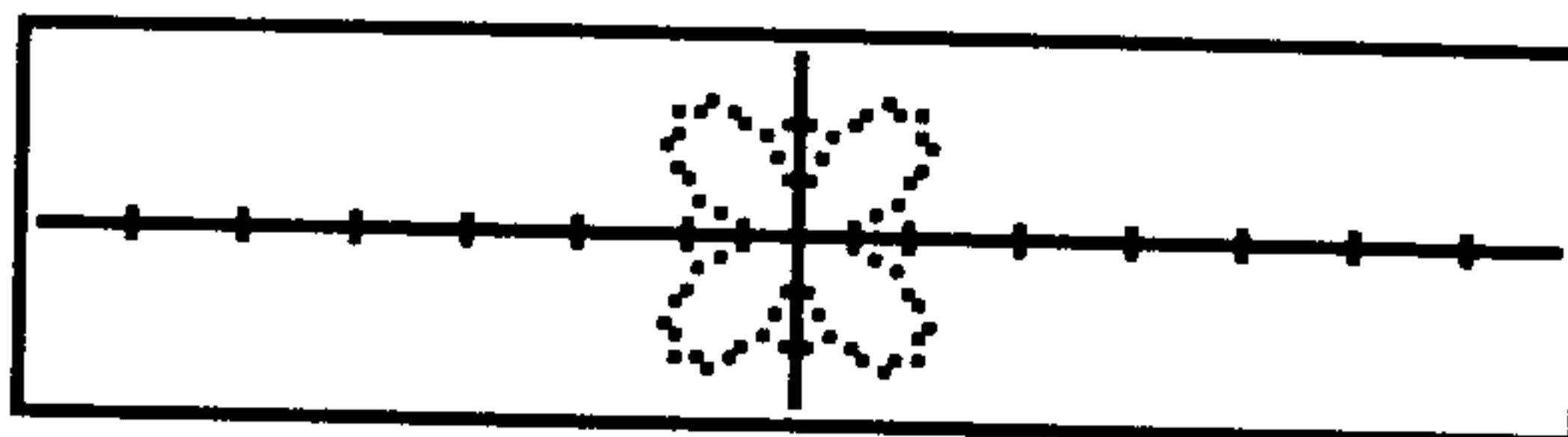
■ *Figure 8.*

```
<< → t 'ABS(1.5*SIN(t))' >> 0 360 5 POLAR
```



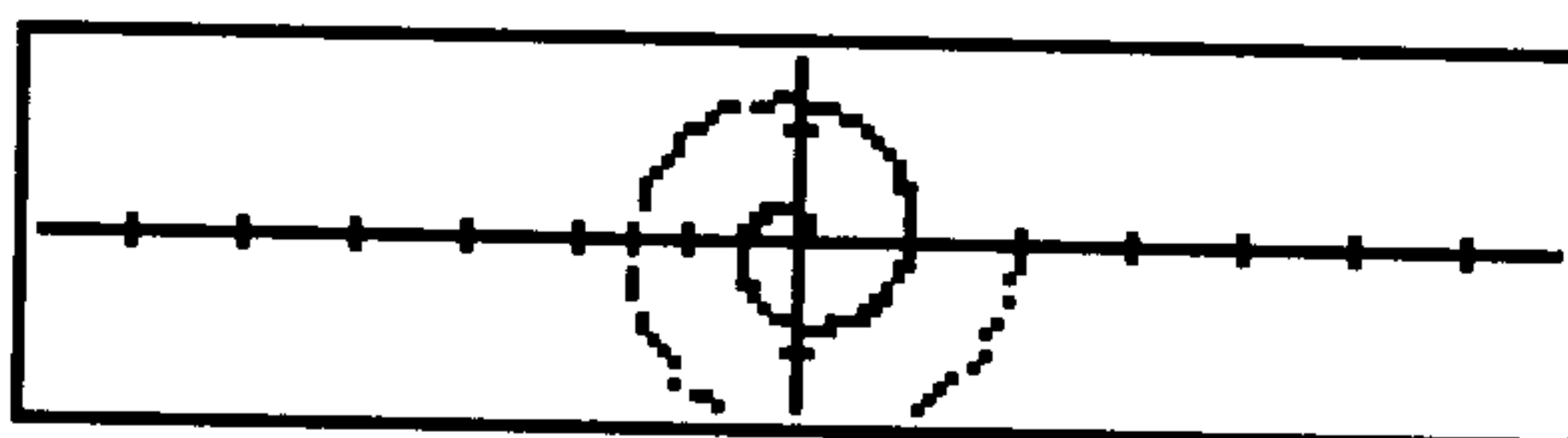
■ *Rose.*

```
<< → t 'ABS(1.5*SIN(2*t))' >> 0 360 5 POLAR
```

■ Spiral.

```
<<  → t 't/360' >> 0 720 5 POLAR
```



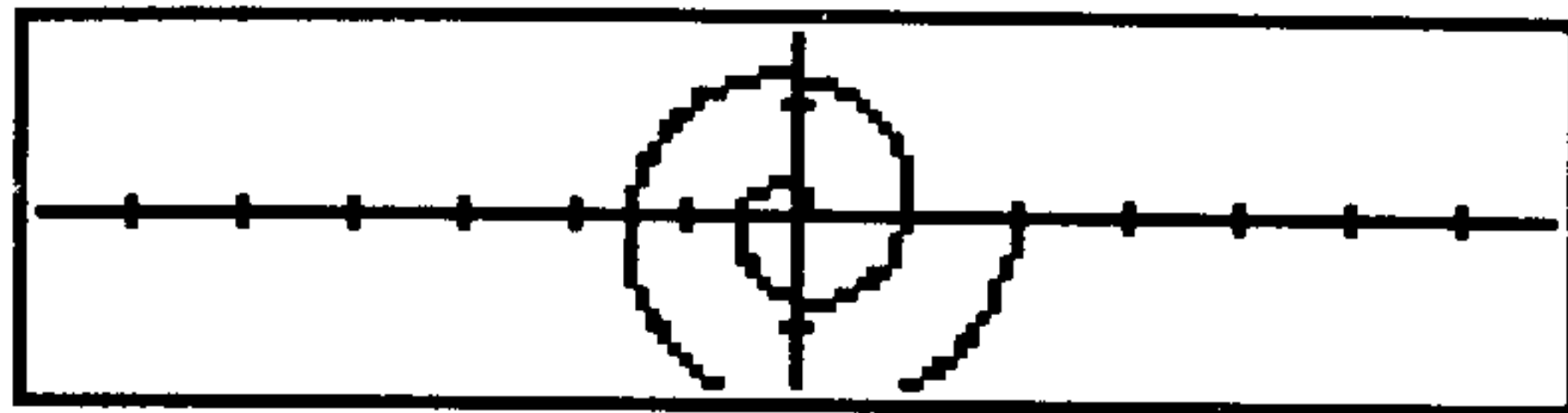
If you plotted the last example, you may have noticed that the plotting went slowly at first. This is because when the radial coordinate is small, the 5° increment is insufficient to cause consecutive points to fall within different pixels, so that some pixels get plotted more than once. This suggests a modification of POLAR, where the new version VTPOLAR itself computes the increment $\delta\theta$ as a function of r .

VTPOLAR				
Variable θ Polar Draw				
	level 3	level 2	level 1	
	procedure	$\theta_{\min}$	$\theta_{\max}$	$\delta\theta$
<pre> << CLLCD DRAX 59 CF ROT → r << FOR theta theta r EVAL DUP theta R→C P→R PIXEL INV 2 MIN 5 * STEP >> >> </pre>				
Initialize the screen. Handle the $r=0$ case. Save the radius procedure. Compute $r(\theta)$. Polar coordinates. Plot the point. Compute the increment. Increment θ .				

VTPOLAR clears flag 59.

You use VTPOLAR in the same manner as POLAR, except that you don't specify $\delta\theta$ as an argument. Try redrawing the spiral with VTPOLAR:

```
<<  ->  t  't/360'  >>  0  720  VTPOLAR
```



VTPOLAR spends less time plotting near the origin than POLAR (with a 5° increment), but more time as the spiral reaches larger radii, as it attempts to insure that no pixels are double-plotted, and also that there are no gaps in the curve. It uses an increment $\delta\theta = r/5$ (with a 10° minimum), which produces reasonable results with the default plot parameters. If you adjust the plot scale, you may also wish to modify this algorithm.

13.6.2 Drawing Lines

A basic necessity for general plotting purposes is a program that draws a straight line between any two points. The program LINE connects two points specified by their coordinates (complex numbers) in levels 1 and 2. LINE uses the plot scale represented by the current value of PPAR, and does not try to verify that either or both points are on the screen. A second program, SKETCH, uses LINE to draw a series of line segments, specified by a list of complex numbers representing points to be connected in order.

LINE draws a line consisting of the series of points

$$\vec{\chi}_n = \vec{\chi}_i + n r (\vec{\chi}_f - \vec{\chi}_i) / N$$

where $\vec{\chi}_i$ and $\vec{\chi}_f$ are the initial and final points, respectively, and $n = 0, 1, \dots, N$. The number of steps N is given by

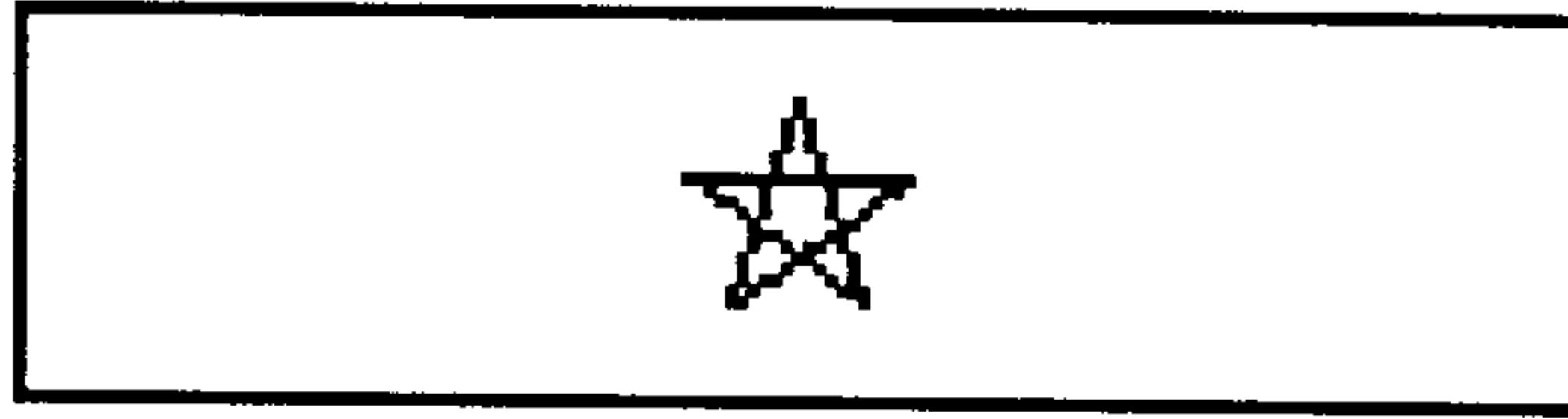
$$N = \text{MAX} \left[\left| \frac{(\psi_f - \psi_i)}{\Delta\psi} \right|, \left| \frac{(\chi_f - \chi_i)}{\Delta\chi} \right| \right]$$

where $\Delta\chi$ and $\Delta\psi$ are the scale per pixel in the horizontal and vertical directions:

$$\Delta\chi = \frac{\chi_{\max} - \chi_{\min}}{136}$$

$$\Delta\psi = \frac{\psi_{\max} - \psi_{\min}}{31}$$

To demonstrate the use of LINE and SKETCH, the program STAR draws a 5-pointed star. Executing STAR produces this picture:



LINE		Draw a Line	
	level 2	level 1	
	(x_i, ψ_i)	(x_f, ψ_f)	
<< OVER PIXEL IF DUP2 $\neq$ THEN OVER - PPAR DUP 2 GET SWAP 1 GET - C-R 31 / SWAP 136 / 3 PICK C-R 4 ROLL / ABS SWAP ROT / ABS MAX PPAR 4 GET / SWAP OVER / ROT 1 4 ROLL START OVER + DUP PIXEL NEXT END DROP2 >>	Plot the first point; insure PPAR exists. Keep going only if the points are different. $ \vec{x}_i \quad \vec{x}_f - \vec{x}_i $ $ \vec{x}_i \quad \vec{x}_f - \vec{x}_i \quad \Delta\psi \quad \Delta\chi $ $ \vec{x}_i \quad \vec{x}_f - \vec{x}_i \quad \psi_{\max} - \psi_{\min} \quad \chi_{\max} - \chi_{\min} \quad \Delta\chi \quad \Delta\psi $ $ \vec{x}_i \quad \vec{x}_f - \vec{x}_i \quad \chi_{\max} - \chi_{\min} \quad \Delta\chi \quad N_\psi $ $ \vec{x}_i \quad \vec{x}_f - \vec{x}_i \quad N_\psi \quad N_\chi $ $ \vec{x}_i \quad \vec{x}_f - \vec{x}_i \quad N $ Divide by resolution. $ \vec{x}_i \quad N \vec{k} = (\vec{x}_f - \vec{x}_i)/N $ $ \vec{k} \quad \vec{x}_i $ Plot a point.		
	$ \vec{k} \quad \chi_n $		

SKETCH <i>Sketch Lines</i>	
<i>level 1</i>	
{ <i>list of points</i> }	
<pre> << → points << 1 points SIZE 1 - FOR n points n GETI 3 ROLLD GET LINE NEXT >> >> </pre>	Store the list. One fewer lines than points. Get the next pair of points. Connect the pair.
STAR <i>Draw a Star</i>	
<pre> << CLLCD 'PPAR' PURGE DEG (0,1) DUP R→P 0 4 START (0,144) - DUP P→R SWAP NEXT DROP 6 →LIST SKETCH >> </pre>	Initialize. Start at (0,1). Rotate by 144°. Add the point to the stack. Combine into a list. Connect the dots.

STAR sets degrees mode.

13.6.3 Elaborating DRAW

Although DRAW is designed for automated plotting of algebraic expressions, the fact that it will work perfectly well with a program as the current “equation” allows you to extend its operation beyond its simple definition. In particular, you can combine the automated properties of DRAW with the customized plotting you can achieve with PIXEL.

■ *Example.* Plot $\sin x$, $\cos x$, and $\tan x$ together.

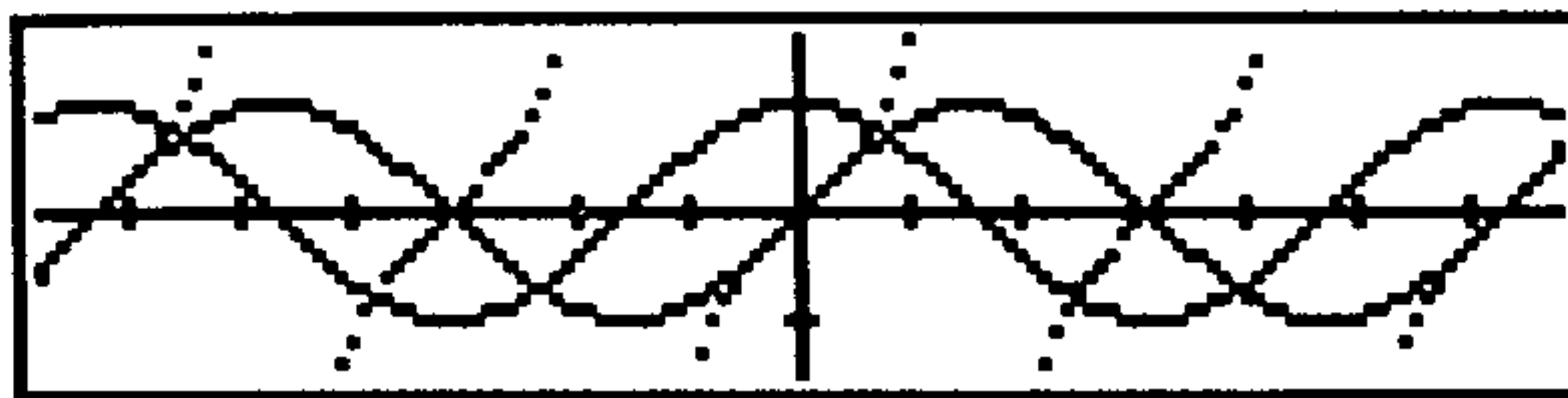
■ *Solution:*

```

<< X X SIN R→C PIXEL X X COS R→C PIXEL X TAN >>

      STEQ RAD CLLCD DRAW

```

Here DRAW accomplishes both plotting $\tan x$ and the automatic incrementing of the independent variable X . DRAW computes each value of X , evaluates EQ, then uses the value returned by EQ to plot one point. In addition to returning one value, EQ itself also plots points corresponding to $\sin x$ and $\cos x$, using the latest value of x stored in X by DRAW.

The general rule for this sort of extension of DRAW is that the program stored in EQ must act like an algebraic expression. This means that it must take no arguments from the stack, and return one real number to be used as the ψ -coordinate for DRAW. Also, you must insure that the independent variable name appears in the program or in one of its subprograms. If these requirements are satisfied, you can do anything else within the program that you want.

The program GDRAW is an extension of DRAW that you can use in conjunction with the Solver. The program plots a specified expression, and also returns a list of independent variable values that are suitable first guesses for the Solver.

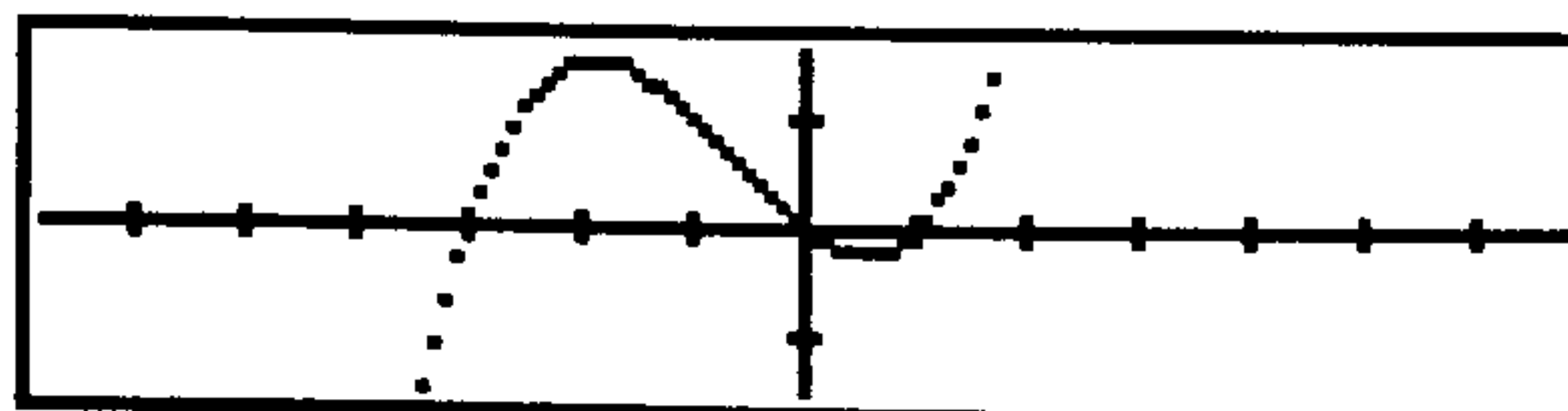
To use GDRAW:

1. Set the plot parameters as you would for DRAW. If you don't specify any parameters, the current PPAR is used; a default PPAR is created if necessary.
2. Enter the plot procedure into level 2, and the name of the independent variable into level 1. The procedure is the same as you would store in EQ for DRAW.
3. Execute GDRAW. GDRAW will make the plot; when you are finished viewing the picture, press **ON** to clear the plot. You will then see (after the "busy" annunciator turns off) the plot procedure in level 2 and a list of values of the independent variable in level 1. The values in the list are values of x at or near which the plotted function crosses the x -axis.

GDRAW		Guesses from DRAW	
level 2	level 1		level 2 level 1
procedure 'name'		☞	procedure { guesses }
<pre> << DUP INDEP PPAR 1 GET RE OVER STO OVER →NUM SIGN { } → eq x sign list << << eq →NUM sign OVER SIGN DUP 'sign' STO SWAP OVER IF - ABS 2 == SWAP NOT OR THEN list x EVAL 1 →LIST + 'list' STO END >> STEQ CLLCD DRAW 'EQ' PURGE eq list >> >> </pre>			Store the independent variable name. Initialize the independent variable with x_{min}. Compute the sign of $f(x_{min})$. Store the function, the name, the sign, and an empty list. Start of EQ procedure: Evaluate ψ. Get the old sign(ψ). Store the new sign. If the sign has changed... Or if $\psi = 0$... ...then add x to the list. Store the procedure in EQ. Make the plot. Discard EQ. Return the function and the list.

■ **Example.** Plot $x(x-1)(x+3)$, and obtain estimates of its roots.

■ **Solution:** 'PPAR' PURGE 4 *H 'X*(X-1)*(X+3)' 'X' GDRAW
☞ 'X*(X-1)*(X+3)' { -3 0 1 }



In this case, the three guesses -3 , 0 , and 1 returned by GDRAW are the exact roots of the expression.

13.7 Printing Plots

If you have an HP 82240A Printer, you can make a printed record of any plot or other picture that you make on the HP-28 display. The primary tool for this purpose is the PRLCD (*P*rint *L*CD) command, which causes transmission of a pixel-by-pixel replica of the screen from the calculator to the printer.

On the HP-28C, you must decide in advance that you want to print the LCD picture *before* you create it, since any keyboard action that you take to execute PRLCD will alter or destroy the picture. For example, if you want to print a DRAW plot, you must execute DRAW from the command line or in a program (that is, don't press DRAW in immediate-execute mode), and include PRLCD as part of the same command sequence. (The shortest such sequence is CLLCD DRAW PRLCD, executed together in a common program or command line.) This is unfortunate, since in many cases you don't know that you want to print a picture until after you've seen it. If you want to print a plot when you did not prearrange the printing, you have no recourse except to clear the plot and redraw it with CLLCD DRAW PRLCD.

The HP-28S addresses this problem in two ways:

- The HP-28S has a "hot-key" version of PRLCD: at any time, you can press and hold **ON**, then press **L** (the PRINT menu selection key). When you release the keys, PRLCD is executed, without disturbing the current appearance of the display. This capability is available even when a special temporary display is showing, including when the plot cursor menu is active.
- You can save a picture with LCD→ (section 13.3), then print it later by returning the image string to level 1 and executing →LCD PRLCD together, or just →LCD followed by pressing **ON** - **L**. See section 13.3.

13.7.1 Extended Plots

PRLCD does not transmit any extra linefeed commands to the printer, so that successive PRLCD's can print a continuous picture. You can take advantage of this property to produce printed plots in which the vertical range is much bigger than is possible on the HP-28 display. As a practical illustration, the program X4PLOT stretches the vertical range represented by the current PPAR by a factor of four, making a roughly square printed plot (137×128 pixels) from four separately plotted pictures.

X4PLOT 4 × Plotting	
level 1	
<< program >>	
<pre> << IF PPAR TYPE 6 == THEN 1 *H END PPAR DUP LIST→ 4 DROPN - IM -127 / 32 * 0 SWAP R→C → proc ppar dy << 'PPAR' 1 OVER EVAL 1 GET dy 3 * + PUT CLLCD proc EVAL PRLCD 1 3 START PPAR LIST→ 6 ROLL dy - 6 ROLL dy - 6 ROLLD 6 ROLLD →LIST 'PPAR' STO CLLCD proc EVAL PRLCD NEXT ppar 'PPAR' STO >> >> </pre>	Does PPAR not exist? Then create a default PPAR. $P_{\min}$ $P_{\max}$ Vertical scale per pixel. Vertical scale per screen. Screen offset. Save the procedure, PPAR and screen offset. $P_{\min}$ New $P_{\min}$ for 1st picture. Replace $P_{\min}$ in PPAR. Print the first picture. Iterate 3 times: Unpack the plot parameters. New $P_{\min}$ New $P_{\max}$ Pack up the new parameters. Print the next picture. Restore the original PPAR.

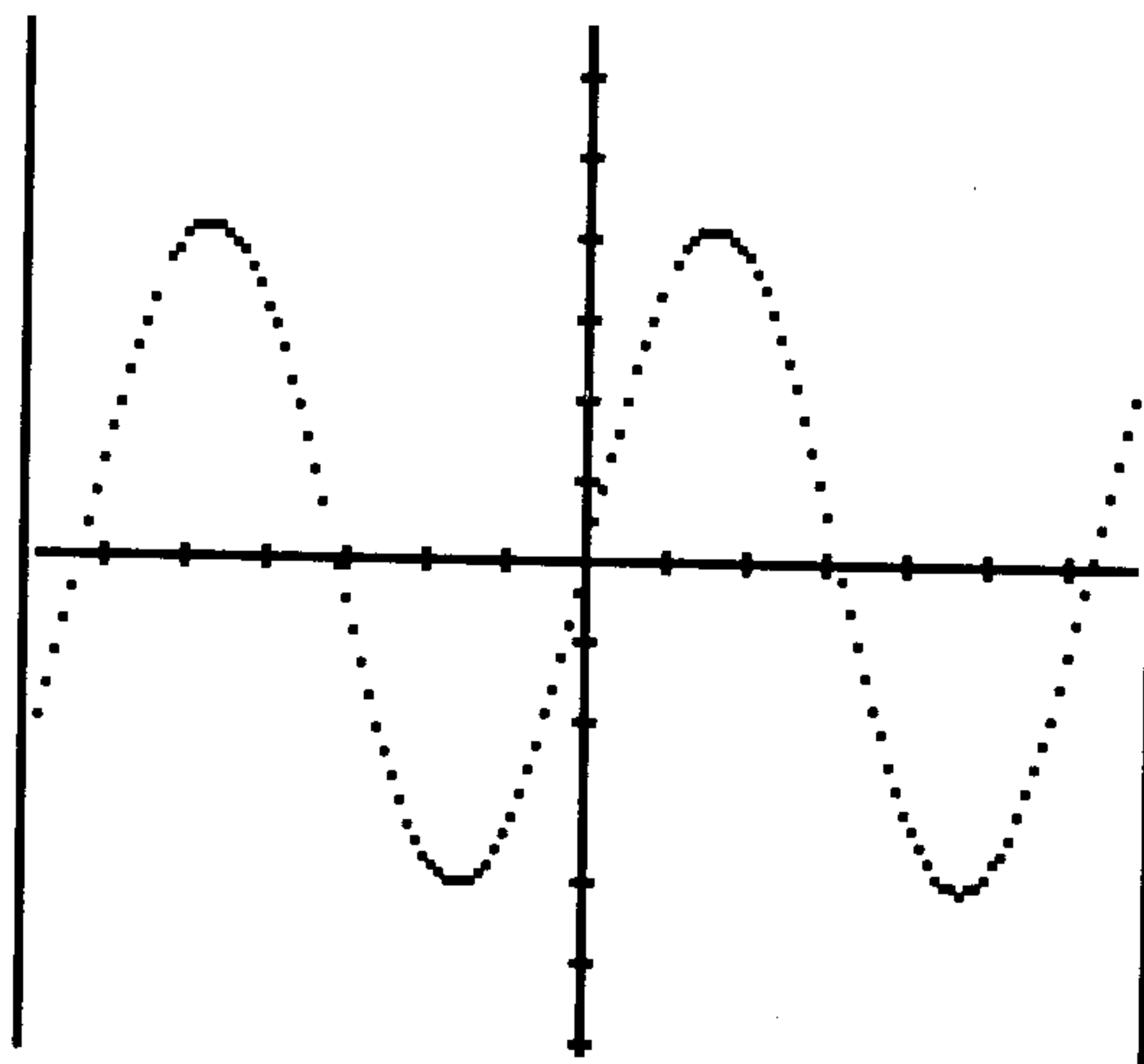
X4PLOT assumes that PPAR already exists.

X4PLOT takes one argument, a program that makes an ordinary plot by any method. It actually executes the program four times, each time with a different set of plot parameters. The easiest way to use X4PLOT is as an extension of DRAW, to make a 4× picture of the current equation. The program argument in that case is << DRAW >>.

■ *Example.* Make a 4× plot of $\sin x$.

■ *Solution:*

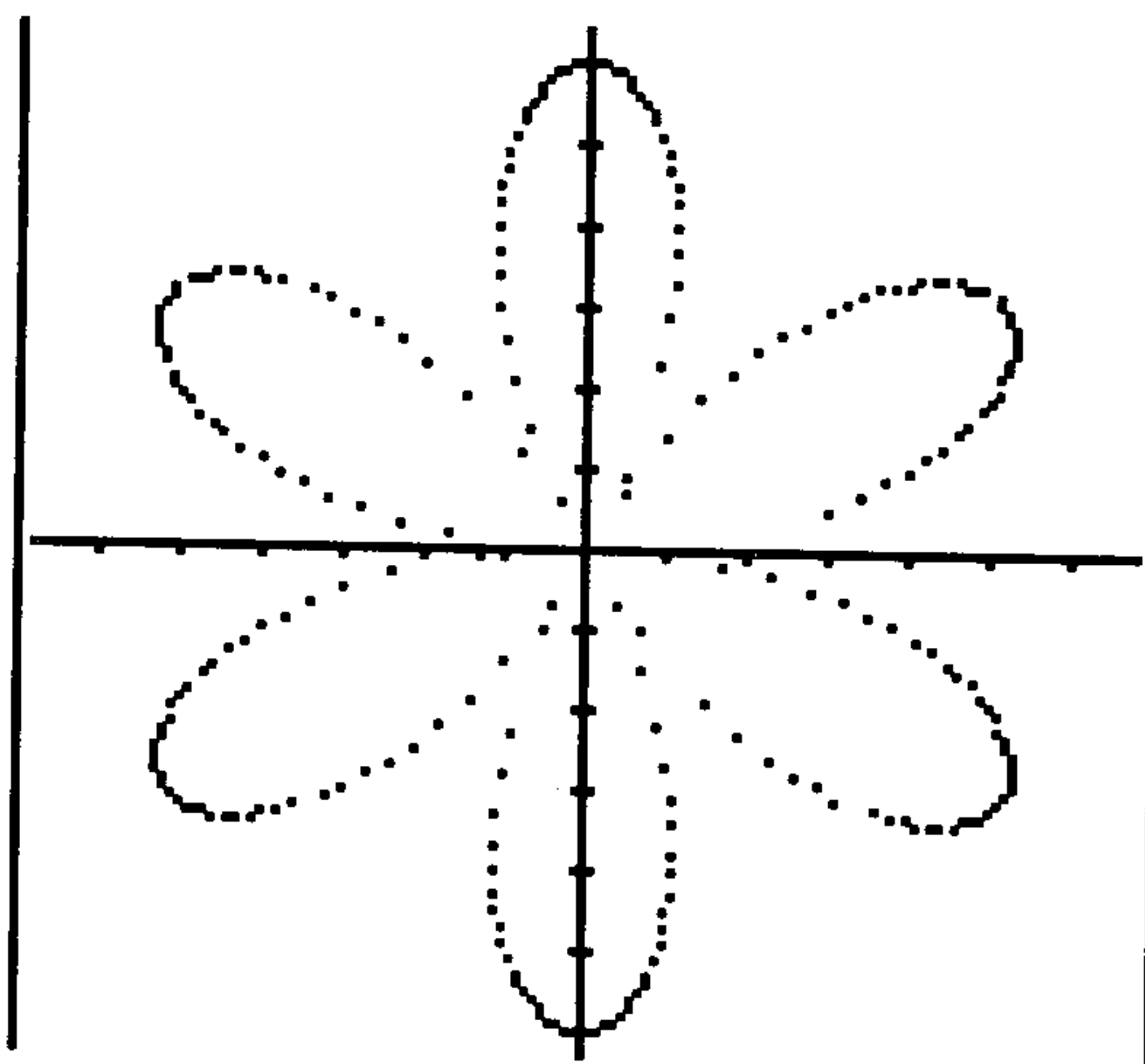
```
'SIN(X)' STEQ 'PPAR' PURGE RAD << DRAW >> X4PLOT
```

- This more elaborate example uses the polar plot program VTPOLAR (section 13.6.1):

```
'PPAR' PURGE (-6.8,-6.3) PMIN (6.8,6.4) PMAX DEG
```

```
<< << -> t 'ABS(6*SIN(3*t))' >> 0 360 VTPOLAR >> X4PLOT
```



[HP-28C version 1BB has a defect in the algorithm for placing “tick” marks on the plotted vertical axis when the origin of the axes is off-screen. When you use X4PLOT, the tick mark spacing is not uniform across the full picture. You can either ignore this cosmetic defect, or use AXES to move the axes out of the picture. This defect was corrected in version 1CC, and is not present in the HP-28S.]

As a final example of extended plotting, and as a slightly whimsical note on which to close this book, we present an HP-28S program MAND to plot the famous *Mandelbrot set*. This set, which is of great importance in the study of fractals, consists of all complex numbers C such that applying the operation

$$z \rightarrow z^2 + C$$

iteratively, starting at $z = 0$, generates a set of numbers z that stays bounded no matter how many times the operation is applied. The program plots all points C in the set as dark pixels. To save time, the program only considers points in a certain region of the complex plane near the origin. It somewhat arbitrarily chooses 15 as the number of iterations for each point; a smaller number would speed up the calculation but the picture would lose detail. A larger number makes the picture more accurate in some respects but takes longer to execute.

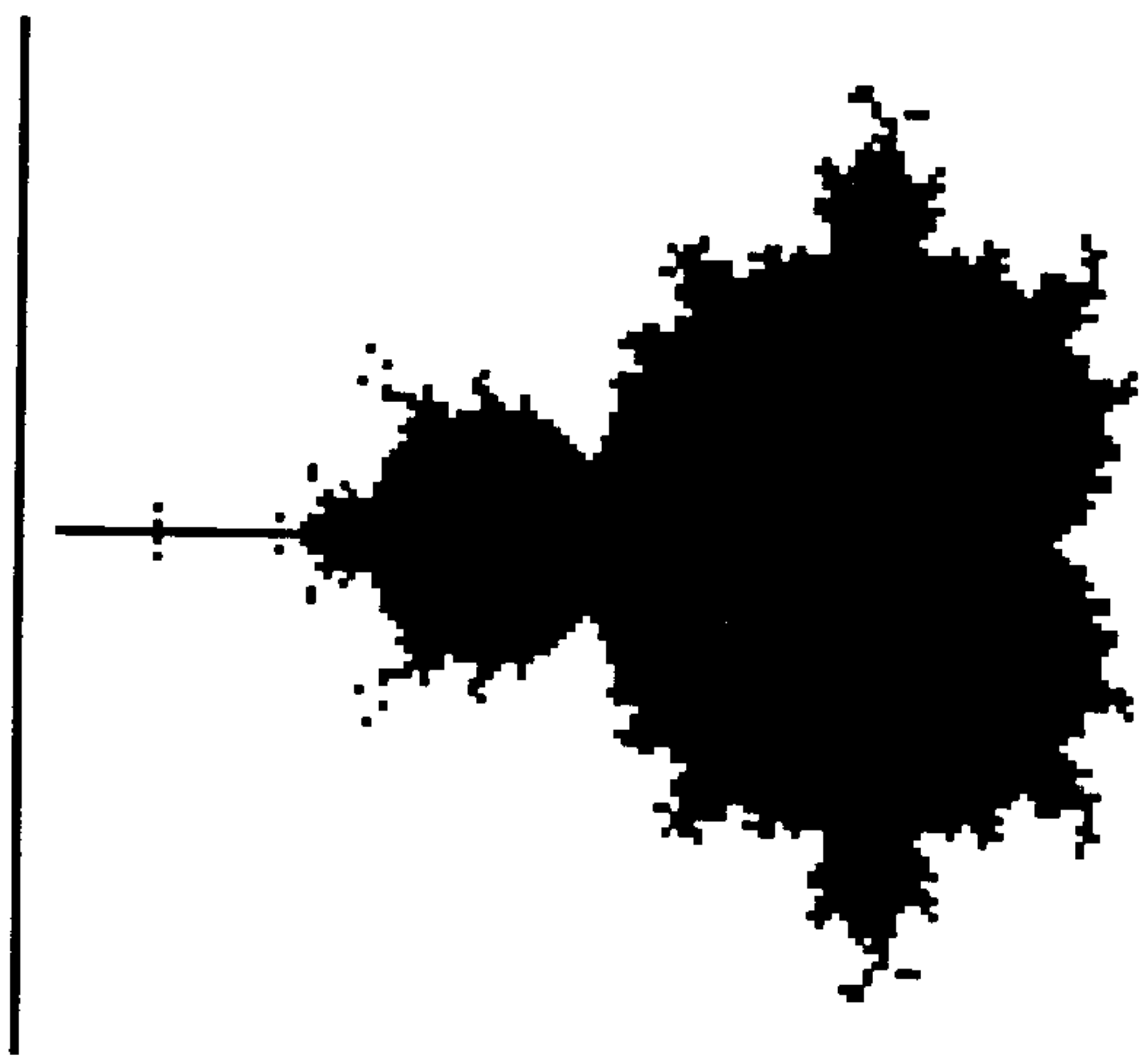
To provide enough detail to make the plot attractive, MAND computes the plot in four “frames,” which make a continuous picture when printed consecutively. Each frame is saved as an image string; the four strings are finally combined in a list and stored in the variable MPIC. The program MPRT makes a continuous printout from the list of image strings.

MAND also illustrates a use of KEY (section 11.3.2) that allows you to interrupt, then resume, program execution. This is desirable because MAND takes over 4 hours (more than 10,000 values of C) to make all four frames. If you press any key (except **ON**) while MAND is running, it saves the frames it has already created in a local variable, and halts. Then you can execute any calculator operations (except KILL or a system halt); when you’re finished, press **CONT**, and MAND will pick up where it left off.

MAND <i>Compute the Mandelbrot Set</i>	
<i>level 1</i>	
☐ { <i>strings</i> }	
<pre> << 1 -2 FOR n CLLCD n .59296 * DUP -2.04 OVER R→C PMIN .57443 + .48 OVER R→C PMAX 1.22 MIN SWAP -1.22 MAX FOR y '1.61*ABS(y)-2.1' →NUM 'MIN(-1.03*ABS(y)+1.19,.5)' →NUM FOR x x y R→C DUP 1 15 START SQ OVER + NEXT IF ABS 3 < THEN x y R→C PIXEL END DROP IF KEY THEN DROP LCD→ '-n+2' →NUM →LIST → pic << HALT pic LIST→ DROP →LCD >> END .01853 STEP -.01853 STEP LCD→ -1 STEP 4 →LIST 'MPIC' STO >> </pre>	Start the next frame. $P_{\min} = (-2.04, .59296n)$ $P_{\max} = (.48, .57443 + .59296n)$ y from $\text{MIN}(y_{\max}, 1.22)$ to $\text{MAX}(y_{\min}, -1.22)$. x start. x end. C Iterate 15 times. $z^2 + C$. If z is still small, plot the point. If a key is hit... save the pictures, and suspend. Restore and resume. Next x. Next y. Save the frame. Next frame. Combine the frames and store.

MPRT	<i>Mandelbrot Printout</i>	
	<i>level 1</i>	<i>level 1</i>
	<i>{ strings }</i>	
<pre><< LIST→ 1 FOR x x ROLL →LCD PRLCD -1 STEP >></pre>		Display the next frame. Print it.

Here's the picture:



The program to plot the Mandelbrot set was inspired by the lecture *The Beauty of Fractals: A Force for Teaching the Public*, given by Heinz-Otto Peitgen at the Atlanta Joint Mathematics Meetings of the American Mathematical Society and the Mathematical Association of America, January 8, 1988. The meetings, which were held to celebrate the centennial of the AMS, also featured the first public introduction of the HP-28S.

Program Index

ADDV	Concatenate Vectors	285
APLY1	Apply to 1 Symbolic Array	289
APLY2	Apply Program to 2 Symbolic Arrays	290
BARBERPOLE	Scroll the Picture	311
BOX	Box Volume	242
C	Combinations	123
CEQN	Characteristic Equation	296
CI	Cosine Integral	271
CINT	Circle in a Triangle	214
CN	Catalan Number	123
COUNT4	Count in 4 Ranges	195
CSEG	Area of a Circle Segment	124
CURR	Return Current Directory Name	83
Clusr	Clear Entire Directory	84
DELROW	Delete a Matrix Row	275
DFACT	Double Factorial	200
DIGITIZE	For HP-28C Version 1BB	306
DIGITIZE	For HP-28C Version 1CC	307
DIM	Symbolic Array Dimensions	288
DOPATH	Execute a Path	84
DRAWPIX	DRAW using PIXEL	325
FCT	Recursive Factorial	257
FIB	Fibonacci Series Generator	281
FIND	Find a Variable	85
GCD	Greatest Common Divisor	205
GDRAW	Guesses from DRAW	333
GSORT	General-purpose Sort	283
IGL	Integral	173
IGL2	Double Integral	175
INFSUM	Compute an Infinite Sum	254
INFSUMM	Compute an Infinite Sum (Monitor)	254
INVERT	Invert a Picture	311
ISOLCK	Isolate and Check	145
ISOLE	Isolate and Return an Equation	143
KEEP	Keep 1 Object	58
KEEPN	Keep N Objects	58
LABEL	Output Labeling Utility	242
LABELV	Output Labeling Utility, Variation	243
LINE	Draw a Line	330
MAKEΣ	Make Statistics Data	323
MAND	Compute the Mandelbrot Set	338
MINL	Minimum in a List (Bad version)	233
MINL	Minimum of a List (Good Version)	234
MINOR	Minor of a Determinant	275
MNORM	Modified Normal Distribution Generator	261
MPRT	Mandelbrot Printout	339

Program Index

MULTLABEL	Label Multiple Outputs	243
NORM	Normal Distribution Generator	261
NPPAR	New PPAR	313
N→S	Numeric to Symbolic	289
P	Permutations	123
POIS	Poisson Generator	260
POLAR	Polar Draw	326
PPER	Polygon Perimeter	124
PRIMES	Find Prime Numbers	263
PROMPT	Input Prompt Utility	241
PSE	HP-41-like PSE	228
PTINFSUM	Infinite Sum from Previous Term	270
Purge	Purge Any Variable	84
QU	Quadratic Root Finder	231, 238
QUADE	Quadratic Solve to Equation	237
RECENTER	Rescale and Recenter	321
RECIP	Compute 10 Reciprocals	63
RMINL	Recursive Minimum of a List	258
RTOP	Rectangular to Polar	194
RTOP2	Rectangular to Polar	197
SADD	Add Symbolic Arrays	291
SA→	Symbolic Array to Stack	288
→SA	Stack to Symbolic Array	288
SCOF	(Unsigned) Symbolic Cofactor	294
SCONE	Surface Area of a Cone	124
SDET	Symbolic Determinant of a Matrix	294
SEC	Secant	119
SI	Sine Integral	271
SIMEQ	Simultaneous Equations	266
SKETCH	Sketch Lines	331
SMINOR	Minor of a Symbolic Matrix	295
SMS	Scalar Multiply Symbolic Arrays	292
SMUL	Multiply Symbolic Arrays	292
SORT	Sort a List in Increasing Order	282
SSUB	Subtract Symbolic Arrays	291
STAR	Draw a Star	331
STRN	Transpose Symbolic Array	291
SUBCOL	Subtract Columns	286
SUBD	Find Subdirectories	83
SUM4	Sum $1/n^4$	252
SUMTERM	Compute an Infinite Sum from TERM	252
S→N	Symbolic to Numeric	289
TAYLRX0	Taylor's Polynomial at x_0	168
TESTPROMPT	Input Prompt Utility	251
TH	*H Substitute	320
TIMED	Timed Execution	255
TW	*W Substitute	320
UFIND	Universal Find a Variable	86
UP	Activate the Parent Directory	83
VCYL	Volume of a Cylinder	123
VSUM	Sum Vector Elements	201
VPOLAR	Variable θ Polar Draw	328

X4PLOT	4× Plotting	335
XMAX	Store $x_{\max}$	313
XMIN	Store $x_{\min}$	313
XPTINFSUM	Infinite Sum in x from Previous Term	270
YMAX	Store $\psi_{\max}$	314
YMIN	Store $\psi_{\min}$	314

Subject Index

- 119, 212
- ⇐ 8, 221
- ' 24, 33, 212
- " 28, 33
- # 27
- | 221
- (,) 26
- [,]
- {, } 29, 246
- <<, >> 24, 33, 179, 212, 246
- = 192
- () 160
- ^1 161
- *1 161
- /1 162
- 1/() 161
- +1 -1 162
- ←A 159
- () 160
- ABORT 227
- aborting programs 227
- abscissa 300
- accuracy factor 171
- activation 23
- add fractions 162
- addressing, indirect 62
- AF 162
- algebraic calculator 10, 16
 - entry mode 38, 40, 120, 246
 - evaluation 130
 - object 2, 16, 19, 23, 24, 29, 34, 64, 90, 129
 - syntax 11, 38, 41, 179, 246
- alpha key 38, 39
- analytic function 20, 144
- AND 309
- A→ 159
- arbitrary integer 147, 148, 149
- arbitrary sign 147, 148, 149
- argument 11
- array object 28, 272, 287
 - symbolic 287
- ARRY 51, 272
- ARRY→ 273
- associate left 159
 - right 159
- association 151, 153, 155, 159
- ATTN 227
- automatic mode changes 40
- automatic simplification 132, 136
- automating calculations 87
- axes 321
- Bad Guess(es) 99
- BASIC 2, 25, 91, 125, 211
- binary integer 27
- built-in program object 24
- busy annunciator 42, 44
- calculus 165
- Can't Edit CHR(0) 310
- CATALOG 172
- center of screen 315
- CENTR 315
- CF 191
- chain-rule 165
- change of sign 102
- change variable name 68
- changing variable contents 69
- characteristic equation 296
- CLEAR 47
- clear stack 47
- clearing 47
- CLΣ 82
- CLUSR 40, 81
- cofactor 293
- COLCT 142, 152, 153
- COLΣ 322
- column number 68
- column vector 274
- combinations 122
- combining RPN and algebraic entry 17
- command 20
- command line 34, 43, 45, 64, 92
- command stack 44
- common notation 12
- commutation 158
- compact format 8
- complex number 26
- complex results 26
- CON 69, 71, 78, 279
- conditional 189
- conservative approach 128
- constant, symbolic 137
- Constant? 100
- CONT 229
- continuous picture 334
- contravariant vector 274
- copying stack objects 49

- cosine integral 268
- covariant vector 274
- CRDIR 75
- current directory 75
 - equation 300
 - path 79
- cursor, algebraic entry 41
 - open box 40
 - solid box 41
- custom input menu 244
- custom output menu 245
- +D 159
- data object 25
- debugging 230
- default guess 105
- default variables 79
- defining expression 119
- defining procedure 212, 221
- definite loop 198, 201
- definition, object 22
- deleting suspended programs 229
- delimiter 21, 43, 179
 - ^ 24, 33, 212
 - " 28, 33
 - # 27
 - (,) 26
 - [,] 28,
 - {, } 29, 246
 - <<,, >> 24, 33, 179, 212, 246
- dependent variable 100
- ∂ 24, 165
- derivative 109, 165
- determinant 293
- DGTIZ 316
- digitizing 102, 105, 305
 - HP-28C 306
 - HP-28S 307
- DINV 161
- directory 74
 - current 75
 - home 75
 - parent 76
 - variable 80
- disappearing arguments 60
- DISP 35, 240
- display messages 240
- distribute 153
- distribute-prefix-operator 160
- distribution 155, 159
- DNEG 161
- DO 206
- DO loop 203
- double guess 102, 104, 107
- double integral 175
- double inverse 161
- double negate and distribute 160
- double negative 161
- double quotes 33
- D→ 159
- DRAW 100, 171, 297, 299, 331
- DROP 47
- DROP2 48
- DROPN 48
- DRWΣ 262, 297, 305, 306, 308, 321
- DUP 49
- DUP2 50
- DUPN 49, 50, 51
- e 137
- E() 162
- E^ 162
- EDIT 222, 232
- editing programs 222
- ELSE 193
- else-sequence 193
- empty stack 59
- END 193
- endless loops 31, 229
- ENTER 14, 20, 37, 180, 181, 246
- entry mode 38, 40
- EQ 69, 78, 96, 297, 300
- equation 129
- ERRM 207
- ERRN 207
- error handling 206
- error-sequence 207
- EVAL 24, 25, 31, 32, 35, 133
- evaluation 11, 19, 24, 29
 - numerical 133, 135, 137
 - procedure 29
 - symbolic 137
- exception 209
- exchange of arguments 48
- execute menu 245
- execution 19, 23, 24, 31, 76, 79
 - action 23, 25, 29
 - by address 35
 - data object 25
 - local name 24, 32, 215
 - global name 24, 31, 76, 215
 - numerical 132
 - postponed 45
 - preventing 24
 - symbolic 132
 - timing 255

Subject Index

- EXGET 152, 162, 163
- exit 201
- EXPAN 131, 142, 152, 155
- expansion 151, 155
- explicit ENTER 37
- explicit variable integration 136, 171, 172, 173
- exponent 25
- EXPR= 111, 115
- expression 11, 129
 - manipulation 150
 - mode 38
 - rearrangement 139
 - simplification 153
 - structure 131
- EXSUB 152, 162, 163
- extrema 108
- Extremum 100, 108
- factors 153
- false 189
- CUSTOM menu 244
- finding extrema 109
- first guess 101
- flag 188
 - 31 52, 207, 208
 - 34 148
 - 35 135
 - 36 134, 135
 - 57-59 209, 210
 - 61-64 210
 - exception action 209
 - infinite result 209
 - overflow 210
 - underflow 210
 - signal 210
 - system 188
 - true 189
 - false 189
 - user 188, 191
- floating-point 25
- FOR 198
- FORM 140, 142, 152, 157, 164
- formal variable 31, 67
- FOR...NEXT loop 212, 198
- FOR...STEP loop 199, 212
- FORTH 3, 46
- four-level stack 16, 57, 211
- FS? 191
- function 11, 19, 20, 118
 - analytic 20, 144
 - keys 39
 - plot 297, 299
- Gaussian distribution 261
- generalized plot 324
- generations, calculator 2
- GET 68, 69, 273, 276
- GET, index for 68
- GETI 68, 69, 273, 276
- global name 24,30, 31, 32, 211
 - variable 30, 66, 74, 211, 214
- guess, default 105
 - double 102, 104, 107
 - first 101
 - obtaining 104
 - single 102, 104, 105
 - triple 102, 103, 104, 107
- guillemets 33
- *H 318
- HALT 224, 226, 229, 232
- helvetica type 7
- hiding variables from solver 111
- home directory 75
- horizontal coordinate χ 299, 300, 322
- hot-key print 334
- HP-11C 3, 62
- HP-15C 3, 62, 94, 95, 97, 174
- HP-17B 9, 114
- HP 18C 9, 96, 114
- HP-19B 9, 114
- HP-27S 9, 114
- HP-28S memory 94
 - user-defined function 121
- HP-34C 95
- HP-35 1
- HP-41 2, 9, 125
 - Advantage Pac 95, 98, 174
 - array 272
 - CLST 47, 55, 59
 - CLX 47, 59, 64, 65
 - END 186
 - ENTER† 36, 37, 42, 44, 49, 64, 65, 179, 181
 - errors 208
 - flag 188, 189
 - flag 24 210
 - flag 25 208
 - FS? 91
 - general 9, 125
 - GTO 185, 186
 - halting 224
 - language 3
 - LASTX 51, 52
 - memory 93, 94
 - prefix commands 61
 - PRGM 179
 - program 29, 87, 91, 181, 182, 184, 185, 186, 190, 202, 203

- PROMPT 240
- PSE 227
- RCL X 49
- R+ 49
- RDN 48, 65
- register 66, 72, 278
- REGMOVE 73
- REGSWAP 73
- return stack 185
- R+ 49
- RUN 35, 225
- $\Sigma +$ 64
- stack 57, 59, 66, 185, 211
- stack-lift 63
- STOP 225
- storage arithmetic 69
- test command 188, 189, 192
- XEQ 32
- X<>Y 48, 55
- HP-65 2
- HP-71B 95
- HP-80 95
- i 137
- identity 157
- identity operations 161
- IDN 69, 71
- IF structure 193
- IFERR structure 206, 207, 230
- IFT 196, 197
- IFTE 196, 197
- IM 26
- image 310
- immediate entry mode 38, 40, 246
- immediate-execute key 38
- implicit ENTER 37, 44, 246
- implicit variable integration 136, 173
- indefinite loop 201, 198, 203
- INDEP 101, 304
- independent variable 82, 100, 101, 138, 300, 303, 322
- index, GET 68
 - loop 199, 212
 - object 163
 - subexpression 163
- indexed variables 286
- indirect addressing 62
- infinite result action flag 209
- infinite sums 267
- infix notation 12
- in-place operations 71
- input 244
 - and output 240
 - list 279
 - menu 244, 247
 - prompting 240
- $\int$ 24, 165
- integral 136, 165, 169, 171
 - double 175
- intermediate result list 282
- intermixing binary and real 27
- intermixing real and complex 27
- internal accuracy 26
- Invalid PPAR 80
- Invalid Σ DAT 81
- Invalid Σ PAR 81
- inverse of power or inverse of inverse-product 161
- ISOL 112, 115, 120, 139, 140, 142, 143, 147, 148
- italics 7
- KEY 227, 337
- key buffer 226
- key types 38
- keyboard operation 20
- keys, cursor 7
 - format 7
 - menu 7
 - shifted 7
 - that do ENTER 39
 - that do not ENTER 39
- KILL 229, 232
- known variable 138
- L() 162
- LAST 51, 207, 223
- L* 162
- LCD 297
- LCD→ 308, 310, 334
- LCD 297, 308, 310, 334
- LEFT= 111
- level, subexpression 131
- like terms 153
- line drawing 329
- liquid crystal display 297
- LISP 3
- list arguments, HP-28S 280
 - object 25, 41, 43, 58, 62, 66, 68, 72 259, 272, 275
 - output 281
- LIST→ 276
- LIST 276
- local name execution 24, 32, 215
 - name object 31, 119, 121, 211, 248
 - name resolution 216
- local variable 31, 32, 56, 66, 73, 89, 118, 119, 211, 214, 226
- local variable structure 212
- locked up keyboard 32
- logical operator 189, 308
- loop 198

Subject Index

- loop index 199, 212
- loop-sequence 203, 204, 205
- LR 324
- Łukasiweicz, Jan 12
- ←M 160
- MacLaurin's formula 157, 167
- Mandelbrot set 337
- mantissa 25
- mathematical approximation 105
- mathematical function 120
 - function plotting 297
 - variable 67
- matrix 272
- maxima 108
- MAXR 137
- memory limitations, HP-28C 93
- Memory Lost 229
- memory reset 229
- MENU 245, 247, 250
- menu, output 245
- merging 151, 160
- minima 108
- minor 293
- MINR 137
- missing points 302
- mode, algebraic entry 38, 40, 120, 246
 - alpha entry 38, 41, 246
 - changes 40, 38
 - entry 40
 - expression 38
 - immediate entry 38, 40, 246
 - numerical evaluation 132
 - principal value 148
 - program 38, 91, 180
 - symbolic execution 132
- mode-dependent key 38
- M→ 160
- multiple roots 146
- name 23, 30, 62, 200
 - execution 76
 - global 30, 31, 32
 - local 121, 31, 119, 121, 211, 248
 - quoted 33
 - resolution 75, 76, 78, 216
 - variable 73
- naming object 67
- nested IF structures 195
- non-analytic function 20, 144
- Non-Empty Directory 81
- non-radix 43
- normal distribution 261
- normal-sequence 206
- No Room to ENTER 223
- NOT 205, 309
- notation 7
 - common 12
 - floating-point 25
 - infix 12
 - Polish 12
 - prefix 12
 - reverse Polish 10
- NUM 24, 133, 135
- numerical definite integral 171
- numerical evaluation 132, 133, 135, 137
- OBGET 162, 164
- object 19
 - algebraic 2, 16, 19, 23, 24, 29, 34, 64, 90, 129
 - array 28, 272, 287
 - binary integer 27
 - classes 25
 - complex number 26
 - data 25
 - list 25, 41, 43, 58, 62, 66, 68, 72 259, 272, 275
 - matrix 272
 - name 23, 30
 - naming 67
 - procedure 29, 90
 - program 23, 29, 90
 - real number 25
 - string 28, 240
 - symbolic 34
 - type 21
 - value 21
 - vector 272, 274
- object string 43
- OBSUB 162, 164
- ON key 224, 227
- operation 19
- operation, prefix 160
- operator, prefix 160
- optimization, program 235
- ORDER 74, 81, 82
- ordinate 300
- output 244
 - labeling 240, 242
 - list 281
 - menu 245
- OVER 50
- Overflow 210
- Owner's Manual 4
- P_{max} 311
- P_{min} 311
- parameterized plot 325
- parent directory 76

- path 79
 - current 79
- pencil-and-paper 13
- permutations 122
- PICK 50, 51
- π 124, 134, 135, 136
- picture 297
- PIXEL 297, 324, 331
- pixel numbers 314
- plot equation 301
 - function 297, 299
 - generalized 324
 - parameterized 325
 - parameters 298
 - polar coordinate 326
 - ranges 311, 315
 - rescaling 318
 - scale 323
 - scatter 321
 - printed 334
- Poisson distribution 259
- polar coordinate plot 326
- Polish notation 12
- POS 277
- postfix syntax 61
- postponed execution 45
- PPAR 69, 78, 298, 322
- precedence 12, 159
- prefix notation 12
 - operation 160
 - operator 160
 - syntax 61
- preventing execution 24
- prime numbers 262
- principal value mode 148
- printed plot 334
- PRLCD 334
- problem solving 87, 126, 138
- procedure 29, 90, 91
 - evaluation 29
 - as argument 250
- program 23, 29, 88, 90
 - body 179, 180
 - comment 28
 - documentation 220
 - editing 222
 - legibility 118
 - listing format 220
 - mode 38, 91, 180
 - optimization 235
 - quoted 34
 - quotes 33
 - structure 92, 178, 180, 187, 239
 - structure word 178, 187
 - suspended 224
 - unquoted 34
- programmable program key 180
- programming 87, 92, 178
 - recursive 257
 - structured 92, 175, 181, 182, 185, 224
- PRVAR 231
- PURGE 73, 81
- PUT 69, 72, 78, 273, 276
- PUTI 69, 72, 273, 276
- QUAD 24, 139, 142, 143, 146, 147, 148
- quadratic equations 146
- qualifying message 97, 99
- quotation mark delimiters 33
- quoted name 33
 - program 33, 34
 - variable 67
- quotes, double 33
 - single 33
 - program 33, 179
- RAND 259
- random numbers 259
- RCEQ 300
- RCL 32, 191
- RCLF 208
- RDM 69, 71
- RE 26
- real number 25
- recursive programming 257
- reference manual 4
- register 66
- reordering terms 150
- REPEAT 205
- rescaling a plot 318
- resolution 322
 - local name 216
 - name 75, 76, 78, 216
- resolved 79
- result 11
- reverse Polish notation 10
- right hand symbol $\rightarrow$ 8, 221
- ROLL 49
- ROLLD 49
- ROOT 24, 78, 91
- root 95
- ROT 49
- row number 68
- row order 272
- row vector 274
- RPL 3, 33

Subject Index

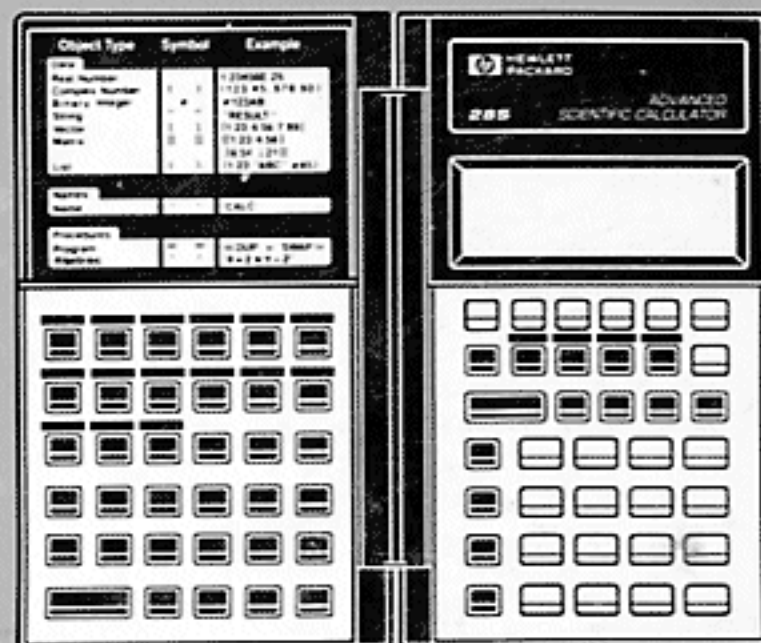
- RPN 2, 10
 - calculator principle 10, 131
 - command 20
- RT= 111
- $\Sigma +$ 64
- Σ DAT 69, 260, 261, 322
- Σ PAR 69, 322
- SAME 192
- scatter plot 321
- SCL 78
- SCL Σ 262, 323
- SCONJ 69, 71
- screen, center
- separator 43
- sequence 8, 178, 198, 200
- SF 191
- SHOW 113, 115
- signal flag 210
- Sign Reversal 99, 102
- simplification, automatic 132, 136
- simplifying polynomials 157
- simultaneous equations 264
- sine integral 268
- single guess 102, 104, 105
- single quotes 33
- single-step 229, 232
- SINV 69, 71
- SIZE 273, 276
- SNEG 69, 71
- solid box cursor 41
- Solver 78, 88, 91, 95, 96, 141, 142, 147, 149, 171
 - constant 96
 - menu 82, 88, 96, 97, 100, 111, 247
- SST 229
- stack 8, 14, 16, 46, 66
 - clear 47
 - diagram 221
 - empty 59
 - four-level 211
 - level 22, 46
 - objects, copying 49
 - roll 48
 - unlimited 57
- stack-lift 37, 47, 49, 63
- stack-lift disable 42, 63
- start 198, 199, 200
- starting and stopping 224
- START...NEXT loop 200
- START...STEP loop 200
- statistics matrix 322
- step 200
- step-wise substitution 31
- STEQ 78, 96, 300
- STO 67
- STO/ 69, 70
- STO- 76, 78
- STO* 69, 70
- STOF 208
- stop 198, 199, 200
- storage arithmetic 69, 70
- storage register 32
- storing pictures 307
- string object 28, 240
- structure, expression 131
 - program 92, 178, 180, 187, 239
- structured programming 92, 175, 181, 182, 185, 224
- SUB 277
- subdirectory 76, 77, 81
- subexpression 130, 131, 158
 - level 131
- subroutine 182
- substitution 35, 141, 162
 - step-wise 31
- summand 154
- suspended program 224
- suspended program annunciator 224
- SWAP 48, 49
- symbolic array 287
 - constants 137
 - evaluation 137
 - evaluation mode 111, 132
 - mathematics 88, 125
 - object 34
 - solution 112, 142
- syntax 37
 - algebraic 11
 - prefix 61
 - postfix 61
- Syntax Error 179, 187, 222, 245
- SYSEVAL 35
- system clock 255
- system flag 188
- System Object 21
- system halt 32, 82, 229, 249
- Taylor's polynomial 167, 168
- TAYLR 24, 146, 157, 165, 167, 168, 296
- term 153
- test 188, 189
- test command 188, 192
- test-sequence 193, 203, 204, 205
- THEN 193
- then-sequence 193
- tick 33
- tick marks 321

time value of money 95
 timing execution 255
 triple guess 102, 103, 104, 107
 TRN 69, 71
true 189
 TVM 95, 96
 Unable to Isolate 120, 145
 Undefined Name 72, 133, 138
 Undefined Local Name 215, 249
 Underflow 210
 UNDO 43, 223, 224, 230, 232
 unknown variable 100, 101
 unlimited stack 57
 unquoted program 34
 UNTIL 204
 USE 172
 user flag 188, 191
 USER memory 66, 74
 USER menu 74, 149, 211, 244, 245
 menu key 74
 user-defined function 88, 89, 117, 118
 value 30, 67
 variable 11, 30, 66, 239
 change contents 69
 change name 68
 default 79
 dependent 100
 directory 80
 formal 31, 67
 global 30, 66, 74, 211
 hiding from solver 111
 independent 100, 101, 138, 300, 303, 322
 indexed 286
 known 138
 local 31, 32, 56, 66, 73, 89, 118, 119, 211, 226
 mathematical 67
 name 73
 quoted 67
 unknown 100, 101
 VARS 81
 vector 272
 contravariant 274
 covariant 274
 row 274
 version 35
 vertical coordinate ψ 299, 300, 322
 VISIT 222, 232
 *W 318
 WAIT 226
 WHILE loop 203, 205, 206
 $X_{\max}$ 299
 $X_{\min}$ 299
 X_{axes} 299
 XOR 309
 $\psi_{\max}$ 299
 $\psi_{\min}$ 299
 ψ_{axes} 299
 Zero 99, 102, 107
 zoom in on plot 316



HP-28 Insights

The HP-28 is a revolutionary calculator. With its symbolic operations, multiple object types, and powerful, flexible programming language, the HP-28 offers the most advanced computational capabilities ever available in a handheld calculator.



In *HP-28 Insights*, Dr. William Wickes offers his own special insights into the operation and application of the HP-28C and HP-28S. He presents in-depth discussions of a wide spectrum of topics, starting with the most fundamental principles of calculator operation. Most of the book is devoted to problem solving methodology, including the operation of the HP Solver, user-defined functions, symbolic mathematics, and general programming. All important ideas are illustrated with specific examples. Included are over 100 practical example programs, that demonstrate programming concepts such as local variables, program structures, recursion, and advanced plotting techniques. As the author of the well-known *Synthetic Programming on the HP-41C*, Dr. Wickes gives special attention to the comparison of HP-41 concepts and their HP-28 counterparts.

Chapter Headings:

1.	Introduction	1
2.	Understanding RPN	10
3.	Objects and Execution	19
4.	The HP-28 Stack	46
5.	Variables	66
6.	Problem Solving	87
7.	The Solver	95
8.	User-Defined Functions	117
9.	Symbolic Math	125
10.	Programming	178
11.	Program Development	220
12.	Arrays and Lists	272
13.	Plotting	297

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.