

An Easy Course In

Using The HP-28S



A GRAPEVINE PUBLICATION

By John W. Loux and Chris Coffin

Cover Illustration by Robert L. Bloch

An Easy Course In Using The HP-28S

By John W. Loux

and Chris Coffin

Cover Illustration by Robert L. Bloch

Grapevine Publications, Inc.

P.O. Box 118

Corvallis, OR 97339-0118 U.S.A.

Acknowledgement

Thanks and appreciation go once again to the Hewlett-Packard Company for continuing to produce such top-quality products and documentation.










© 1988, Grapevine Publications, Inc. All rights reserved. No portion of this book or its contents, nor any portion of the programs contained herein, may be reproduced in any form, printed or mechanical, without written permission from Grapevine Publications, Inc.

Printed in The United States of America
First Printing – February, 1988

ISBN 0-931011-18-3

DISCLAIMER: Neither the authors nor Grapevine Publications, Inc. make any express or implied warranty with regard to the keystroke procedures and program material herein offered, nor to their merchantability nor fitness for any particular purpose. These keystroke procedures and program material are made available solely on an "as is" basis, and the entire risk as to their quality and performance is with the user. Should the keystroke procedures or program material prove defective, the user (and not Grapevine Publications, Inc., nor the authors, nor any other party) shall bear the entire cost of all necessary correction and all incidental or consequential damages in connection with, or arising out of, the furnishing, use, or performance of these keystroke procedures or program material.

CONTENTS

INTRODUCING...The Introduction	8
What Is This Tool?	10
What Is This Book?	11
What's In This Book – and What's Not?	13
 How to Picture Your HP-28S	 14
The Display	17
The Keyboards	18
Posting Memos: Interactions Between the Keyboards and the Display	19
The Menu Keys: Your Command Card File	27
The  ("Shift") Key	27
Immediate Execution ("Do-It-Now") Keys	33
Messages From the System – Memos From Your Staff	36
Status Messages: The Annunciator Area	38
A Tricorder Reading	39
Quickie Quiz	40
Quickie Answers	41
 Making Your HP-28S Work For You: The Command Line	 42
Typing Characters Into the Command Line	43
Changing a Character in the Command Line	44
Adding and Removing Characters	48
 INS ,  DEL and ATTN	51
 NEWLINE ,  and 	54
The LC Key	57
The  Key	58
Item Delimiters and ENTER	62
 EDIT and  COMMAND	64
Command Line Quiz	67
Command Line Answers	68
Notes	69

Real Numbers, the Stack, and Postfix Notation **70**

Real Numbers – and the Real World	71
Representing Real Numbers On the HP-28S	73
Scientific Notation on Your HP-28S	73
12-Digit Accuracy: Rounding Error	74
Magnitude: How Big (or Small) Can You Get?	76
Posting Real Numbers: [CHS] , [EEX] and Display Modes	79
Display Formats	81
The Stack and Postfix Notation	84
Real Number Commands: 0-, 1-, and 2- Number Operations	85
Arithmetic Practice	91
Arithmetic Practice Solutions	93
Stack Operations	94
[ENTER] 's Second Job	95
The [SWAP] Command	96
How to [CLEAR] the Stack	96
Strenuous But Practical Stack Practice Problems	107
S.B.P.S.P.P. Solutions	108

The "Stuff" Upon Which the HP-28S Works **112**

An Equal Opportunity Calculator	113
The HP-28S's Philosophy of Information	114
Real Numbers	115
Complex Numbers	118
Simple Questions About Complex Numbers	124
Simple Answers to Simple Questions About Complex Numbers	125
Vectors	128
A Visit With Vectors	138
Results of A Visit With Vectors	140
Arrays	143
Array Aptitude Test	147
A.A.T. Results	148
Characters	151
Character Strings	152
Character String Query	159
C.S.Q. Answers	160

Names	161
Name Games	167
Name Game Winners	168
Bits	171
Binary Integers	172
Binary Integer Test	175
B.I.T. Answers	176
A Pause For the Cause	177
Lists	178
List Lessons	183
List Lessons Learned	184
Procedures: (a) Postfix Programs	185
Program Problems	189
Program Problem Solutions	190
Procedures: (b) Algebraic Expressions	193
Algebraic Aptitude Test	201
A.A.T. Scores	202
Procedures: (c) User-Defined Functions	205
User-Defined Function Fun	209
U.D.F.F. Consequences	210
Directories	212
Directory Discussion	219
Directory Assistance	220
Menus	222

Problem Solving **226**

Introduction **227**

Postfix Programming **228**

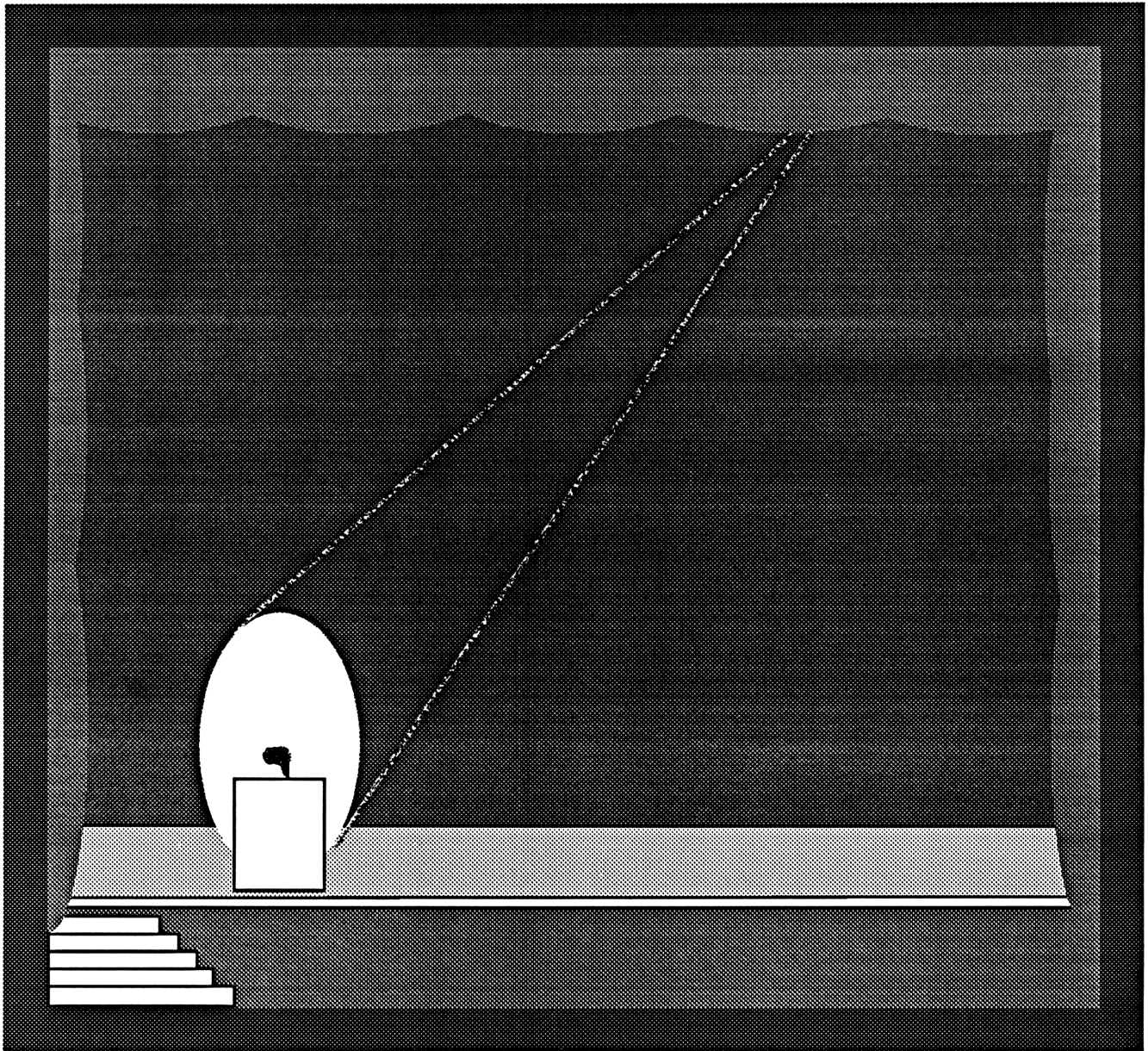
So	229
Answers	230
Local Names	232
Local Name Lesson	235
Local Name Moreon	236

Some Comments Before You Go On	237
Making Decisions	238
Conditional Curves	241
Conditional Conclusions	242
Variants of IF...THEN...ELSE...END	246
Iffy Situations	249
Iffy Answers	250
Doing Things More Than Once	252
Loop Practice	257
Loopy Answers	258

Solving Problems Using Algebra **260**

Why Use an Algebraic Object?	260
Peculiarities of Representation	262
Some Practice	263
Some Answers	264
How Do You Use an Algebraic Object?	266
Basic Algebra Problems	269
B.A.P. Answers	270
Name Substitution	271
More Substitutions	274
FORM	276
Less Basic Algebra Problems	281
Less Basic Algebra Solutions	282
Some Short-Cuts	286
The Equation Solver	290
Still More Algebra Problems	295
S.M.A.P. Answers	296
Calculus	299
Symbolic Constant and Symbolic Function Evaluation	301
A Final Visit with Algebra/Calculus	303
Final Answers	304

Plotting	308
Scaling	310
Digitizing	311
 Keyboard Error Recovery	 312
■ COMMAND : The Command Stack	312
■ UNDO : UNDOing a Command	313
■ LAST : Recalling the Stack as It Was Before the LAST Command	314
Enabling and Disabling Error Recovery	315
 Editorial	 316



INTRODUCING...The Introduction

Well now...so you have an HP-28S, eh?

Now the questions are, "What do you want to do with it?" and (mostly), "*How?*"

These are the right questions to ask, of course. And you may have heard that once you've decided *what* you want to do, the *how* should be intuitively obvious – even to the most casual observer.

That's just not true. There's nothing wrong with your intuition or your personal casualness index. It's simply that this machine is not all that simple. Even if you're experienced with other HP calculators, this one is so *radically* different that you may find yourself "starting over" in many respects. You may still recognize some familiar HP stack-oriented arithmetic, but that's about where the similarities end. For the most part, the HP-28S will be a "brave new world."

Of course, as with all calculators, the HP-28S is only a tool, a problem-solving tool. So is a hammer. And though it's fairly obvious (even to extremely casual hammer-observers) that a hammer is good for pounding, it takes more than casual observation to use it effectively to build a house. It takes time and practice.

So it is with the HP-28S. Being just a bit more complex than a hammer, it does require more effort on your part to use it effectively. But once you make that effort, you'll be amazed at the "houses" you can build with it.

That's the purpose of this book – to help you learn to use a tool. Just be sure to remember that it *is* just a tool, not a magic box that gives you the answer to your every question. It can't check to see if you've given it the right numbers to "crunch," nor can it catch you when you're attacking a problem altogether wrongly. It's an inanimate mechanical aid – not a replacement for your understanding of the problem. You *must* understand both your tool *and* your problem in order to use the one on the other.

What Is This Tool?

Before you begin to use the HP-28S as a problem-solving tool, you'd better have at least some idea that it's actually the *right* tool for the job.

This calculator is not omnipotent. It does some things very well and other things not so well. It's flexible, but for some tasks, it may cost you more effort to bend it to your will than it's worth. In those cases, you would come out ahead by choosing another, more appropriate tool.

So what is the HP-28S really "good at?"

Mostly, it's a math engine. It provides you with an extensive set of mathematical operations. And it uses these operations on a fairly comprehensive set of mathematical "things": real numbers, vectors, arrays, complex numbers, and algebraic expressions, to name a few. So if a lot of your problems involve this kind of math, then the HP-28S is probably as good a "hammer" as any you will find.

But it's *not* a generalized computer. For example, it doesn't have the means to save your calculations anywhere else (i.e., on magnetic tape or disc). It doesn't have a full typewriter keyboard. You wouldn't want to try to type your doctoral thesis on it.

Of course, with some effort you *could* coerce it into doing many different things, but don't be surprised (or upset) if the results are not the best. After all, you *can* drive a screw with a hammer, but if you do and then things don't turn out very well, don't go blaming the hammer for not being a screw driver. It just wasn't built for that.

What Is This Book?

There are (at least) four ways to approach your learning about the HP-28S:

1. Be Joe Computer-Whiz, for whom it is either intuitively obvious or the essence of joy to play with such a machine until it yields all of its secrets;
2. Apply brute force – not knowing where to start, but pressing buttons anyway, hoping that something meaningful will result;
3. Resort to tears and despair (usually as a result of method 2);
4. Ask for some help and explanation (usually as a result of method 3).

If you're now using method 4, then this book is meant for you. And there's absolutely nothing wrong with that. It carries no shame or stigma to say "I don't understand this yet." You just haven't yet seen it explained in a way that "clicked" for you. This book is merely a different way to explain the HP-28S.

Admittedly, it doesn't appeal to everybody. You may find the pace too slow or the explanations too meticulous. But chances are there is *something* presented here that could "shed more light on your HP-28S" for you. So relax and browse if nothing else. A lot of people discover the same thing – that such a slow, classroom-style approach seems to work better than the "brute force" method.

Above all, please don't feel "talked down to" by this Easy Course.

Just because the printing is large and spread out and the wording is simple and "folksy," you shouldn't take this as any commentary on your technical expertise or vocabulary. The subjects here are *not* trivial, nor is your intellect being trivialized by seeing them presented in this fashion. The only reason for all this is to communicate to you the skills and knowledge you need to make the best use of a very sophisticated tool. And this method of communication often helps.

And how does this method go? Here are a few things to know about the book and its classroom approach:

1. Every so often, you'll come across a little set of quiz problems. These are just some exercises to help you make sure you "have things under control" before you move on. If you have any major difficulties with the questions, you'll find the answers immediately following, along with page numbers so you can go back and review if you wish.
2. At certain points along the way, you'll be given the option to skip ahead if you feel that you already know the material being discussed. If you do skip, it will usually be to the quiz at the end of the section, so you can be sure you're really as knowledgeable as you thought.
3. There's no race, no time limit, no clock, no exam proctor, and no #2-pencil-grading-machine breathing down your neck. This is *your* Course, to be taken at *your* speed. Who cares if you go back and reread something a couple of times? The idea is to learn about your calculator, not to break a speed record for doing so.

What's In This Book – and What's Not?

This book is *not* a re-packaging of the manuals that came with your HP-28S. Many keys, features and functions on your calculator just don't appear anywhere in this book – and this is no accident. Why should anyone try to document every last aspect of the machine? That's what the HP manuals do so well; why try to improve on them?

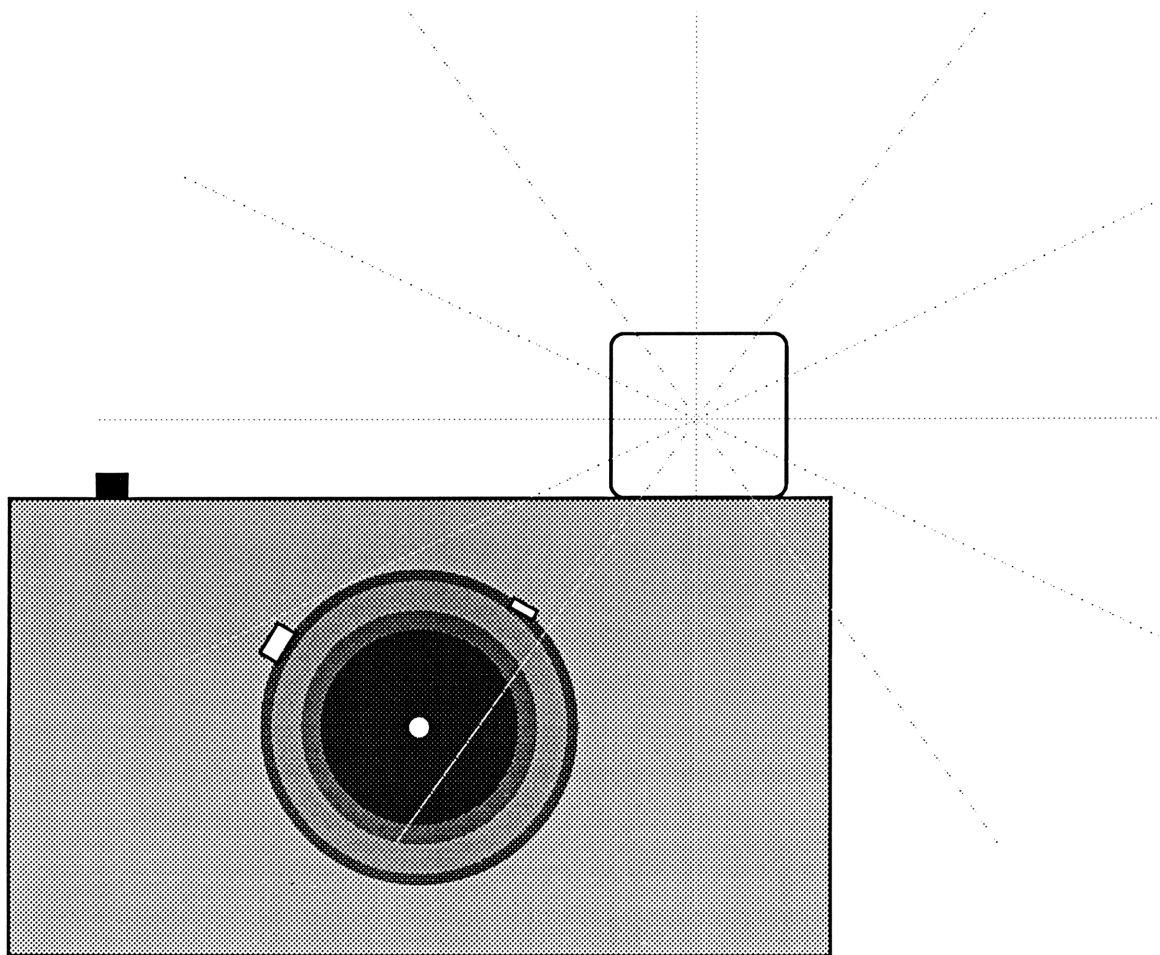
Instead, what you're going to see here are the fundamental concepts and principles of the HP-28S. Of course, you'll need to learn the mechanics of the keyboard and the display first, but the real idea here is to orient yourself and move around in the generalized data-manipulation world of the HP-28S. By the time you finish this Easy Course, you should feel quite comfortable in using and combining the different available data "objects" to help solve your math problems.

But all the while, keep in mind that this is only an *introduction* to the HP-28S.

Why just an introduction? Two reasons:

First of all, this book is meant to help you get enough "calculator savvy" to begin building a more exhaustive understanding of this tool on your own terms – and in your own way. Hopefully, it's enough to get you on your way down that road, without unduly burdening you with a load of preferences and biases as to how you ought to actually apply this tool to your everyday tasks. At some point *you* must take over and decide for yourself exactly when and how to use it.

Secondly, there's only so much room in one book!



How To Picture Your HP-28S

Before you can really do anything in the world of the HP-28S, you'll need to know how to move around in it. Learning these mechanical skills isn't always a whole lot of thrills – but it is necessary.

So you need to realize right up front that these next three chapters are really just a set of lessons in controlling and communicating with your calculator. Boring as that may sound, don't underestimate the importance of these skills.

OK?

Now then: A picture is still worth a thousand words, so it makes some sense to have a picture of the HP-28S's world to help you understand it.

Of course, the picture you'll need isn't exactly an 8-by-10 color glossy of the calculator (with circles and arrows and writing on the back).

First of all, who needs such a photo when you have the real thing? And anyway, you may already know from personal experience that you can stare at the physical HP-28S until drops of blood bead up on your forehead and you still won't come any closer to understanding how the machine works.

So, because that's what you really want to know – *how* the thing works and not what it looks like – you'll need instead a picture of something that doesn't physically exist, the *logic* of the calculator.

Unfortunately, the camera that can take that picture hasn't been invented. But you might try another method instead: mind games....

The name of this particular mind game is "Easy-Course-Warmer-Upper-HP-28S-Mental-Picture-Of-Its-Logic." (Sort of catchy, don't you think?) It's for all ages and requires only one player and one mind (and since you seem to "have a mind" to use your HP-28S, you'll do quite nicely, thank you).

As you might suspect, the object of the game is to paint for yourself a mental picture you can use as a map to explore the unfamiliar world of the HP-28S. But there are no rules; you just make it up as you go along.

So, if you're ready, flex your mental muscle, and sure enough, a picture begins to form in your mind's eye....

The first thing you see is, not surprisingly, the HP-28S (see it there?). Of course, as you've already observed, this doesn't get you very far (especially if, in your mental picture, you've forgotten to open the calculator).

So you concentrate even harder, focusing in on its two most obvious features, the display and the keyboards, and slowly but surely, a better picture forms....

You're the newly-elected president of a very talented little company of mathematicians who make their livings by solving problems for others. By prior arrangement, these mathematicians have their offices inside the HP-28S.

Of course, as president, your job is to properly delegate and assign tasks, so that the overall results are those requested by your clients. You're the go-between, understanding and translating your clients' needs into terms that your staff can understand and act upon.

The Display

Think of the display of your HP-28S as a bulletin board, and picture it that way. It's how you communicate with your staff (the calculator).

Upon reflection, you'll find that this makes quite a bit of sense because the display is *interactive*. That is to say, it changes as you and the calculator do things to change it.

As with a real bulletin board, messages are posted in the display by *you* (for the machine) – problems to solve, numbers to store or "crunch", etc.

And messages are posted by the *machine* (for you) – status reports, information and graphs for your inspection and correction.

You'll see as you go along that your bulletin board is quite well organized with different messages from different departments posted in different areas on the board. And you'll also find that, very much like a real-world bulletin board, your display bulletin board can become cluttered. New messages can obscure or even "bump off" old messages.

Not to worry though. There are ways to tell your "staff" that whenever they need to post messages that would "bump off" other ones, they should save such bumped messages, just in case you want to look at them again. Your staff will obey this – and all your instructions – if you make them clear.

Actually, all things considered, you have a fairly well-organized, imaginary math-problem-solving business here.

The Keyboards

The next areas to notice are the keyboards (and you should probably continue to think of them as two distinct keyboards rather than as two halves of one keyboard, because there are some significant differences in how each is used).

Continue with your mental picture: If the display is your bulletin board, then you can envision the keyboards as your typewriter or dictation recorder. After all, as president you need some way of creating memos and messages for posting on the bulletin board, right? OK, draw it in your mind as a typewriter (got it?), and look at how this typewriter is arranged.

First, look at the left-hand keyboard. If you pay attention only to the white letters on the keys, the left-hand keyboard really does look like a typewriter with its keys rearranged. And indeed, these keys are used for typing words and phrases (ignore the other, less obvious things on the keyboard for now. You'll come back to them later as you need them).

Likewise, if you look at the right-hand keyboard and notice only the white keys with black lettering, you'll see what appears to be a simple, 4-function calculator. Again, this is how it ought to appear; that's exactly what those keys are for (and again, ignore for now the other, less self-explanatory keys).

So, as a new president, you're beginning to at least find your way around the office. Review your picture up to this point:

You have a bulletin board (the display) through which you communicate with your staff (the HP-28S system). You also have a simple, desktop calculator and a typewriter to use in writing memos for posting. Not bad.

Next thing to figure out: How do you actually post memos?

Posting Memos: Interactions Between the Keyboards and the Display

As you might expect, in order to get any work out of your staff, you need to tell them what to do – i.e. post a memo, after composing it on your typewriter.

Of course, right now is when you realize that your typewriter actually types directly onto the bulletin-board (quite a high-tech office, really).

Unlikely? Well, yes, it's true that things don't work exactly like this in the real world, but it doesn't stretch your imagination too much to picture it this way nevertheless.

Now then, it's time for everyone's favorite game (yep – even company presidents like to play):

"Press the Pretty Buttons and See What Happens."

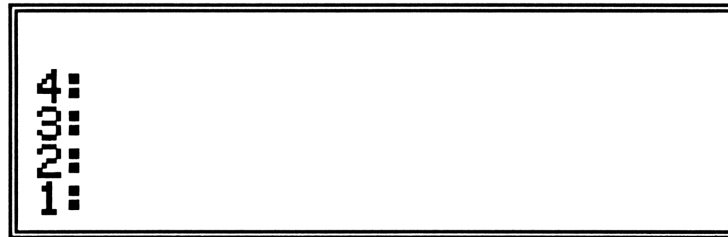
But before you do that, stop and think for a minute: Whatever memos are posted on that bulletin board now are from the previous administration (heaven forbid). Better clean up the bulletin board so that everyone will know exactly who to blame (you) for anything that appears hereafter.

Ready?

Scrub And Dust: Clean up your HP-28S bulletin board.

Here's How:* Press `#4001BFC400000000` `LC` `H` `ENTER` `S` `T` `O` `F` `ENTER` `'` `A` `B` `PURGE` `H` `O` `M` `E` `ENTER`. Next, press and hold down the `ON` key. While holding this down, press the `▲` key (the upper middle of the right-hand keyboard). Now release the `▲` key. Now release the `ON` key.

After doing all this, your display will look like this:



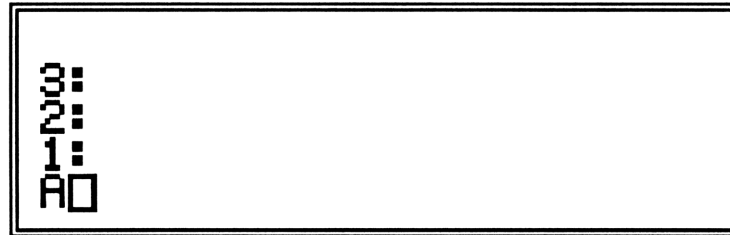
OK, now everyone in your company should be ready to receive instructions from the new chief.

And now you're ready to try your typewriter to see its effects on the bulletin board. Of course, you'll notice that the board (the display) isn't totally bare. Don't be too concerned about what those remaining numbers and colons mean. For now, just watch them move around as you begin to use your typewriter....

*This is, admittedly, a rather complicated procedure to start with here, but there's only one other way to ensure that you're starting "in step" with this book – clearing the machine's 32K-byte memory *entirely*. And since you may not want to re-key in the 31.5K-bytes of stuff you may already have stored...

(At this point, if you already know how to type and post message and command memos, how to use menus and immediate-execute keys, then you can probably skip ahead now to page 40. Otherwise, stick around.)

Go: Find the **[A]** key in the upper left-hand corner of the left-hand keyboard. Press it once, and then look at the display. You should see this:



"Whoa!" (you undoubtedly say), "that's quite impressive!" Not at all (shucks). Actually, here are the really important things to notice:

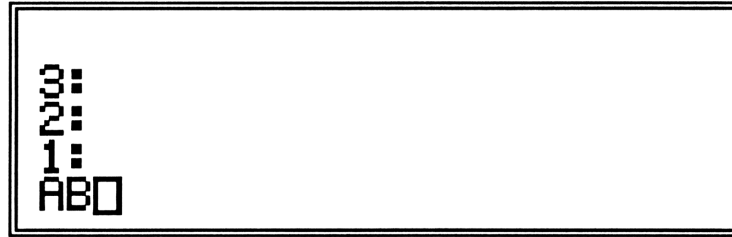
Firstly, notice that almost everything that was already in the display has been pushed up one line – to make room for the newcomer on the bottom line.

The space opened up at the bottom is lovingly known as the *command line*. In your mental picture, this is where the things you type on your typewriter are first put onto the bulletin board.

Secondly, there's a flashing, empty box immediately to the right of the **A**.

This box, called the *cursor*, shows where the next character will be placed if you type another one (notice that your typewriter does indeed produce *characters* – not just letters; it can type other things, such as numerals and special symbols. All of these things are collectively called characters).

Next: To prove to yourself what this cursor (the flashing box) is for, find **B** on the left-hand keyboard and press it.



The **B** is placed where the box was, and the box is moved one space to the right – to where the *next* character will be placed. And so on.

Important point: You will see the cursor *only* when typing in the command line.

By the way, if you press the wrong letter key while using the command line, use **⏮** to correct it. **⏮** is the same as the backspace key on a typewriter keyboard. That is, by pressing it, you move the cursor one space to the left *and* remove the character that was there.

If you use **⏮** to remove the last remaining character, the command line goes away. If you keep pressing **⏮** after that, nothing more will happen.

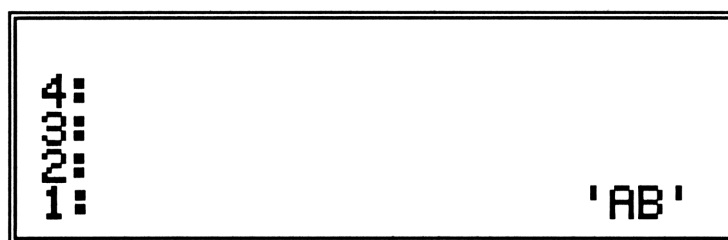
Play with it, if you wish (then restore your display to the way you see it above).

OK, since you're a new chief executive trying to learn the ropes around here, do a trial run: Pretend that what you've typed so far is actually something meaningful that you'd like to post on the bulletin board.

Give It A Whirl: Seeing that the cursor is still blinking merrily, you press **ENTER**.

The command line goes away and the message is posted.

You see:



A rectangular box representing a bulletin board. On the left side, there are four vertical slots numbered 4, 3, 2, and 1 from top to bottom. The bottom slot (1) contains the text 'AB'.

What can you learn from this?

1. The message was posted at the bottom of the bulletin board. It was put in the first spot, indicated by the **1**.

That's what those numbers on the left-hand side of the bulletin board are – level markers. They just tell you the age of each message on the board, the youngest (most recent) ones going on at the bottom.

And as in any normal office, those newest postings are always the most interesting. If anything is to be done by your staff, therefore, they will look first at that bottom (the last) memo you've posted.

2. You're already seeing the work of the "memo poster" – that loyal "office boy" on your staff, whose job it is to make sure that your posted memos are given the space and attention they merit.

Of course, this memo poster has worked here longer than you have, so he knows enough to do certain things without being told all the details. After all, weren't you wondering just who was actually cleaning and rearranging the bulletin board to make room for the command line? And who was putting that cursor up there?

And note that the memo poster has put single quotation marks around your message. Why? Because he didn't recognize the memo as anything but a message to be remembered (e.g. "softball practice today at 6:00"). Therefore he didn't do anything special to or with the message; he just posted it.

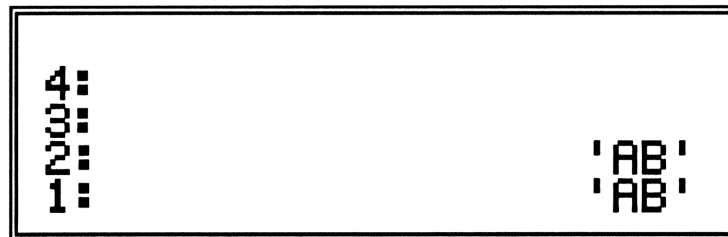
3. The command line then went away. By posting something with the **ENTER** key, you've told the memo poster that you don't need the command line anymore, so he clears it away – to leave more space on the bulletin board for messages.
4. The cursor went away, too. As you know, that cursor will appear only when you're typing in the command line – and the command line is gone now.

Now for the real test: Post something that really *is* a command – a memo that someone in your staff knows is an explicit request to *do* something.

Try This One: Type, from the left-hand keyboard, `DUP`.

The display at this point shows nothing that you haven't seen before (just different characters). And everything was pushed up to make room for the command line, and the cursor is sitting there, telling you where you are. No surprises, right?

Now press `ENTER`. Here's what you should see:



```
4:
3:
2:
1:
'AB'
'AB'
```

And here's why you see it:

DUP is indeed a command that someone in your staff understands. In this case, that someone is the memo poster himself; he recognizes it as a command intended for him and, without hesitation, he does what it tells him to do. He doesn't even bother to post it – he just does it.

DUP is shorthand for "DUPLICATE the last message on the bulletin board."

The memo poster reads this and quickly makes a copy of the bottommost memo (i.e., the memo at Level 1). Then he pushes the old memo ('AB') up the board, and posts the new memo (the copy of 'AB') as the last message on the board.

And keep this in mind: To you, there's no real difference between posting a command memo and posting any other kind of memo. Either way, you can just type it in and press **ENTER**.

The difference to your staff is whether or not someone knows what to do with it. In this case, your office clerk – the memo poster – was the person responsible for carrying out the command, and he did so immediately.

Now stop and recap for a minute: What all do you now know about this HP-28S "staff" you have working for you?

You've seen basically how you and your office/business work together – how the common language currency is the memo. You also know how to write and post these memos, *and* you know that there are basically "information" memos and "command" memos.

Are you starting to feel more at home in your new position (a couple of ferns and some pictures of the family ought to just about do it, then, eh)?

Weeell...unfortunately, being somewhat new at the job of president, you don't yet quite know all the commands you might need for working with your staff.

But all is not lost. You do have a command card file.



The Menu Keys: Your Command Card File

The key to any efficient office is organization. And though you may not realize it yet, your office is organized "to the max."






You have a command card file, a file containing virtually every command that your staff can execute. Not only that, being a card file, it has index tabs (those little category names that stick up out of the card file, effectively dividing the file into sections). It's about time to explore this card file and see how it works, but before you do that, you should first know about this:




The ("Shift") Key

See that red key on the right-hand keyboard? Now notice that most of the keys on both keyboards have red words or symbols written above them. This is not a coincidence.



Up to now, you've assumed that when you press a key, it will produce the action or character written on the key face (e.g., pressing  causes an  to be placed into the command line).

Well, by pressing the red key and then any key with a red word or symbol over it, you'll produce the action or character that's written in red over that key.

For example, press  . What happens? A  is written into the command line. And  just happens to be what's written in red above the  key. The red key is called the *shift* key, because it shifts the operation of the keys to a second set of operations – just like the shift key on a typewriter.

(If you have indeed pressed   , then press  now, before you go on.)

Back to this card file you were going to explore. Notice the top three rows of keys on the left-hand keyboard. Most of these keys have red words inside light-grey boxes above them – as do the keys in the second row on the right-hand keyboard. These keys are the index tabs for your command card file. As in a real card file, if you select one of these index tabs, you should find a logically-related group of "things" under it.

Try One: Press  **A** (which is really  **ARRAY**). You should see:



As you look at this, you should realize:

1. The words in the black boxes at the bottom of the display are all commands. What's more, they're all related – they're all *array* commands. **ARRAY** is the red word over the **A** key, which you just pressed. In other words, you've selected the **ARRAY** index tab, so you're given this set of array commands.
2. A set of commands such as this is called a *menu*, because it's a list of items from which you choose, just as in a restaurant.
3. A menu's appearance in the display moves everything else on the bulletin board up one line. Notice that this doesn't make the memos any older; it only moves everything up out of the way – just as the command line does.

Now then: You've seen how the command line will take the bottom line of the display. But what happens if there's a menu already there when you activate the command line?

One Way To Find Out: Type **STD**.

You'll see:



As you can see, the menu stays on the bottom line, and the command line takes the next line, pushing everything else up one line farther than usual.

Why does the menu remain? Because you might want to use one of its choices in the command line.

Now press **ENTER** to execute what you've just keyed in (apparently it was a command that was meaningful to someone on your calculator staff. You can see this by the fact that it wasn't simply put up on the bulletin board as a message. Instead, someone recognized it and did it – immediately).

Try Another Menu: Pick another index tab from the card file, say,  **REAL**
( **F**).

You should see:



This is the REAL number menu. Because it's a menu, you should be able to pick and use an item from it.

To do so, first notice that the black boxes around the items are lined up over the top row of keys on the right-hand keyboard – and those keys are blank.

Another non-coincidence.

Whenever a menu is shown in the display, the keys in that top row take on the meanings of the names in the menu. By pressing the key under an item, you will be choosing that item from the menu.

Order From This Menu: For example, press the key under **MAXR** and see:



MAXR, in this case, is another command to the message poster. It just says "Post this name (MAXR) as the message." As you'll see later, posting such names can be very useful in certain situations.

But here's an important point: Menus are a convenience feature, *not* a vital necessity. You could have *typed* the name, MAXR, to get the same result. In fact,


Try It: Type **M****A****X****R** **ENTER**.



You accomplished the same thing on your typewriter as you did with your card file! You can therefore think of your card file as your stock of ready-typed memos.


In case you were wondering, it's true that many menus have more than six items. To see the others, you simply need to flip to the next "page" of the menu by using the **NEXT** key. To flip pages in the other direction, use **PREV**. Practice now with these two keys by looking through the entire REAL number menu.

Once again, tick off the things you now know:

You know how your memo poster obeys your keyboard by posting or acting upon memos. And you know that you can change the meanings of keys with the  ("shift") key.

You also know how to pull out various collections of related commands from your command card file. Each such collection is called a menu, and when you want to, you can select from it by using the blank keys on the top of the right-hand keyboard.

But those menus just give you easy access to the names of the commands. What if you forget the particular rules for using them?

Ask your office boy. If you press  CATALOG, you can get him to show you the details for each command – rules and limitations that might appear on the bottom of each card in a real, paper card file. You can check the spelling, fetch, or refresh your memory on the use of these commands, either in straight alphabetical order or beginning with whatever letter you specify.

Play around with this special CATALOG menu. The commands on *this* menu are fairly self-explanatory, so go ahead – try 'em out!

Immediate Execution ("Do-It-Now") Keys

Now that you know where to find commands and menus, the next thing to notice is that menu-related keys work a bit differently than the typewriter keys.

When you pressed the typewriter keys, the command line came on, and the characters that you typed were placed there – but the message you were typing wasn't considered by the memo poster as being ready for posting until you pressed `ENTER`.

By contrast, when you press a menu key (either an index tab or a command from a menu), the effect is to "do-it-now." Such keys *don't* wait for you to press `ENTER` before they present themselves to the memo poster; in essence, they "press the `ENTER` on themselves," thus saving you a keystroke.

And some of these immediate-execution keys are so frightfully useful that they've been awarded keys of their own. Of course, `ENTER` itself is one such vital "do-it-now" key, but now it's time to introduce `ENTER`'s counterpart, which is also a "do-it-now" key:

`DROP`

As you know, `ENTER` tells the memo poster to put memos on the bulletin board. But what if you want to take memos *off* of the board? What if you make a mistake and don't notice it until after that erroneous memo is already posted? How do you discard and replace it?

You press `DROP`. The `DROP` command tells the memo poster to rip down and trash the last memo and move the rest of the memos on the board down a level.

So if you were to press **DROP** right now, what would you expect to see?

Try It: Use **DROP** (it's on the right-hand keyboard, just above **9**). Press it once and voila:



Memo 1 is trashed and everything else is moved down one level – just as you would have expected, knowing the rules for DROP.

Notice, by the way, that **'AB'** sitting up there at Level 3. Where did it come from?

It used to be up at Level 4.

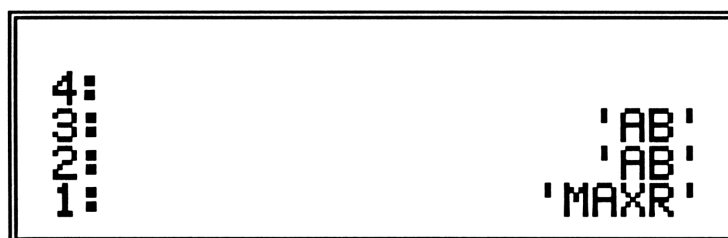
Nothing had really "happened" to it; you just couldn't see it while it was on Level 4. The bulletin board is, for all practical purposes, "infinitely tall." But the display isn't (an infinitely tall display doesn't fit very well in a calculator).

So for practical reasons, the HP-28S shows you, at most, the bottom four levels of the bulletin board. *But*, any items you have posted which have been bumped up above the fourth display line are still on the bulletin board, safe and sound.

All right, so you've seen some immediate-execution ("do-it-now") keys, a couple of which you'll be using quite a bit: **ENTER** and **DROP**.

But there are plenty of other such keys. For example, notice that weird-looking one next to the shift key: **↔**.

What Does It Do? Press it once and see the following:



What happened? The menu went away.

↔ will turn the menu display either on or off, whichever makes sense at the moment (press it again and the menu comes back; once more and the menu goes away again, etc.).

The big advantage of this sleight-of-hand is that when you don't need the menu, you don't have to keep it around cluttering up the bulletin board. That **↔** key is your quick, convenient way to tell your memo poster to set that current menu aside until you ask for it again.

Messages From the System – Memos From Your Staff

At this point, you've explored some of the ways that you can use to communicate with your staff, but you really haven't seen much about how your staff responds to your commands and messages.

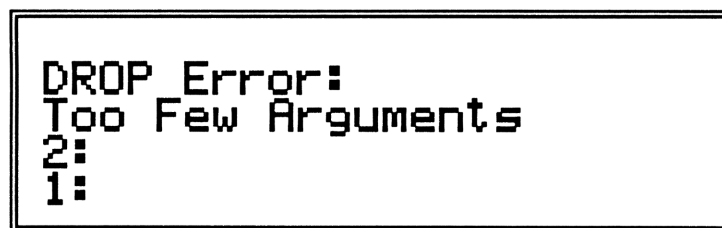
Now, everybody knows that one of an employee's most important jobs is to tell the boss when he's messed up. It's time to see how your staff does this for you.

First, of course, you have to make a mistake (this may come as a shock to you personally, not having done such a thing in so long; but of course, if you make a mistake on purpose, then it's not really a mistake, is it?). All right then,

Mess Up: Press `[DROP]` three times. All the memos are gone. You've dropped them all.

But now, what happens if you tell the memo poster to drop a memo when there's not one there to drop? ("Let's find out ... 1 ... 2 ... 3 ...")
Press `[DROP]` once more.

You'll hear a beep (to get your attention), and you'll see:



```
DROP Error:
Too Few Arguments
2:
1:
```

Is your staff bored because there hasn't been enough bickering in the office lately?

Not really. Actually, this mathematical staff of yours is just guilty of using big words. When they say argument, they mean "something to work on."

So your memo poster is simply telling you that when you told him to drop something off the bottom of the bulletin board, he didn't have anything to drop. A reasonable objection, don't you think?

But forgetting for a moment about what this particular message says, you should examine in general what your staff does whenever they notice a mistake of yours.

1. They yell at you (remember the beep? Yep – that was them yelling at you).
2. They post a memo for the whole office to read. The memo says, in effect,

"You Blew It and This Is Why"

3. This message is posted at the top of the bulletin board – as are all error messages. But these messages don't bump others off or push them up the board; they just temporarily cover up what's there.

The next time you do anything to the bulletin-board, the memo poster will remove the error message (and if you just want to be rid of the error message without otherwise changing the board, you can do so by pressing **ATTN**, which is the **ON** key).

So error messages are one way your employees talk to you.

Status Messages: The Annunciator Area

There's another way your staff can tell you things.

There is actually *another* line visible on the display/bulletin board – above the fourth "active memo" line. Up to now, this area has been largely irrelevant as you've been learning your way "around the office." But now take a look at it.

That line is the *annunciator* area, a place where little "wait-a-minute-I'm-busy" and "remember-your-lunch-money" messages are posted by your staff, for your benefit.

For example, you may have noticed – though it wasn't pointed out – that many times during the process of posting a memo (especially after pressing **ENTER**) the symbol ((•)) will appear briefly on the top line.

Simply put, your staff is telling you that they're busy at the moment. As you may have observed, in most cases, they're so fast that this "busy signal" only flashes (but later on you'll know how to issue commands that will keep them occupied for quite some time).

Another symbol you may have noticed is the one that comes on when you press the shift key. The symbol is **⇧**. It's there to remind you that the next key you press will perform its shifted function (written in red above the key). You can turn the **⇧** off by pressing **■** a second time, thus shifting all keys back to their main functions.

There are other annunciators that can appear on this top line, but you'll encounter them as you go along; no sense crossing those bridges now.

A Tricorder Reading

As usual, before going on, it's a good idea to get your bearings in this mental "world" of your HP-28S.

This first Monday at the office was all about learning to communicate with your staff through memos and messages on a bulletin board.

You saw how the keyboards are connected to this bulletin board (the display), and how the keys produce either immediate actions (the "do-it-now" keys) or characters for building memos.

You specifically know about 3 immediate-execution keys: **ENTER**, **DROP**, and **↔**.



You know that as you type in characters, your "office boy" will show you your memos-in-progress on the command line. And if these typed-in memos are commands recognized by anyone on your staff, they'll be carried out promptly after you officially give your OK to post them (by pressing **ENTER**). If nobody recognizes them, they'll stack up on the bulletin board, with the oldest memos on top.

You know how the **■** key changes the meaning of keys on both keyboards and how a lot of these red-printed functions bring to the menu keys various sets of related commands for your use. And you know that you can review your entire repertoire of commands by pressing **■****CATALOG**.





You know how your calculator staff can give you signals and temporary error messages when they need to – by using either the annunciator area or the top lines of the actual bulletin board.

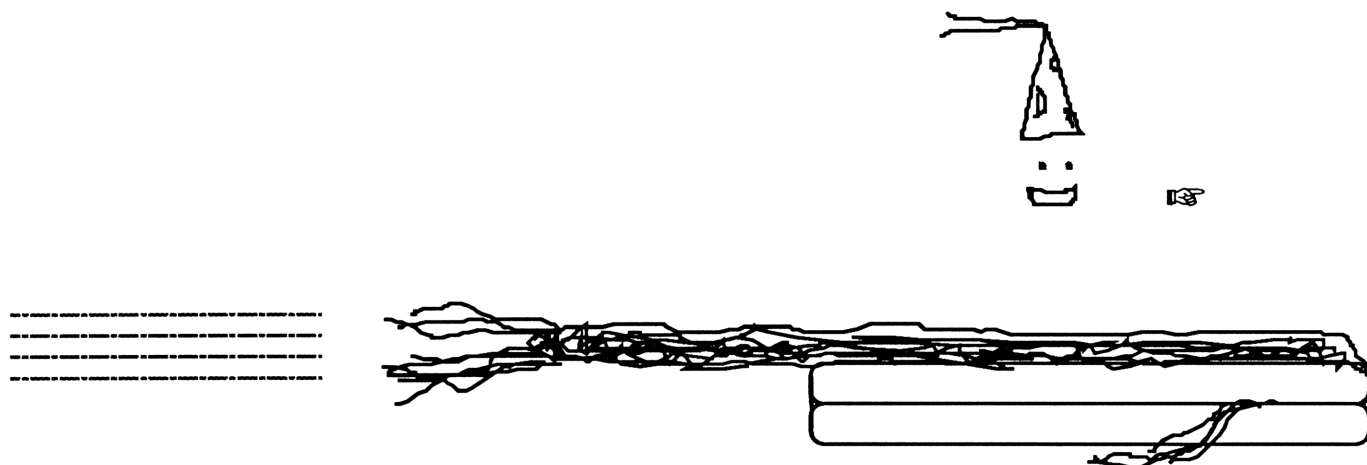
So here's a set of questions to let you test your understanding before you go on. The answers are on the next page, so check yourself; if you need to go back to review, just look on the pages noted after each answer.

Quickie Quiz

1. What's the command line for?
2. What's the main purpose of the left-hand keyboard?
3. What's a menu key?
4. What are ((•)) and  and how do you get rid of them?
5. What's a character?
6. When would you expect to see this:  ?
7. How many days hath September?

Quickie Answers

1. The command line is for typing and editing memos for posting (page 21).
2. The left-hand keyboard is mostly used for typing, especially the alphabetic characters A through Z (page 18).
3. A menu key is one of the six blank keys at the top of the right-hand keyboard which take on the functions of the displayed menu (page 30).
4. ((•)) and  are both annunciators, appearing on the very top of the display. ((•)) is the busy annunciator, which you would get rid of simply by waiting for the machine to finish what it's doing. The  is the shift annunciator, and you would press  to turn it off (page 38).
5. A character is any alphabetic letter, numeral, or special symbol that the HP-28S can generate (page 21).
6. You see  (the cursor) when the command line is active (page 22).
7. September hath thirty days.



Making Your HP-28S Work For You: The Command Line

The command line is where you'll be spending much of your time and energy as you communicate with your HP-28S. So now that you've seen most of the various communication channels you have with your office staff, it's time to concentrate on this particular one. This chapter is all about the editing and presentation options you have in the command line.*

Typing Characters Into the Command Line

As you know, the command line is where you type in numbers and words – as series of characters – preparing them for posting or for issuing as commands. Indeed, you've seen how directly it can be compared to the output portion of a typewriter. It is, in effect, a very simple text editor.

But have you noticed that there's no command that says "Start the command line"? Rather, certain keys that you often use in spelling out commands and memos *automatically* tell that "memo poster" to start the command line.

The most commonly used of these keys are the alphabetic and numeric keys, [A] through [Z] and [0] through [9]. Invariably, if you press one of these keys when you're not yet typing in the command line, the memo poster will start a command line for you and begin with the character you typed into it.

And of course, once you've keyed in all the characters you want on the command line, you press [ENTER] to post it.

So if all the command line allowed you to do were to type out commands and other memo postings, life would certainly be fruitful – but it wouldn't be very easy. That is, being not quite perfect, you'll sometimes need simply to correct your typing errors – and mercifully, the command line allows you to do this.


*Of course, if you're already feeling quite comfortable with all that, then you may skip ahead to page 67.

Changing a Character in the Command Line


You already know about the most commonly used correction key: backspace(←).




In the command line, it removes the character immediately to the left of the cursor. In this way, it's quite convenient, especially if you notice your error before you've typed too many more characters.

But if you type something like **CKARACTERISTIC**, then backspacing over all but the first character is a waste, especially since almost everything is correct. Somehow, you need to be able to move the cursor to the second character and replace the **K** with an **H** – without erasing everything else along the way.

Fortunately, you can: Remember the  key? You've seen how it turns on and off the menu area of the display – but that's not its most important talent. The arrows on its face are the tell-tale signs:


The  key *enables* and *disables* the cursor-movement keys.


Those cursor-movement keys are, non-coincidentally, the same as the menu keys. This is because the  key – much like the shift key – *shifts* the function of the blank menu/cursor keys between the current menu's functions and those printed in white *above* the menu keys. *The cursor-movement functions of these keys are available only when there is no menu in the display.*

And notice that, *unlike* the shift key, the  key changes the functions of those keys *until the next time the  key is pressed*. In other words, you don't need to repeatedly press  to maintain the menu selection keys' current functions.

So look now at those cursor-movement keys (called cursor keys, for short).

As you might expect, since they affect the cursor (which exists only when the command line is active), these keys work only with an active command line.


Time For Some Practice: (If at first you see a menu in the display, just press  to get rid of it for now.)

Type in **CKARACTERISTIC**, "mistake" and all (but don't press  afterwards; you're going to "catch" this "mistake" before actually posting it onto the bulletin board).

You should see:



As you know, characters can be added to the command line only at the current location of the cursor. Thus, typing a character key now would add the character to the end of the word and move the cursor one character to the right.

As you also know, you *could* use  repeatedly to delete all of the characters between the cursor and the first **C**, thus deleting the **K** in the process. But all you really want to do is to move the cursor on top of the **K** and overwrite it with an **H**.

How can you do this?

You do it by pressing the key with the white ◀ over it (but remember: the white cursor symbols are only active when there's no menu in the display. When you press a menu key, if there's any menu visible, the function of that menu key – not the cursor control function – will be performed).

So press the ◀ key.

The cursor moves to the left by one character, but it doesn't delete that character. Press it again, and it moves one more character to the left. Press it and hold it down, and the cursor will continue to move to the left until you let up on the key. When the cursor has moved all the way to the left – over the top of the first character – pressing ◀ will no longer move it at all.


Now press ▶. What happens?




No real surprises here, right? ▶ moves the cursor to the right, but notice that if you keep pressing the ▶ key until you reach the last character of the word, the cursor doesn't stop there; it goes one space farther, to exactly where it was when you stopped typing the word in the first place – and for the same reason – so that you can add more characters to the end of the word.

Now, go fix that typo.


Playing Editor: Press . What happened?

The cursor moved all the way to the left-most character.

Press . What happened? The cursor moved all the way to the right.

These are shortcuts. You could have accomplished the same thing simply by pressing and holding down either the  or  keys, respectively; but pressing  saves you some time.

So press  and then .

The cursor will now be over the **K**. Since characters are added to the command line at the position of the cursor, pressing  now will put an **H** in the command line – right where the **K** used to be. Do it.

As you've come to expect, the cursor then moves one space to the right.

Adding and Removing Characters

Now, what if you had simply omitted a character, rather than accidentally typed the wrong one?

To see how you would deal with this, use ◀ and ▶ to move the cursor so that it's positioned over the **E** in **CHARACTERISTIC**.

Now press ⬅. What happened?

The backspace key did what it always does. It deleted the character immediately to its left. In this case, since there were characters to the right of the deleted character, they were all moved one space to the left, to fill up the hole.

Next, press ⬆▶.

This is what the command line would look like if you had originally forgotten to type the first **T**.



3:
2:
1:
CHARACTERISTIC␣

The image shows a rectangular box representing a command line interface. Inside the box, on the left side, there is a list of three items, each preceded by a number and a colon: '3:', '2:', and '1:'. To the right of this list, on the same line as '1:', is the word 'CHARACTERISTIC' followed by a small square cursor icon.

Notice that you have just learned the way to remove a character or characters if you've typed too many:

You use ◀ and ▶ to move to the space immediately *to the right* of the offending character, then press ⬅ to delete it.

Now the command line is all set up to look just as it would if you had just keyed in **CHARACERISTIC**. You want to correct the omission.

Use **◀** to move to the **E**, which is the character that your missing **T** will *precede*.

Now, can you simply type in a **T** to fix things? Nope. Remember that if you type a character now, it will *replace* the **E**. What can you do?

Press **INS**.

What happens? Look closely and you'll see that the cursor – which was a flashing box (**□**) – is now a flashing arrow (**◄**).

INS is the **INS**ert key. It tells the command line that you want to insert one or more characters before (to the left of) the character that *was* sitting under the flashing box cursor (**□**).

So the arrow is now pointing to the place (between the **C** and **E**) where a character would be added. Make sense? OK, do it: Press **T**. Now what happens?



```
3:
2:
1:
CHARACT◄RISTIC
```

First, the **T** was added to the command line at the place where the cursor was. Then the cursor moved one space to the right. What's different is that everything to the right of the arrow's point moved with the arrow. You have now corrected the omission!

Press ◀ and ▶ a couple of times. Notice that they work the same way with this arrow cursor as with the box cursor. The only difference is when you press a character key.

When the cursor is a *box*, the new character will *replace* the one on which the cursor is sitting.

When the cursor is an *arrow*, the new character will be *inserted before* the character on which the cursor is sitting.

Finally, press [INS]. What happens?

The cursor changes from an arrow back into a box. Repeatedly pressing [INS] will change the cursor back and forth between a box and an arrow. In this way, it's a "toggle key" – like ■ and ◀▶ – shifting alternately between two modes.

Another key that you should find useful when editing the command line is the [DEL] (delete) key. [DEL] works just like ◀, except that instead of removing the character to the *left* of the cursor, it removes the character *under* the cursor.

And just like ◀, all the characters to the right of the deleted character are moved to the left one space to fill up the hole.

Also, both [DEL] and ◀ will repeat their functions if you hold their keys down.

You can see now that you have a number of different ways to correct minor errors you may make while keying in a memo on the command line!

■ **INS**, ■ **DEL** and **ATTN**

Suppose that your error isn't so minor this time: you need to delete more than one character.

Of course, you could always fix things by moving the cursor with ◀ and ▶ and then using either **DEL** or ◀ – as you just saw.

But what if it's a whole string of characters that you need to remove?

In that case – whenever you need to delete *all* characters to the right or to the left of the cursor – you have yet another option....

Using the word **CHARACTERISTIC** from the previous examples, assume that what you really wanted was the word **CHARACTER**.

Assume also that the cursor is now sitting over the **E** – because you just inserted the **T** (so if you've pressed **INS** to get the ■ cursor, then for the purposes of following along here, press **INS** again. Just bear in mind that the example will work no matter which kind of cursor you use).

So you should see the following:

```
3:
2:
1:
CHARACT4RISTIC
```

Obviously, you want to delete **ISTIC**. You could move the cursor to the first **I** and use **[DEL]**. Or you could move the cursor to the right end (with **[▶]**) and use **[←]**. Or, you *could* move the cursor over the first **I** and press **[DEL]**.

Try It (You'll Like It): Move the cursor over the **I** and press **[DEL]**. What happens? Everything to the right of – and under – the cursor is deleted, right? *It's exactly as if you had pressed and held down the **[DEL]** key.*

You're left with **CHARACTER**, and the cursor has been left at the end of the new word, so that you can add more to it if you like.

And Now This: Type in **MITE** and move the cursor so that it's sitting over the first (left-most) **T**. Press **[INS]**.

See? Everything to the *left* of the cursor is deleted, and the remaining characters are shifted to the left. *It's exactly as if you had pressed and held down the **[←]** key.*

Last Resort: If all else fails, you can always press **[ATTN]** (**[ON]**) to clear the whole command line and start with a clean slate.

Try it now. Notice that the **[ON]** key serves two functions: When the HP-28S is off, this turns it on; when it's already on, **[ON]** functions as ATTN (attention), interrupting and effectively shutting off the command line, discarding everything that was in it.

While you're paused here with such a clean slate, take a minute to review all these options for correcting errors on the command line – just to be sure you have them all straight in your mind.

The ◀ and ▶ keys move your cursor to the left and right, respectively. You can't go any farther left than the first character on the line; you can go exactly one place farther than the last character – to be ready to type another, of course.

Pressing ■◀ and ■▶ are shortcuts for moving to the very ends of the command line.

The □ cursor lets you type a new character right *over* an existing one (thus replacing it). The ✚ cursor lets you *insert* a new character *between* existing ones. You alternate back and forth between these two cursors by pressing the INS key.

To delete an unwanted character, you can press ◀, which would delete the character to the *left* of the cursor. Or, you can use the DEL key, which would delete the character *under* the cursor.

Pressing ■INS and ■DEL are shortcuts for deleting all characters from the cursor to the left and right ends of the command line, respectively. The one difference is that ■DEL also deletes the character under the cursor, while ■INS doesn't.

OK so far?

NEWLINE, ▲ and ▼

If you're at all verbose with your commands, you can certainly overrun the visible 23 characters of the command line. But this is no problem, really, because the command line is effectively infinite; you can type as much as you want.

Test This: Type in the 26 letters of the alphabet onto a fresh, clean command line (i.e., first press **ATTN** if there's anything on the command line):

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

You'll see:




Notice that when you exceeded 23 characters, the command line scrolled to the left and showed an ellipsis (...) as the first character to tell you that the command line does indeed continue to the left, but that the beginning part isn't currently visible.

Now press **▲** to get to that far left end.

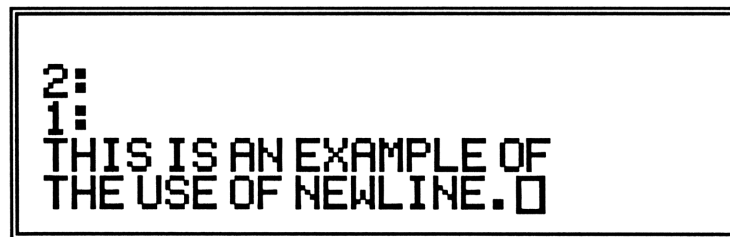
The command line now scrolls to the right and places the ellipsis at the right end of the display. Makes sense, right?

Well, that's all good and fine, but it doesn't take advantage of the other 3 lines in the display that are available to you.


Happily, if you want to see more of the command line, you do have the option of using  **NEWLINE** to separate words....

Try This: Press **ATTN** and type **T H I S SPACE I S SPACE A N SPACE E X
A M P L E SPACE O F  N E W L I N E T H E SPACE U S E SPACE O F
SPACE N E W L I N E .**

This is what you should see:



```
2:
1: THIS IS AN EXAMPLE OF
  THE USE OF NEWLINE.
```

You can use as many  **NEWLINE**'s as you want in order to make things more readable, and you don't need to fill up each line before going on to the next line.

Your command line is essentially unbounded, since you can add lines – separated by **NEWLINE**'s – to the point where the text scrolls off the top of the display. And because the bulletin board is unbounded, even when these lines do scroll up out of sight, they're faithfully preserved and usable!

OK, but how can you see or edit these lines that disappear off the top?

Simple: You move from line to line and scroll lines back into the display with **▲** and **▼**.

These two vertical cursor-movement keys work in the same way that **◀** and **▶** do – except that they move the cursor up and down rather than from side to side. And, as you might expect, **▲** and **▼** also function similarly, sending the cursor to the very top or very bottom line, respectively.

Next question: How can you get rid of these **NEWLINE**'s that you've embedded in your command line?

Next Answer: Use **DEL** to "undo" a **NEWLINE**.

Try It: Press **▲** **▶** to move to the end of the first line, where you pressed **NEWLINE**, and press **DEL**. The two lines are joined into one, with the one that was on the bottom extending off to the right.

Notes: **DEL** and **INS** affect only the line containing the cursor. And you can't use **◀** or **▶** to move from line to line; you must use **▲** and **▼**.

The **[LC]** Key

Up to now, when you've typed something into your HP-28S, it has come out in upper case. But that's not the only way to do things. If you need to use Lower Case letters, just press **[LC]** (it's down there on the bottom line of keys on the left-hand keyboard).

Go ahead and do that now.

Nothing obvious happens, but if you now use any of the alphabetic keys you'll find that they all put lower-case letters into the command line.

Notice that **[LC]** is like **[INS]** in one respect. Once you press it, it stays in effect until you press it again (or press **[ENTER]**), much like the upper case ("Caps") lock key of a standard typewriter.

This may not seem like a very important feature, but you must realize that the case of a character in any command is taken quite literally by the HP-28S. If you accidentally capitalize some character in a command that's not supposed to be capitalized, the machine won't recognize it.

In order to have your commands recognized, you must spell them exactly the way they appear in the command CATALOG, including all upper and lower-case characters.

The α Key

So far, you've been concentrating on the keys that function only to put their symbols into the command line. You've ignored most of the immediate-execution ("do-it-now") keys, such as \oplus or \ominus . Recall that pressing one of those keys all by itself normally causes the calculator to perform that function.

Watch: Press ATTN AB \oplus .

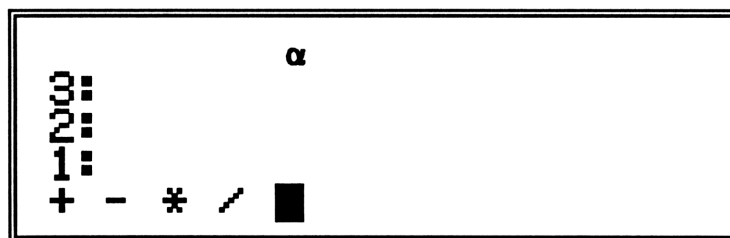
What happened? ' AB ' was posted just as if ENTER had been pressed, and an error message was displayed. Don't worry right now about why this error occurred. Just realize that immediate-execution keys will normally try to "do their things" *even when the command line is active*.


Sometimes this is convenient; sometimes it's not. After all, what if for some reason you wanted a *symbol* such as \oplus or \ominus to appear in the command line?



You would press α first. The α ("alpha") key tells the system (your memo-obeying staff) to treat the keys pressed as character keys rather than as "do-it-now" keys.

As you might suspect, certain vital immediate-execution keys, such as ENTER and ATTN , are exempt from the α key's disabling influence.







So Try It: Press      .




See how convenient  can be when you want to type merely symbols rather than the commands usually associated with those symbols? All of those symbols have been entered as plain old, garden-variety characters into the command line.

Notice that when you pressed , the α annunciator appeared in the top line of the display and the cursor changed to a solid block () to remind you that you are in this mode where most of the immediate-execution keys are "blocked" from executing immediately.

Alpha mode will stay on until you press  or  *twice*.

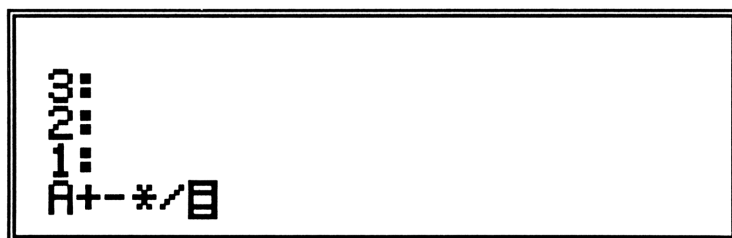
"Twice?" Yes, twice. The  key is a *three-way* toggle. You've already seen two-way toggles (, ,  and ) that turn certain modes on or off. Well,  cycles between three different modes.

Think about it this way: normally, when the command line is active, you get the  cursor – the immediate-entry cursor – and are in immediate entry mode. Character keys put their characters in the command line and command keys do their commands.

But if you press α immediately before or while the command line is active, the α annunciator comes on, you get \blacksquare – the alpha cursor – and you are in alpha mode. Most keys will then put characters into the command line, but notice that keys like \oplus , \ominus , \otimes and \oslash will put spaces around their names, too.

Now, if you're in alpha mode, the command line is active, and you press the α key, you'll get the \equiv cursor – the *algebraic* cursor – and be in *algebraic* mode. You'll see most of the usefulness of this mode later, but for a sneak preview,

Looky Here: Press ATTN A $\alpha\alpha$ $\oplus\ominus\otimes\oslash$.












There are no spaces between the characters.


Notice, though, that you must have at least one character in the command line (to activate it) before you can activate algebraic mode.

If you press α again while in algebraic mode, the cursor changes back into \blacksquare .

You're adding rapidly to your bag of tricks for controlling the command line. First, you learned how to correct errors. Now you've seen some ways to key in lower-case letters, long strings of characters, or strings involving symbols normally reserved for immediate execution. Review:

The   key produces an invisible character that lets you break a long command strings into manageable segments, so that you will see these segments in your display (your bulletin board) on adjacent lines. When you have such a multi-line command line, you can move around between lines with the help of  and  and their short-cutting versions,   and  .

The  key lets you type in lower-case letters – until you press it again or post the memo.

The  key toggles between three different entry modes – immediate, alpha and algebraic.

Now go on and look at some convenient variations of skills you already have....

Item Delimiters and **ENTER**

In past examples, you've almost never completed the memo on your command line to the point of pressing **ENTER**.

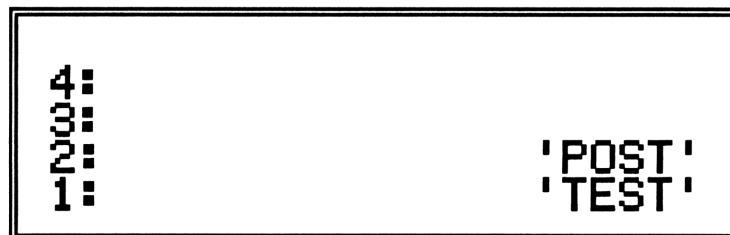
That is, you didn't actually give the go-ahead to your faithful office boy, the memo-poster, to officially post a message or otherwise try to evaluate the command line.

This is because most of what you've keyed in so far just wouldn't make much sense – either to you or to your office staff (the calculator system) when it was evaluated.

And on those occasions when you have pressed **ENTER**, you may have noticed that the command line may not have been posted as a single memo.

For Example: Type in **ATTN POST SPACE TEST ENTER**.

What you'll get is this:



```
4:
3:
2:
1:
      'POST'
      'TEST'
```

What's going on here, anyway?

Three things of interest:

1. The space between the two words in the command line effectively separates them into two postings when **ENTER** is pressed. In this case, then, the space is called a *delimiter*, because it acts as a marker, denoting the end of one memo and the beginning of another.
2. The memos are posted from left to right; the word on the left was posted before the word on the right (and as a consequence, **POST** now appears farther up on your positionally-"dated" bulletin board).
3. Neither of these words was recognized by your calculator's system, so they were posted as is – with single quotation marks to let you know this.

Conclusion: *You can use the command line for posting more than one memo at a time by marking each successive item with a delimiter character!*

So, besides **SPACE**, what other characters will play this role of delimiter?

NEWLINE will.

So will the comma: **,** or period: **.** – whichever the HP-28S is *not* currently using as the radix mark (decimal point). In other words, if the current radix mark is the period (i.e. if **1.5** is interpreted to mean one-and-a-half), the comma is a delimiter; but if the current radix mark is the comma (**1,5** = one-and-a-half), then the period is free to be used as a delimiter.

There are many other delimiters too: **{**, **}**, **[**, **]**, **#**, **"**, **'**, **⌘**, and **⌘**. But these all have special meanings to the calculator – meanings you'll see later.

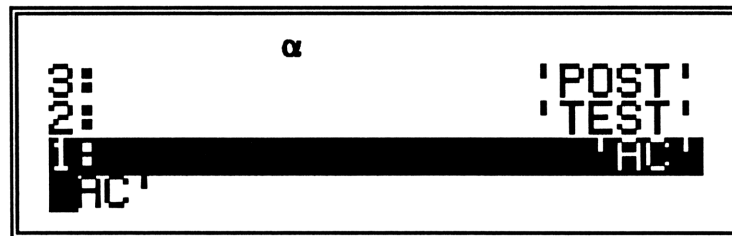
COMMAND and EDIT

Another variation on something you've already seen: You've seen how to correct errors in the command line – as long as you catch them before you press **ENTER**. But what if you don't catch them that soon? How do you "undo" an error that has gone so far as to be officially posted on your bulletin board?

Of course, you could simply unpost the memo (using **DROP**) and totally retype it. But this seems like a colossal waste of time if the error is minor and the memo is major. Wouldn't it be nice if you could just *edit* the posted memo?

Good News: Press **AC****ENTER**; but now decide that you really wanted 'ABC'.


So press **EDIT**. You should see this:




Notice what has happened: The command line is activated *containing the contents of Level 1*, which is highlit to show that it's being edited. Alpha mode is activated for your convenience, as indicated by the α annunciator and the solid cursor.

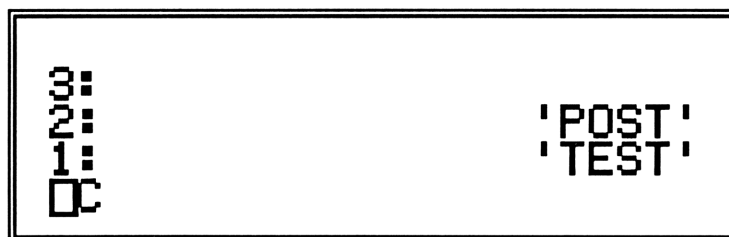
You may now edit the memo in the same way that you would edit anything in the command line. And when you're finished, pressing **ENTER** *replaces* the highlit line with what's in the command line. Or, if you change your mind midway, pressing **ATTN** aborts the edit and does *not* change the highlit line.


Even if you do DROP an erroneous memo, there's still a time-saving shortcut to re-post it:





DROP the bad memo and then press  **COMMAND**.

Try It: Press **ATTN** (to discard whatever command line you may now have in progress).

Then: **DROP**  **COMMAND**. You should see the following:*



The last thing you typed and followed with **ENTER** is what will be in the command line after you press  **COMMAND**. Then you can edit it in the old familiar way(cursor keys and all that) and re-post it with **ENTER**.

* This feature is actually an option – one that you can disable ("turn off") if you wish. Only those commands entered while the command memory is enabled will be remembered. It's your choice – and this and other such preference options live in the MODE menu. Thus, you would press  **MODE** **NEXT**  **CMO** to enable this command memory (it's a command "stack," actually).  **CMO** is off,  **CMO** is on.

By the way, **■****COMMAND** has an even better memory than you might have first supposed. Not only does it remember the last memo you posted, it remembers the three before that, too!

If you press **■****COMMAND** a second, third and fourth time, you'll see the second-to-last, third-to-last, and fourth-to-last commands (or "memos") you have **ENTER**'ed, respectively. And as each one of these comes to the command line, you may edit it or repost it with **ENTER**.

If you press **■****COMMAND** a fifth time, it will cycle around and show you the most recently posted command again. As always, you can get rid of the command line altogether by pressing **ATTN**.



This extra-good, short-term memory may hint to you of another advantage to using **■****COMMAND**.

If you're doing a lot of posting and many of the memos are the same, you don't need to retype any that were already posted within the last four postings. You can just use **■****COMMAND** to call them up again.














































As you might imagine, this is especially useful if the postings were long or tricky!

Now it's time to put it all together and see how well you know your way around the command line....

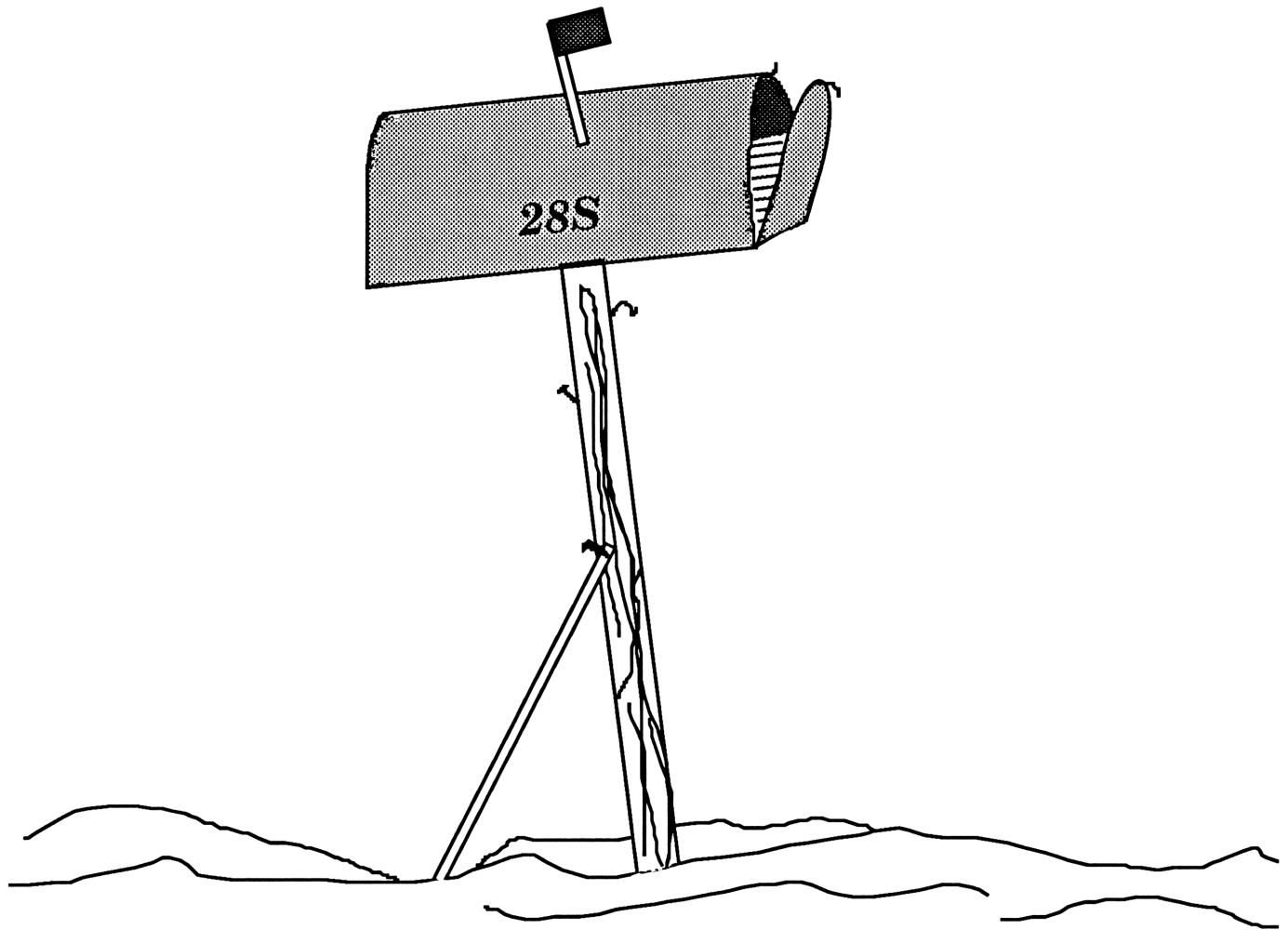
Command Line Quiz

1. How do you turn on the command line?
2. What's the difference between the functions of  and  ?
3. What's a delimiter? Name two.
4. Change **'CONFRONTABLE'** to **'COMFORTABLE'** (assume that the memo **'CONFRONTABLE'** is now sitting at Level 1 of your bulletin board).
5. Change **'centimeter'** to **'centipede'** (again, assume that what you start with is sitting at Level 1).
6. Change **'Apples'** to **'Oranges'** (again, at Level 1).
7. What's the capital of Montana?

Command Line Answers

1. Press any character key (see page 43 to review this).
2.  deletes the character *to the left of* the cursor;  deletes the character *under* the cursor (pages 48-50).
3. A delimiter is a separator. In the case of the HP-28S, it is a character separating two memos in the command line.  and  are two examples (pages 62-63).
4.     M    R  or     M F O R  , for example; of course, there are many different ways to accomplish such an editing job (page 53).
5.       LC P E D E   or        LC P E D E  or        LC P E D E , for example (pages 53 and 57).
6.   LC R A N G E S  (page 57).
7. Helena.

Notes



Real Numbers, The Stack, And Postfix Notation

How does this imaginary HP-28S world seem to you now? On your first Monday at your new job as president, you familiarized yourself with your office, the bulletin board, your typewriter, your command file, and your faithful office boy, the memo poster. On Tuesday, you spent all day learning all those skills for efficiently posting and editing memos.

It's now Wednesday morning (time flies when you're having fun).

Real Numbers – and the Real World

There are a lot more high-powered brains working back there in the offices beyond the bulletin board. It's time y'all were introduced.

As you know, many of your HP-28S "staff" system's talents lie in the area of numeric problem-solving. So you can't really relate to them unless you're brushed up on their language: numbers.

It's probably best to begin by using real numbers, since they're probably familiar to you already.*

Numbers can be broken up into different classes which are useful in different circumstances. *Real numbers* form a collection of most of these classes into the one big group that you normally think of as being numbers: the positive and negative integers (1, 2, -3, -5, etc.), the positive and negative rational numbers (4.56, -.23, etc.), the positive and negative irrational numbers ($-\pi$, e) and zero (0).

*If you already know how to key in, format, and otherwise represent real numbers on the HP-28S, then now's a good time to skip over to page 84.

Right, then: Your first priority is to learn to communicate with your calculator's math brains. That is, when it shows you a number, you'd better be able to recognize it.

Usually this isn't too much of a problem, except for extremely large and extremely small real numbers. These are always a bit awkward to deal with (in any tool – from paper and pencil to a high-speed computer), because their representations use a lot of digits.

For example, the number one-half is relatively easy to write. Its representation is .5, ("point-five" or "five tenths"). But much smaller numbers, such as one thousandth (.0001), are more cumbersome and less easy to read.

And *really* small numbers – like one hundred-millionth (.00000001) – or *really* big numbers – like one billion (1,000,000,000) – are actually unpleasant to deal with, precisely because of all of these zeros you need to carry around.

For this reason, an alternate representation has been developed, called *scientific notation*.^{*} In this notation, you take a number and split it into two parts. The first part consists of all the digits except leading or trailing zeroes. The second part tells you how many of these leading or trailing zeros you also need and whether they're leading or trailing.

Thus, 5,280 is 5.28×10^3 , 0.00023 is 2.3×10^{-4} , and 1 is 1×10^0 .

Notice the convention here. The first part of the number (called the "mantissa") shows its *precision* and is written with its first digit just to the left of the decimal point, with the rest of the digits, if any, to the right. The mantissa is then multiplied by a power of 10 (called the "exponent") in order to show the number's *magnitude*.

^{*}It's called scientific notation not because it's in any way more "scientific" than other notations, but because in science one commonly deals with very large or very small numbers. It could as easily have been called "national debt notation," for example.

Representing Real Numbers On the HP-28S

Scientific notation is especially useful for representing numbers on machines. As you would expect, the HP-28S can be used to represent and manipulate real numbers of extraordinary magnitudes. But being just a finite machine, it has some limitations, peculiarities, and rules that you need to understand if you're going to communicate with it well. Fortunately, these are few and reasonable.

Scientific Notation on Your HP-28S

First of all, the HP-28S does not use strict scientific notation. It uses a slightly compacted, computerized version of it.

For example, 2.5×10^4 is represented as **2.5E4**

And 3.9×10^{-6} appears as **3.9E-6**

As you can see, the **E** means "...times ten to the...". That is, the number following it is the Exponent of 10.

It's just a convenient way to write scientific notation without resorting to superscripts in the display of the calculator.

12-Digit Accuracy: Rounding Error

Secondly, keep in mind that some real numbers have representations that are just plain *infinite*. For example, the decimal representation of $1/3$ is $.333\dots$, where the 3's continue forever.

Of course, it's unreasonable (and fortunately, unnecessary) to try to use all of those 3's during real-number arithmetic. What you do, naturally, is *round* it, shortening it to a value that is both convenient and accurate enough for your purposes. To be sure, the rounded number is *not* the same as the original, but the difference is negligible in practice.

Well, the HP-28S rounds, too. In dealing with infinite or extremely long representations, it rounds the number, remembering 12 digits of the original number.

The inaccuracy that results is called *rounding error*. And as you would suspect, multiplying together two rounded numbers will multiply this error.

So, just how great an error is this?

"Let's find out."

Say that you're the pilot of a plane flying from Los Angeles to New York – a distance of 3,000 miles. Well, it's a lovely day, and once airborne, your navigator lets it slip that he's been using his HP-28S to do fuel calculations.

Not only that, he freely admits that his computations of the number of miles per pound of fuel are only accurate to $.000000000001$ miles (the 12th digit).

Uh-oh. If his calculations are off by that much per mile, how big an error will this make over a *lot* of miles (3,000)...?

Oh, about one two-hundredth of a millimeter.

Not a lot, really.

If you'd flown clear to the moon and back, instead (roughly 500,000 miles), the accumulated error would be an entire eight-tenths of a millimeter.

And in a round trip to the sun (about 186,000,000 miles) you'd gain or lose about a foot (*now* you're talking *gross* error).

As you can see, digital accuracy to 12 decimal places as given to you by the HP-28S is slightly more than barely adequate. So if an answer isn't exactly what you were expecting, it's very, very close.

Magnitude: How Big (or Small) Can You Get?

A third limitation of the HP-28S is the *magnitude* of a real number (i.e. the numeric value – not the number of digits) it can represent. And again, it's the finite nature of the HP-28S that imposes this limitation; you simply cannot expect it to be able to represent arbitrarily large or small numbers (everyone has his limit; you do and so does your machine).

The largest real-number value representable on the HP-28S (which you can produce with the command MAXR – "MAXimum Real") is

$$9.99999999999 \times 10^{499}$$

And the smallest representable real-number value (which will result when you use the command MINR – "MINimum Real") is

$$1 \times 10^{-499}$$

These numbers are *extremely* large and small, respectively. It's difficult, if not truly impossible, to convey – or even contemplate – the enormity and tininess of these values.

"It's a tough job...but someone's gotta do it..."

On the small end of things, picture in your mind that packed pile of MAXR electrons. Then picture yourself picking out just ten of those electrons. That ten – in relation to the whole – is the fraction you're talking about when you say "MINR."

So you see, the magnitude limits of the HP-28S aren't all that restrictive. Indeed, to further put things in perspective, you may have heard of human societies whose numbering systems went something like:

"1...2...3...more than 3."

And that was all the farther they could describe numerical magnitude.

So it is in every society. In this modern-day technical world, for example, the numbering goes well beyond 3, but at some point, it runs out of names and meanings, too. "Millions...billions...trillions...quadrillions..." etc, up to about "nonillions" (?), which are on the order of 10^{30} . But what do you call numbers on the order of 10^{100} – or 10^{400} ?*

Truly, there is a limit to practical needs to describe numbers. One society's limit may simply be a little higher than another's – but not much.

*The authors recommend the term "several gadzillion."

Posting Real Numbers: **[CHS]**, **[EEX]** and Display Modes

Now then: Those are the details about how the HP-28S can and cannot represent real numbers. Knowing these rules and limitations, it's time you started posting real numbers as memos to your calculations staff. You'll see right away that it isn't much different than posting any other kind of information, except that you use the number keys to key them in.

For Example: Post 5,280, 365.25 and 6.022×10^{23} .

Solution: Press **[ATTN]** **[CLEAR]** **[STD]** **[ENTER]** **[5]** **[2]** **[8]** **[0]** **[ENTER]** **[3]** **[6]** **[5]** **[.]** **[2]** **[5]** **[ENTER]** **[6]** **[.]** **[0]** **[2]** **[2]** **[E]** **[2]** **[3]** **[ENTER]**.

You should see:

4:	5280
3:	365.25
2:	6.022E23
1:	

Notice that when keying in that last number, **6.022E23**, you used the **[E]** key. You *could* have used **[EEX]** (Enter EXponent).

[EEX] works the same as **[E]** except for one case. Press **[EEX]**. What happens? Since you hadn't specified the number to the left of the **E** (the *mantissa*), **[EEX]** supplied you with one: **1**. It's just an added nicety of the EEX function.

(Now press **[ATTN]** to clear the command line).

OK, now how about posting a negative number? You have two ways to do this:

First method: You can post the positive number in the usual way (up to and including pressing **ENTER**) and *then* press **CHS** (CHange Sign).

Try It Now: Press **CHS**. The number in Level 1 becomes negative.

Press **CHS** again – to make that number positive again.

Second method: You can change the sign of either the mantissa or the exponent at any time while you're keying in that part.

Examples: Post **-1.3**, **4.5E-24**, **-7.8E3** and **-9E-54**.

Solutions: Press **1** **.** **3** **CHS** **ENTER** **4** **.** **5** **EEX** **2** **4** **CHS** **ENTER** **7** **CHS** **.** **8** **E** **3** **,** **CHS** **9** **EEX** **CHS** **5** **4** **ENTER**.

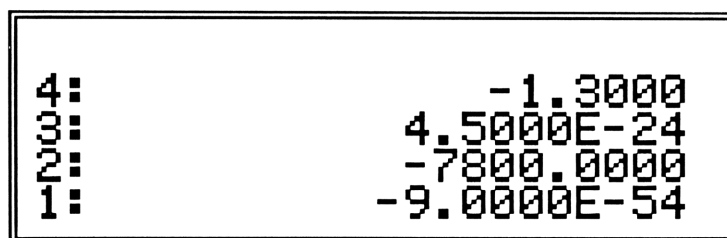
You'll see this:

4:	-1.3
3:	4.5E-24
2:	-7800
1:	-9.E-54

Notice that pressing **CHS** *before* you start to key in the number will work only if the command line is already active. If not, then **CHS** will change the sign of the number in Level 1, as in the first method, above.

Display Formats

Try this: Press `4` `,` `F` `I` `X` `ENTER`. Continuing on from the previous example, you should see this:




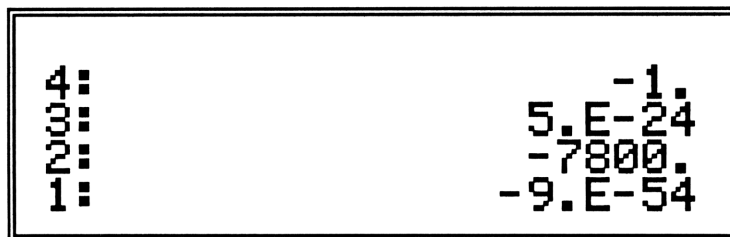
4:	-1.3000
3:	4.5000E-24
2:	-7800.0000
1:	-9.0000E-54

What's-a-goin' on?

Well, you just told the HP-28S to change the *format* of real numbers in the display. That is, the *values* of the numbers haven't changed – just the way you *see* them.


`4` `,` `F` `I` `X` `ENTER` told the HP-28S to display a FIxed number of digits – four in this case – to the right of the decimal point. As you can see, that's just what it did. When there are no more significant digits to be displayed, one or more zeroes are added to the end of the number to fulfill the FIx requirement.

Next: Press  **COMMAND** **0** **ENTER**. Here's what you'll see:



The calculator display shows a list of numbers with their indices on the left and the values on the right:

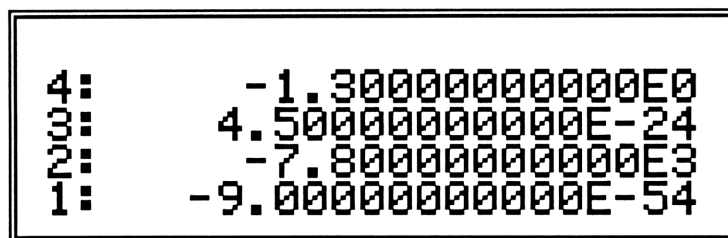
Index	Value
4:	-1.
3:	5.E-24
2:	-7800.
1:	-9.E-54

First of all, recall how  **COMMAND** retrieves your most recently posted command (**4, FIX**, in this case). Since the first character, **4**, is the one you wanted to change, you overwrote it with **0** and then pressed **ENTER** to execute the rewritten command.

So there are now zero digits to the right of the decimal point. Again, the numbers haven't changed in value; only their appearances have.

Remember! *All these display formats are only the display's "editing" of the numbers for presentation to you. The internal representations of the numbers – for purposes of computation – are always fully 12 digits of precision.*

Once Again: Press `11'SCI``ENTER`. Result:



```
4:      -1.300000000000E0
3:      4.500000000000E-24
2:      -7.800000000000E3
1:      -9.000000000000E-54
```

This example brings up an important point: In the previous examples there were numbers in the display formatted in scientific notation even though the mode was FIX. That's because there are only 12 digits possible to display a real number. Therefore, any number larger than 999999999999 (twelve nines) or smaller than .000000000001 ("point-eleven-zeros-and-a-one") will be displayed in scientific notation *by default*, because its magnitude exceeds the ability of the display to show in an explicit, one-part number.

But now, with SCI mode, not only have you set the number of digits to be displayed, but you've *forced* the display to use scientific notation for *every* number – regardless of whether or not that number could otherwise be correctly represented in the display. To see this, compare how Level 4 looks now to how it looked after the previous example. Although the exponent is 0, the number is still expressed in scientific notation here.

Finally: Press `STD``ENTER`.

This is STandarD display format, where you started on page 79. All significant digits of all numbers are displayed and scientific notation is used only when the number overruns the display's magnitude limits.

The Stack and Postfix Notation

"OK, ok: Scientific notation... real-number representation limits... display formatting... when are we going to get to the part where I start *doing* things – like arithmetic – with real numbers?"

Right now. Begin by noticing that what you see in the display is, quite literally, a *stack* of numbers. It's true. Everything you've posted so far has been "stacked up" on the bulletin-board.

This particular stack may look upside down to you, compared to other stacks of things you've seen, because you put the latest additions on the *bottom* here. Aside from that, it works in much the same way as any stack of "stuff."

If you think about it for a moment, you'll realize that a stack is merely a Last-In-First-Out type of arrangement, where the last thing you put on the stack is the first thing you take off.

But – as you've so eloquently put it – putting things on and taking things off the stack doesn't accomplish a lot. You want to be able to do other things with the stuff on the stack. With real numbers, for example, it would be nice to do some math. Of course, you can.

But here's the idea to hang onto as you begin: An HP-28S command that uses this stack – any math operation, for example – assumes that *something to operate on will already be in the stack when you invoke the operation itself*.

In effect, you must first put onto the stack any number(s) you want to manipulate, and *then* perform the operation. This way of doing things is called postfix (post-affix: literally, "to add after"), because the operation itself comes *after* the operands.

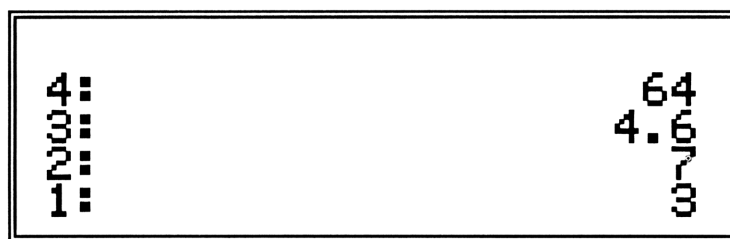
Real Number Commands: 0-, 1-, and 2- Number Operations*

Just so that you have some simple arithmetic to follow,

Do This: Key in these numbers: 100, 64, 4.6, 7 and 3 – in that order.

Solution: 100 ENTER 64 ENTER 4.6 ENTER 7 ENTER 3 ENTER.

And here's how your stack should look:



This is the reliable general procedure anytime you want to put a series of numbers in your stack, right? You just key in each number and press ENTER to post it.

*If you already know how to do simple postfix arithmetic on the HP-28S, skip ahead to page 91.

Start Crunching: With your stack set up like that, press $\boxed{\times}$.

You should see this:

4:	100
3:	64
2:	4.6
1:	21

The numbers in Levels 1 and 2 were multiplied together, and the result was left at Level 1.

Try Another: Press $\boxed{+}$.

4:	-9.E-54
3:	100
2:	64
1:	25.6

The same thing happened – except that the result on Level 1 is now the sum of the (previous) bottom two levels.

Notice that, because two numbers were combined into one number here, there is one number fewer in the stack now, and the rest of the stack has therefore dropped one level. This is the way each *2-number* math command works. It takes the bottom *two* numbers from the stack, combines them, and puts the result back on the bottom of the stack.

Notice also that both addition and multiplication are *commutative operations*. That is, their results do not depend upon the order of the two numbers involved. Clearly, $1 + 2 = 2 + 1$; and $2 \times 3 = 3 \times 2$.

So for this addition, it wouldn't have mattered if the **4.6** had been above **21** or below it in the stack. This is *not* the case with other arithmetic operations, such as division and subtraction.

To Wit: Press $\boxed{\div}$.

4:	-7800
3:	-9.E-54
2:	100
1:	2.5

Notice that the order of evaluation is "Level 2 divided by Level 1."

Then Of Course: Press $\boxed{-}$. Here's the result – and now you know why, right?

4:	4.5E-24
3:	-7800
2:	-9.E-54
1:	97.5

Notice that throughout this little set of examples, all those other numbers you had floating around above Level 4 have successively made their reappearances. Your stack has been steadily "settling" downward as you perform these arithmetic operations that combine two numbers into one.

This settling is a very important part of the stack's operation. It becomes obvious with any problem that forces you to compute several intermediate results before combining them into a final answer.

For Example: Find $((2.4 \times 6.8) + (5.9 - 2.3) - (17.5 \div 4)) \times 43.2$

Solution:

Press

$\boxed{2} \boxed{\cdot} \boxed{4} \boxed{\text{ENTER}} \boxed{6} \boxed{\cdot} \boxed{8} \boxed{\times}$	(2.4×6.8)
$\boxed{5} \boxed{\cdot} \boxed{9} \boxed{\text{ENTER}} \boxed{2} \boxed{\cdot} \boxed{3} \boxed{-}$	$(5.9 - 2.3)$
$\boxed{1} \boxed{7} \boxed{\cdot} \boxed{5} \boxed{\text{ENTER}} \boxed{4} \boxed{\div}$	$(17.5 \div 4)$
$\boxed{-}$	$(5.9 - 2.3) - (17.5 \div 4)$
$\boxed{+}$	$(2.4 \times 6.8) + (5.9 - 2.3) - (17.5 \div 4)$
$\boxed{4} \boxed{3} \boxed{\cdot} \boxed{2} \boxed{\times}$	

The result is 671.544. Notice how you worked from the inner parentheses outward, thus eliminating the parentheses as you go. That harkens back even to your earliest days in arithmetic class, doesn't it?

Notice also how you combined the values in each parenthesized portion, "melting" them into intermediate results, which you allowed to "stack up" while you computed the next portion!

OK, that probably gives you some idea of the workings of the 2-number math functions. What about 1-number math functions?

You Asked For It: First, get rid of some of the extra numbers, by pressing **DROP** **DROP** **DROP** **DROP** **DROP** **DROP** **DROP** (that's 7 times).

Then press **■****x²** (the shifted **+** key). Here's what you should see at this point:

4:	
3:	
2:	5280
1:	133407.5625

x^2 is a 1-number function, since it takes only one number off the stack. Since it replaces that one number with its result, only Level 1 is affected.

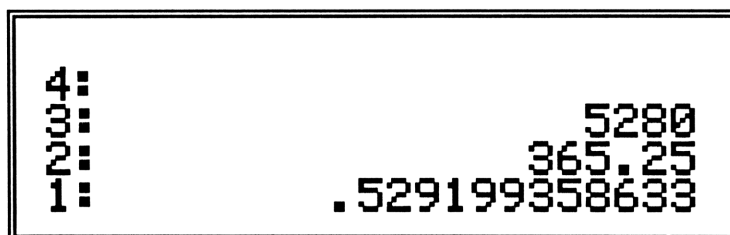
Affect It Again: Press **■****√x** (**■****-**). **■****√x** is also a 1-number function – and no prizes for guessing what operation you just did:

4:	
3:	
2:	5280
1:	365.25

So now you've seen 2- and 1- number operations.

Believe it or don't, there are even some *0-number* operations. That is, there are some operations that take nothing from the stack, but leave a value there nevertheless.

Pick A Card, Any Card: Press **R****A****N****D****ENTER**. You'll see something like this:*



4:	
3:	
2:	5280
1:	.529199358633

RAND is the RANDom number generator. It takes no numbers from the stack but leaves a random number there at Level 1, thus bumping everything else up by one level.

*Because the number on Level 1 has been chosen at random by the calculator, the number in your display may not be the same as the one shown above. Your calculator *should* generate this number the first time you invoke RAND after a machine reset (via **ON****INS****▶**).

Arithmetic Practice

Here are some not-so-trivial problems to let you practice your postfix arithmetic skills and some of those 2- and 1-number functions.

A reminder: You'll notice an abundance of parentheses here, since that's how you're used to seeing such problems expressed on paper. But there's no need for parenthesis keystrokes when solving these on your HP-28S. The way these numerical "memos" stack up on your "bulletin board" allows your arithmetic "staff members" to work with them without using any parentheses!

And keep in mind the rule of thumb for handling parenthetical expressions:

"Work from the innermost parentheses outward."

1. $((((1 + 2) \times 3) + 4) \times 5) = ?$

2. Calculate
$$\frac{-12 + \sqrt{12^2 - (4 \times 3 \times (-5))}}{2 \times 3}$$

(You might recognize this as one of two solutions to the quadratic equation, $ax^2 + bx + c = 0$, where $a = 3$, $b = 12$ and $c = -5$.)

3. $\sqrt[3]{7} = ?$

(This is the cube root of 7. Notice that there is no cube root command on the HP-28S and that you "raise-to-a power" via $\blacksquare \uparrow$. Notice also that $\sqrt[3]{7} = 7^{1/3}$.)

4. $173e^{\left[\frac{-16 + (43 \times 0.004)}{32 - 16.3} \right]} = ?$

(e^x is EXP from the LOGS menu.)

5. $1 + 0.5 + \frac{0.5^2}{2!} + \frac{0.5^3}{3!} + \frac{0.5^4}{4!} = ?$

(x! is the FACTorial of x. The command FACT lives in the REAL menu. In case you're curious, this problem is asking you to add the first five terms of the Taylor series approximation of $e^{0.5}$. You might want to compare your result here with the result of the HP-28S's EXPonential function.)

Arithmetic Practice Solutions

1. $\boxed{1} \boxed{\text{ENTER}} \boxed{2} \boxed{+} \boxed{3} \boxed{\times} \boxed{4} \boxed{+} \boxed{5} \boxed{\times}$

The result = **65**

2. $\boxed{1} \boxed{2} \boxed{\text{X}^2} \boxed{4} \boxed{\text{ENTER}} \boxed{3} \boxed{\times} \boxed{5} \boxed{\text{CHS}} \boxed{\times} \boxed{-} \boxed{\sqrt{\text{X}}} \boxed{1} \boxed{2} \boxed{\text{CHS}} \boxed{+} \boxed{2} \boxed{\text{ENTER}} \boxed{3} \boxed{\times} \boxed{\div}$

The result = **.38047614285**

3. $\boxed{7} \boxed{\text{ENTER}} \boxed{3} \boxed{1/\text{X}} \boxed{\wedge}$

The result = **1.91293118277**

4. $\boxed{4} \boxed{3} \boxed{\text{ENTER}} \boxed{\cdot} \boxed{0} \boxed{0} \boxed{4} \boxed{\times} \boxed{1} \boxed{6} \boxed{\text{CHS}} \boxed{+} \boxed{3} \boxed{\cdot} \boxed{2} \boxed{\text{ENTER}} \boxed{1} \boxed{6} \boxed{\cdot} \boxed{3} \boxed{-} \boxed{\div} \boxed{\text{LOGS}} \boxed{\text{EXP}} \boxed{1} \boxed{7} \boxed{3} \boxed{\times}$

The result = **579.135149585**

5. $\boxed{1} \boxed{\text{ENTER}} \boxed{\cdot} \boxed{5} \boxed{+} \boxed{\cdot} \boxed{5} \boxed{\text{X}^2} \boxed{2} \boxed{\text{REAL}} \boxed{\text{FACT}} \boxed{\div} \boxed{+} \boxed{\cdot} \boxed{5} \boxed{\text{ENTER}} \boxed{3} \boxed{\wedge} \boxed{3} \boxed{\text{FACT}} \boxed{\div} \boxed{+} \boxed{\cdot} \boxed{5} \boxed{\text{ENTER}} \boxed{4} \boxed{\wedge} \boxed{4} \boxed{\text{FACT}} \boxed{\div} \boxed{+}$

The result = **1.6484375**

$\boxed{\cdot} \boxed{5} \boxed{\text{LOGS}} \boxed{\text{EXP}} = 1.6487212707$

Of course, there's a whole lot more to this machine than just your basic arithmetic. You're surely itching to crack into all that (and why not? It's never any fun to hammer out the fundamentals before getting to the really good stuff). Nevertheless, there's quite a bit more hammering to do before you get into serious number-crunching. In fact, the only reason you're seeing arithmetic with *any* numbers right now is to learn about how the stack works. And there's a whole lot more to see before you're ready to manage your staff. This part of the course, then, is still a filling-in of the details of the everyday operations of your staff and bulletin board. After all, you need to *fully* construct this imaginary world in your mind before you can operate with its help.

Stack Operations

Here's a quick run-down of all the things you know so far about the stack.

- You've already been introduced to some of the stack's basic math operations.
- You know how `ENTER` and the command line are used to put things on the stack and how `DROP` is used to remove them.
- You've even seen how the operation called DUP can be used to duplicate the first level of the stack (remember way back on page 25?).

There are a lot more stack commands than just these 3, and since the stack is basically your work area, you'd better know your way around it.* The next few pages, therefore, are a continuation of your introduction to the HP-28S's stack commands. Don't expect to inscribe all of this in your mind or on your fingertips after a single pass here; it will take time until you're fluent in using all these different specialized tools. But take a look now, and begin your practice....

*But if, on an outside chance, you feel at home there already, then by all means, jump now to page 107.

ENTER's Second Job

Before you get to any new commands, take a second look at the hardest-working key of all: **ENTER**. Apart from posting (or evaluating) the command line, it has another use altogether!

This second use of **ENTER** is that, when the command line is not active, **ENTER** functions as a "do-it-now version" of DUP. This is another convenient extra, because most commands "eat" items off the stack; having back-up copies becomes important.

Try It: First, be sure there's no half-built command line (by pressing **ATTN** if necessary).

Key in a number and press **ENTER** several times. See how it duplicates that number?

Now DROP all those duplicate entries out by pressing **DROP** several times.

The **SWAP** Command

Another commonly used stack operation is **SWAP**. It functions to exchange the contents of Levels 1 and 2.

So what good is that? Well, remember that division and subtraction are not commutative operations; their operation depends on the order of the numbers in Levels 1 and 2.

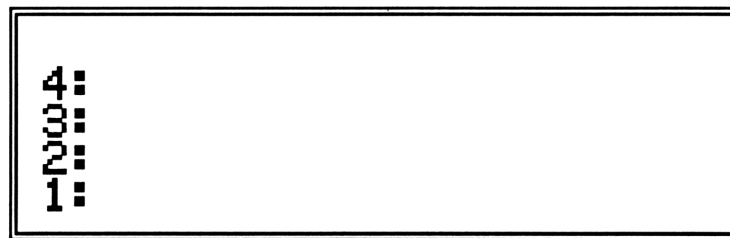
SWAP gives you the ability to reverse the order of these two stack levels, and because there are many operations (besides $-$ and \div) that are *not* commutative, **SWAP** becomes very much in demand.

How to **CLEAR** the Stack

Remember **DROP** – that command that throws away the bottom (most recent) memo and therefore "drops" all the rest of the memos down one level?

Well, **CLEAR**, like **DROP**, is a stack clean-up command. If there are items on the stack that you don't need any more, **CLEAR** removes them. But you'd better be sure about what you're doing: **CLEAR** clears the *whole* stack.

Clean Your Slate: Press **CLEAR** now – and see this:



Up to now, the stack commands you've been using are all important enough to have keys of their own.

But everyone has to live someplace; the less commonly used stack commands live in the STACK menu (■G).

Get That Menu: Press ■STACK. You should now see this:



You see that DUP lives here. But why? If ENTER does the same thing, and it's sitting right there on the keyboard, why put DUP in a menu also?

Well, recalling that ENTER is exempt from the effects of α, if you wanted DUP to appear on your command line, you'd have to type DUP manually (perish the thought) unless it were available on a menu key.

All this planning – just for your convenience!

Continue your perusal of this STACK menu. Reading from left to right in the menu, the next command over is OVER. OVER makes a copy of whatever is in Level 2 and then pushes this copy onto the stack (to "push" something onto the stack means to put it on at Level 1, thus bumping everything else up one level). In effect, then, OVER makes a copy of Level 2, jumps *over* the current Level 1 and pushes the copy on the stack "beneath" it.

Drum Roll, Please: Press **1** **2** **OVER**. And here's the result:



See how this works? Remember, although you never pressed **ENTER** to put the **1** and **2** on the stack in the first place, OVER is an immediate-execution function, which means that, for all practical purposes, you had a command line that read **1, 2, OVER** before an **ENTER** was "caused" by the OVER. Since the HP-28S posts (or obeys) memos from left to right, this explains how the numbers got onto the stack by the time the OVER happened!

Horse around with this some more, but after you've finished, set up your stack so that it looks like this (from the above example, all you would need to do is press **DROP** **3** **ENTER** or **OVER** **+**):



Now then: Just to keep things interesting, skip over to the fifth item in the STACK menu – to the command called ROT.

This command ROTates the bottom three levels of the stack "upward." In effect, Level 3 is *removed* (not copied) and then pushed onto the stack.

Prove It: Press **ROT**. The result:



See? Just a simple rotation of the three bottom-most "things" in the stack. Now press **ROT** twice more to return the stack to its original order.

And Just For Laughs: Press **SWAP** **ROT**.

What happened? The bottom three levels of the stack have been reversed.



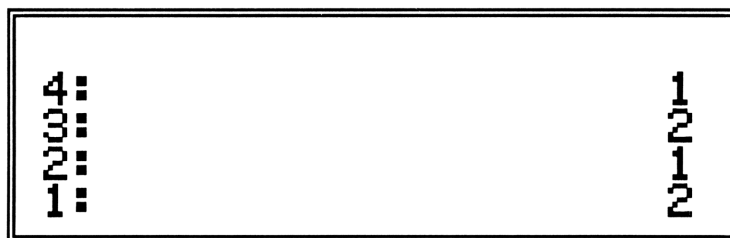
(Now press **CLEAR** to shred all evidence of levity.)

Going back now to pick up those two commands you skipped:

DUP2 and DROP2 are analogous to DUP and DROP but – as their names imply – *they operate on both the first and second levels at the same time.*

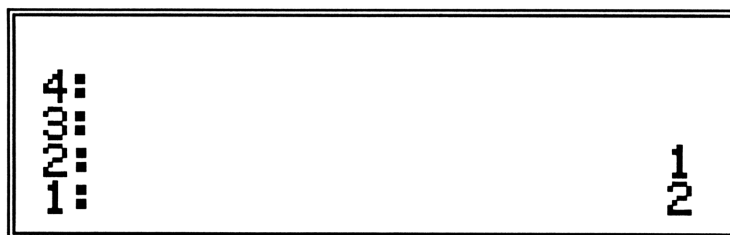
DUP2 makes a copy of both the first and the second levels and then pushes them on the stack. That is, it DUPlicates the contents *and ordering* of the bottom 2 levels of the stack.

Watch: Press **1** **2** **DUPE**. Then press **↔** to get the menu out of the way for a minute so that you can see the bottom four levels of the stack:



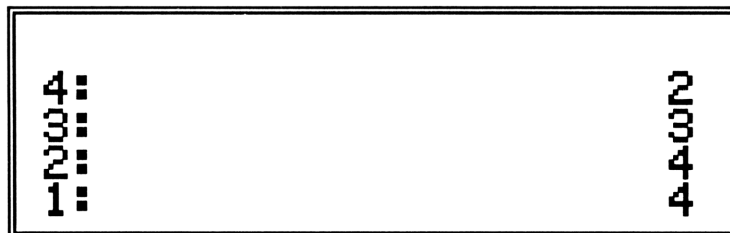
And DROP2 drops (discards) the bottom two levels of the stack,

Like So: Press **↔** **DROP2** **↔**.

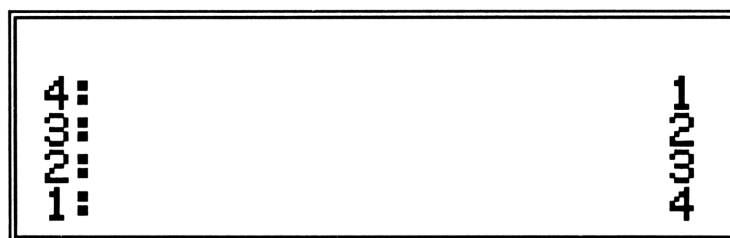


No sweat, right?

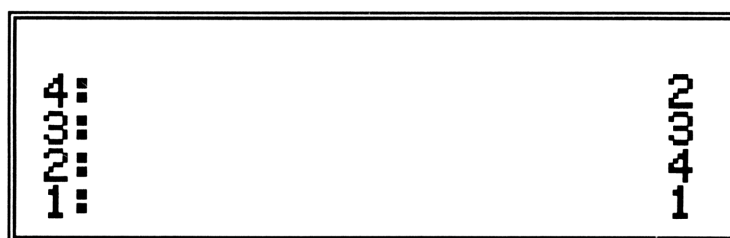
ROLL is a do-it-now function, so it has a built-in **ENTER** after it. Therefore, the argument, 4, that you just keyed in *does* go onto the stack at Level 1 *before* the ROLL gets under way (so keep in mind there's a 1 up on Level 5):



Next, your memo posting "office boy" obeys the do-it-now ROLL, which says, in effect, "DROP (throw away) that bottom **4**, but *note its value* on the way to the dumpster." He obeys, and so the stack (bulletin board) looks like this once again:



He now climbs up the stack to Level 4 (because he just threw away a **4**), removes whatever's there (a **1**) and pushes it back onto the stack at the bottom:



As you get more practice, of course, you won't even need to think about all these intermediate steps. After awhile, it'll feel obvious that if you want the 4th "thing" in the stack to "come on down," you just key in a 4 and use ROLL.

The main point of this backstage tour is that ROLL is a good example of a one-argument, postfix operation.

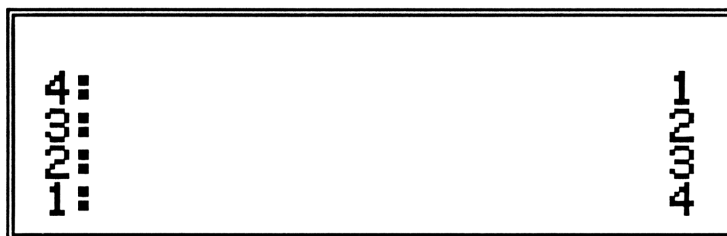
It's postfix because whatever it operates on (and with) had better be in position by the time it comes along – and this is indeed the case: everything is sitting on the stack exactly right – so that it does what you want it to.

And it's a one-argument function because it needs one parameter ("argument" – remember your staff's math jargon?) *in addition to the stack's current contents* to tell it the "where's" and "how-many's" of its operation.

The way in which ROLL simultaneously uses and discards its argument is very typical of the HP-28S's treatment of arguments that you load into the stack. It notes them while throwing them away (DROPPing them).

Another good example of this is ROLL's twin sister, ROLLD (in the STACK menu). ROLLD is the reverse of ROLL. It ROLLs the stack Down in the same way that ROLL rolls it up – sending what's in Level 1 up to the specified level.

Going The Other Way: Press **4** **↔** **ROLLD** **↔**. Thus, the stack is returned to the order it had before you executed the ROLL:



And you know the reasons for the parameter, 4, and the messing about with the **↔** key, right?

Onward and upward to more strange and wonderful stack manipulation stuff (on the second "page" of the STACK menu)...

DUPN and DROPN are generalizations on DUP and DROP in the same way that ROLL is a generalization of ROT – including the treatment of the one parameter. Both commands first DROP a number off the stack and use it to tell the number of levels on which to operate.

For Example: There are now four items on the stack. Press `4` `↔` `DUPN` `↔`.

Voila! There are now eight levels on the stack. The bottom four levels have been duplicated and then pushed back onto the (bottom of the) stack.

Press `DROP` four times to prove this to yourself....

Thus, `1` `DUPN` is the equivalent of `DUP`, and `2` `DUPN` is the equivalent of `DUP2`.

Likewise: DROPN will *remove* the specified number of levels. Press `4` `↔` `DROPN`. All four of the remaining levels are dropped (nuthin' left)!

Thus, `1` `DROPN` is equivalent to `DROP`, and `2` `DROPN` is equivalent to `DROP2`.

"But wait – there's more!"

PICK (same menu) is a generalization of OVER. It drops an argument from the bottom of the stack, using its value to count up the stack. It then makes a *copy* of that level (unlike ROLL, which extracts that entry altogether) and pushes this copy onto the stack.

Thus, **2 PICK** is equivalent to **OVER**, and **1 PICK** is equivalent to **DUP**.

And more: DEPTH is a command that takes nothing from the stack. It merely counts the number of levels currently on the stack and then pushes its resulting count onto the stack (as the new bottom number, of course).

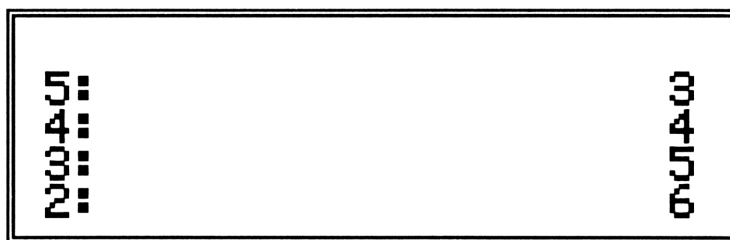
Do This: Be sure the stack is clear, then press **DEPTH**, and it returns a **0** – no mystery, right? The stack was empty.

Press it again and it returns a **1**. (Why ?)

Would you believe...more? **VIEW↑** (**CHS**) and **VIEW↓** (**EEX**) don't actually change the stack in any way. As their names imply, they allow you simply to *view* portions of the stack.

Take A Look: Press **CLEAR** **1** **'** **2** **'** **3** **'** **4** **'** **5** **'** **6** **'** **7** **ENTER** **↔**.

Now press **VIEW↑**. What happens?



The whole stack was *scrolled* (not rolled) down. You can now see what was beyond the top line of the display.

Pressing **VIEW↑** again will move the view up one more line. Pressing and holding **VIEW↑** will scroll until the top line of the stack is visible. **VIEW↓** has the opposite effect.

One more thing: **VISIT** (**'**) is just like **EDIT**, except that it uses the number in Level 1 as an argument – DROPping it and noting it in the usual manner – to select the stack level to be edited.

Then, in terms of actual editing, **VISIT** is exactly like **EDIT**, except that when you press **ENTER** the altered contents of the stack level you've been editing are placed back at that level – not at Level 1 (and you would expect this, right?). Try **VISIT**-ing various stack levels, just to get the hang of it.

Strenuous But Practical Stack Practice Problems*

Solve the following as efficiently and expertly as you now know how:

1. $\left(\frac{2}{7} - \frac{14}{15}\right) + \left(\frac{14}{15} - \frac{2}{7}\right) = ?$

2. Assume the bottom three levels of the stack contain the coefficients of a quadratic equation (i.e., $ax^2 + bx + c = 0$), where Level 3 is a , Level 2 is b , and Level 1 is c . Give a sequence of key-strokes that will produce the equation's lesser root, whose formula is

$$\frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

3. You've seen (on page 99) how you might reverse the bottom three stack levels. What are some key-strokes that will reverse the bottom *four* stack levels? How about reversing Levels 2, 3 and 4?
4. Swap Levels 1 and 2 for Levels 3 and 4. Then swap Levels 1, 2 and 3 for Levels 4, 5 and 6. How about swapping Levels 1 and 4?
5. $(((((12.45 + 3) \times 12.45 + 3) \times 12.45 + 3) \times 12.45 + 3) \times 12.45 + 3) \times 12.45 + 3 = ?$
6. Who played Elliot Ness in the television series "The Untouchables?"

*Try saying *that* three times in a row.

S. B. P. S. P. P. Solutions

1. **2** **ENTER** **7** **÷**
1 **4** **ENTER** **1** **5** **÷**
STACK **DUP2**
SWAP **−**
ROT **ROT** **−** **+**

(2/7 in Level 1)
 (2/7 in Level 2; 14/15 in Level 1)
 (Copy both values)
 (14/15 − 2/7)

Result = $\frac{1}{15}$

2. One solution:

STACK **ROT** **2** **×**
DUP **ROT** **×**
2 **×**
3 **NEXT** **PICK**
X²
SWAP **−** **√x**
NEXT **ROT** **CHS** **SWAP** **−** **SWAP** **÷**

(Stack = b; c; 2a, in descending order)
 (Stack = b; 2a; 2ac)
 (Stack = b; 2a; 4ac)
 (Stack = b; 2a; 4ac; b)
 (Stack = b; 2a; 4ac; b²)
 (Stack = b; 2a; $\sqrt{(b^2 - 4ac)}$)

3. **STACK** **SWAP** **ROT** **4** **ROLL**;
STACK **4** **NEXT** **ROLL** **SWAP** **NEXT**
ROT **4** **ROLL**

4. 4 **ROLL** 4 **ROLL**;
 6 **ROLL** 6 **ROLL** 6 **ROLL**;
 4 **ROLL** **SWAP** 4 **STACK** **NEXT** **ROLLD** or
 4 **STACK** **NEXT** **ROLLD** **NEXT** **ROT**

5. 1 2 • 4 5 **ENTER** **ENTER** **ENTER** **ENTER** **ENTER** **ENTER** (Six copies.)

3 **ENTER** 1 **ENTER**

MODE **NEXT** **CMO**

(Enable **COMMAND**.)

STACK **α** **ROT** **×** **OVER** **+** **ENTER**

(Remember these keystrokes.)

COMMAND **ENTER**

COMMAND **ENTER**

COMMAND **ENTER**

COMMAND **ENTER**

COMMAND **ENTER**

(Do them five times.)

Result = 4699789.91278


See how easily you can repeat any given set of keystrokes?

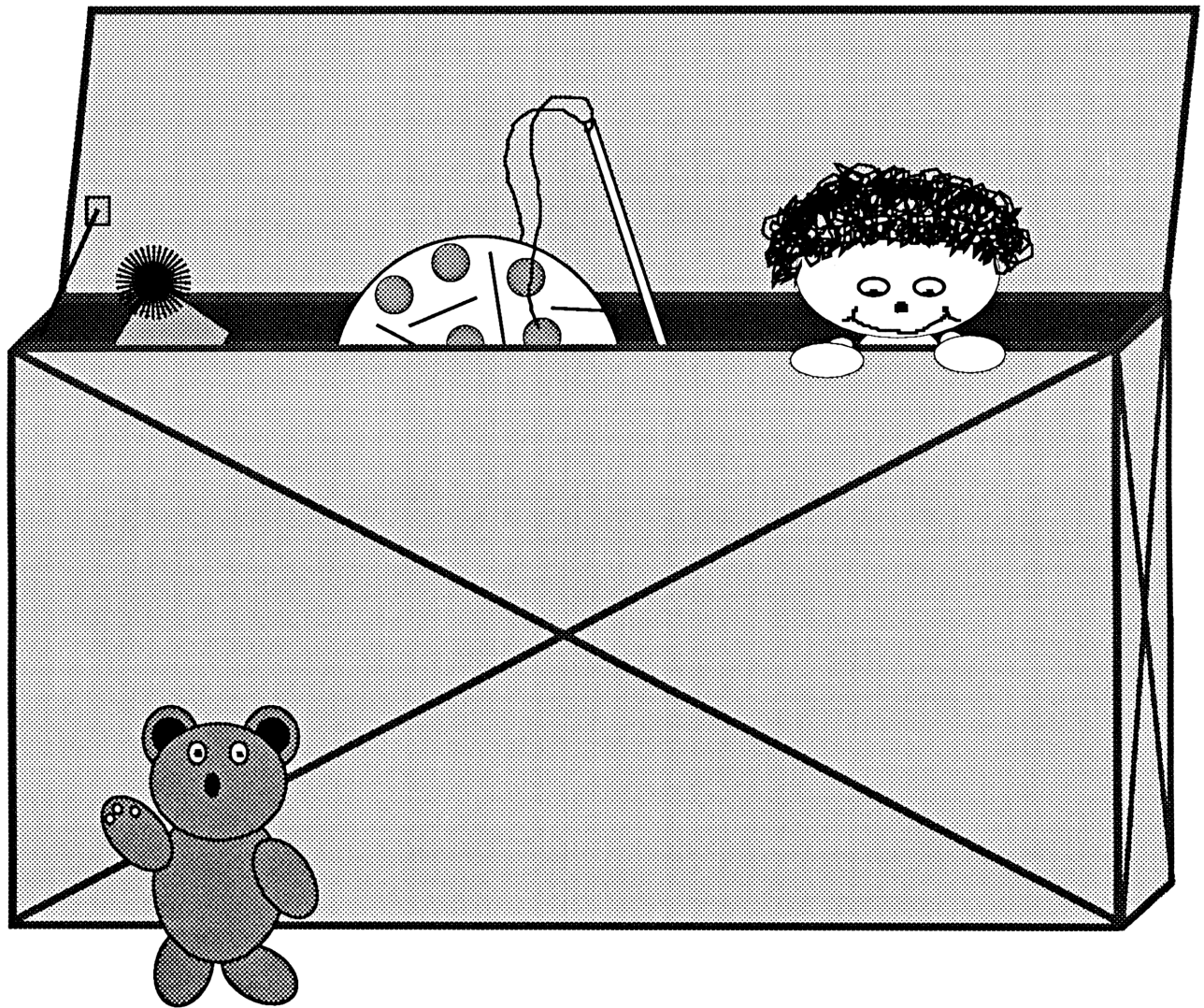
6. Robert Stack.



This is it. You've fully constructed your office surroundings in your mind. Now you're ready to learn how to work with all the high-powered brains on your HP-28S staff.

Looking back for a moment, you can see how far you've come as "president" of this "collection of mathematicians" called the HP-28S....

- You know how the keyboard connects to the "bulletin board" display, and you have a basic understanding of the layout of the keyboard – how the character and command keys are arranged and how the  key serves to change the meanings of most of the keys (changes to the red-printed functions).
- You know how to use the command line to edit and post "memos" or commands on the bulletin board. You know that you have a "card catalog" of reserved words (menu items) that the HP-28S will recognize as commands rather than simple messages. And you know that many of these are "do-it-now" commands, so they won't be posted onto the stack but rather, executed immediately (unless you have switched to α mode).
- You know how arithmetic works on the HP-28S, with its postfix logic, where both operands precede the operator. And you know about 0-, 1-, and 2- number arithmetic operators.
- You know several gadzillion charming and convenient stack-manipulation commands – ways to rearrange the numbers in the stack.
- You're familiar enough with the HP-28S that you can now dispense with this "office world" mind game and instead see this machine for what it really is – just a mindless but powerful calculator that will obey your commands.



**The "Stuff" Upon Which
The HP-28S Works**

An Equal Opportunity Calculator

Unlike most calculators, the HP-28S is not limited to working only with real numbers. Though real numbers are tremendously useful, and much real-world work involves their manipulation, you might also want more flexibility in your problem solving – the ability to manipulate other sorts of information, too.

Unfortunately, with increased flexibility comes increased complexity (you don't get somethin' fer nuthin'). With all these new sorts of information and new ways of manipulating them, there comes a whole slew of new rules – and new exceptions to those rules.

Fortunately, however, there is an underlying, unifying logic to how things work in the HP-28S. Once you get a grip on this general operating scheme, you should be able to move from manipulating one sort of information to another without much discomfort.

How can that be? How can you treat characters, for instance, in *any* way similarly to the way you treat real numbers?

Be assured, you can. Though you must remember certain details for certain types of information, *you will find that the machine treats most every "thing" in very much the same way!*

The HP-28S's Philosophy of Information

Despite all evidence to the contrary, there are really only three basic kinds of information that the HP-28S understands. They are:

Real Numbers
Characters
Bits (short for Binary digits).

This is because these three basic information types are so very useful, forming the backbone of almost all information processing that goes on in the real world.

But that isn't nearly the whole story.

Though these three information types are used almost universally, *they are almost never used just as they are* (the major exception being real numbers).

Specifically, **characters** are more commonly used as elements of character *strings* or pages of *text*.

Bits are more often grouped into *bytes* and *words*, or used as *binary integers*.

Even **real numbers** are often grouped into *vectors*, *arrays*, *tables*, and other more exotic "things."

The good news is this: HP-28S has anticipated your need for such complex groupings of these three simple information types. Built right into this calculator is the capability to allow you to build and manipulate compound objects in familiar, useful and convenient ways!

Real Numbers

You've already had an introduction to real numbers – what they are and some of the things you can do with them. *Now you must begin to look at them as part of the larger scheme of things.*

Real numbers are "things," *objects* – one specific *kind* of information. Sure, they're somewhat familiar to you because you often use them to solve problems in your daily life – including the problem of how to introduce yourself to the HP-28S's stack and arithmetic.

But stop and take a good, hard look at that. Exactly how have you used real numbers in this Course so far?

Two ways:

1. The first way is probably so familiar to you that you didn't even notice it. You used real numbers as *object information*. That is, they were data – information for its own sake – to be manipulated in order to get other numeric information.

This is the way that you look at real numbers when you do things like addition, subtraction, multiplication and division. *You are working on numbers that are meaningful to you in order to get another number that is meaningful to you.*

2. You have also seen numbers used not as data, but as parts of commands. For example, ROLL doesn't use the Level-1 stack value as data. Rather, it uses the number as an indicator of how it is supposed to work. In other words, *real numbers used in this way are meaningful to you only because they help you control the machine.*

3. And here's yet a third way to use real number – a way you've not yet encountered in this book: use them as truth values.

That is, use two different real numbers (conventionally 0 and 1) to represent opposing states or responses (e.g., yes or no, on or off, set or clear). In this sense, then, a real number can be used to represent qualitative information.

For example, if you're comparing two real numbers to see if the first is greater than the second, you would use the > command.

Try It: Press `2``ENTER` `3``ENTER` `█``>``ENTER`. The result is `0`. Then press `3``ENTER` `2``ENTER` `█``>``ENTER`. The result is `1`.

To the HP-28S in this context, `0` means that the comparison is false (i.e. the answer is "no, 2 is not greater than 3"). Conversely, a `1` as a result means that the comparison is true.

All such comparison operations (<, >, ==, SAME, ≤, ≥, ≠) will return either a one or a zero depending upon whether the result is true or false.

You can begin to see that a number or any other type of information is actually quite meaningless unto itself. *It gains meaning based on how it is used.* That's a basic concept – an underlying truth – of the HP-28S.

You've also seen another underlying truth of the HP-28S – the stack. It may have been a rather new idea to you. It does present some new problems and new ways of doing things, but it also opens up many new possibilities – new and powerful concepts, such as the postfix operation you've already begun to explore.

So it's time to begin exploring these different uses of information, seeing first-hand just how the HP-28S builds upon the three fundamental information types, creates other information "objects" and combines them on its stack....

Complex Numbers

A good starting point: With real numbers as your building blocks, the simplest compound-data-type you can build is the *complex number*.

As an information structure, a complex number is simply an *ordered pairing* of real numbers – a *list* with two real-number *elements*. The first element of the pair is considered the *real* part of the number; the second is called the *imaginary* part.



On the HP-28S, complex numbers are represented by bracketing two real numbers between parentheses, separating the two numbers with a delimiter.

Build One: Press **MODE** **STO** **↔** **CLEAR**.

Then press **(** **2** **SPACE** **3** **.** **4** **)** **ENTER**. This is what you'll see:




4:
3:
2:
1: (2, 3.4)

Notice these things:

1. You didn't need to press  at the end of the number. If the complex number has other things following it in the command line, then you must use  as a delimiter. If not, the HP-28S will automatically add the right-hand parenthesis before posting the number.
2. The space you used as a delimiter in this example was replaced by a comma when you posted it. Regardless of the delimiter you use when keying in the complex number, the HP-28S will use a comma (unless you're using the comma as the radix mark, in which case it will use a period*) in the posted form.
3. The real and imaginary portions of the number are on the *same stack level*. Both this and the fact that they are grouped together inside parentheses tells you clearly that *this is a single object – one number*. From now on, unless you purposely break it into its components, it will be treated as one number – one object.

That's all there is to it. That's how you key in and post any complex number "object."

Now, what good is it?...

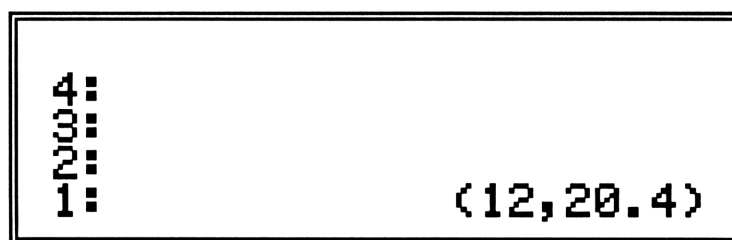
*The choice of radix is a selectable *mode*. The commands used to alter such HP-28S modes are found in the MODE menu (). RDX, controls the radix and is on the second "page" of the mode menu.  indicates that the radix is the period while  is the comma. Pressing this key toggles between the two modes.

Complex numbers have, by definition, mathematical properties quite similar to those of real numbers. Thus, many HP-28S operations meaningful for real numbers are also meaningful for complex numbers.

You can, for example, perform arithmetic operations (+, -, \times , \div , etc.), trigonometric functions (SIN, ASIN, COS, ACOS, etc.) and logarithmic functions (LOG, ALOG, LN, EXP, etc.) on complex numbers.

And in all these operations you can perform arithmetic using a mixture of complex and real numbers. The real number is converted by the machine into a complex number (with a 0 imaginary part) before the operation commences. The result is always complex.

Watch: Press `ENTER` `5` `⊗` `+`.



Pretty slick, right? With those few keystrokes – and the stack logic you now know all about – you multiplied a real number by a complex number, then added the result to a complex number.

So you can see right away that one major advantage of a compound data type such as a complex number is that the components of the object are manipulated together as a unit in new and meaningful ways – and you don't have to expend any energy trying to keep track of all the parts!

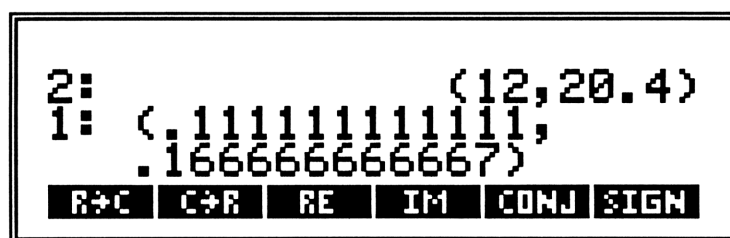
However, another (complementary) advantage is that you can pull apart the complex number into its component pieces, do things to them individually, and then reassemble them into a complex number. In fact, you can build a complex number from pieces generated by other, unrelated operations.

So in addition to the strictly mathematical operations in your complex-number repertoire – operations they share with the real numbers – you have other operations designed *specifically* for complex numbers.

Any guesses as to where these type-specific operations might "live?" In the COMPLX (■C) Menu, of course!

Try This: Press **1****ENTER****9****÷** **7****ENTER****4****2****÷**.

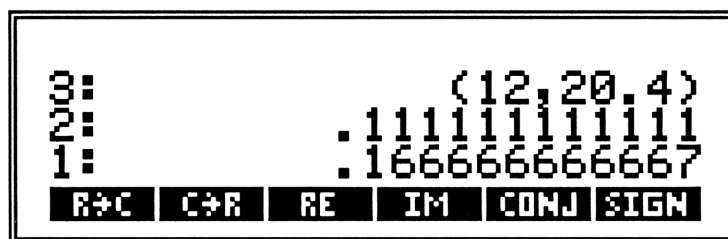
Then **■****COMPLX** **■R→C** ("Real to Complex"). Here's what you get:



R→C allows you to construct a complex number from two real numbers that are in stack Levels 1 and 2. The real portion is taken from Level 2; the imaginary portion from Level 1.

As you might expect, the function C→R (also in this menu) allows you to go the other way – decompose a complex number into two real numbers – where the real portion goes to Level 2 and the imaginary portion to Level 1.

Now Try This: Press **C \leftrightarrow R**. You see:

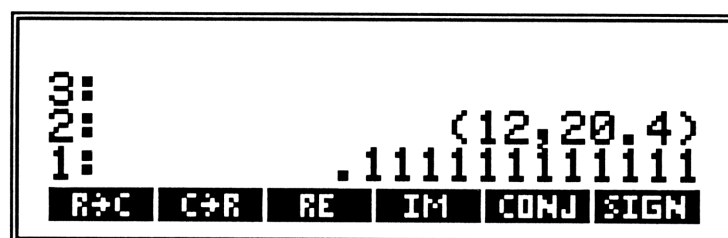


```
3:                               (12,20.4)
2:                               .111111111111
1:                               .1666666666667
R+C  C+R  RE  IM  CONJ  SIGN
```

You've decomposed the complex number that was at Level 1 into its two components. The real part is now at Level 2; the imaginary part is at Level 1.

Next, press **R \leftrightarrow C** to recombine those two parts once again. Simple, right?

OK: Press **RE**.



```
3:                               (12,20.4)
2:                               .1666666666667
1:                               .111111111111
R+C  C+R  RE  IM  CONJ  SIGN
```

Now what have you gone and done?

Nothing too awful: **RE** replaces the complex number on Level 1 of the stack with only that number's **REal** component. **IM** does the same with a complex number's **IMaginary** component.

Here's a point to help you wrap your mind around the HP-28S's idea of data objects:

Although complex numbers are a meaningful form of information and there are plenty of operations that use them as such, you're *not limited to using them just as complex numbers*.

You can also use complex numbers as two-dimensional vectors (which are very similar to complex numbers), or as coördinate pairs in plotting, or just as a convenient way to group two related numbers.

And because you can pull complex numbers apart and put them back together again, you can actually define new operations that give new meanings to the complex number *data type* in the HP-28S but which have no relationship to the mathematical concept of complex numbers in the real world.

As far as you should be concerned, then, complex numbers are just ordered pairs of real numbers – *information objects for you to use in whatever way you see fit*.

And just so you're fully confident of your skills with the mechanics of complex numbers, here's a "little something for the occasion..."

Simple Questions About Complex Numbers

1. How would the HP-28S represent these complex numbers?

$$3 + 4i \quad 2.3 - 1.1i \quad -1 - i \quad \frac{7}{8} + \frac{4}{3}i$$

2. How would you key in the numbers in question 1?

3. Calculate the following: $\sqrt{-1}$, $\sqrt[3]{-7}$, $(1,1)^2$, $\sin^{-1}(2)$, $\ln(-1)$.

4. Change (1,2) into (2,1).

5. Calculate the following: $(0.34(2 + 3i)(32.4 - 12.2i)) \div (33.42 - (12.2 + i\sqrt{2}))$.

6. Consider the two-dimensional vector (3,4). What is its magnitude and direction? What unit vector has the same direction? Change the sign of the original vector and answer the same three questions.

7. The heights and circumferences of a set of trees are as follows: 18 feet and 2 feet; 25 ft and 1 foot, 11 inches; 28 feet and 2 feet, 5 inches. What are the mean height and circumference of these trees (use complex number objects).

Simple Answers to Simple Questions About Complex Numbers

1. $\langle 3, 4 \rangle$;
 $\langle 2.3, -1.1 \rangle$;
 $\langle -1, -1 \rangle$;
 $\langle .875, 1.333333333333 \rangle$.

2. (3 , 4 ENTER
 (2 . 3 SPACE 1 . 1 CHS ENTER
 (1 , 1 ENTER CHS
 7 ENTER 8 ÷ 4 ENTER 3 ÷ COMPLEX R↔C

3. 1 CHS \sqrt{x} = $\langle 0, 1 \rangle$
 7 CHS ENTER 3 $1/x$ ^ = $\langle .956465591387, 1.65664699997 \rangle$
 (1 , 1 x^2 = $\langle 0, 2 \rangle$
 2 TRIG $\pi \div IN$ = $\langle 1.57079632679, -1.31695789692 \rangle$
 1 CHS LOGS LN = $\langle 0, 3.14159265359 \rangle$

4. (1 , 2 ENTER COMPLEX C↔R SWAP R↔C or
 (1 , 2 ENTER (1 , 1 CHS +

5. . 3 4 ENTER (2 , 3 X (3 2 . 4 , 1 2 . 2 CHS X
 3 3 . 4 2 ENTER 1 2 . 2 ENTER 2 \sqrt{x} COMPLEX R↔C - ÷

Result = $\langle 1.54011490435, 1.2690881897 \rangle$

$$6. (3, 4) \text{ [COMPLX] [NEXT] [R}\rightarrow\text{P]} = (5, 53.1301023542).$$

R→P converts the Cartesian (rectangular) coordinates to polar coordinates. Thus, the "real" part is the magnitude while the "imaginary" part is the angle. Alternately:

$$(3, 4) \text{ [ENTER] [ENTER] [COMPLX] [NEXT] [ABS]} = 5$$

$$\text{[SWAP] [ARG]} = 53.1301023542$$

ABS gives the magnitude. ARG gives the angle. The unit vector is given by SIGN:

$$(3, 4) \text{ [COMPLX] [SIGN]} = (.6, .8).$$

Notice that the unit vector multiplied by the original magnitude (5) does return the original vector. Also, the magnitude (ABS) of the unit vector is 1.

Using the negative vector:

$$(3, 4) \text{ [ENTER] [CHS] [COMPLX] [NEXT] [R}\rightarrow\text{P]} = (5, -126.869897646).$$

The magnitude is the same (of course), but the angle is different by 180°.

$$\text{[P}\rightarrow\text{R] [PREV] [SIGN]} = (-.600000000002, -.799999999998).$$

The resulting vector points in the opposite direction (i.e., 180° different) from the unit vector for (3,4). Although it is not exact, the unit vector is effectively (-.6, -.8). This is because R→P could not generate the exact angle (within 12 digits) so P→R could not return the exact original vector. Compare this result with that of the SIGN of (-3, -4).

7. (1 8 , 2 ENTER
 2 5 ENTER 1 ENTER 1 1 ENTER 1 2 ÷ + COMPLEX R↔C
 2 8 ENTER 2 ENTER 5 ENTER 1 2 ÷ + R↔C

The three data pairs are on the stack where each complex number object is in the form (height, circumference). The circumferences have been converted from feet and inches to just feet. To find the means, sum the components and divide by the number of components.

(+)(+) 3 ÷ = (23.6666666667, 2.1111111111)

This is the mean height and mean circumference in decimal feet. To convert each to feet and inches:

C↔R REAL NEXT NEXT
 ENTER IP SWAP FP 1 2 X
 ROT ENTER ENTER IP SWAP FP 1 2 X
 ⇄

Result:

4:		2
3:	1.3333333333	2
2:		23
1:	8.0000000000	4

The mean height is in Levels 2 and 1 (23 feet, 8 inches) and the mean circumference is in levels 4 and 3 (2 feet, 1 1/3 inches). As you might have surmised, IP gets the Integer Portion of a real number while FP gets the Fractional Portion.)

Vectors

Put most succinctly, a vector is an ordered list of numbers. It's ordered such that the left-most element is numbered 1, and the rest follow in ascending order.

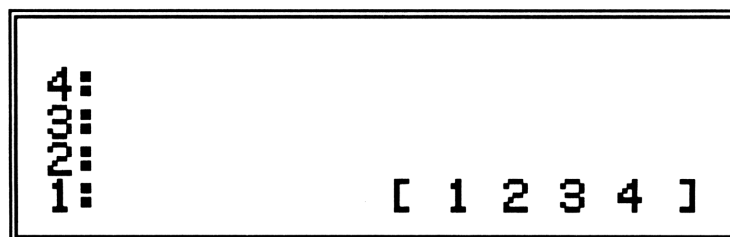
You're probably familiar with two- and three-element vectors, but the HP-28S imposes no upper limit on the size of a vector. Vectors are represented by the HP-28S as a list of numbers, separated by delimiters and enclosed within *square* brackets ([and]).

And here's a key point: The number elements of a vector may be real or complex (aha)! You can see already how to build a compound data type from simple data types *and other compound types*.




Better to start simply, though, so

Watch This: Press   1  2  3  4  .

You'll see:



Again, notice certain things:

1. You didn't need to press   at the end of the number. If the vector needs to be separated from other things following it in the command line, you must use  as a delimiter. If not, the HP-28S will automatically add the right-hand square bracket (and these rules should sound quite familiar, since you just heard similar ones for complex number objects).
2. The commas you used as delimiters were replaced by spaces. Regardless of the delimiters you use to enter the vector, the HP-28S will use spaces to display the vector (also a familiar rule).
3. All elements of the vector are on the same stack level. Both this and the fact that they're grouped together inside square brackets tell you that this is a single object. From now on, unless you purposely break it into its components it will be treated as one object.

By definition, vectors may be combined via certain mathematical operations, but unlike complex numbers, only a few of these are in any way analogous to corresponding operations on real numbers (operations such as +, -, x, ÷, ABS, and NEG).

For the most part, then, you have *vector-specific* operations, collected under the ARRAY (■A) menu. Why not a VECTOR menu? Because vectors are really examples of an object type called an *array*.

So an array is an ordered arrangement of objects, and a vector is just one particular arrangement – a one-row arrangement.

Another Vector: Press [((1,2) (3,4) (5,6) (7,8)] [ENTER].

The calculator display shows a vector of four complex numbers. The first three are displayed as [1 2 3 4] and the last one as [(1,2) (3,4) (5,6) ...].

The usual, little things to notice:

1. Since something follows each complex number in this vector (except the last one), you need to use the right-hand parenthesis to delimit each.
2. Neither the last complex number nor the vector needs the final parenthesis or bracket. Since it found both an opening bracket and parenthesis, the HP-28S knows it must close them at [ENTER].

Notice that the vector is longer than the display can hold. When this happens, the HP-28S will use an ellipsis (...) to indicate the over-run – just as you saw with the command line (page 54).

To view the whole vector, you must do one of three things:

- (i) Edit Level 1 (with either EDIT or VISIT), using the cursor keys to scroll through the entire object. Then press **ATTN** to discard the command line so that you don't inadvertently change the vector;
- (ii) Decompose the object with **ARRAY→** from the **ARRAY** menu, and if necessary use **VIEW↑** and **VIEW↓** to examine the elements in the stack. Remember to rebuild the vector with **→ARRAY** when you are done;
- (iii) Use **GETI** to step through the vector components.

No sense exploring (i). You already know how to EDIT.

Try Door Number (ii): Press **ARRAY** **ARRAY→**. Here's what you get:




The object in Level 1 is something new called a *list*. You can tell that this is a different object type because it's represented within braces (`{` and `}`). A list is a different animal altogether, so you won't see it in all its glory until later. For right now, just learn to recognize it by its braces.

You can see that the numbers up farther in the stack are the former values within the vector you just dissected. But what's that bottom number inside the braces?

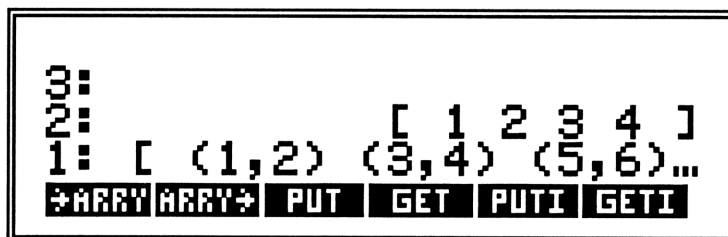
It's the number of elements the original vector contained: 4 (complex) numbers, in this case. `ARRY→` will always put this extra value on Level 1, so that you'll know how many stack levels contain vector elements *and* so that the machine will know this too (the array-building command, `→ARRY`, needs to know how many stack levels to use in making the new vector).

Now, use  `VIEW↑` and  `VIEW↓` to scan the stack.

The vector has been decomposed from left to right (like the complex number with `C→R`); the first element (the left-most) has been put on the stack first and the last vector element (the right-most) last.

Notice that you can modify any single vector element by using  `VISIT` to edit that element right where it is (see page 106).

Rebuild: Press **→ARRY**.

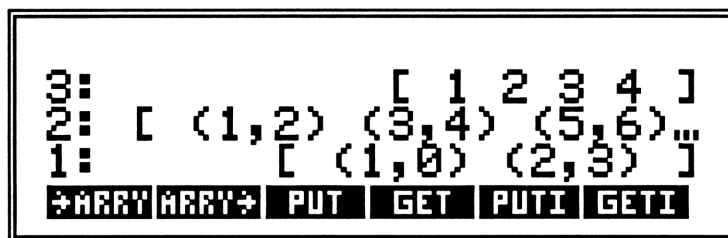


A calculator screen showing a vector rebuild operation. The display shows three levels of a stack: Level 3 contains the number 3; Level 2 contains the number 2; Level 1 contains a list of four coordinate pairs: (1,2), (3,4), (5,6), and an ellipsis. Below the stack, the sequence of operations performed is shown: →ARRY, →ARRY, →, PUT, GET, PUTI, and GETI.

The vector has been recomposed from its pieces. The **→ARRY** operation used the **{ 4 }** to know how many stack elements to put into the vector.

Note that you could have filled the stack with real and/or complex numbers of your own, put the vector size in Level 1 and used **→ARRY** to build an entirely different vector.

Like This: Press **1, (2, 3 ENTER { 2 →ARRY**.



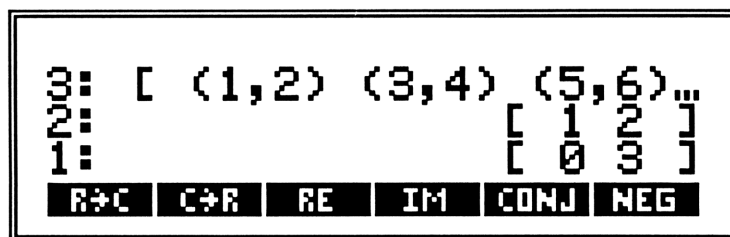
A calculator screen showing a vector rebuild operation with different data. The display shows three levels of a stack: Level 3 contains the number 3; Level 2 contains the number 2; Level 1 contains a list of two coordinate pairs: (1,0) and (2,3), followed by an ellipsis. Below the stack, the sequence of operations performed is shown: →ARRY, →ARRY, →, PUT, GET, PUTI, and GETI.

What happened? First of all, the real value 1 was changed to the complex value $(1, 0)$, which is its complex equivalent (remember that complex numbers with a zero for the imaginary component are mathematically identical to real numbers). The HP-28S likes to be consistent within a vector, so if any element of the vector is complex, all of the elements are made to be complex.

Secondly, notice that you didn't need to press **ENTER** before pressing **⌘ARRAY**. Most do-it-now operations will automatically evaluate the command line before they execute, thus saving you a step.

Since you can have a vector full of complex numbers, it would be reasonable to use the complex number operations on them, too. What happens if you do? To find out, press **⏮PREV** to get to the previous page of the **ARRAY** menu.

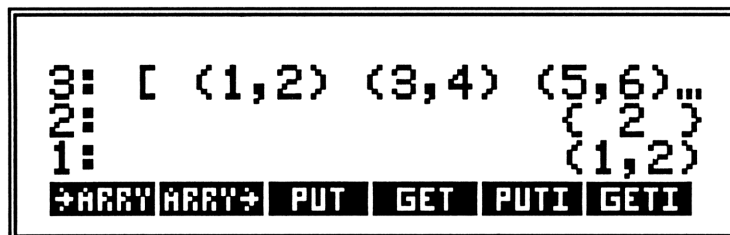
Then Press: **⌘R.**



The complex vector has been decomposed into two real vectors, just as a complex number would be decomposed into two real numbers. In the same way also, the real components of the original complex vector are in the Level-2 vector, and the complex components are in the Level-1 vector.

As you might expect, **R→C** will recompose the original complex vector from the two real vectors in Levels 1 and 2.

Now Try This: Press **DROP** **DROP** **{** **1** **NEXT** **GETI**. What happened?



First, you brought that 4-element complex vector to Level 1.

Second, you keyed in a list containing the number 1. This list was created to be an *index* into the vector.

Third, you pressed **GETI** which told the HP-28S to GET the indexed component from the vector in Level 2, then Increment that index and push the selected component on the stack – in that order. Sure enough, component 1, **{ 1,2 }** was pushed onto the stack and the index was incremented to **{ 2 }**.

Encores: Press **DROP** **GETI**. Now the second component, **{ 3,4 }**, is pushed onto the stack, and the index is incremented to **{ 3 }**.

Press **DROP** **GETI**. The third component, **{ 5,6 }**, is pushed onto the stack, after the index is incremented to **{ 4 }**.

Press **DROP** **GETI**. Now the fourth component, **{ 7,8 }**, is pushed onto the stack, but since there are only 4 components in this vector, the index is cycled back to **{ 1 }**.

At this point, your display will look like this:

```
3: [ (1,2) (3,4) (5,6) ]
2: { 1 }
1: { 7,8 }
→ARRY←ARRY→ PUT GET PUTI GETI
```

You can see how easy it is to gain access to any element within a vector, using the GETI command. Remember that the I in GETI stands for Incrementing the index (not simply the fact that you're using an index) while GETting the indexed element.

A Variation: Press **DROP** **GET**. Here is the result:

```
3:
2: [ 1 2 3 4 ]
1: { 1,2 }
→ARRY←ARRY→ PUT GET PUTI GETI
```

Unlike GETI, GET *consumes* both the index and the vector and leaves only the indexed component on the stack. Therefore, you would probably use this form of GETting when you don't care about keeping the vector in the stack. When you do want to keep the vector, you could either make a copy of it (with DUP) prior to using GET, or you could use GETI.

By now, you've probably noticed that many commands come as matched sets of complementary functions, such as $C \rightarrow R$, $R \rightarrow C$, $\rightarrow \text{ARRY}$, and $\text{ARRY} \rightarrow$.

Speaking Of Which: Press **DROP** **{** **1** **ENTER** **5** **ENTER**. Here's what you'll see:

```
3:          [ 1 2 3 4 ]
2:          {  1 }
1:          } 5
+ARRY+ARRY+ PUT GET PUTI GETI
```

Now try to guess what will happen if you press **PUTI**. Hint: **PUTI** is the complement to **GETI**. Therefore, it should put the number in Level 1 into the vector in Level 3 at the location specified by the index in Level 2. It should also increment the index.

Yes, But Does It Really? Press **PUTI**.

```
3:          [ 5 2 3 4 ]
2:          {  2 }
1:          }
+ARRY+ARRY+ PUT GET PUTI GETI
```

Shur 'nuff: The machine put the **5** into the vector as element number 1 (the indexed element), and the index was incremented to 2.

And Notice This: Press **6** **PUT**. **PUT** is the complement of **GET**, functioning similarly to **PUTI**, except that it doesn't preserve the index.

A Visit with Vectors

Like complex numbers, vectors are a meaningful form of information, and there are operations that use them as such. But you're not limited to using them solely as mathematical vectors (i.e. representations of "reality"). Keep in mind that because you can "dissect" vectors and reassemble them, you can define new operations that give new meanings to the vector data type – meanings that have no relationship to the concept of vectors in the real world. So before going on, be sure you understand vectors, how to build them, and how to take them apart.

1. Give three ways of putting the vector $[(1,2) (3,4) (5,6)]$ into Level 1.
2. If $\mathbf{A} = [1 \ 2 \ 3]$, $\mathbf{B} = [-3 \ 2.5 \ -.2]$ and $\mathbf{C} = [\frac{1}{3} \ \sqrt{2} \ -6]$ what's $14.5\mathbf{A} - 0.2\mathbf{B} + (1,1)\mathbf{C}$?
3. Sum the real and imaginary components of the result of the second problem. I.e., split that vector into two real-valued vectors and sum them.
4. Using the result of problem 3, find the corresponding unit vector ($\mathbf{u} = \frac{\mathbf{v}}{|\mathbf{v}|}$).
5. For the following table, find the total hours worked per person and overall.

	Abe	Ben	Carl	Dan
Mon	8.00	7.75	4.50	6.40
Tue	7.50	8.25	5.50	7.40
Wed	3.50	6.50	4.75	7.10
Thu	8.00	7.50	4.00	7.50
Fri	8.10	8.00	4.50	7.25

6. Create the vector **[1 2 3 4 5 6 7 8 9]**. Change element 4 to **40**. Change element 7 to **70**. Change element 1 to **(1,1)**.

7. Create the vector **[1 2]**. Redimension it to a 5-element vector. Change element 3 to **3**. Redimension the vector to 2 elements

8. How would you key in these vectors?

$$3\mathbf{i} + 4\mathbf{j}; \quad 2\mathbf{i} - 4\mathbf{j} + 2.3\mathbf{k}; \quad \sqrt{2}\mathbf{i} + \sqrt{3}\mathbf{j} + 1/3 \mathbf{k}$$

9. Evaluate the following:

$$[3 \ -4 \ 5] + [1 \ 1 \ -2]$$

$$3 \times [2 \ -7 \ 1] - 4 \times [-3 \ 0 \ 4]$$

$$[1 \ -8 \ 0 \ 5] \bullet [4 \ 1 \ -2 \ 5]$$

(\bullet is the symbol for *dot product*.)

$$[-3 \ 4] \times [8 \ -2]$$

(\times for vectors is the cross product.)

Results of A Visit With Vectors

1. `[(1,2) (3,4) (5,6) ENTER;`
`(1,2) ENTER (3,4) ENTER (5,6) ENTER { 3 ARRAY \div ARRY;`
`[1,3,5] ENTER [2,4,6] ARRAY PREV $R\div C$.`

2. `14.5 ENTER [1,2,3] X [CHS 3,2.5,.2] CHS ENTER .2 X -`
`3 $\frac{1}{X}$ 2 \sqrt{X} 6 CHS ENTER { 3 ARRAY \div ARRY (1,1) X +`

Result = `[(15.4333333333, .333333333333)`
`(29.9142135624, 1.41421356237) (37.54, -6)]`

3. `ARRAY PREV $C\div R$ +;`

Result = `[15.7666666666 31.3284271248 31.54]`

4. `ENTER ARRAY NEXT NEXT $M\div S$ $\frac{1}{X}$ X;`

Result =
`[.334265455504 .664186741847 .668672249469]`

Note that since the HP-28S won't allow you to divide a vector by a scalar (a real number), you must invert the magnitude (via $\frac{1}{X}$) and multiply.

5. [2 , F I X] ENTER

(Mon) [[8 , 7 . 7 5 , 4 . 5 , 6 . 4] ENTER

(Tue) [[7 . 5 , 8 . 2 5 , 5 . 5 , 7 . 4] +

(Wed) [[3 . 5 , 6 . 5 , 4 . 7 5 , 7 . 1] +

(Thu) [[8 , 7 . 5 , 4 , 7 . 5] +

(Fri) [[8 . 1 , 8 , 4 . 5 , 7 . 2 5] +

Totals = [35.10 38.00 23.25 35.65]

[ARRAY] NEXT NEXT **CNRM** or [ARRAY] **ARRY→** [DROP] [+] [+] [+]

Sum total = 132.00 hrs.

Notes: CNRM returns the sum of the elements of a vector. Also, you may want to return to STD mode at this point.

6. [[1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9] ENTER

[4] ENTER [4] 0 [ARRAY] **PUT** ;

[7] ENTER [7] 0 **PUT** ;

Notice that you can use a real number (as opposed to a list) as the index to PUT, PUTI, GET, GETI, and →ARRY although ARRY→ and SIZE will always return one-element lists when used on vectors.

Since you can't PUT a complex number into a real vector, make the vector complex first.

[(1 , 0 X] 1 ENTER [(1 , 1] **PUT**

7. [1][1][,][2][ENTER]

Result = [1 2]

[5][ENTER][][ARRAY][NEXT][][ROM]

Result = [1 2 0 0 0]

[3][ENTER][ENTER][][ARRAY][PUT]

Result = [1 2 3 0 0]

[2][][ROM]

Result = [1 2]

Things to notice:

- Redimensioning a vector to a larger vector fills the new elements with 0's to the end of the vector.
- Redimensioning a vector to be a smaller vector throws away elements at the end of the vector.

8. [3][,][4][ENTER] Result = [3 4];

[2][,][4][CHS][,][2][.][3][ENTER] Result = [2 -4 2.3];

[2][][3][][3][][3][][ARRAY][][ARRAY]

Result =

[1.41421356237 1.73205080757 .333333333333]

9. [3][,][4][CHS][,][5][ENTER][1][,][1][,][2][CHS][+]; Result = [4 -3 3]

[3][2][,][7][CHS][,][1][X][4][3][CHS][,][0][,][4][X][-]

Result = [18 -21 -13]

[1][,][8][CHS][,][0][,][5][4][,][1][,][2][CHS][,][5][ENTER][DOT][ENTER]

Result = 21

[3][CHS][,][4][8][,][2][CHS][][ARRAY][NEXT][NEXT][][ROSS]

Result = [0 0 -26]

Arrays

You've already heard the word "array." In fact, you know that a vector is one form of an array. Mathematically, an array is nothing more than an ordered arrangement of numbers in rows and columns, and a vector is merely a one-row array.

So in general, you can think of an array as a list of vectors. The vectors form the rows, and corresponding elements of the vectors form the columns. The lengths of the component vectors must therefore be the same, but the number of rows (vectors) does not need to be the same as the number of columns (elements per vector).

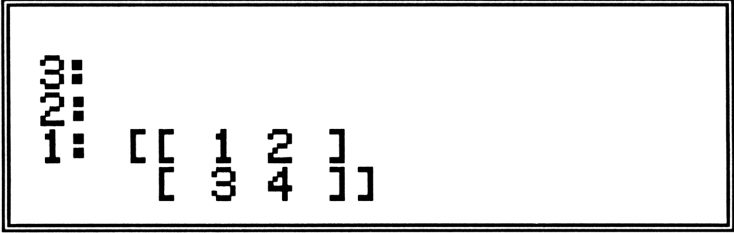
On the HP-28S, an array is represented by bracketing a list of vectors with square brackets (**[** and **]**). Thus, you'll see a double set of square brackets, since vectors themselves use a set, too.

But here's a less obvious point: The HP-28S allows you to deal with arrays only as a two-dimensional list of numbers (i.e. not as a "vector of vectors"), which of course conforms with the normal, mathematical notions of matrices.

In other words, it may seem logical to be able to extract entire rows (vectors) from any given array and then horse around with them, just as you can with the components of complex numbers and vectors. There are no conceptual reasons why you can't do this; there just aren't any commands built into the HP-28S to let you do it. You can decompose arrays only on an element basis, not on a row (vector) basis.



An Example: Press          .



Here's what you'll see:



The calculator display shows a 3x2 matrix. On the left, the dimensions are listed vertically: 3:, 2:, and 1:. To the right of these dimensions, the matrix elements are displayed in two rows. The first row contains the elements 1 and 2, and the second row contains the elements 3 and 4. The entire matrix is enclosed in square brackets, with the first row's opening bracket being part of the dimension list.

Things To Notice That Shouldn't Be Very Surprising Any More:

Notice that you didn't need to delimit the first vector with  before starting the second vector with .

Since the HP-28S is expecting the second vector (you told it that there were to be more than one when you typed ), it doesn't need the closing bracket. The new starting bracket simply tells it that you're finished with the first row/vector (and therefore tells it how many elements are in each row).

Nothing very new, right? The HP-28S has dealt with the array in much the same fashion in which it deals with vectors. This is in fact the case with almost all of its other operations as well. You really wouldn't want it any other way.

Now, the only major mechanical difference between working with vectors and arrays on the HP-28S is in the index – that list denoted with braces – associated with →ARRY, ARRY→, GET, GETI, PUT and PUTI.

Because of its two-dimensionality, the size of an array is represented as a *pair* of numbers within braces. The first number of the pair is the number of rows (vectors) while the second is the number of columns (elements per vector).

Take It Apart: Press **▢** **ARRAY** **ARRY→** .



Use **▢** **VIEW↑** and **▢** **VIEW↓** to look over the stack. The array has been decomposed into its component numbers, with the elements having been pushed onto the stack in "row-major order." That is, they were pushed on starting with the first row, proceeding from left to right until that row was exhausted. Then the second row was taken left-to-right, and so forth.

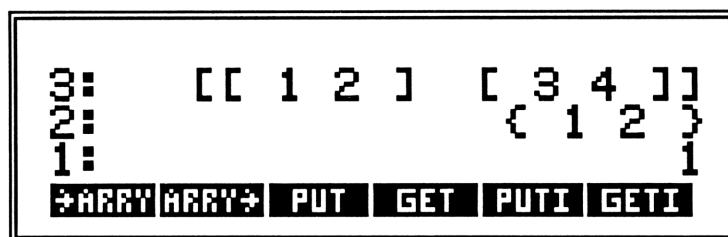
The list on Level 1 contains 2 numbers specifying two rows and two columns, reminding you (and the HP-28S) where those numbers above actually originated. As with all decomposing commands, the stack is left in a state that allows you to immediately recompose.

Do It: Press **▢** **ARRY**. The array is recomposed, using that index list on Level 1 as the blueprint for rebuilding the array.

Again, remind yourself that you're working in the same **ARRAY** menu as you did with vectors. How, then, do these same commands work on two-dimensional objects?

Find Out: Press **{ 1 , 1 ENTER**. This is the array's index, pointing to the first row, first column.

Now press **GETI**. Here's the result:



The calculator screen displays the array index **{ 1 1 }** and the data **[[1 2] [3 4]]**. Below the data, the command **GETI** is shown, indicating the current index is **{ 1 1 }**. The screen also shows the command **PUTI** and the command **GETI** again.

This works exactly analogously to the way it does with vectors, with an allowance for the second dimension. Indeed, the only new thing to notice is how the index is incremented: **PUTI** and **GETI** increment the index in row-major order.

Do it a few more times to make sure you see the pattern. Press **DROP GETI**. The index is now **{ 2 1 }**.

Press **DROP GETI**. The index is now **{ 2 2 }**.

Press **DROP GETI**. The index cycles back to **{ 1 1 }**.

Care to speculate on how **GET** and **PUT** work? Go ahead – experiment with them.

Array Aptitude Test

1. Convert the vector [1 2 3 4 5 6 7 8 9] (a 1 x 9 array) into a 3 x 3 array. Change element {2 2} to 10.
2. Convert the array result from problem 1 into a complex array such that all of the imaginary components are 0. Are there different ways to do this?
3. Given $\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 9 \end{bmatrix}$, double \mathbf{A} and subtract 1 from every element (i.e., find $2\mathbf{A}-1$).
4. Given $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix}$ show that $\mathbf{A} \times \mathbf{B} \neq \mathbf{B} \times \mathbf{A}$ (thereby showing that matrix multiplication is not commutative).
5. How might you extract individual rows (single vectors) from a 3 x 3 array?

6. Given \mathbf{A} and \mathbf{B} from problem 4, calculate the following:

$$\mathbf{A} \cdot \mathbf{B}$$

(the *dot* product of \mathbf{A} and \mathbf{B})

$$\mathbf{AB}$$

(the product of \mathbf{A} and \mathbf{B})

$$\mathbf{A} \div \mathbf{B}$$

A.A.T. Results

1. [1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ENTER **ARRAY**
ARRY **DROP** { 3 , 3 **÷ARRY** or **NEXT** { 3 , 3 **ROM** ;
 { 2 , 2 **}** 1 0 **ARRAY** **PUT** or 5 , 1 0 **ARRAY** **PUT** ;

Notice that the index to PUT can either be a list or a real number. If it is a list, the list-elements specify the row and column of the array-element. If the index is a real number, it specifies the position of the array-element counting in row-major order.

2. Method 1: ENTER **ARRAY** **NEXT** **SIZE** 0 **CON** **NEXT** **NEXT** **R÷C** ;

I.e., find the SIZE of the array and use it to create a CONstant array that is filled with 0's. Next, convert these two real arrays into a complex one.

Method 2: (1 , 0 **X**)

3. [[1 , 2 , 3 [4 , 5 , 9 ENTER
 2 **X** ENTER
SIZE ENTER (Find the size of the array.)
 1 , **CON** ENTER (Create a CONstant array of the same size filled with 1's.)
=

Result = [[1 3 5] [7 9 17]]

4. `[[1 , 2 [3 , 4 ENTER`
`[[4 , 3 [2 , 1 ENTER`
`DUP 2 ENTER` `×`
`SWAP ROT ENTER` `×`
`== ENTER`

SWAP ROT reverses Levels 1, 2 and 3. See page 99.

`==` tests Levels 1 and 2 for equality. If they are equal, the test will produce a 1. If not, it produces a 0.

Result = 0. The values at Levels 1 and 2 were not the same.

5. Take the array `[[1 2 3] [4 5 6] [7 8 9]]` as an example. The strategy is to create the `1 x 3` array `[[1 0 0]]` and multiply this by the first array. Thus:

`[[1 , 0 , 0 ENTER` `[[1 , 2 , 3 [4 , 5 , 6 [7 , 8 , 9` `×`

Using `[[0 1 0]]` will give the second row, and `[[0 0 1]]` will give the third.

Be sure to keep in mind that the *order* of the two arrays is important in multiplication!

6. `[[1 , 2 [3 , 4 ENTER` `[[4 , 3 [2 , 1 ENTER`
`DUP 2 ENTER DOT` `= 20 ;`
`DROP DUP 2` `×` `= [[8 5] [20 13] ;`
`DROP ÷` `= [[4 5] [-5 -6] ;`

You've now seen three different objects you can construct with real numbers as your basic building blocks. Before building anything else, see if you can put all these different real-number constructions into some perspective. How do they all relate to one another?

- Each of these information objects is built from real numbers, which are one of the three fundamental information types (characters and binary digits are the other two).
- Throughout these last 33 pages, you've seen that innocent-looking little word, "list." The ordered, indexed list is really the key when it comes to thinking about how the HP-28S associates information. In order of increasing sophistication, you can think of the real-number-based objects like this:
 - A complex number is a 2-element list of real numbers;
 - A vector is an "n"-element list of either real or complex numbers. Therefore, it can actually be a list of other lists (complex numbers).
 - An array is an "n"-element list of vectors. Therefore (in the case of a complex array), it can actually be a list of lists of lists.

Lists of "things." That's the compound-information object "concept" in its pure form – the truly consistent, generalized way to think about these objects.

However, as you've already seen, you don't have exactly the same set of "list arithmetic" and "list decomposing tools" available for each of these objects. Clearly, the HP-28S's command set has been tailored toward the real-life math "meanings" of each of the objects. For example, it's true that an array is nothing more than a list of vectors, but the HP-28S won't decompose it into component vectors for you – probably because this isn't a commonly needed application.

Characters

Characters are another sort of simple information that you use every day without thinking much about it; you're using them right now as you read this book. They are so simple that they convey very little information by themselves. But in mathematics, if you associate a certain character, say X , with a value or operation, it gains information value. Notice that this value is not intrinsic; you have given it this value *by association*.

Characters also gain in information value when they are used to make *words*. The characters on this page are only meaningful because of their association with other characters to make words. Thus they attain a higher level of information.

The words on the page in turn gain meaning by being associated with other words in *sentences*. The process goes on through all of the sentences in each *paragraph*, all the paragraphs on a *page*, to all the pages in this *book*. And it doesn't stop there. The book is only meaningful in the context of your HP-28S, and your HP-28S is only meaningful in the context of what you want to do with it.

Although its ability to gain meaning from higher and higher levels of associated characters is far more limited than yours, the HP-28S can indeed go a few steps up the ladder. However, it actually has *no* facility to deal with characters simply as characters – only with characters as members of larger information types.

For example, you cannot place a single character on the stack. That is, there is no data object type called a "character." This is different than with a real number, which may, of course, appear on the stack as itself – in its elementary building-block form. Not so with characters. They must always appear within a compound data object.

Character Strings

A *string* is simply a *list* of characters, displayed within quotation marks ("). And although you can't have a single character on the stack, you *can* have a string of one character(!).

If this distinction sounds a little strange, don't worry. In practice, you'll find it to be irrelevant (i.e. you can "mess with" a 1-character string just as if you were "messaging with" a single character), but you'll see it to be logically true. In fact, the logic is consistent even to the point of allowing strings which contain *no* characters ("empty" strings).

Strings may be arbitrarily long and may contain *any* character.

Build One: Press     H E L P ,  I ' M 
T R A P P E D  I N  H E R E .

Here's your string at Level 1 on the stack:





A screenshot of a calculator's stack display. The stack has four levels, numbered 1 to 4 on the left. Level 1 contains the string "HELP, I'M TRAPPED ...". Levels 2, 3, and 4 are empty.








The Notice-These-Things-Drill:

1. As always, you don't need to key in the final delimiter. The HP-28S closes the expression for you.
2. All characters except " – including those that usually act as delimiters in other objects – are included in the string.
3. All elements of the string are on the same stack level. Both this and the fact that they are grouped together inside double quotation marks tells you that this is a single object.
4. If a string is too long, it will run off the right hand side of the display. As always, the HP-28S indicates this with an ellipsis. The only (convenient) way to view the entire string is to EDIT it and scroll from end to end using the cursor keys.

Strings are, of course, information objects and can therefore be placed on the stack and manipulated with stack commands. But you can't use them to do math since math isn't defined for such objects.

You can, however, use one command that is normally associated with math. Since the concept of adding two strings together (appending one to another) is similar to the concept of adding two numbers, the + command effectively adds two strings.

Other than +, though, you'll need to rely mainly on the string-specific commands that you'll find in the STRING ( ) menu.

Explore: Press             .



The image shows a calculator screen with a stack of three items. The top item is '3:', the middle is '2:', and the bottom is '1: "HELP, I'M TRAPPED!"'. Below the stack, the command line shows the sequence of commands entered:      .

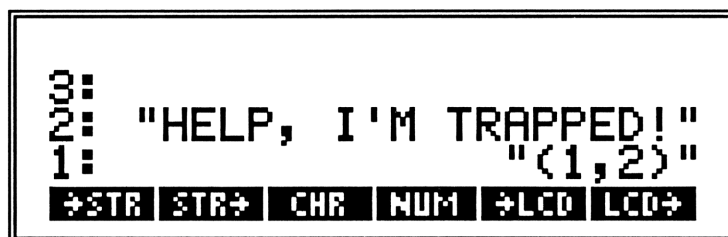
Here's what you did:

1. You selected the second level of the STRING menu.
2. You entered two parameters for the SUBstring operation. The first number indicates the first and the second number the last character of the original string that you want to keep.
3. You invoked the SUBstring command. The result was the string **"HELP, I'M TRAPPED"** in Level 1 of the stack – characters 1 through 17.
4. You keyed-in the argument for the CHaRacter operation. You may have noticed that not all of the characters you might expect to be available are on the HP-28S keyboards (and there are some you probably didn't expect, too). In the HP-28S Reference Manual in the section on strings there is a table of characters and character codes. The CHR operation allows you to convert a real number (dang, those things just keep popping up) into a one-character string. For example, 33 is the character code for **!**.
5. You moved back to the first level of the STRING menu (there are only two) and performed the character code conversion. **"!"** was left in stack Level 1.
6. Finally, you added (appended) the string in Level 1 onto the end of the string in Level 2 using **[+]**. The result was left in Level 1. Notice that the order is Level 2 + Level 1, as always.

Since you can't have individual characters on the stack (only one-character strings), there's no convenient way to change a string into characters (sure, you *could* do some tricks with SUB, using several copies of the original string, but total string decomposition isn't generally very useful).

There is, on the other hand, a very powerful method of changing a string into other, very useful things. Strings may be converted to and from *any* information object by using →STR (convert to STRing) and STR→ (convert from STRing).

For Example: Press ((1 , 2 ENTER →STR.



The calculator display shows a stack with three levels. Level 3 contains the number 3. Level 2 contains the string "HELP, I'M TRAPPED!". Level 1 contains the string "(1,2)". Below the stack, the function keys →STR, STR→, CHR, NUM, →LCD, and LCD→ are visible.

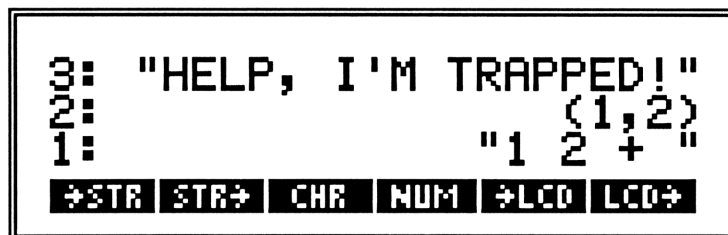
Now press STR→.



The calculator display shows the same stack as before, but the string "(1,2)" on Level 1 is now enclosed in quotation marks, making it a string object. The function keys below the stack remain the same.

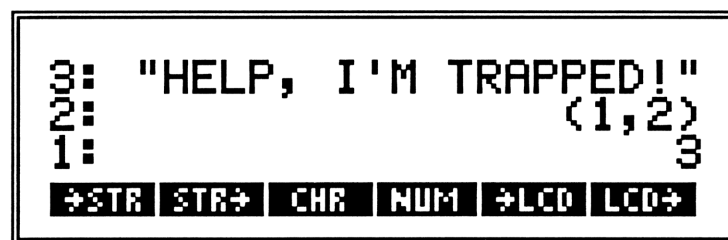
These two objects on Level 1 don't look much different, but they are. "**(1,2)**" is a *string*, not a complex number. The HP-28S (and you) can tell this by those quotation marks. **(1,2)**, on the other hand, is a complex number and the two objects are used in radically different ways.

Try This: Press **⏏** (Notice that the α annunciator comes on. Remember what that means?) **1****SPACE****2****+****ENTER**.



3: "HELP, I'M TRAPPED!"
2: (1,2)
1: "1 2 +"
→STR STR→ CHR NUM →LCD LCD→

Press **→STR**.



3: "HELP, I'M TRAPPED!"
2: (1,2)
1: 3
→STR STR→ CHR NUM →LCD LCD→

What happened? **STR→** recognized that the string contained three different objects, and in converting the string into those objects, it posted them.

It read the string from left to right and first found the **1** (delimited by a space). It converted that character into a real number and pushed it onto the stack. It then kept reading and found the **2**. It converted that to a real number and pushed it onto the stack. Finally, it found the **+**, recognized it as the name of a command, and performed it. The first two stack items were therefore added together, with the result landing at Level 1, as usual. **STR→** then reached the end of the string and stopped reading.

The important point: This is *exactly* how the command line would have responded if it had contained those characters when **ENTER** was pressed.

So you see that strings may actually form "pretyped" command lines that you can then post in their command form by converting out of string notation.

However, you probably won't use strings for this purpose nearly as much as you will for other information. In fact, strings may quite possibly be the most information-packed data objects available to you, because they allow the HP-28S to communicate with you in English (or whatever language you prefer). This type of information is probably how you'll encounter and use strings most often.

Test your understanding of them now....

Character String Query

1. Given that Level 1 contains the number **100.01**, use it to build the string **"Vol.= 100.01 gal."**
2. How would you go about pulling the number back out of this string? Assume that you don't actually know what that number is.
3. Taking the number **6.022E23** from Level 1, format it within a string so that it looks like this: **"6.022 * 10^(23)"**.
(Hint: use MANT and XPON from the REAL number menu.)
4. Starting with the result of problem 3, what would you expect the result to be if you invoked STR→? Why? Rewrite the string so that STR→ gives you back the original real number.
5. Change the string **"You understand?"** to **"You understand!"**

C.S.Q. Answers

1. S T D ENTER " V LC O L . = SPACE ENTER SWAP STRING →STR " " SPACE LC G A L . ENTER ++

Recall why you use \oplus here – to concatenate (join) strings.

2. ENTER ENTER " SPACE ENTER STRING NEXT POS 1 + SWAP SIZE SUB ENTER " SPACE ENTER POS 1 SWAP SUB NEXT STR→

Notice how you find the spaces on either side of the embedded number by using the POS function. You then use the position of the space as one of two indices you need to extract a SUBstring.

3. ENTER REAL NEXT MANT SWAP XPON SWAP STRING →STR " × 1 0 α ^ (ENTER + SWAP →STR + ") ENTER +

Same idea as in problem 1, really – except for the use of MANT and XPON. Remember that to get $^{\wedge}$ without spaces around it, you press α to get the \equiv cursor before pressing \wedge .

4. The result is a **Syntax Error** because the expression is not in postfix form. If you want a string that breaks the original number up into mantissa and exponent but will still evaluate back to the original number, then use "6.022 10 23 ^ *"

5. ENTER STRING NEXT SIZE 1 - 1 SWAP SUB 3 3 NEXT CHR +

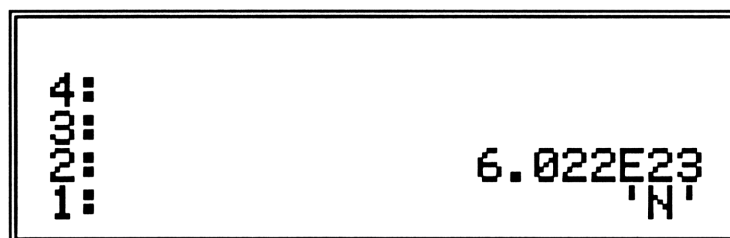
Names

Names are also character strings, but they have special restrictions and a special purpose in life. They're represented by bracketing lists of characters with *single* quotation marks (').

A name is a descriptive word used to describe an object in the HP-28S. In other words, if you have an object – any object – you can give it a name and thereafter refer to the object by that name. This name will be associated with the object until you change the association, which you may do at any time.

You can, of course, elect not to name objects, and as a matter of fact, you may also have a name that has no object associated with it.

For Example: Clear the stack, clear the menu line, and load 6.022×10^{23} and 'N' onto the stack. After doing so, here's how things look:



And what have you done? First, of course, you cleared the stack and the menu line. Then you put the real number 6.022×10^{23} onto the stack.


Then you put the *name* 'N' on the stack: `'N'ENTER`. You can tell the HP-28S regards this as a name because of the single quotation marks.

That 'N' on the stack isn't yet associated with any object. Although it need never be so linked to an object, it's often useful to do so, and you can link it to any other object (even another name) by using the STO (STOre) command.

Go For It: Press **[STO]**.

Both the name and the number are removed from the stack, right?
What happened to them?

To find out, press **[N][ENTER]**. Here's what happens:



4:	
3:	
2:	
1:	6.022E23

You can see that when you put a name associated with an object onto the stack *without* single quotation marks, this tells the HP-28S to "evaluate" the name, thus replacing it with the object itself (and you're going to appreciate this more and more as time goes on).

Do It Differently:

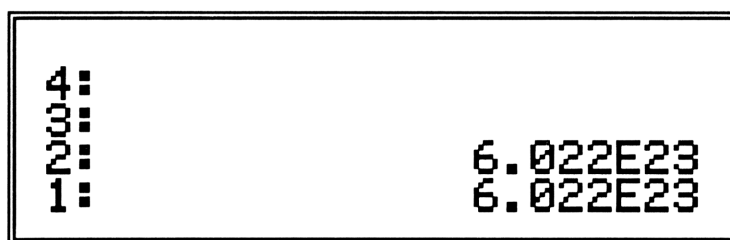
Press **[']****[N]****[ENTER]**.



A screenshot of an HP-28S calculator display. On the left side, there is a vertical stack of four labels: '4:', '3:', '2:', and '1:'. On the right side, the display shows the value '6.022E23' followed by a single quote and the letter 'N' followed by another single quote.

When you use **[']**, you're telling the HP-28S that you *do* want just the name on the stack. So there you have it.

Now Change Your Mind: You've decided you wanted the object after all – not just its associated name? No problem. Just press **[EVAL]**.



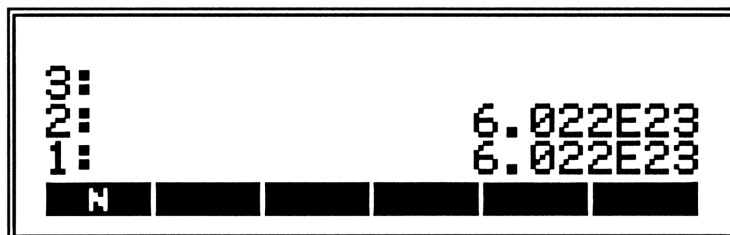
A screenshot of an HP-28S calculator display. On the left side, there is a vertical stack of four labels: '4:', '3:', '2:', and '1:'. On the right side, the display shows two instances of the value '6.022E23' stacked vertically.

Can you make an educated guess as to what **[EVAL]** does?

It EVALuates the name in Level 1, thereby replacing it with its object.

But typing in an object's name isn't the only way to put the object on the stack.

Watch: Press **USER**. Assuming that you don't have any other named objects in your calculator yet, you'll see:

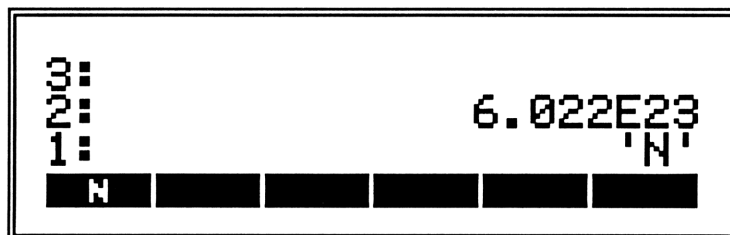


The USER menu is your own personal menu. *You* are the user. Whenever you **STO**re an object in a name (i.e., associate a name with an object) that name will appear in your USER menu.

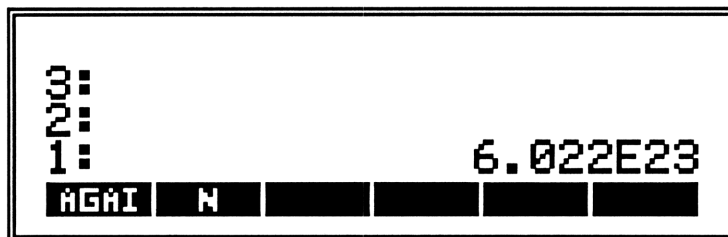
Use Your USER: Press **CLEAR** **N**. Pressing the menu key is a quick way to get the named object to be put onto the stack.

Now press **'** **N** **ENTER** (preceding the menu key with **'** allows you to use just the name, rather than the object it represents).

Your display should now look like this:



Next: Press **[']** **A** **G** **A** **I** **N** **[STO]**.



You have now stored the name **'N'** in the name **'AGAIN'**! This is an example of using one name to refer to another, which is quite "legal," of course, since a name is just an object like all the others.

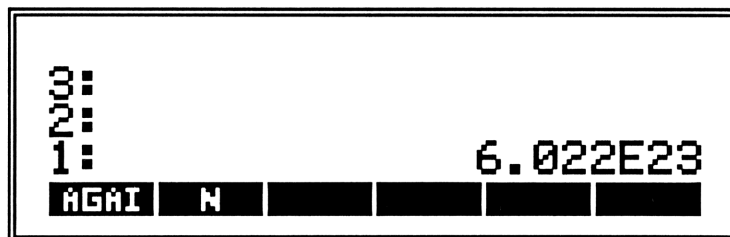
And notice that **NAME** has been added to the USER menu. The first four or five characters in any name are all the menu has room for (not all of the characters are of the same width).

But the name itself has not been shortened.

Prove this by pressing **[']** **NAME**. The complete name is loaded into the command line. Now press **[ATTN]** to clear it.

But what is **AGAIN**'s "value?" Is it **'N'** or is it **6.022E23**?

Here Goes Nothing: Press **CLEAR** **AGAIN**.







There you have it. Whenever you evaluate a "name of a name of a name...etc.," the HP-28S continues to follow its nose, evaluating each named object until it encounters one that's not a name (or a name that doesn't point to anything).

In this case, **AGAIN** points to **N**, and **N** points to **6.022E23**. Evaluating **AGAIN** causes **N** to be evaluated. And since the object that **N** points to is not a name, the HP-28S stops evaluating and places that object on the stack.

This chain of evaluations has a potential hazard. If you were to store, say, **'A'** in **'B'** and **'B'** in **'A'** and then try to evaluate either name, the HP-28S would go into an infinite loop. **A** would point to **B**, and **B** would point back to **A**, and so on, forever. In fact, the only way to stop this would be to press **ATTN** and **▲** at the same time (a special command just for such emergencies).











Another caution: Not all characters can be used in names. For obvious reasons, you can't use delimiters (**#**, **[**, **]**, **"**, **'**, **{**, **}**, **(**, **)**, **&**, *****, **,**, **SPACE**, **NEWLINE**). You are allowed to use numerals, but not as first characters. And it's a definite no-no to use symbols or names which are already commands in the HP-28S (**+**, **-**, *****, **/**, **^**, **√**, **=**, **<**, **>**, **≤**, **≥**, **≠**, **÷**, **DROP**, **SWAP**, etc.).

Name Games

1.   will dissociate a name and a value. Use   to rename 'H'.

2. The HP-28S has no *data registers*, unlike many calculators. Data registers are usually a numbered series of slots for storing real numbers. Create a named data object that looks like a series of data registers. How would you store a number into it? How would you recall a number?

3. Store 4 in "register" 1 of the object you created in problem 2. Evaluate the named object so that it's in Level 1 of the stack. Store 3 in "register" 1 of the named object. Evaluate the named object so that it's in Level 1 of the stack. Why are Levels 1 and 2 different?

4. Purge the name H. Press    . Before trying it to find out, try to guess what the result of STR→ would be on this string. Type      . What will →STR do to this object?

5. Store 1 in A, -8 in B and 15 in C, then solve the following:

$$A^2 + B^2 + C^2$$

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

$$\frac{\sin(A) - \cos(B)}{\ln(C)}$$

Name Game Winners

1. `N` `ENTER` `'N` `PURGE` `'Q` `STO`

The first entry of N is its value; the second is its name. You dissociate the two and then, with the value still on the stack, you associate it with 'Q'.

2. `{` `1` `0` `ENTER` `0` `ENTER` `C` `O` `N` `ENTER` `'M` `E` `M` `O` `R` `Y` `STO`

Store : `'M` `E` `M` `O` `R` `Y` `ENTER` `{` `1` `ENTER` `1` `'` `P` `U` `T` `ENTER`

Recall : `'M` `E` `M` `O` `R` `Y` `ENTER` `{` `1` `ENTER` `G` `E` `T` `ENTER`

You have created a list of ten "registers" which you may then store into and recall out of by using the list commands PUT and GET.

3. `'M` `E` `M` `O` `R` `Y` `ENTER` `ENTER` `ENTER`, then `{` `1` `ENTER` `4` `ENTER` `P` `U` `T` `ENTER` `EVAL`
`SWAP` `ENTER` `{` `1` `ENTER` `3` `ENTER` `P` `U` `T` `ENTER` `EVAL`

The point here is that the value on the stack is *not* what the name is referring to. Once you have evaluated the name, the object left on the stack is effectively a *copy* of what was referred to by the name. Changing the value on the stack will not change the named object and vice versa.

4. Since performing STR→ on a string is identical to keying in the contents of the string into the command line, 'N' is left on the stack as a name that has no object to point to. This will happen to any valid name the HP-28S doesn't already recognize as the identifier of another object. →STR will make the following string out of the name: "'N'"

5. 1 ENTER 8 CHS ENTER 15 'C STO 'B STO 'A STO USER

$$A \cdot X^2 + B \cdot X^2 + C \cdot X^2 = 290$$

$$B \cdot CHS \cdot B \cdot X^2 + A \cdot C \cdot X^4 - \sqrt{X} + A \cdot 2X = 5$$

$$A \cdot \text{TRIG} \cdot \sin B \cdot \cos - C \cdot \text{LOGS} \cdot \ln \div = -.359231029731$$

You can see how the use of names might make calculations easier. If the numbers in this problem were a little hairier, keying them in and keeping them straight would be more difficult and the use of names would almost be essential.

You can see how you might also create and use constants (names with values that don't change) and give them meaningful names (like JIM, FRED and PETE).

Notice that names don't need to be short to be easy to use since you can always key them in with one keystroke from the user menu.

So that's what you can build from characters. Now summarize for yourself the differences and relationships between these objects:

Although a character is a fundamental information type, it is *not* recognized as an object type on the HP-28S; therefore, you can't place a character on the stack or manipulate it in any way. You need to build a compound object from one or more characters.

The main object to build is the character *string*, which is just a list of characters within quotation marks ("). Such a string may be broken down into smaller strings – even a string with one (or zero) characters. But the object is still a string. You can, however, convert a string into another object (and back again).

A specialized form of string is the *name*, which is denoted by single quotation marks ('). The main purpose of a name is to associate itself with another information object (even another name), thus giving you an easy way to refer to large or complicated objects as you manipulate them.

Your repertoire is growing:

- You can use **characters** to build **strings** and **names**;
- You can use **real numbers** to build **complex numbers**, **vectors**, and **arrays**.

It's time now to look at what you can do with the third fundamental information type – **bits**.

Bits

You can think of bits (Binary digits) as being 1's and 0's, true and false, on and off, or any other pair of mutually exclusive states.

As such, they're used to indicate that some thing or state is either there (valid) or not there (invalid). As with characters and real numbers, bits gain meaning only within the context of their use. And since they are the simplest possible kind of information, they have almost no useful meaning unto themselves.

The HP-28S can use bits individually as *flags*. The word flag is computerese for a value that indicates the current state of something else. When that something only has two possible states, the flag can be a bit.

Many of the HP-28S flags signal certain system states of the machine. For example, there's a flag (48) that it examines whenever it needs to remember whether to use the ■ or the ▀ as the radix (as you'll recall from page 63, you do have this choice). But there's also a generous supply of flags that have no intrinsic meaning to the system – flags that you can therefore define for your own purposes.

As with characters, the HP-28S has no facility to deal with bits as bits on the stack – only with bits as members of larger information types. You can't place a single bit on the stack (i.e. the command TYPE, which tells you the type of object currently at Level 1, has no provision for a bit type). But – also as with characters – this is no serious limitation, since a larger data type may contain a single bit as its only member, and bit oriented operations will deal with this as if it were an elemental bit.

Binary Integers

A binary integer is a *list* of bits. On the HP-28S, the list may be from 1 to 64 bits long. The length of the list is called its *word size*.

You have several choices for the display and entry of a binary integer. You can use either binary (0 or 1), octal (0 - 7), decimal (0 - 9) or hexadecimal (0 - F) digits. A binary integer is entered and displayed preceded by #.

Like So: Press **CLEAR** **BINARY** (**B**) **#** **1** **1** **0** **1** **ENTER**.



You cleared the stack, selected the BINARY menu (because that's where most of the binary-integer operations can be found) and put the binary integer **# 1101b** onto the stack. You didn't need to type the **b** because you were in Binary mode (indicated by the **■** next to **BIN** in the menu) so the HP-28S assumed the number was keyed-in in – and should be displayed in – binary digits.

Had you tried to use any other digits when keying-in the number, the command line would have caught your error when you pressed **ENTER**. At that point you would have been obliged to either re-enter the number using only ones and zeros, select the digit entry mode (DEC, OCT, or HEX) compatible with the digits you keyed in, or type the trailing letter indicating the type of digits you used (**d** or **h**).

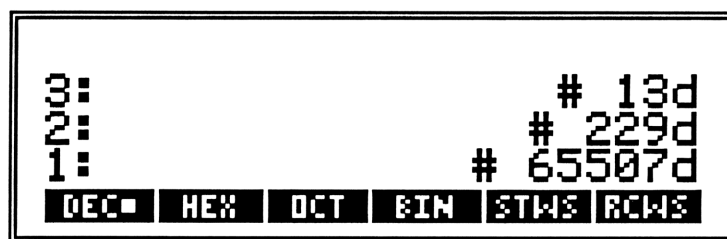
Now Press: **DEC** **#** **2** **2** **9** **ENTER**. What you see:



Upon pressing **DEC**, the HP-28S understands that DECimal digits are expected when keying-in binary integers. Not only that, it also changes the representation of *all* binary integers to use decimal digits (notice Level 2).

Now press **OCT** and **HEX** and notice how the display changes.

Now This: **#** **F** **F** **E** **3** **LC** **H** **ENTER** and see:



Putting the **h** at the end of the binary integer allowed you to key-in a hexadecimal representation while in decimal mode. Notice, though, that the display mode didn't change. **b**, **o** and **d** work the same ways for their respective representations.

Like characters, literal bits cannot live on the stack, so there's no convenient method for breaking a binary integer into its component bits. But binary integers *are* information objects; they can be placed on the stack and manipulated with stack commands. You can even use them to do a smattering of math (limited to +, -, x and ÷).

In fact, since a binary integer and a real number are both fundamentally numbers, the HP-28S will allow you to mix them within this same restricted set of math operations. The conversion is performed "on the fly," with the result always being a binary integer (any fractional portion of the real number is lost).

There are also several binary-number specific commands in the BINARY and PROGRAM TEST (■○) menus. If you're interested, by all means explore those menus. For now, however, this is enough of an introduction to binary numbers.

Binary Integer Test

1. What are the binary (base 2), octal (base 8), and hexadecimal (base 16) representations of the decimal (base 10) number 1000 (a.k.a. 1000_{10})?
2. What is $2_{10} \times FF_{16}$?
3. Calculate $2 \times (FFF_{16} \div 2)$. Why is the result not FFF_{16} ?
4. Set decimal mode, key in **# 100**, duplicate it, and convert the Level 1 copy to a character string. Now set binary mode. Why didn't the "number" in Level 1 change like the number in Level 2?

B.I.T. Answers

1. **1000** **BINARY** **DEC** **NEXT** **R→E** (Result = # 1000d);
PREV **HEX** (Result = # 3E8h);
DEC (Result = # 1750o);
BIN (Result = # 111101000b).

R→B converts a real number in Level 1 to a binary integer in the current base and word size.

2. **HEX** **#FF** **ENTER** **2** **×**; (Result = # 1FEh).
3. **HEX** **#FFF** **ENTER** **2** **÷** (Result = # 7FFh);
2 **×** (Result = # FFEh).

The point here is that the result of the division of a binary integer is also a binary integer; any fractional portion in the result is lost. Therefore, the result of the division is accurate only to the next lowest whole digit, and doubling this result might give you a number 1 smaller than your original.

4. **BINARY** **DEC** **#100** **ENTER** **ENTER** **→STR** **ENTER** **BIN**

Level 2 has # 1100100b, a binary integer. Level 1 is "# 100d", a character string. A character string, even though it may look like another type of object, is simply a string of characters and as such has no binary-integer meaning.

A Pause For The Cause

Take another compass reading here. You have now rounded out your repertoire of compound objects that can be built purely from one of the three fundamental information types:

Real Numbers may form *complex numbers*, *vectors* and *arrays*;

Characters may form *strings* and *names*;

Bits may form *binary integers*.

Now what? Where do you go from here? Is this the sum total of the objects you can build and use on your HP-28S?

Not quite. You've seen most of the possible objects, but the few remaining are the most powerful of all. And they're different – because they aren't constructions built from only a single information type. Every object you've built so far has been a list of simpler, *similar* objects (i.e. based upon the same fundamental information type).

This again corroborates what you read on page 150, that the HP-28S deals simply with "lists of things." Up to now, the main concern has been those "things" and what they can mean to you.

But what exactly is a list itself? What good is it? Can you have lists that combine any objects you want?

It's time to answer these questions....









Lists

The actual description of a list as an object ought to sound quite familiar by now:

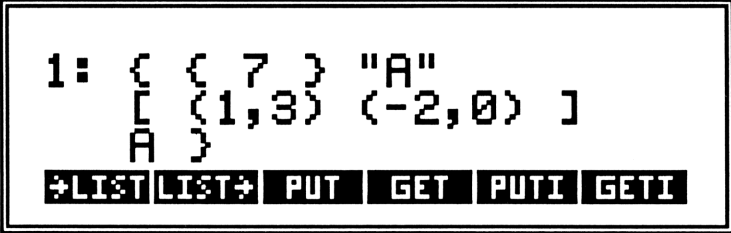
A list is a one-dimensional ordering of objects – *any* objects. It is ordered so that the left-most element is numbered 1, with the rest of the elements numbered in ascending order. It may be arbitrarily large or small; in fact, it may even be completely empty.

As you've also seen already, a list is represented by bracketing a collection of objects within braces ({ and }).

Build One: Press 

  7   A  [(1 , 3 ) 2 CHS  A ENTER
 LIST.

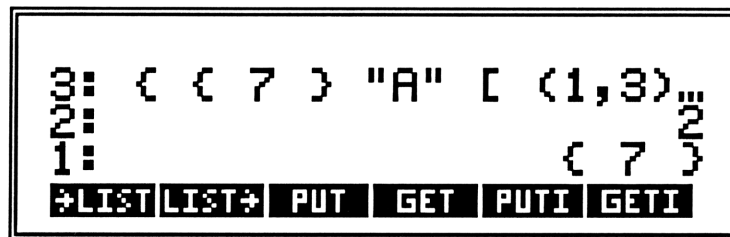
Here's what you'll see:



Notice these things:

1. As always, you don't need to press **■****[]** at the end of the list unless another object follows it in the command line.
2. All elements of the list are on the same stack level. Both this and the fact that they are grouped together inside braces tell you that this is a single object. From now on, unless you purposely break it into its components, it will be treated as one object.
3. This list contains a list (**{ 7 }**). This is the first object you've seen that can contain an object of the same type as itself.
4. If the list were longer than the display can hold, to view the whole list you must do one of several things:
 - (i) Edit Level 1 (with either **EDIT** or **VISIT**). If the object is still too large, you may use the cursor keys to scroll the display.
 - (ii) Use **■****[VIEW↑]** and **■****[VIEW↓]** to scroll through the display (if the objects within the list are quite long, this method is not too helpful).
 - (iii) Decompose the object, in this case with **LIST→**, and if necessary, use **■****[VIEW↑]** and **■****[VIEW↓]** to then examine the individual elements in the stack. Remember to rebuild the list with **→LIST** when you're done;
 - (iv) Use **GETI** to step through the list's components.

Try That Last Choice: Press **1** **GETI**. Result:



```
3: { { 7 } "A" [ (1,3) }
2:
1:
>LIST LIST+ PUT GET PUTI GETI
```

The index on Level 2 has been incremented to a value of 2, and Level 1 now contains { 7 }.

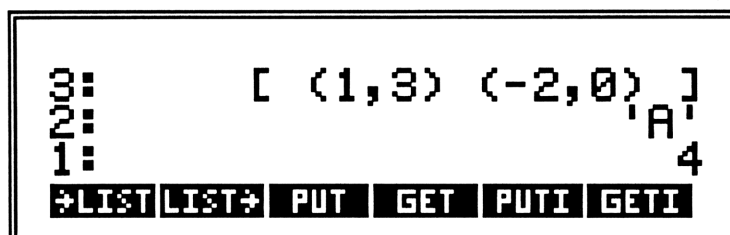
Notice that the index for a list is a real number. The index for an array is typically a list because arrays may have *either* one or two dimensions (a vector is the one-dimensional version) and thus one or two indices. But a list has only one dimension, so a single real number is used for an index.

While The GETting Is Good: Press **DROP** **GETI**. The index is incremented and Level 1 contains the character string "A", which is the second element in the list.

Press **DROP** **GETI**. The index is incremented and Level 1 contains the complex vector (element number 3 in the list).

Press **DROP** **GETI**. The index is incremented and Level 1 contains the name 'A'. Notice that the name has its single quotation marks when it's on the stack by itself – but not when it's a member of a list.

Secession From The Union: Press **DROP****DROP** **LIST** to decompose the list.



The number of list elements is in Level 1, all ready in case you want to rebuild the list by pressing **→LIST**.

Notice that, because the index/list-length is a real number, the list elements now "stacked up" become easily accessible to several other commands, particularly the stack commands **ROLL**, **ROLLD**, **DUPN**, and **DROPN** (you may recall that **→LIST** and **LIST→** are also available in the **STACK** menu).

It's quite feasible, therefore, to do some substantial list operations by decomposing the list, manipulating the elements in the stack, and then recomposing the list.

Reconstruction: Press **→LIST**. Very convenient, no?

As with character strings, there's a certain analogy between numerical addition and the addition of an element to a list. If you have two lists at Levels 1 and 2, you can "add" them to get one list with all the elements of the original two.

Like So: Stack up two lists by pressing \leftarrow { 1 2 3 }.

```

1: { { 7 } "A"
    [ { 1,3 } (-2,0) ]
    A }
{ 123 }

```

Now press \oplus to combine them:

```

2:
1: { { 7 } "A"
    [ { 1,3 } (-2,0) ]
    A 123 }

```

The contents (123) of the list formerly at Level 1 have been added to the *end* of the list that was in Level 2 – the pattern for "list addition."

Remember that a list may contain *any* number of *any* data object. This makes the list the most general purpose data object available to you. In keeping with this idea of generality, the HP-28S doesn't restrict you by imposing too many list-specific commands (note that the LIST menu isn't all that whopping huge).

But the convenience of the ability to decompose lists onto the stack, manipulate the various elements, and then restore the lists allows you to dream up new commands to manipulate them however you like!

List Lessons

1. What's the difference between { 1 2 3 4 } and [1 2 3 4]? How would you convert between one and the other?
2. Since a vector's components are limited to either complex or real number objects, how might you "represent" a "vector" whose elements are the *name objects* **I**, **J** and **K**?
3. You can add elements to a list using \oplus , but how might you *delete* the last element? The first element?
4. Say that you work with lumber. You therefore work with "lumber numbers" in terms of feet, inches, and fractions of inches. What are some ways in which you might use a list on the HP-28S to meaningfully represent six feet, five and three-quarters inches?
5. You want to record the height and weight of a number of people so that you can later do some statistical analyses on them. How might you use lists to store/organize this information?

List Lessons Learned

1. `{ 1 2 3 4 }` is a four-element list containing the real numbers 1 through 4. `[1 2 3 4]` is a four-element real vector containing the numbers 1 through 4.

Convert from the list to the vector: `▢ LIST LIST+ 1 →LIST ▢ ARRAY →ARRY`

Convert from the vector to the list: `▢ ARRAY →ARRY+ ▢ LIST LIST+ DROP →LIST`

Each of these decomposes the original object, alters the index so that it matches the new object, and forms the new object out of the items and the index on the stack. (Since `→ARRY` will also take a real number as the index in making a vector, converting the index to a list is not really necessary.)

2. `{ I J K }`

3. `▢ LIST LIST+ ▢ SWAP DROP 1 - →LIST` or
`▢ LIST ENTER NEXT SIZE 1 - 1 ▢ SWAP SUB` (To delete the last element);
`▢ LIST LIST+ 1 - →LIST ▢ SWAP DROP` or
`▢ LIST ENTER NEXT SIZE 2 ▢ SWAP SUB` (This deletes the first element).

4. `{ 6 5.75 }` or
`{ 6 "feet" 5 "inch" 3 "quarters inch" }` or
`{ { 6 "ft" } { 5 "in" } { 3 "/4" } }`, et cetera.

5. `{ { "HEIGHT" "WEIGHT" } { 6.2 210 } { 5.9 170 } { 5.7 135 } }`, for example.

Procedures: (a) Postfix Programs

Programs are objects just like any other object discussed up to now. They can be put on the stack, associated with a name, and put into a list. In fact, programs are merely one specialized version of a generalized one-dimensional list of objects. But to signify their special differences, programs are represented and treated as an object type in its own right.

A program is indeed entered and displayed as a one-dimensional list of objects, separated by delimiters in the usual manner, but it is bracketed between French quotation marks (⌘ and ⌘) to distinguish it from a generic list.

As data objects used to *store* information, programs are relatively useless. Except in the crudest manual sense – through the command line – you can't add elements to a program, nor can you break it into its components nor build it from its components. You can't perform math on it nor can you convert it to another object type.








Well then, what good is it?

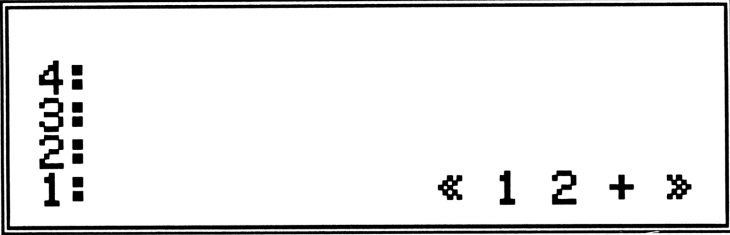
A program is a *dynamic* object, not in the sense that it changes itself or can be changed by any other object, but rather that it does things; *it causes changes to other objects*.

You've already been introduced to the idea of evaluation with names. Remember when a name is evaluated, how the HP-28S actually produces the value of the object associated with that name?

Well, programs can also be evaluated (indeed, that's their purpose in life). And when evaluated, *a program sequentially evaluates its elements*.

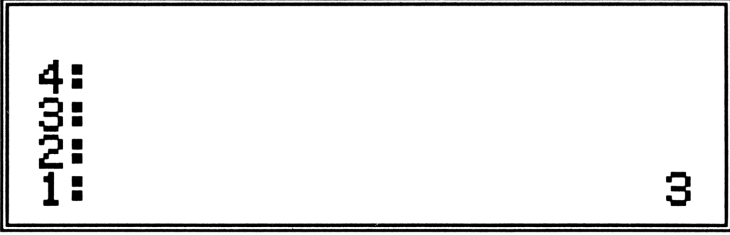
The best way to see this is with an example.

Watch The Birdie: Press    (notice that alpha mode is automatically activated) , ,  . Result:




A calculator display showing a list of four items on the left, labeled 1 through 4 from bottom to top. On the right, the expression « 1 2 + » is displayed.

Now press .



A calculator display showing the same list of four items on the left. On the right, the number 3 is displayed, indicating the result of the evaluation.

This should look vaguely familiar. In learning about strings, you saw how you could put a similar sequence of objects into a string and then evaluate them by using STR→.

This might also look familiar from your earlier work with the command line. Remember when you lined up several items on the command line (separating each with a legal delimiter) and then pressed , how this "posted" all of them at once, one after another? If they were numbers or other data, they went onto the stack. If they were commands, they were executed immediately, right?

So there are actually three roughly identical methods of doing this same thing:

Command Line:	<code>1</code> <code>,</code> <code>2</code> <code>,</code> <code>+</code>
String:	<code>"1 2 + "</code> <code>STR→</code>
Program:	<code>« 1 2 + »</code> <code>EVAL</code>

Any operation that can be performed through the evaluation of any one of these expressions can also be performed by the others. So how are they different? Funny you should ask....

The Command Line:

The command line is interactive and immediate. Once you've keyed in the string of characters representing objects, `ENTER` evaluates them. Thus, you have immediate evaluation *and* immediate error detection. The HP-28S tells if you've made detectable errors in your typing, and then you get immediate feedback on execution errors by pressing `ENTER`.

Strings:

In sharp contrast to the command line, "stringed" collections of evaluable objects are non-interactive and non-immediate – but portable. They are non-interactive because a string may contain any character. The HP-28S won't look at a string for syntax errors. Therefore, you won't know until you convert the collection into non-string form whether or not the string contained errors.

It's also not as easy to evaluate a string as it is a program or a command line. For a string, you must explicitly use `STR→`. And if you have a name associated with it, you must first evaluate the name. But strings do have advantages over programs: they can use less memory and can be modified by other commands.

Programs:

So how do program objects stack up beside those other two ways to collect evaluable sequences of objects? A little of this and a little of that: Programs are somewhat interactive, non-immediate, and portable.

They are somewhat interactive because entry errors are detected just as when you use the command line. This happens because a program, though not immediately evaluated, is immediately scanned and turned into objects for storage. And during this scan, certain input errors can be detected.

A program is portable because it's an information object; you can put it onto the stack and "store" it in a name. Its association with a name makes it very convenient to use, because, as you remember, typing an unquoted variable name evaluates the name and all other objects the name points to.

Therefore, named programs are virtually identical to HP-28S system commands.

Program Problems

1. Rewrite the solutions to problem 1 from page 183 as postfix programs. Name the first $L \rightarrow V$ and the second $V \rightarrow L$.
2. Use $L \rightarrow V$ to convert $\{ \ 0 \ \}$, $\{ \ 1 \ 2 \ 3 \ \}$ and $\{ \ 1 \ 0 \ (1,0) \ \}$ to vectors.
3. Try to convert $\{ \ \}$ to a vector using $L \rightarrow V$. What happens? Why?
4. Using $L \rightarrow V$ and $V \rightarrow L$, write a program named LADD ("List ADD") that will add two lists together such that the resultant list's elements are the sums of the corresponding elements of the original two lists.
5. Use the program from problem 4 to add the following:
 - a. $\{ \ 1 \ 2 \ 3 \ 4 \ \} \ \{ \ 5 \ 6 \ 7 \ 8 \ \}$
 - b. $\{ \ (1,1) \ (-3,4) \ \} \ \{ -5.4 \ (4.3,-8.1) \ \}$
 - c. $\{ \ 9 \ 6 \ 8 \ \} \ \{ \ 1 \ 1 \ \}$
 - d. $\{ \ [\ 1 \ 2 \] \ [\ 3 \ 4 \] \ \}$
 $\{ \ [\ -3 \ 1 \] \ [\ 6 \ 9 \] \ \}$

Why do c and d fail?

Program Problem Solutions

1. \ll \blacksquare LIST LIST \div 1 \div LIST \blacksquare ARRAY \div $\overline{\text{ARRY}}$ ENTER
 'L \blacksquare \rightarrow V STO;

\ll \blacksquare ARRAY $\overline{\text{ARRY}}$ \div \blacksquare LIST LIST \div DROP \div LIST ENTER
 'V \blacksquare \rightarrow L STO.

It's certainly not very hard to translate postfix keystroke sequences into postfix program objects, is it?

2. USER { 0 \blacksquare L \div $\overline{\text{U}}$	Result = [0];
{ 1 , 2 , 3 \blacksquare L \div $\overline{\text{U}}$	Result = [1 2 3];
{ 1 , 0 (1 , 0 \blacksquare L \div $\overline{\text{U}}$	Result = [(1,0) (0,0) (1,0)].

Notice – as you know – that a *list* will tolerate components of differing types, but a *vector* will not. Therefore, you get a vector with either all real or all complex components.

3. Here's what you do: **{ L→V**. And here's what you get:



Why? To find out, mentally "walk through" the program $L \rightarrow V$:

$LIST \rightarrow$ puts the contents of a list onto the stack, followed by the number of elements in Level 1. **{ }** has no elements; its size is 0. Therefore, **{ }** is replaced with **0**.

$1\ LIST \rightarrow$ then makes a 1-element list from the **0**, thus preparing the stack for the use of $\rightarrow ARRAY$. So at this point, **{ 0 }** is left on the stack at Level 1.

Then $\rightarrow ARRAY$ tries to use this index to build a vector, but there's no such thing as a zero-length vector on the HP-28S, so the error is generated and the stack is left as it was when the error occurred. Notice that the error message tells you where the problem was.

4. **« L→V SWAP L→V + V→L »** would be one reasonable approach. After all, you can't sum lists directly – but you can sum vectors!

Here are the keystrokes to create the program and name it LADD:

⏪ USER L→V SWAP L→V + V→L ENTER 'LADD' STO

5. a. { 1 , 2 , 3 , 4 ENTER { 5 , 6 , 7 , 8 LADD

Result = { 6 8 10 12 }

b. { (1 , 1) (3 CHS , 4 ENTER { 5 • 4 CHS (4 • 3 , 8 • 1 CHS LADD

Result = { (-4.4, 1) (1.3, -4.1) }

c. { 9 , 6 , 8 ENTER { 1 , 1 LADD

Result: The two lists are of different lengths and are therefore converted to two vectors of different length. The **Invalid Dimension** error occurs because the HP-28S can't add two vectors that aren't the same length. Note that the vectors are left on the stack after the error is reported.

d. { [1 , 2 [3 , 4 ENTER [3 CHS , 1 [6 , 9 LADD

Result: **Bad Argument Type** occurs when LADD attempts to make a vector out of vectors (i.e., when L→V tries to perform →ARRY on a stack full of vectors).

Procedures: (b) Algebraic Expressions

Algebraic expressions are exactly like programs, only different.

They are programs whose syntax is algebraic (i.e. operand-operator-operand) rather than postfix.

Algebraic expressions are represented by bracketing a *syntactically correct* list of *algebraically meaningful* objects within single quotation marks ('). Since those single quotation marks also apply to names, this explains why you can't have a name that looks like a syntactically correct (and therefore executable) expression (see page 166).

For example, compare `1 2 +` and `'1+2'`.

Both evaluate to `3`.

The major difference between them is the *order of the objects within them*. The ordering within a program is postfix (i.e. just like the stack – with the operands first and the operator last).

By contrast, the ordering of an algebraic expression is (astonishingly) algebraic.

Notice that they're called *algebraic* objects, not simply *mathematical* objects. The reason is that algebraic objects may contain name objects (remember them?), thus giving the expressions the classical form of algebra with one or more *variables*.

Suppose you want to solve quadratic equations. The form of one solution is:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

You want to create an algebraic object which, when evaluated, will give you x .

No Problem:

Here are the key-strokes to do this, along with a play-by-play analysis.

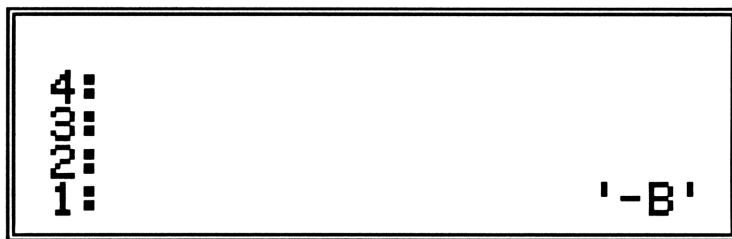
First, press   .

Here, you're clearing the stack and disabling the menu. This is just "clearing the decks for action."

Next, press       .

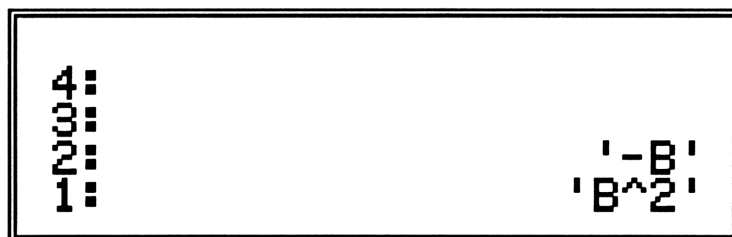
Why? Well, you want to use the names 'A', 'B', and 'C' (you may prefer 'a', 'b', and 'c', but lower-case is somewhat cumbersome). So you PURGE these names, dissociating them from any objects that they might otherwise belong to. This allows you to key in names without single quotation marks – not necessary, but convenient.

Now you start to build your algebraic expression. Press **B****ENTER****CHS**. Here's what you have so far:



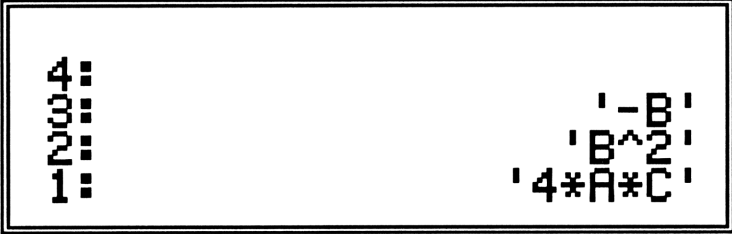
You've keyed in "negative B." The CHS placed a negative sign in front of the **B**. Notice that before you did the **CHS**, **B** was a *name* sitting on the stack, not an algebraic expression. But since the single quotation marks can mean either object type, whenever you perform any allowable mathematical operation – such as CHS – on a name, *the operation's effect will be to build an algebraic object*.

Next, press **B****ENTER****2****■****^**



Here you key in **B** again and square it by raising it to the second power (you could have used **■****x²** and the result would have been **'SQ(B)'**. The two versions of B^2 evaluate to be the same thing; they just look different). Notice that circumflex, **^**. Because the HP-28S can't display superscripts, it uses the circumflex to indicate "raising to a power."

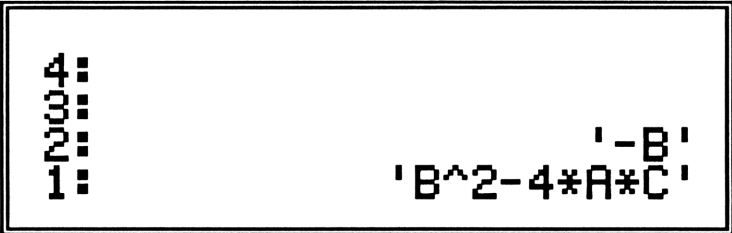
Now press $\boxed{4}\boxed{\text{ENTER}}\boxed{\text{A}}\boxed{\times}\boxed{\text{C}}\boxed{\times}$



4:
3:
2:
1:
'-B'
'B^2'
'4*A*C'

You key in **4** and multiply it by **A** and **C**. Notice that the result is **'4*A*C'** and not **'4AC'**. If the HP-28S didn't use ***** to indicate multiplication, neither it nor you could distinguish $A \times B$ (**'A*B'**) from the name AB (**'AB'**). In written algebra, you can omit the multiplication sign (it's implied) because you typically use only single character variables, such as x , y , and z .

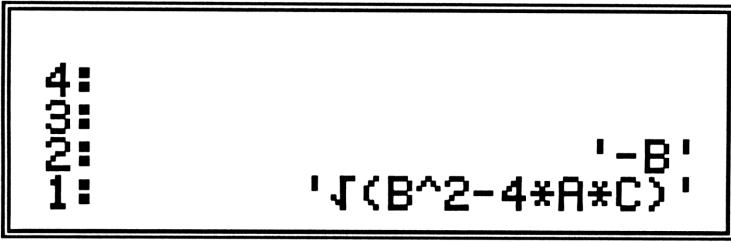
Then: $\boxed{-}$



4:
3:
2:
1:
'-B'
'B^2-4*A*C'


You subtract **'4*A*C'** from **'B^2'**. Notice again that when objects – even algebraic objects – are on the stack, you use *postfix* logic commands. Therefore you pressed $\boxed{-}$ *after* the two arguments were on the stack. The HP-28S *then* interprets that arithmetic command in terms of the objects on which it must act. When acting upon two algebraic expressions, it just so happens that $\boxed{-}$ means to combine them into one, with a minus sign embedded in the resulting expression.

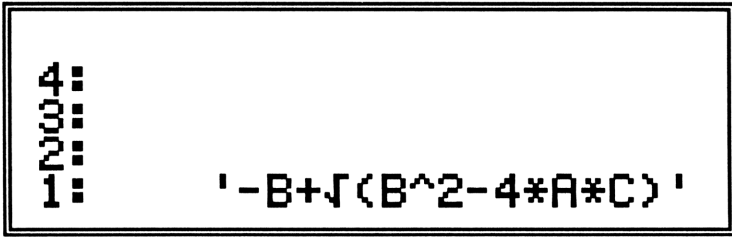
Next step: 



The display shows a list of four lines. Line 1 contains the expression $\sqrt{B^2 - 4AC}$. Line 2 contains $-B$. Line 3 is blank. Line 4 is blank.





You take the square root of ' $B^2 - 4AC$ '. Notice the parentheses. Again, because the HP-28S's display is limited, it cannot draw the radical so that it includes the entire expression under it. Instead, the radical sign is represented as a mathematical function, like $f(x)$ (read "f of x"), and in the same way, parentheses are used to enclose the argument. $\sqrt{(x)}$ is therefore "square root of x."

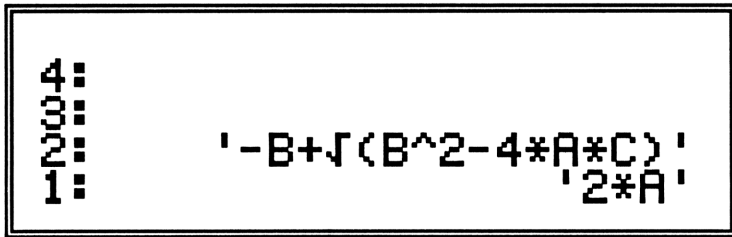
Then press 



The display shows a list of four lines. Line 1 contains the expression $-B + \sqrt{B^2 - 4AC}$. Line 2 is blank. Line 3 is blank. Line 4 is blank.

You add ' $-B$ ' to ' $\sqrt{B^2 - 4AC}$ '. No surprises, right?

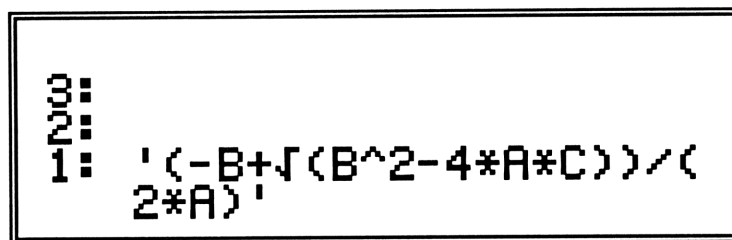
Keep going:    



The display shows a list of four lines. Line 1 contains the expression $2 \times (-B + \sqrt{B^2 - 4AC})$. Line 2 is blank. Line 3 is blank. Line 4 is blank.

You multiply **2** by **A**. Again, no surprises.

At last: $\boxed{\div}$



3:
2:
1: '(-B+√(B^2-4*A*C))/(
2*A)'

You have divided ' $-B+\sqrt{B^2-4AC}$ ' by ' $2A$ '.

Notice the extra parentheses. Since the display's limited capacity forces the division sign onto the same line with the rest of the expression, it needs a way to indicate what is divided by what. That's where the parentheses come in. They group the things that are in the numerator, ' $-B+\sqrt{B^2-4AC}$ ' and the things in the denominator, ' $2A$ '.

If these extra parentheses weren't there, the order of evaluation of the expression would be different, because in algebraic notation, the convention is that multiplication and division are performed before addition and subtraction.

Notice also that the final object doesn't fit on one line. In multi-line mode (check your MODE menu for this option), an algebraic object in Level 1 will be broken between internal objects and displayed on several lines.

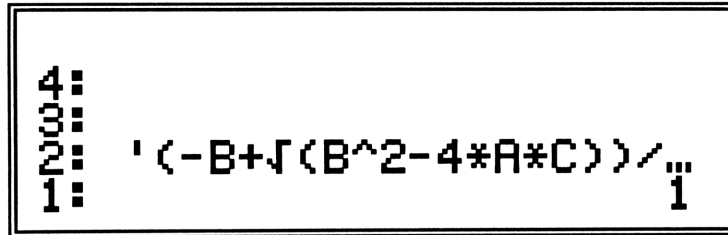
Of course, you could have simply *typed in* the expression above; the result would be the same. It's up to you which you find more convenient.

("Now they tell me.")

Now that you have an algebraic object, what do you do with it?

You use it as a mathematical procedure to solve problems.

Like This: Press **ENTER** **1** **,** **A** **STO** **2** **CHS** **,** **B** **STO** **1** **,** **C** **STO** **EVAL**.



What have you done? You've associated a real number 1 with the name '**A**', a -2 with the name '**B**' and a 1 with the name '**C**' (and you could verify this in your USER menu right?).

Then you've evaluated your algebraic expression, at which time all of the names that have associated objects were replaced by the objects themselves, and all mathematical expressions were performed.

Thus, you get the mathematical result of this expression – for the case where $a=1$, $b=-2$, and $c=1$.

In this case, the result was a real-number object, 1. With other coefficients, it would, of course, be another result, possibly a complex-number object.

So you see the idea behind an algebraic object. It's merely a way to list certain mathematical objects and operations in a collection that is evaluable in algebraic ("left-to-right") notation – rather than in postfix. And even this collecting process can be done with postfix operations on the stack, unless you choose to type in the entire expression manually.

Happily, this notation *can* refer to named objects as its variables and thus it resembles written algebraic logic quite closely. And since so much of symbolic math is represented in algebraic form, these capabilities of the HP-28S open up vast horizons for you.

Algebraic Aptitude Test

1. Build an algebraic object for the expression $x^2 - 2x + 1$ and evaluate it for:

- a. $x = 1$
- b. $x = -2$
- c. $x = (2,3)$
- d. $x = \sqrt{2}$.

2. Evaluate the expression ' $A+B*C-D$ ' for:

- a. $A = 1, B = -2, C = 3, D = 4$
- b. $A = (1,2), B = (-2,-2), C = (.5,1.3), D = (104,.2)$
- c. $A = [1 \ 2], B = -2, C = [.5 \ 1.3], D = [104 \ .2]$
- d. $A = 14_{10}, B = 20_8, C = 34_{16}, D = 101_2$

3. Evaluate the expression ' $2*X+Y$ ' for:

- a. $X = -2Y$
- b. $Y = -2X$
- c. $X = T, Y = T - 1$
- d. $X = Z - 3Y, Y = Y - 3Z$

4. Evaluate the expression ' $A+B+C+D$ ' for:

$A = \text{"THIS "}, B = \text{"IS "}, C = \text{"ODD"}, \text{ and } D = \text{"."}$

A.A.T. Scores

1. $\boxed{1}\boxed{X}\boxed{\text{ENTER}}\boxed{2}\boxed{\blacksquare}\boxed{\wedge}\boxed{1}\boxed{X}\boxed{\text{ENTER}}\boxed{2}\boxed{X}\boxed{-}\boxed{1}\boxed{+}\boxed{1}\boxed{EQ}\boxed{\text{STO}}\boxed{\text{USER}}$

a. $\boxed{1}\boxed{1}\boxed{X}\boxed{\text{STO}}\boxed{\blacksquare}\boxed{EQ}\boxed{\text{EVAL}};$ Result = $\boxed{0}$

Notice that invoking the (unquoted) name of an algebraic expression does not immediately evaluate it numerically; rather, it is left on the stack in its symbolic form.

This is an exception to the normal immediate-EVALuation rule for other named objects, but there's a good reason for this: Often the forms of the expression itself – and the possibilities for modifying those forms – are as much of interest to you as the numeric results of "plugging in" variables. Therefore, the HP-28S asks you to confirm that you are indeed interested in a numeric answer; you must explicitly use EVAL to tell it so.

b. $\boxed{2}\boxed{\text{CHS}}\boxed{1}\boxed{X}\boxed{\text{STO}}\boxed{\blacksquare}\boxed{EQ}\boxed{\text{EVAL}};$ Result = $\boxed{9}$

c. $\boxed{(}\boxed{2}\boxed{,}\boxed{3}\boxed{\blacksquare}\boxed{)}\boxed{1}\boxed{X}\boxed{\text{STO}}\boxed{\blacksquare}\boxed{EQ}\boxed{\text{EVAL}};$ Result = $\boxed{(-8,6)}$

d. $\boxed{2}\boxed{\blacksquare}\boxed{\sqrt{x}}\boxed{1}\boxed{X}\boxed{\text{STO}}\boxed{\blacksquare}\boxed{EQ}\boxed{\text{EVAL}};$ Result = $\boxed{.17157287525}$

2. `'A+B×C-D` `ENTER` `'EQ` `STO`

A reminder: According to conventional algebraic notation, you don't necessarily evaluate an expression strictly from left to right. Instead, your evaluation order is based upon the priority of various operators.

Thus, since multiplication and division have a higher priority than addition and subtraction, the expression $A + B \times C - D$ means this: $A + (B \times C) - D$. Only *after* performing the multiplication are all remaining operations of the same priority; *then* the evaluation proceeds from left to right.

So the fact that you use no parentheses in keying in this expression means that you, too, are reading it and understanding it not just from left to right but also according to this hierarchy of priorities (which is called infix notation, by the way).

a. `[1]` `'A` `STO` `2` `CHS` `'B` `STO` `3` `'C` `STO` `4` `'D` `STO` `EQ` `EVAL`;

Result = **-9**

b. `(` `[1]` `,` `2` `ENTER` `'A` `STO` `(` `2` `,` `2` `ENTER` `CHS` `'B` `STO` `(` `.` `5` `,` `1` `.` `3` `ENTER` `'C` `STO` `(` `104` `,` `.` `2` `ENTER` `'D` `STO` `EQ` `EVAL`;

Result = **(-101.4, -1.8)**

c. `[` `[1]` `,` `2` `ENTER` `'A` `STO` `2` `CHS` `'B` `STO` `[` `.` `5` `,` `1` `.` `3` `ENTER` `'C` `STO` `[` `104` `,` `.` `2` `ENTER` `'D` `STO` `EQ` `EVAL`;

Result = **[-104 -.8]**

d. `BINARY` `DEC` `#14` `'A` `STO` `OCT` `#20` `'B` `STO` `HEX` `#34` `'C` `STO` `BIN` `#101` `'D` `STO` `USER` `EQ` `EVAL`;

Result = **# 1101001001b** (because you're still in BIN mode).

3. `'2XX+Y'ENTER 'EQSTO`

a. `'-2XY'ENTER 'XSTO 'Y'PURGE EQ EVAL;`

Result = `'2*(-2*Y)+Y'`;

`COLCTENTER`; Result = `'-(3*Y)'`

Notice how the COLCT (COLleCT) command affects an algebraic result. It collects like terms and attempts to reduce the expression to lower terms.

b. `'-2XX'ENTER 'YSTO 'X'PURGE EQ EVAL;`

Result = `'2*X+-2*X'`;

`COLCTENTER`; Result = `0`

c. `'T'ENTER 'XSTO 'T-1'ENTER 'YSTO EQ EVAL;`

Result = `'2*T+(T-1)'`;

`COLCTENTER`; `'-1+3*T'`

d. `'Z-3XY'ENTER 'XSTO 'Y-3XZ'ENTER 'YSTO EQ EVAL;`

Result = `'2*(Z-3*Y)+(Y-3*Z)'`;

Now use EVAL and COLCT a couple of times on this result. Each time you use EVAL, the expression becomes more complex, because Y is replaced with a more complex expression containing Y.

4. `'A+B+C+D'ENTER '"THIS SPACE'ENTER 'ASTO '"IS SPACE`

`ENTER 'BSTO '"ODD'ENTER 'CSTO '"•'ENTER 'DSTO EVAL`

Result = `"THIS IS ODD."`

This works only because addition is defined on character strings.

Procedures: (c) User-Defined Functions

You've seen how algebraic objects can be used to solve problems. You simply create an algebraic object of the proper form and assign values to the names in it. Then, when you evaluate this algebraic object, it combines the values represented as the expression specifies, and you get a result. Fine and dandy.

But if the algebraic object contains a lot of named objects, then associating (STOring) the data objects with their names can be a lengthy and therefore error-prone process.

One way around this – a method that stream-lines the use of algebraic objects – is the *User-Defined Function*.

Take, for example, the old standby: one of the two roots of a quadratic equation (an algebraic object or expression *can generate only one result*, and therefore you can't get both roots at once).

You've already generated an algebraic expression (pages 194-198) to do this. And you've seen that in order to use this algebraic object to solve for numerical roots, you must assign values to the names – the *variables* – in the algebraic expression.

So you did that (e.g.,

1	'	A	S	T	O
---	---	---	---	---	---

2	C	H	S	'	B	S	T	O
---	---	---	---	---	---	---	---	---

1	'	C	S	T	O
---	---	---	---	---	---

) and then EVALuated the expression.

But Try This: Create the following object:

⌘ → a b c '(-b-√(b^2-4*a*c))/(2*a)' ⌘

And give a name to this odd-looking hybrid (it's a sort of cross between a postfix program and an algebraic object).

For this example, use the name '**RUTE**' (there's an HP-28S system command already called ROOT, so you can't use that); type: **[R][U][T][E][S][T][O]**.

Now type **[USER][1][,][2][CHS][,][1][RUTE]**.

What happened?

RUTE took the three objects *off the stack* and used them in its algebraic expression(!). The result, **1**, was left on the stack.

That's a real step saver, eh? But how did RUTE do it? Look at the object itself, to see if you can surmise some things from what you already know....

First, the French quotation marks (❖) make it look similar to a postfix program. And postfix programs have the feature that when evaluated, they evaluate their contents *element by element, from left to right*. This thing should do the same.

So, going from left to right, the first few symbols, ➔ **a b c** , seem to be something new. But see if you can guess what they mean after looking at the rest of the object.

Skipping therefore to the next (and last) object, you find it to be an algebraic object. This is an algebraic object inside *another kind of program*, which is perfectly "legal" and often very useful.

So, the question is: Where does this user-defined function get the values for its variables, a, b, and c? After all, you certainly haven't created such names nor STORed any values into them. "Aha! The answer must have something to do with the ➔ preceding the list of variable names."

That's right: *This ➔ symbol inside a postfix program associates objects on the stack to whatever names follow the ➔.*

In other words, the bottom three stack entries will be associated with (stored into) **a**, **b**, and **c**. And pay close attention to the order: **a** will be associated with Level 3, **b** with Level 2 and **c** with Level 1 (because you would naturally load the stack in the order a [ENTER] b [ENTER] c [ENTER]). So this is the way your listing of the variable names will be interpreted also. Makes sense, right?).

Of course, once these *stack* values have been associated with the names, the algebraic object is evaluated in the normal algebraic manner.

So that's how the U.D.F. works. It's truly a function, because it's an algebraic expression (which, as you'll recall, can produce exactly one result), but instead of looking into your USER menu collection of named objects to find its variables, it pulls objects off the stack – in the quantity and order you specify with the \rightarrow in the function definition.

Once again, you can see the hybrid nature of the U.D.F. It evaluates like an algebraic object, but it uses values from off the stack like a postfix program.

And here's an added bonus: U.D.F. names that are created and associated with \rightarrow are *temporary*. In other words, they are created at the beginning of the program and PURGE'd at the end.

Not only that, they are created in such a way that they don't conflict with other names that might already exist and have the same spellings. So if you had a list or some other object already STORed under the name of \rightarrow , and you nevertheless evaluated RUTE as written (with its *temporary* \rightarrow \rightarrow), the contents of your named object would *not* be changed!

User-Defined Function Fun

1. Build a user-defined function for the expression $x^2 - 2x + 1$ and evaluate it at:
 - a. $x = 1$
 - b. $x = -2$
 - c. $x = (2,3)$
 - d. $x = \sqrt{2}$.

2. Build and evaluate a U.D.F. for the expression ' $A+B*C-D$ ' when:
 - a. $A = 1, B = -2, C = 3, D = 4$
 - b. $A = (1,2), B = (-2,-2), C = (.5,1.3), D = (104,.2)$
 - c. $A = [1 \ 2], B = -2, C = [.5 \ 1.3], D = [104 \ .2]$

3. Evaluate the expression ' $2*X+Y$ ' using a U.D.F. for:
 - a. $X = -2Y$
 - b. $Y = -2X$
 - c. $X = T, Y = T - 1$
 - d. $X = Z - 3Y, Y = Y - 3Z$

4. Define the U.D.F. $\text{X} \rightarrow \text{X} + 1$ and name it INCR (increment). Then create both a postfix program and an algebraic object that use INCR to sum a number and its increment (i.e., $X + \text{INCR}(X)$).

U. D. F. F. Consequences

1. $\ll \blacksquare \rightarrow X \blacksquare X \blacksquare \wedge 2 - 2 X X + 1 \text{ ENTER } \blacksquare Q \text{ STO } \text{USER}$

- a. $1 \blacksquare \blacksquare$; Result = 0
- b. $2 \text{ CHS } \blacksquare \blacksquare$; Result = 9
- c. $(2 , 3 \blacksquare \blacksquare$; Result = (-8, 6)
- d. $2 \blacksquare \sqrt{x} \blacksquare \blacksquare$; Result = .17157287525

Note that, unlike a plain algebraic expression, a U.D.F. does follow the rule for immediate evaluation – just like a postfix program and the other objects; if you invoke its (unquoted) name, it will produce its ultimate result right away, without stopping to let you see its algebraic form.

2. $\ll \blacksquare \rightarrow A , B , C , D \blacksquare A + B X C - D \text{ ENTER } \blacksquare Q \text{ STO } \text{USER}$

- a. $1 , 2 \text{ CHS } , 3 , 4 \blacksquare \blacksquare$; Result = -9
- b. $(1 , 2 \text{ ENTER } (2 , 2 \text{ ENTER } \text{CHS } (. 5 , 1 . 3 \text{ ENTER } (1 0 4 , . 2 \blacksquare \blacksquare$; Result = (-101.4, -1.8)
- c. $[1 , 2 \text{ ENTER } 2 \text{ CHS } \text{ENTER } [. 5 , 1 . 3 \text{ ENTER } [1 0 4 , . 2 \blacksquare \blacksquare$; Result = [-104 - .8]

3. \ll \rightarrow X SPACE Y ' 2 X X + Y ENTER ' Q STO

a. ' Y ENTER 2 X CHS ' Y ENTER \rightarrow ; Result = ' 2 * (- (Y * 2)) + Y '

b. ' X ENTER ENTER 2 X CHS \rightarrow ; Result = ' 2 * X - X * 2 '

c. ' T ENTER ENTER 1 - \rightarrow ; Result = ' 2 * T + (T - 1) '

d. ' Z - 3 X Y ENTER ' Y - 3 X Z ENTER \rightarrow ;

Result = ' 2 * (Z - 3 * Y) + (Y - 3 * Z) '

4. \ll \rightarrow X ' X + 1 ENTER ' INCR STO

\ll DUP , INCR , + ENTER

' X + INCR (X ENTER

which is: \ll DUP INCR + \gg

which is: ' X + INCR (X '

Note the different form that INCR takes in a postfix program and an algebraic object. This is another special advantage of User-Defined Functions – the fact that you may use them in algebraic objects in the form you normally expect to see for mathematical functions: $f(x)$ or $f(x,y)$, etc.

Naturally, you also use this conventional form when you invoke standard HP-28S functions – like LOG and SIN – in an algebraic expression.

Directories

A directory is an organizational thing.

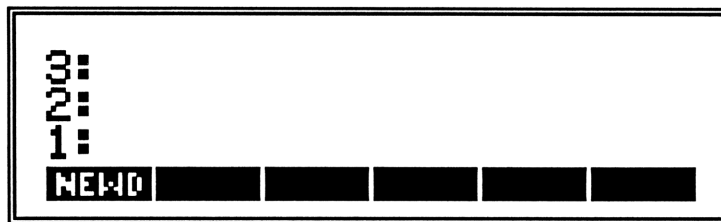
Like other directories that you may have already seen, HP-28S directories exist to form collections of related names or of names of related things. In essence, the HP-28S *directory* is a *special purpose list-object whose contents can only be name-objects*.

Consider this: Everything you create that doesn't "live" on the stack must be given a name; otherwise you wouldn't have access to it. Every named thing must live somewhere. That somewhere is a directory.

Look at the USER menu (i.e., press **USER**). What you see is a menu. What you see is also a directory. In a sense, if you aren't already in another directory, this is *the* directory, the *main* directory, or the HOME directory, the current "living quarters" of all named objects you've created up to now. To get them there, you didn't need to do anything special. When you associated an object with a name (via STO), that pair of objects was automatically placed in the current directory.

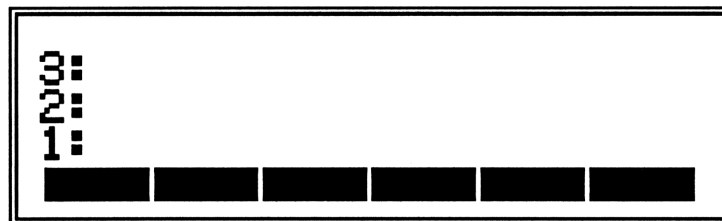
When you activate the USER menu, you are actually selecting a menu whose content is the current directory.

But Try This: Press **MEMORY** **HOME** **NEWD** **CRDIR** **USER**.



You selected the MEMORY menu, where most of the directory commands are. Then you made sure you were in the HOME directory (the main one), typed in a name, and used CRDIR to CReate a DIRection with that name. The name of the new directory is now in the USER menu.

Next: Press **NEWD**.



What happened? The USER menu is empty! Why? Well, you just moved into the NEWD directory and since it's new, it's empty. There are no names in NEWD, and since NEWD is separate from HOME (the name of the main or default directory), you can't see any of the HOME names.

You see the idea? NEWD is a directory just like HOME, except that the name NEWD lives in the HOME directory. Evaluating the name NEWD moves you to a new directory, the one named NEWD.

To verify where you are, type **P****A****T****H****ENTER** (PATH is also in the MEMORY directory). What you get is a list like this: **{ HOME NEWD }**. This is the *path* you took to get where you are. You started in the HOME directory and moved to the NEWD directory – the last name in the list – where you are now.

Another Level: First, press **1****'A** **STO**. **A** is created and placed in the *current* directory: NEWD. Now press **MEMORY****N****E****W** **2****ENTER**
ENTER **CRDIR** **EVAL** **PATH**.



The calculator display shows a list of three items: 3: { HOME NEWD }, 2: { HOME NEWD }, and 1: { HOME NEWD NEW2 }. Below the list, the memory contents are displayed: MEM MENU ORDER PATH HOME CRDIR.

Because you were in the NEWD directory when you created NEW2, the name NEW2 is placed in the NEWD directory, not in the HOME directory. Evaluating '**NEW2**' then moved you to that directory – and PATH tells you how you got there.

Now how do you get out of there (i.e., the directory NEW2)? The easiest way is to invoke HOME. No matter where you are, HOME will always move you to the HOME directory – from which you can then follow a path to any sub-directory.

Do It Now: Press **HOME** **USER**.



The calculator display shows the same list of three items: 3: { HOME NEWD }, 2: { HOME NEWD }, and 1: { HOME NEWD NEW2 }. Below the list, the memory contents are displayed: NEWD followed by four empty slots.

Yes, but what are directories good for?

You can probably see that directories are useful for grouping and separating related things. But this separation goes beyond just organization. *Objects stored in sub-directories are invisible to their parent directories.*


Try This: Press `'A` `ENTER` `ENTER` `PURGE` `RCL`.



A screenshot of a calculator screen with a black background and white text. The text reads: "RCL Error:" on the first line, "Undefined Name" on the second line, "1:" on the third line, and "'A'" on the fourth line. Below the fourth line is a row of six black rectangular boxes, representing the calculator's display buffer.

Although `A` exists (remember you created it in the NEWD directory), RCL can't find it in the current directory. And typing `A` `EVAL` doesn't evaluate `'A'` either, for the same reason.

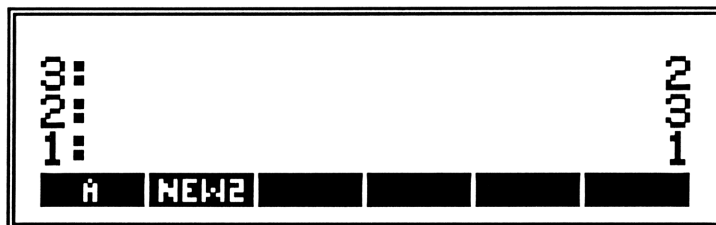
But: Press `NEWD` `NEW2` `EVAL`.



A screenshot of a calculator screen with a black background and white text. The text shows a directory stack: "3: { HOME NEWD }" on the first line, "2: { HOME NEWD NEW2 }" on the second line, and "1: 1" on the third line. Below the third line is a row of six black rectangular boxes, representing the calculator's display buffer.

EVAL found `A` in the NEWD directory even though you're in the NEW2 directory. Names will be found if they exist in directories *above* the one you are in, but *not* in those below.

And Another Thing: Press **H O M E** **ENTER** **3** **'** **A** **STO** **NEW0** **NEW2** **COMMAND**
2 **ENTER** **STO** **CLEAR** **A** **ENTER** **H O M E** **ENTER** **A** **ENTER**
NEW0 **A** **ENTER** .



You now have three different objects named **'A'** in three different directories. What this says is that directories *restrict* the evaluation of their contents.

What this means to you is that directories function best as places where you store special commands or data. They are *localized environments* where the things you do don't have much impact on the rest of the HP-28S world.

This suggests that you should place all of your generally accessible objects in the HOME directory where they will be found from any other directory (because HOME is above all other directories) and create sub-directories for your special activities.

For example, all of your calculation routines can be in HOME and the data for different experimental runs can be in different sub-directories (i.e., **'DATA1'**, **'DATA2'**, etc.).

On the other hand, you *could* create a directory that contains only ARRAY utilities, but if you do you must do most of your ARRAY calculations there or in a directory below it.

Now To Clean Up: Press **H O M E** **ENTER** **' N E W D** **PURGE**.

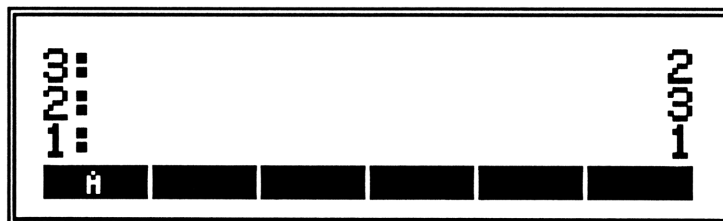


```
PURGE Error:
Non-Empty Directory
1: 'NEWD'
A NEWD
```

The error here is pretty self-explanatory. The HP-28S requires a directory to be empty before it will purge it. The reason behind this is a good one: you wouldn't want to inadvertently purge a directory that has loads of useful objects in it.

Therefore, to purge **'NEWD'**, first you must purge its contents. Now, **'NEWD'** contains the directory **'NEW2'** and the same restriction applies: **'NEW2'** must be empty before you can purge it.

So: Press **NEW0** **NEW2**  **MEMORY** **NEXT** **CLUSR** **ENTER** **ENTER** **EVAL** **CLUSR** **ENTER**
NEXT **HOME** **USER**  **PURGE**.




What you did: First, you moved to '**NEW2**' and emptied it using CLUSR.

Notice that CLUSR (CLear USeR) requires you to press **ENTER** after pressing its key in the MEMORY menu. It does this as a safeguard – so that you have the chance to change your mind about destroying the entire contents of a directory.

Second, you made use of the fact that the name '**NEW0**' was in Level 1 of the stack. You duplicated it and evaluated the Level-1 copy, thus putting you in the '**NEW0**' directory.

Third, you emptied the '**NEW0**' directory with CLUSR. Notice that CLUSR removed '**NEW2**' without complaint, because the directory was empty.

Fourth, you moved HOME and since a copy of the name '**NEW0**' was in Level 1 of the stack, you pressed  **PURGE** to purge the directory.

Now that you've had a smattering of an introduction to directories, try your hand at some ...

Directory Discussion

1. When does evaluating a directory name move you to that directory?
2. Write a postfix program that moves you to the directory immediately above the one you are in. Call it '**DU**' (Directory Up).
3. Say that you want to write a little telephone directory program. You want to be able to key in a person's name (as a character string) and to have the program return the person's telephone number.

You have a lot of names, so you decide to set up a directory called '**PHONES**' in which there are 26 sub-directories named '**A**' through '**Z**' to separate the names into groups and thus (hopefully) speed things up a bit.

Write a little program that takes a character string from Level 1 of the stack and uses its first character to select the appropriate sub-directory.

4. How would you create two names that point to the same directory? In other words, create two different names such that when you evaluate either one, you are moved to the same directory.

Directory Assistance

1. When the directory name exists either (i) in the current directory or (ii) in one of the directories in the current directory's PATH (i.e., in one of the directories *above* the current directory).

2.

<code>⌵ PATH,</code>	Find out where you are.
<code>DUP, SIZE, 1 -</code>	Get the position of the previous directory.
<code>GET,</code>	Get the name from the PATH list.
<code>EVAL</code>	Evaluate the name to go there.
<code>ENTER 'DU STO</code>	

Notice that this routine will fail when you are in the HOME directory because PATH returns { HOME } which has SIZE == 1. GET would be called to get element 0 (SIZE - 1), and there is no such animal.

Notice also that, to be useful, this routine should live in the HOME directory (why?).

Remember: As you noted in the answer to question 1, above, EVALuating the second to the last name in the PATH list will always work (i.e., will always move to another directory) as long as you're not in the HOME directory, of course.

Menus

As you saw in your new-job orientation, "menus are index tabs [handy collections] to your command-card file." They are collections of object names – just as are directories.

Now, as with any good card file, there are ways to create new cards (i.e. new named objects – with STO – which you know already) *and* new index tabs (menus): You create new index tabs (menus) with the MENU command. This command (found in the MEMORY menu) takes a list of objects from Level 1 of the stack and creates a CUSTOM menu for your own use.

However, *the use of a menu is limited by the current directory structure*. You can create a menu containing names of any objects you want (including built-in commands, of course), but you won't be able to invoke a given name on that menu *unless the name is "visible" from the current directory* (i.e. unless it's named there or in a higher directory). Menus may be your index tabs for noting collections of object names, but directories are the file cabinets in which you keep those object names. OK?

Give It a Shot: α { A SPACE B SPACE C R O S S SPACE D O T + - X ENTER
MEMORY MENU.



You now have a menu containing two names and some commonly used array/vector commands.

So Try It Out: Press `[1][2]ENTER` `'A` `STO` `[3]CHS` `'6` `ENTER` `'B` `STO`

`H` `E` `CROSS`

`H` `E` `DOT`

`E` `7` `NEXT` `%`

3:				[0	0	12]
2:								9
1:				[-21	42]
	%							

Ponder what happened for a moment:

You created a list of names. Some of these were just common name objects while others were the names of system commands. You needed to be in alpha mode to do this, so that `[+]`, `[-]`, and `[X]` wouldn't execute immediately.

When you executed MENU the names in the list were placed into a menu in the same order and the menu was displayed.

There are only six "display boxes" for the menu but you had seven items in your list. The seventh item wrapped around and was placed in the next "page." You change levels with `NEXT` and `PREV`, as you'll recall.

The common names behaved just as they normally would in the USER menu, while the command names behaved as any other immediate-execute menu key.

Notice that `[+]`, `[-]` and `[X]` are poor choices for a custom menu since it's easier to press their keys on the keyboard than to hunt them in the menu.

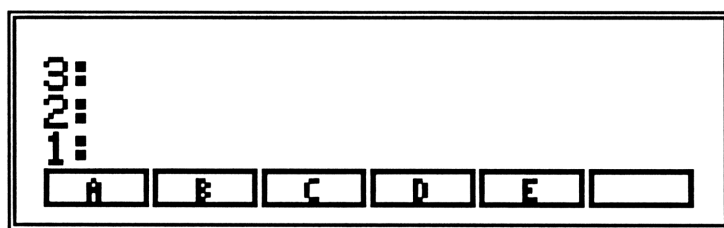
You can even create a menu that contains names that have no *known* associated objects (and pressing the corresponding menu keys will then place the name on the stack).

By contrast, you *cannot* create a *directory* with any unassociated names, because the only way to place a name into a directory *is* to associate the name with an existing object (via STO or CRDIR).

The utility of having your own custom menu like this is fairly obvious: You can create a collection of related or personal-favorite commands so that you can do your work with a minimum of keystrokes. *And...*

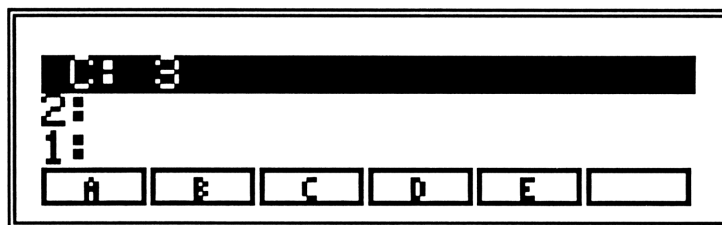
Something Else From the Menu:

Press   STO  SPACE  SPACE  SPACE  SPACE
 ENTER  MENU.






This is just a little something extra that the MENU command gives you. It's a *storage menu*, giving you a convenient method of storing values into names.

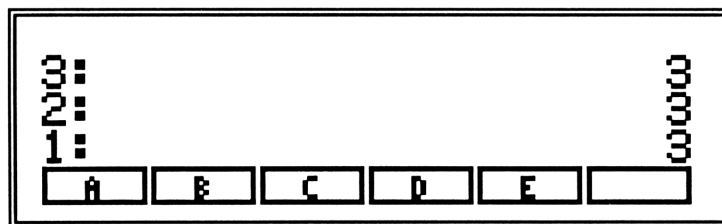
Par Example: Press (1)  (2)  (3) .





You've stored 1 into 'A', 2 into 'B' and 3 into 'C'. A storage menu conveniently shows you the object name and value at the top of the display – in case you've done something wrong.

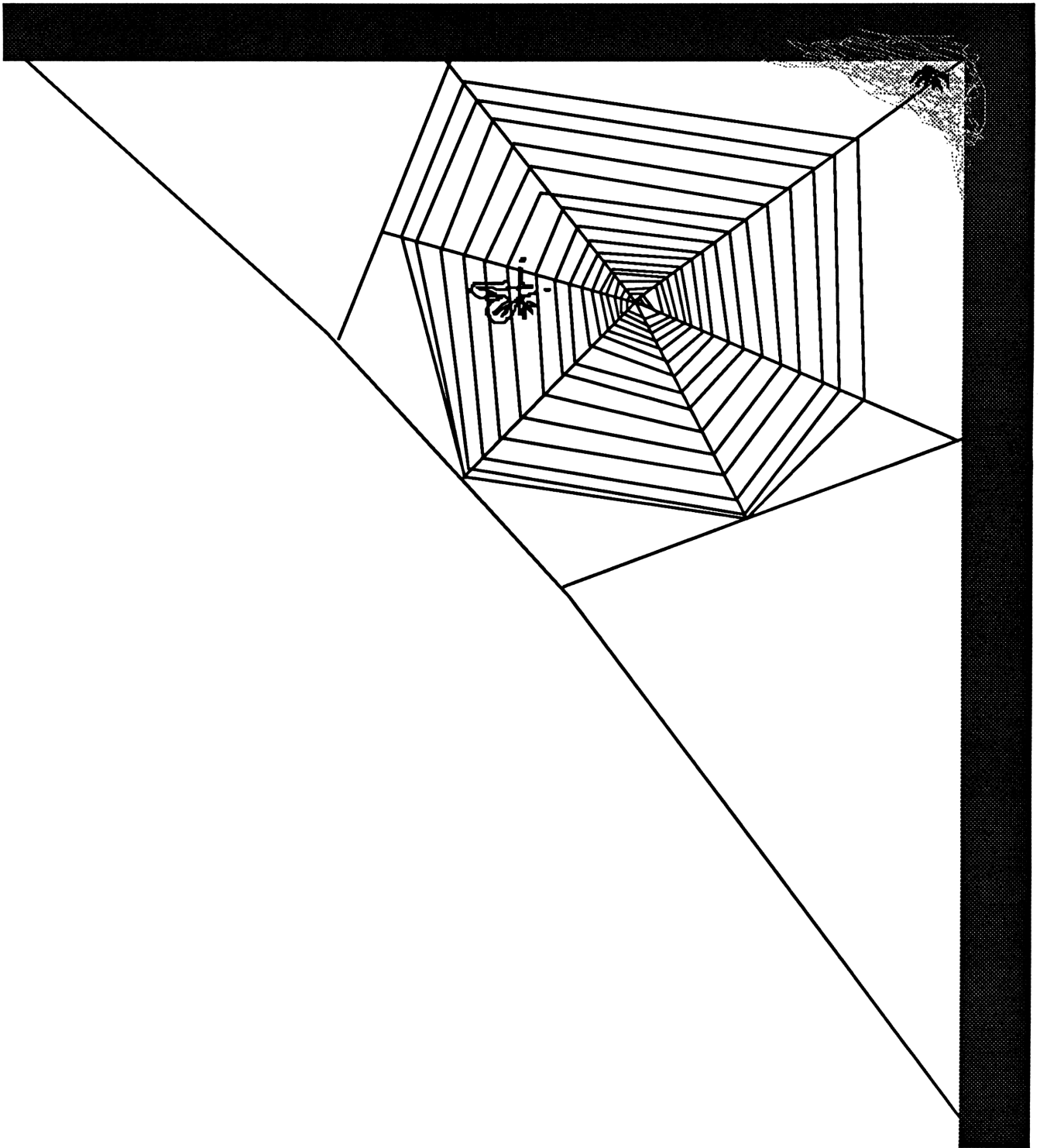
If you want to use only the *names* with which you've associated values, either (i) type the names from the left-hand keyboard, (ii) press  before using the menu keys, (iii) press  before pressing a menu key, or (iv) press  to access the current directory, since this is where the names will be stored.

Try Them All:                    



The first way used the command line in immediate mode; the second used it in alpha mode; the third, in algebraic mode; and the fourth way just used the immediate- execution keys.

Pressing   reactivated the current custom menu.



Problem Solving

Introduction

What we have here is a problem.

This book has been your introduction to the basics of HP-28S operation, but there is much more available to you. These basic lessons gave you a feel for the different kinds of information objects and their use. But there are programs and groups of operations living in the machine that do more sophisticated problem solving for you. These operations carry far-ranging implications for your use and application – so much so, in fact, that a complete description of any one of them would require a book in itself.

That's the problem: a book cannot be 10 books, no matter how hard it tries. So although there's a lot more to say about the following topics, there just isn't the room here to tell the whole story for each one. But they're definitely worth mentioning – if for no other reason than to round out your introduction to the machine. Even with these very cursory sketches of these topics, maybe you'll get some feel for their uses, potentials, and limitations.

Above all, these "surface- scratching" pages are meant to give you a little courage and curiosity about all these other capabilities. So take these first impressions and run with them.

Postfix Programming

Most programming languages are just that, languages. Learning them is akin to learning something like French or Spanish. You must first learn some words and then go about the more difficult task of using the words to make meaningful statements. This is usually a long and difficult process, especially for someone who just wants to solve a simple problem.

Not so with the HP-28S. As you've already seen (on pages 185-188), postfix programs are just "captured" command lines. The keys you'd press to solve a problem from the keyboard (interactively) are the same ones you'd press to write a program (preceded by ⌘). As you've also seen, a program has the relatively massive advantage of being invokable with one keystroke (via its name) from the keyboard, making it virtually equivalent to a built-in HP-28S command.

Therefore, before you go on to bigger and better things, a little more practice in postfix programming is probably appropriate.

So

1. There are two vector commands, RNRN (Row NoRM) and CNRM (Column NoRM), that find the maximum absolute value and sum the absolute values of the elements of a vector, respectively. Write two postfix programs, '**LMAX**' and '**LSUM**' to do these operations on lists.
2. Write a program called '**SPLIT**' that breaks the character-string in stack Level 2 into two sub-strings. The break should be *before* the character pointed to by the real-number in Level 1. The two commands SUB (SUB-string) and SIZE (the length of a string) will be useful to you.
3. Write the solution to problem 4 (pages 138 and 140) as a postfix program and call it '**UNIT**'. Use it to calculate the unit-vectors for $[1 \ 1 \ 1]$, $[-3 \ 2 \ -4 \ 6]$, and $[5 \ 5 \ 5 \ 1 \ 2 \ 1 \ 2]$.
4. Write the solution to problem 3 (pages 147 and 148) as a postfix program. Call it '**DS1**' (Double and Subtract 1) and use it on the following objects: $[[1]]$, $[[1 \ 2 \ 3] [4 \ 5 \ 6] [7 \ 8 \ 9]]$, and $[(2,4) (-4,6)]$.
5. Write a program to generate the character string "!" (recall from page 155 that it's not on the keyboard). Name it '!'.
6. Write the answers to problem 3 (pages 183 and 184) as postfix programs.

Answers

1. `LSUM:⌘ LIST→ →ARRY CNRM ⌘`
`LMAX:⌘ LIST→ →ARRY RNRM ⌘`

2. <code>⌘ DUP2</code>	Two copies of the index and string.
<code>1 SWAP OVER -</code>	From 1 to index - 1.
<code>SUB</code>	Get the sub-string.
<code>ROT ROT</code>	Get the original index and string.
<code>OVER SIZE</code>	From index to the length of the string.
<code>SUB</code>	Get the sub-string.
<code>⌘</code>	

The original string is split so that the indexed character is the first character of the second string. The two strings are left in the stack such that \oplus will re-join them.

3. `⌘ DUP ABS INV * ⌘`

`[1, 1, 1] USER UNIT =`
`[.577350269189 .577350269189 .577350269189]`

`[3 CHS, 2, 4 CHS, 6] UNIT =`
`[-.372104203767 .248069469178 -.496138938356`
`.744208407534]`

`[5, 5, 5, 1, 2, 1, 2] UNIT =`
`[.542326144545 .542326144545 .542326144545`
`.108465228909 .216930457818 .108465228909`
`.216930457818]`

4. « 2 * DUP SIZE 1 CON - »

[[1] USER [08]] = [[1]]

[[1, 2, 3] [4, 5, 6] [7, 8, 9] [08]] =
[[1 3 5] [7 9 11] [13 15 17]]

[(2, 4) (4 CHS, 6) [08]] = [(3, 8) (-9, 12)]

5. « 3 3, C H R ENTER ENTER EVAL [] STRING STR+ STO ».

6. Last element:

« DUP SIZE 1 - 1 SWAP SUB » or, in an optimized version,
« 1 OVER SIZE OVER - SUB » *or*
« LIST→ SWAP DROP 1 - →LIST »

First element:

« DUP SIZE 2 SWAP SUB » or, in an optimized version,
« 2 OVER SIZE SUB » , *or*
« LIST→ 1 - →LIST SWAP DROP »

Local Names

Local names are name objects. They are born, live and die with the particular program to which they are attached. They are *local* to that program.

Local names exist to enhance/ease the writing of postfix programs and user-defined functions (see pages 185-192 and 205-211). As such, they cannot exist without being attached to some procedure object.

An Example: Write a program to calculate $(X+1) \times (X-1)$ where X takes its value from Level 1 of the stack.

The direct approach: `« DUP 1 + SWAP 1 - * »`

Unfortunately the direct approach is not intuitively obvious to the more casual HP-28S user.

Be less direct: `« 'X' STO X 1 + X 1 - * »`

It's a little more clear here what is to be accomplished because instead of manipulating the stack to get 'X', you just "call it up" when you need it. But you're left with the name 'X' hanging around after you're done. Not only that but the routine assumes that you don't already have something of value in the name 'X' and goes right ahead and stores into it.

A touch of elegance: `« → X « X 1 + X 1 - * » »`

What's this? It's very similar to the previous program. The calculation portion is exactly the same except that you've made it a program unto itself (by placing it within its own set of French quotation marks).

The real difference is how you store stack Level 1 into the name 'X'.

That is, 'X' STO and → X are very similar operations – but they have some very significant differences:

You know that 'X' STO stores the value in Level 2 ('X' is in Level 1) into the name 'X'. You also know that the name 'X' is created if it didn't already exist. Regardless, the name 'X' exists in a directory and is now accessible by this and other programs – even after this program is finished.

On the other hand, → X stores the value in Level 1 into the name 'X', but *this* name 'X' is a local name and is therefore *not created in any directory*. If a real name 'X' already exists in a directory, it is *not* over-written. And once the program is done, the *local* name 'X' is gone – along with its value. Any named object, 'X', in any directory has not been affected by the transitory existence of the local name 'X' in this program.

If you haven't guessed by now, the command → makes the name local. You can see how it comes by its name – its presence is felt locally within the procedure object that immediately follows it, but the rest of the HP-28S world knows nothing about it.

So then there's the age-old question of "what's it good for?"

Well...

First, local names are extremely attractive. You don't have the danger of storing over some important value in an already existing name, and you don't need to worry about erasing them or otherwise cleaning them up when you're done.

Second, they help you organize your program. The assignment part comes first and the calculation part comes second – as a separate procedure object.

Third, they can ease calculations. In a calculation that requires the use of many objects taken from the stack, or the repeated use of a single object, keeping track of the stack contents can be difficult. You'll also soon find that the majority of your program just moves the stack around so that you can get to the object you want. Local names let you avoid such stack "gymnastics" by allowing you simply to "call up" an object when you need it within the calculation.

Fourth, they don't require the repeated use of STO. $\rightarrow a b c d e$ will take the bottom five objects off the stack and store them in the names 'a' through 'e'. \rightarrow will store stack objects into *all* the following names, no matter how many there are. *The order is:* the first one on the stack is the first one to be stored into a name. Thus, in this example, Level 5 goes into 'a' and Level 1 goes into 'e'.

Fifth, if a procedure object starts with $\& \rightarrow$, it is effectively a User-Defined Function (see pages 205-211) and can be placed in an algebraic object. Be careful, though! User-Defined Functions *should* return only one object as a result, and although it's relatively easy to write a U.D.F. that returns one object when the calculation part is an *algebraic* object, you must pay close attention to your *postfix* object to make sure that it does too.

Local Name Lesson

1. What's the difference between the results of

$\leftarrow x \ y \ 'x*y' \ \rightarrow$ and $\leftarrow x \ y \ \leftarrow x \ y \ * \ \rightarrow \ \rightarrow$?

2. Write two postfix programs, one with local names and one without, to calculate $((A + B) \times (A - B)) / C$ when stack Level 3 contains 'A', Level 2 contains 'B' and Level 1 contains 'C'.
3. Most HP-28S commands that would be useful in algebraic objects act like User-Defined Functions. For example, $\leftarrow \text{SIN} \ \rightarrow$ and $\text{'SIN(X)'} \rightarrow$ are both possible uses of the SIN command. Two commands that don't have algebraic forms are COMB (COMBINations) and PERM (PERMutations) in the STAT menu. Both take two real numbers from the stack. Write the U.D.F.'s that will allow COMB and PERM to be used in algebraic objects.
4. Write the solution to problem 1 page 167 as a postfix program using local names. Why does this work?

Local Name Moreon

(Answers to Local Name Lesson)

1. Nothing.

2. `« ROT ROT DUP2 + ROT ROT - * SWAP / »`
`« → A B C « A B + A B - * C / » »`

3. `« → A B « A B COMB » »`
`« → A B « A B PERM » »`

4. `« → oldname newname « oldname RCL newname STO`
`oldname PURGE » »`

Unlike *global* names, local names that contain other names (as '**oldname**' and '**newname**' must) don't evaluate their contents when they themselves are evaluated. Thus **oldname** puts the old name on the stack and **RCL** recalls the object associated with this name. **RCL** does not recall the contents of **oldname**. You could say the same thing about **STO** and **PURGE**.

By the way, is this a User-Defined Function? No. It doesn't return a value to the stack.

Some Comments Before You Go On

You probably haven't thought about it much, but you've been assuming that it's true: all of the programs and user-defined functions that you've written up to now have assumed that something has been on the stack to work on.

That's no real surprise. It's the nature of the postfix environment of the HP-28S. By now it's probably become pretty natural to you – and that's good, because you're learning to think like the HP-28S – a habit to help you to communicate better with the machine.

It will also help you to write better programs. That is, the best programs are those that work similarly to the built-in commands of the HP-28S. They take things from the stack and (most of the time) leave things there, too – ready for another command. In this way, your programs effectively *become new HP-28S commands*. You are extending your machine – customizing your tool – by building a library of short, sweet machine-extensions that will make the HP-28S exactly what you need to do your job.

And here's another thought for you. Once you've built your "new commands" from the built-in system commands, what's to keep you from writing yet newer commands from the "new" ones you've just built?*

*Absolutely nothing.

Making Decisions

Now back to more practice with postfix programming: Not every program is as straight-forward as the ones you just finished. Sometimes you need to test a value before you know what calculation to perform on it.

For example, your train of thought to solve a particular problem might be:

"IF x is less than 0
THEN divide x by 2

or

IF y is equal to z
THEN multiply x by 2
ELSE multiply x by 2 and add y"

In the first case, you only do the calculation IF the condition is true. In the second case, IF the condition is true THEN you do the first calculation, otherwise (ELSE) you do the second.

Well, *you* do this sort of decision-making all the time, right? So wouldn't it be nice if your calculator could do it too?

You're in luck. "This is what HP-28S's do best."

Decisions... : Write the following program:

```
⌘ IF x 0 <
    THEN x 2 /
    END
⌘
```

This is quite similar to your first train of thought, isn't it? The only difference is that the condition and the calculation are both written in postfix notation (what else?). And the **END** indicates the end of the calculation part. You need this special delimiter because there might (and probably will) be more commands following.

...Decisions: Try another:

```
⌘ IF y z ==
    THEN x 2 *
    ELSE x 2 * y +
    END
⌘
```

This is your second "train," from above. The same comments apply here as before, plus notice how the **ELSE** signals the **END** of the **IF-THEN** part and the beginning of the **ELSE** part.

Putting together the calculation part is something that you've done before – just straight postfix programming. The conditional portion (between the IF and THEN) may take some getting used to – but not too much, because these commands, such as `<` and `==`, are all just two-number postfix operations (page 85), similar to `-` and `÷`, because the order of the stack objects is important:

<code>-</code>	Level 2 - Level 1	subtraction
<code>÷</code>	Level 2 ÷ Level 1	division
<code><</code>	Level 2 < Level 1	less than
<code>≤</code>	Level 2 ≤ Level 1	less than or equal to
<code>==</code>	Level 2 == Level 1	equal to
<code>≠</code>	Level 2 ≠ Level 1	not equal to
<code>≥</code>	Level 2 ≥ Level 1	greater than or equal to
<code>></code>	Level 2 > Level 1	greater than

The conditionals all compare the objects in Levels 1 and 2 and return a 1 if the comparison is true or a 0 if the comparison is false. THEN takes this *truth value* and uses it to decide whether to (i) execute the commands that immediately follow it, then skip to END (when the truth value == 1, true); or (ii) skip to ELSE and execute the commands that follow it (when the truth value == 0, false).

Notice (and here's the best part) that there is nothing to keep you from using named program objects as the conditional and calculation parts. In other words, you could write:

```
IF this
THEN that
ELSE other
END
```

where **this**, **that** and **other** are names of programs or other objects that you've defined!

Pretty great, right? Well then, it's time to throw you some...

Conditional Curves

1. In problem 2 on page 219, the program **'DU'** caused an error if you invoked it in the HOME directory. Rewrite the program with a conditional that corrects the problem.
2. The command TYPE is useful in creating programs that work on different object types. It returns a number corresponding to the object type. Write the program **'BREAK'** that will decompose either a list (via LIST→) or an array/vector (via ARRY→). The TYPE numbers are 5 for a list, 3 for a real vector or array, and 4 for a complex vector or array.
3. Write a new conditional (like ==, or >) called **'LIST?'** that returns a **1** if the Level 1 object is a list and a **0** if otherwise.
4. Write another new conditional that uses **'LIST?'** and determines if the list is not empty. If the Level-1 object is a list and not empty, return a **1**. Return a **0** otherwise.
5. Conditional statements are useful for determining whether or not a real number is within a certain range. Write a program that uses DISP to display **"Out of Range"** if the Level-1 value is not between 1 and 5. (DISP will DISPLAY the Level-2 object in the display line indicated by the Level-1 number.)
6. Rewrite the previous program so that it displays **"In Range"** for values between 1 and 5, inclusive.

Conditional Conclusions

```
1. « PATH DUP SIZE 1 -  
    IF DUP 0 ==  
    THEN NOT  
    END GET EVAL  
»
```

If Level 1 equals 0,
then change the 0 to a 1.

The program first calculates the size of the **PATH** list and subtracts **1** to get the previous directory position. This number will be **0** only if the original list was **{ HOME }**.

The conditional portion (**DUP 0 ==**) first makes a copy of the position number because **==** eats both Level 1 and Level 2 objects. If the number isn't **0**, the program proceeds as normal. If it is **0**, then the logical operator **NOT** will change the **0** to a **1**. The **1** is then used as the index into the list (pointing to **HOME**) and is therefore valid – though the result is that the current directory is not changed.

NOT considers the real number in Level 1 to be a truth value (recall page 240). **0** represents *false* while any other number, typically **1**, represents *true*. Then **NOT** effectively turns *false* into *true* (**0** into **1**) and *true* into *false* (**1** or any other number into **0**). In other words, **NOT true == false** and **NOT false == true**. In this case, **NOT** is effectively equal to **DROP 1**.

Notice that the conditional portion could have been **DUP 1 <, DUP 0 SAME, 0 OVER ==, DUP NOT** or even **DUP → × 'x==0'**. The choice of **DUP 0 ==** is good, because it's both short and easy to understand.

```
2. ⌘ IF DUP TYPE 5 ==
    THEN LIST→
    ELSE ARRAY→
    END
```

If TYPE is 5

Otherwise assume an array.

- ⌘ This program is simple because it assumes a lot. It will cause an error for any object other than an array/vector or list. Here's a safer version:

```
⌘ DUP TYPE → t
  ⌘ IF t 5 ==
    THEN LIST→
    ELSE
      IF t 3 == t 4 == OR
      THEN ARRAY→
      END
    END
  END
```

⌘

- ⌘ This is more complex. It first associates the object type with the local name 't' to avoid having to keep track of or recalculate that type. Then begins an IF...THEN...ELSE...END group to test if 't' is 5. If so, then the program uses **LIST→** on the object. If not, then the ELSE part begins – a complete IF...THEN...END group. This OR conditional tests whether 't' is either 3 OR 4. OR is a logical operator that evaluates to true (1) if either Level 1 OR Level 2 contains a true (non-zero) value. If *both* are false, then OR returns false (0).

```
⌘ DUP TYPE
  IF { 3 4 5 } SWAP POS DUP
  THEN { ARRAY→ ARRAY→ LIST→ } SWAP GET EVAL
  ELSE DROP
  END
```

- ⌘ This version is effectively the same as the previous one, but without the "nested" IF routines (one inside the other). Can you follow it?

3. **« IF TYPE 5 SAME
THEN 1
ELSE 0
END**

- » While it's clear what this routine is doing, it's redundant. **« TYPE 5 SAME »** does the same thing.

4. **« IF DUP LIST?
THEN
IF SIZE
THEN 1
ELSE 0
END
ELSE DROP 0
END**

- » Again you see nesting. The DUP before the LIST? command is there to keep a copy of the original list around for SIZE to use later. The DROP after the second ELSE is to remove the list since SIZE wasn't performed.

**« IF DUP LIST?
THEN SIZE NOT NOT
ELSE DROP 0
END**

- » This routine gets rid of the nesting at the expense of some clarity. Why does it work?

5. **⌘ IF DUP 1 < SWAP 5 > OR
THEN "Out of Range" 1 DISP
END
⌘**

6. **⌘ IF DUP 1 ≥ SWAP 5 ≤ AND
THEN "In Range" 1 DISP
END**

⌘ AND is another logical operator. It returns true (1) only if both Levels 1 and 2 are true (non-zero) and false (0) otherwise.

Variants of IF...THEN...ELSE...END

You've seen that conditionals are useful for testing to see whether some object is a valid input for some calculation. You can check the type or range of a value and then proceed accordingly. But what if you don't *know* all the possible conditions to be met, or there are just too many to test for them all? You just want to try your operation and if there's a problem – an error – *then* you'll deal with it.

Well, the HP-28S even gives you *this* option. Rather than testing for some true or false condition, it can test for an *error condition*.

How It Works: Type the program

```
⌘ IFERR INV  
  THEN "Infinity" 1 DISP  
  END
```

- ⌘ Call this program 'INF?', and try it on 2, -4, and 0. The 2 and -4 invert properly, but $1 \div 0$ is not mathematically defined and is commonly assumed to be infinite. The program tells you this – without a lot of testing!

IFERR (IF ERRor) is much like the IF command, but rather than obtaining a truth value from the commands between it and THEN, IFERR waits to see if they generate an error. If so, IFERR causes a skip to the THEN part. If not, the commands between IFERR and THEN are completed and those between THEN and END are skipped.

You can also use IFERR...THEN...ELSE...END in a method similar to your use of IF...THEN...ELSE... END. Just remember that the condition that causes the branch (skip) is an error condition and not a logical (truth value) condition.

Another version of the IF command is IFT. IFT (IF Then) is a one-step version of IF ... THEN ... END. It takes its conditional and command portions from Levels 2 and 1 of the stack, respectively.

For example, take the solution to problem 5 on page 241. It might be rewritten as follows:

```
« DUP 1 < SWAP 5 > OR
  « "Out of Range" 1 DISP
  » IFT
»
```

The condition is taken from Level 2 and the object in Level 1 (in this case a program) is evaluated if the condition is true.

Look at another version:

```
« « "Out of Range" 1 DISP » IFT » 'IFFY' STO

« DUP 1 < SWAP 5 > OR IFFY »
```

IFFY assumes that a truth value is in stack Level 1 and displays its message if the value is true.

IFT's sister command IFTE (IF Then Else) is not too difficult to understand. It just expects the conditional (Level 3), the THEN commands (Level 2) and the ELSE commands (Level 1) to be on the stack.

IFTE has the side benefit of being able to be used within an algebraic object (and this means, of course, that all the conditional operators are usable within algebraic objects, too)!

F'rinstance: ' IFTE(X==0,1+B,4*X+B) '

IFTE takes three arguments separated by commas. The first is the conditional, the second is the THEN clause and the third is the ELSE clause.

This algebraic form works for IFTE but not IFT, because IFTE *always returns one value* (which IFT doesn't).

Since these IF ... THEN variants will probably take some getting used to, you'd better practice your skills with some ...

Iffy Situations

1. Since a directory is always a named object (i.e., its name can be put on the stack but the directory itself cannot), you can't use the command TYPE to determine whether the object you have is a directory. You *can* make use of the fact that attempting to recall from a directory is an error. Write a program using IFERR to determine if an object is a directory.
2. Combine the solutions to problems 5 and 6 on pages 241 and 245. Use IFTE.
3. IFT can be used to do quick conversions – changing a bad input into one that is acceptable. Write a routine that tests Level 1 for 0. If it is 0, it is changed to 1; otherwise, nothing is done.
4. Write a routine to change the Level 1 object into a real number only if the Level-1 object is complex.

Iffy Answers

```
1.  « IFERR RCL
    THEN RCLF 64 STWS
      IF ERRN # 12Ah ==
      THEN 1
      ELSE 0
    END SWAP STOF
    ELSE DROP 0
  END
```

- » The program assumes that there is an object on the stack. The test is to see if recalling from the object produces an error. If an error is produced, the program then tests to see if it was error # 12Ah, which is "Directory Not Allowed".

ERRN returns the ERRor Number for the most recent error. Since the number returned is a 12-bit binary integer (3 hexadecimal digits), the binary word size must be at least 12 bits. Since you can't assume the word size is already large enough, you must set it (STWS).

Programs that arbitrarily change mode settings can be annoying. So you recall the system flags (RCLF) that contain all the system mode settings before you change them and restore them (STOF) to their original settings (including binary word size) when you are finished.

As before, the program can be shortened as follows:

```
« IFERR RCL
  THEN RCLF 64 STWS ERRN # 12Ah == SWAP STOF
  ELSE DROP 0
END
»
```

2. « DUP 1 < SWAP 5 > OR "Out of Range"
"In Range" IFTE 1 DISP »

3. « DUP NOT « NOT » IFT »

Short and sweet. Again, the first NOT can be replaced by 0 == and its equivalents while the second NOT could be DROP 1 instead.

4. « DUP TYPE 1 == « C→R » IFT »

Doing Things More Than Once

What if you want to do something, oh, say, a lot of times?

Some problems require you to do the same or similar things over and over again. While it *is* possible to write a program that includes all of these redundant calculations or operations, wouldn't it be nice to be able to say just "Do it again"?

Fortunately, this is another "thing that HP-28S's do best."

For Example: Sum all the elements of the list {1 27 3 2 8 9 43}.

The program:

```
« LIST→ 1 SWAP OVER - START + NEXT »
```

Try it. The answer is 93.

Yes, but how did it work?

Well, to start with, the thing that you wanted to do over and over again was add. And you wanted to add one fewer times than the number of elements you have in the list (right?). LIST→ helped you there; it not only put the list elements onto the stack (so you can use +), but it left the number of elements in Level 1.

1 SWAP OVER – puts the number 1 in Level 2 and listsize – 1 in level 1. START, the *looping* command, takes these numbers to mean "count from 1 to listsize – 1." Not only that, but the commands between START and NEXT are performed once for every time START counts one higher. NEXT shows the end of the commands to be executed and tells START it's time for the NEXT loop, if there is one.

The postfix logic here is: **start finish START commands NEXT**

START counts from start to finish, executing commands for every step.

One limitation is that NEXT adds only 1 each time, so you can only count using integers.

No problem.

Use this instead: **start finish START commands increment STEP**

The STEP argument allows you to count by whatever increment that you want. For example,

« LIST→ 1 SWAP OVER - START + 2 STEP »

will sum every other element of a list. You can even use a negative increment to count down. Of course, then the start number would need to be larger than the finish number – a minor detail.

STEP is really quite convenient. Suppose you wanted to count from 0 to 1 in steps of .1? You'd use this:

« 0 1 START commands .1 STEP »

Of course, there's really no difference between this last example and counting from 0 to 10, because you can't use or evaluate the mental index counter the HP-28S is using to keep track of its looping.

But say that you *do* want to know what the index is during each step. Why, then you'd use:

start finish FOR index commands NEXT

or

start finish FOR index commands increment STEP

FOR ... NEXT and FOR ... STEP are exactly the same as START ... NEXT/STEP except that the index is a *local name* that you can use. Try:

« 1 10 FOR K K 1 DISP NEXT »

Evaluating this program causes the numbers from 1 to 10 in steps of 1 to be displayed in Line 1 of the display. Look closely at that FOR K K part. The first K is actually a part of the FOR command. It simply tells FOR the local name to use as the index. The second K is just a command that puts the current value of K onto the stack.

START ... NEXT/STEP and FOR ... NEXT/STEP are fine and dandy if you know beforehand *how many times* you want to repeat a set of commands. But what if you *don't* know this?

Suppose you know only that you must keep repeating *until a certain condition is met?*

You use: **WHILE condition REPEAT commands END**

This looks alot like your old friend IF ... THEN ... END. And it is. The same logic holds, except that the commands are *repeated* while the condition is true.

Notice that this means that if the condition is not initially true then the commands won't be executed even once. Not only that, but the commands to be repeated *must* change the condition at some point or they will be repeated forever (or some very long time).

A Trivial Example: Sum the elements of the stack.

Solution: **⌘ WHILE DEPTH 1 > REPEAT + END ⌘**

In other words, only while the stack has more than one level do you want to execute +. And you want to execute + until the stack has only one level. Notice that the action of the command + does indeed have an effect on the outcome of the conditional test DEPTH 1 >.

But what if you must execute the commands once before the test is even valid? In other words, say you want to do something like the following:

```
« getvalue WHILE DUP 0 > REPEAT getvalue END »
```

Because WHILE performs its test *before* the calculation, you must pull the calculation out and do it once before the test.

Or you could:

```
« DO getvalue UNTIL DUP 0 ≤ END »
```

DO ... UNTIL performs its test *after* the calculation. Notice that the conditions are the opposite of each other. The statements "WHILE this is true REPEATedly do this" and "DO this UNTIL this is false" are equivalent statements except for when the tests are performed.

Loop Practice

1. Use FOR ... NEXT to reverse the characters in a string.
2. Write a program that will delete all occurrences of the character in Level 1 from the string in Level 2.
3. Say that you have the PATH to some directory in Level 1. Write a program that will sequentially evaluate the elements of this list, thus moving you to that directory.
4. Write a routine that will wait for a key to be pressed and tell you what that key was. The command KEY returns a 0 for no key having been pressed or a 1 and the key name, but it doesn't wait for you.
5. Write a program that takes a list and a conditional routine from the stack, applies the condition to each element of the list and leaves a list on the stack that contains only those elements for which the condition was true.

Loopy Answers

```
1. « "" SWAP 1 OVER SIZE
    FOR I DUP I DUP SUB ROT + SWAP
    NEXT DROP
    »
```

FOR 1 to the SIZE of the string, take each successive element of the string and add it to the *beginning* of a new string you are building. Notice that you couldn't easily use START ... NEXT because you need the loop index to index the string.

```
2. « → S
    « WHILE DUP NUM CHR S ==
        REPEAT 2 OVER SIZE SUB
        END
    »
    »
```

WHILE the first character of the Level 2 string is the same as the test character, REPEATedly make the Level 2 string one shorter from the left (i.e., take the sub-string from the second character to the end of the string).

```

3. « 1 OVER SIZE
   FOR I DUP I GET EVAL
   NEXT DROP

```

» You use FOR again because you need the loop index to index the PATH list.

```

4. « DO UNTIL KEY END » or
   « WHILE KEY NOT REPEAT END »

```

```

5. « → list condition
   « { } 1 list SIZE
   FOR I list I GET
       IF DUP condition EVAL
       THEN +
       ELSE DROP
       END
   NEXT
   »

```

» This is rather long, but it's not too difficult to follow. Try it with the following objects:

```

{ 1 2 3 4 5 6 7 8 9 }    « 5 < »
{ 1 2 3 4 5 6 7 8 9 }    « 2 MOD »
{ 1 -1 2 -2 3 -3 4 -4 }   « 0 < »

```

Solving Problems Using Algebra

Why use an algebraic object?

Algebra is a universal language.

Almost every technical field has its own set of algebraic equations and expressions that are meaningful to it. These equations describe and define the relationships that exist within the discipline. It's only reasonable, therefore, that the HP-28S provide you with the ability to manipulate and solve algebraic problems.

Much of your work can also be simplified by writing ad hoc algebraic expressions for the problem at hand. In this way, HP-28S algebraic objects are programming without a programming language; the programming language is algebra!

To understand the full utility of an algebraic object, you should understand exactly what one is to the HP-28S.

For example:

1. An algebraic object, when evaluated, should yield one and only one result. This result is *typically* a real or complex number, but it need not be. This "restriction" is in keeping with the standard rules of mathematics.
2. An algebraic object may contain only names, algebraic operations, algebraic functions, real numbers and complex numbers.

If this is the case, then how do you get other types of objects to result from the evaluation of an algebraic object? By storing other types of objects under the variable names used by the algebraic object!

3. An algebraic object may or may not contain one equal sign ($=$). If it has no equal sign, it is called an algebraic *expression*. If it has an equal sign, it is called an algebraic *equation*.

That's basically it. There are no more restrictions. There are only ramifications and applications. You'll spend the rest of this discussion in looking at these.

Peculiarities of Representation

Inherent to the manipulation of algebraic objects on the HP-28S are its representation limits, which can at times be both annoying and confusing. The main limitation for algebraic objects is that all expressions must be represented as lines in the display.

In other words, there's not enough display to be able to present an algebraic expression in the pleasant and easy-to-read formats that you have seen in textbooks and have often written yourself. The HP-28S (and you too, when you're using it) must organize all algebraic expressions and equations so that they can be written and read from left to right, one character at a time.

This can (and does) often result in extremely long and complex-looking expressions, with a prodigious use of parentheses to keep things grouped properly. But there's no alternative.

Here's a word of comfort, however: The presentation of algebraic information is one of the most difficult aspects of computer algebra even on large computers with large displays.

Your HP-28S is in good company.

Some Practice

1. Identify the algebraic objects among the following:

- a. `'A+B'`
- b. `'SIN(A)'`
- c. `'1'`
- d. `A`
- e. `'ONE÷TWO'`
- f. `"8+cos(A/B)"`
- g. `'FACT(IP(ABS(LN(X-Y*√2+1/3))))'`
- h. `'1/X'`
- i. `'INV(X)'`
- j. `'[1 2]*D'`
- k. `'a>2'`
- l. `'IFTE(A(I,J)≥0,ABS(A(I,J))+1,A(I,J))'`
- m. `'(1,-2)*z/(-3.4,8)'`
- n. `'H DUP'`
- o. `''`
- p. `« 'F≠1' »`
- q. `'SIZE+3'`
- r. `« → s 's^2+2*s+1' »`
- s. `'a*F''`
- t. `'6+C→R(cmplx)'`

2. How would you key in the valid algebraic objects from problem 1?

Some Answers

1. a, b, g, h, i, k, l, m, and s are valid algebraic objects.
- c. This appears to be valid but you can't put it on the stack. It automatically evaluates to the real number 1.
- d. This is a name, but you can't put it on the stack without apostrophes (').
- e. This is a name, not an algebraic object. \div is a valid character to use within name objects. Pressing **"ONE** **ENTER** **129** **STRING** **CH** **"TWO** **ENTER** **++** **STR** will give you this name.
- f. This is a character string. Note the quotation marks (").
- j. This can't be keyed-in because vectors aren't allowed in algebraic objects.
- n. This example does not follow algebraic syntax. No stack command (like DUP) is allowed within an algebraic object.
- o. This is invalid because an algebraic object cannot be empty.
- p. This is a postfix program that contains an algebraic object.
- q. This is invalid because it contains the command SIZE. Generic object-oriented commands like SIZE are not allowed within algebraic objects.
- r. This is a user-defined function.
- t. This object contains the command $C \rightarrow R$ which returns two objects to the stack and is therefore not allowed.

2. Note, of course, that there are other possible solutions besides these:

a. `'A+B` `ENTER` or `'A` `ENTER` `'B` `ENTER` `+` or, if the names don't already exist, `A` `ENTER` `B` `+`.

b. `'SIN` `(` `A` `ENTER` or `'TRIG` `SIN` `A` `ENTER` or `'A` `ENTER` `TRIG` `SIN` or, if the name already exists, `A` `TRIG` `SIN`.

g. `'FACT` `(` `IP` `(` `ABS` `(` `LN` `(` `X-Y` `X` `√` `2+1÷3` or
`'` `REAL` `FACT` `NEXT` `NEXT` `IP` `PREV` `HES` `LOGS` `LN` `X-Y` `X`
`√` `2+1÷3` `ENTER` or
`'X` `ENTER` `'Y` `ENTER` `'` `√` `2` `ENTER` `X-` `'1÷3` `ENTER` `+` `LOGS` `LN`
`REAL` `NEXT` `HES` `NEXT` `IP` `REAL` `FACT`.

h. `'1÷X` `ENTER`.

i. `'X` `ENTER` `1/X` or `1` `'X` `÷` or, if X doesn't already exist, `X` `1/X`. This object is equivalent to the one in h.

k. `'LCA` `>2` `ENTER` or `'LCA` `ENTER` `2` `ENTER` `>` `ENTER`.

l. `'A` `(` `I` `,` `J` `ENTER` `ENTER` `0` `ENTER` `>` `ENTER` `SWAP` `ENTER` `ABS` `ENTER` `1` `+`
`SWAP` `IFTE` `ENTER`.

m. `(` `1` `,` `2` `CHS` `ENTER` `'LCZ` `ENTER` `X` `(` `3` `•` `4` `CHS` `,` `8` `÷`.

s. `'LCA` `X` `LCF` `•` `ENTER`.

How Do You Use an Algebraic Object?

Obviously, you use an algebraic object to do algebra. The obscure parts are the steps you take to do so.





You've already seen one method of manipulating an algebraic object (pages 194 to 199) when you built the solution to the quadratic equation. Remember how you used postfix commands to *build* the object out of items on the stack?

Well, you can do much of the *manipulation* of algebraic objects in this same way.

For Example: Assume that you know that $R = A + B + C$. You want to isolate B(i.e., to solve for B). To do so you must subtract A and C from both sides of the equation. You press:

  {  ,  ,  ENTER  
 ENTER  ENTER  ENTER    ENTER.

Then:

You get:

4:	
3:	
2:	
1:	'R-A-C=A+(B+C)-A-C'

Now, what good is that?

Well, it does show you what you've done. But it doesn't give you any more information.

But Press:  



4:
3:
2:
1:
 $'-A+R-C=B'$

Much better. The left side of the equation is a little out of order, but the information is correct – and simplified.

Take a moment to look at what you've done.

You first PURGE'd the names you were to use, so that when you were keying in these names to the stack, you didn't need to precede each with an apostrophe.

You used postfix operations to combine the names. This isn't new to you, but you could have posted the whole expression at once, like this:

$$['R=A+B+C\text{ENTER}]$$

Notice that $=$ is a postfix operator just like $+$ or $-$.

You then had the original expression. Next, you subtracted A and C from the algebraic object on the stack. You did this again with postfix operations. Notice that pressing $\boxed{A}\boxed{-}$ subtracted A from both sides of the equation at once. The same is true for C.

Finally, you invoked COLCT (COLleCT). This did a couple of things. First, it collected like terms. That is, looking at each side of the equation independently, it both combined positive and negative A and positive and negative C. It then eliminated the resulting zeros, leaving the final expression. Unfortunately, COLCT doesn't respect the order of the names in an algebraic object, so the left-hand side of the equation was somewhat rearranged.

In general, this is the basic mode of manipulating an algebraic object: Create the initial object, perform the desired manipulations using postfix commands, and collect terms when you're finished.

Fear not – there's more you can do with algebra than this, but first you should have some practice with this method....

Basic Algebra Problems

Convert the objects on the left into those on the right.

1. $'Y = T * (V_0 + V) / 2'$

$'2 / T * Y = V + V_0'$

2. $'X^2 = 2 * X + 3'$

$'-3 + X^2 - 2 * X = 0'$

3. $'-X = 1'$

$'X = -1'$

4. $'\sqrt{X} = \sqrt{Y}'$

$'X = Y'$

5. $'A = \pi * R^2'$

$'A / R / \pi = R'$

6. $'A * B * C - 1'$

$'A * B * C = 1'$

7. $'Z * 4 - 2 = 6'$

$'Z = 2'$

8. $'2 * A + B - C'$

$'A = .5 * (-B + C)'$

B. A. P. Answers

To make things easier, do this first: $\{\} \text{A} \text{SPACE} \text{B} \text{SPACE} \text{C} \text{SPACE} \text{X} \text{SPACE} \text{Y} \text{SPACE} \text{Z} \text{SPACE} \text{R} \text{SPACE} \text{T} \text{SPACE} \text{V} \text{SPACE} \text{V} \text{0} \text{PURGE} \text{ALGBRA}$. This is to make sure that you can enter these names on the stack without using ' '.

1. $\text{Y} \text{ENTER} \text{T} \text{ENTER} \text{V} \text{0} \text{ENTER} \text{V} + \text{X} \text{2} \div = \text{ENTER} \text{T} \div \text{2} \text{X} \text{COLT}$
2. $\text{X} \text{ENTER} \text{ENTER} \text{2} \text{^} \text{SWAP} \text{2} \text{SWAP} \text{X} \text{3} + = \text{ENTER} \text{2} \text{ENTER} \text{X} \text{X} - \text{3} - \text{COLT}$
3. $\text{X} \text{ENTER} \text{CHS} \text{1} \text{ENTER} = \text{ENTER}$ then either CHS or $\text{1} \text{CHS} \text{X}$
4. $\text{X} \text{ENTER} \text{Y} \text{ENTER} = \text{ENTER} \sqrt{\text{X}} \text{X}^2$
5. $\text{A} \text{ENTER} \pi \text{ENTER} \text{R} \text{ENTER} \text{2} \text{^} \text{X} = \text{ENTER} \text{R} \div \pi \div \text{COLT}$
6. $\text{A} \text{ENTER} \text{B} \text{X} \text{C} \text{X} \text{1} - \text{0} \text{ENTER} = \text{ENTER} \text{1} + \text{COLT}$
7. $\text{Z} \text{ENTER} \text{4} \text{X} \text{2} - \text{6} \text{ENTER} = \text{ENTER} \text{2} + \text{4} \div \text{COLT} \text{COLT}$
8. $\text{2} \text{ENTER} \text{A} \text{X} \text{B} + \text{C} - \text{0} \text{ENTER} = \text{ENTER} \text{C} + \text{B} - \text{2} \div \text{COLT} \text{COLT}$

This level of manipulation is fairly minimal, and it doesn't give you a lot of flexibility or control, it does give you a method of constructing algebraic objects and a degree of ability to manipulate them. As you've seen (pages 201-204), you can use algebraic objects to generate numeric results. If this is your primary purpose, then the appearance of the object isn't too important, since the numeric result of its evaluation will be the same either way.

Rest assured, though, that you do have the option of using more powerful operations on algebraic objects. For instance:

Name Substitution

You know already that a name can contain another name (page 165). This feature is useful for simplifying algebraic expressions and for allowing you to substitute one name for another in an algebraic object.

There are basically two commands that give you this capability: EVAL and SHOW. You've already seen that EVAL evaluates an algebraic object and replaces its names with whatever the names "point to." If the names point to other names, those names are substituted. If the names point to other objects, those objects are substituted.

SHOW, on the other hand, gives you more control as to what is evaluated. You can actually pick and choose which objects are to be evaluated, but the objects you want substituted *must* be other names.

Here's How: Say you know that $v = v_0 + at$ where v is velocity, v_0 is the initial velocity, a is the constant acceleration and t is time. You also know that $x = x_0 + (v_0 + v)t/2$ where x_0 is the initial position and x is the final position. Now, you want to build a relationship between x and a , position and acceleration. What you do is:

Press: $\{V, V_0, X, X_0, T, A\}$ **PURGE**
 V_0 **ENTER** A **ENTER** $T \times + V$ **STO**
 X_0 **ENTER** V_0 **ENTER** $'V$ **ENTER** $+ 2 \div T \times +$
ALGBRA **NEXT** A **SHOW**

And Get: $'X_0 + (V_0 + (V_0 + A * T)) / 2 * T'$

Clean It Up: **NEXT** **COLCT** **EXPAN** **EXPAN** **COLCT**

To Get: $'.5 * A * T^2 + T * V_0 + X_0'$

Since the starting expression was equal to x , so is this last one. You can press $X, =$ **ENTER** to create the actual equation, if you want.

Look for a moment at the steps you took:

First you purged the names you were going to be using and created the expression for v . Purging the names just made keying them in easier. Notice that you didn't key in the whole equation that solves for v , but just that part which is the expression equal to v . You then stored this expression in the name ' Ψ '.

Next, you keyed in the expression for X . At this point, you could've keyed in the entire equation – it's up to you. You picked the name you wanted to *show* in place of the name(s) that refer to it. You then invoked `SHOW` which replaced all V 's with their equivalent expressions containing A .

Since none of the other names referred to objects, you could have used `EVAL` to the same effect. Remember that `EVAL` evaluates *all* references and not just the ones you select.

The resulting expression didn't contain the name V , but does contain an expression with the name A .

Now this expression is fine and does what you want it to (it is a valid expression relating X to A), but it's not very clean. For instance, there are two occurrences of V_0 which could be combined. Since `COLCT` will combine like terms, you invoke that. Now it would be nice to distribute that $2T$ term throughout the expression. To do that you invoke `EXPAN`.

`EXPAN` `EXPAN`s an expression by distributing terms where it can. The first invocation of `EXPAN` distributes the $.5$ (i.e., $1/2$). The second distributes the T .

Finally, `COLCT`ing once more combines $T*T$ into T^2 and $.5*2*V_0$ into V_0 .

More Substitutions

It's fairly obvious that SHOW and EVAL are useful for replacing names and *increasing* the complexity of an expression (replacing names with expressions). But many times you want to *simplify* expressions by replacing all or parts of them with names or other, shorter expressions.

For Example: You know that $V = q(\ln(b/a))/(2\pi\epsilon_0 l)$ where everything except V and q is constant. Also, $C = (2\pi\epsilon_0 l)/\ln(b/a)$. Therefore, you'd like to substitute C into the first expression.

Key in the expression ' $q*(\ln(b/a)/(2*1*e0*\pi))$ ',
press **7** **ENTER** **'** **1** **÷** **C** **ALGBRA** **EXSUB**.

You get ' $q*(1/C)$ '.

Press **COLLECT** to get ' q/C '.

EXSUB (EXpression SUBstitute) is something new. It lets you substitute one expression for another within an algebraic object. In this case, you substituted $1/C$ for $\ln(b/a)/(2*1*e0*\pi)$. You did this by placing the object to be modified in stack Level 3, the position in the object of the expression you want to replace in Level 2, the object to be inserted in Level 1 and executing EXSUB.

The hardest part of this is determining the numeric position of the expression to be replaced. You do it by looking at the Level 3 object and counting the objects from left to right – excluding parentheses. In this case, $/$ is the seventh object. You want the number of $/$ because you want to replace *it and its arguments*.

A Variation: Key in ' $q*(\ln(b/a)/(2*1*e^0*\pi))$ ' once again. Now press **ENTER** **7** **NEXT** **EXGET**
1/X **'** **C** **STO**
7 **'** **1** **÷** **C** **NEXT** **EXSUB** **COLT**.

You get the same thing, but now press **EVAL**. You get back the original expression.

EXGET will pull the indexed expression out of an algebraic object in much the same way that GET pulls objects out of vectors, arrays and lists. You used it this time to make a copy of the expression you were going to replace and to store it into the name you were going to replace it with.

An easy way to use EXGET is, with the source object in Level 1, to press **ALGBRA** **FORM**. FORM is used to allow you to change or examine the form of an algebraic object. Press **[+]** until you've moved the black box over the second \swarrow . Now press **EXGET**. The stack is loaded with the source object, the sub-expression position and the sub-expression.

You can see how you might use EXSUB and EXGET in a long and complicated expression to reduce it to manageable size (by substituting names for expressions) before manipulating it. Then you replace the substituted names with their objects/expressions when you're done.

By the by, OBSUB and OBGET are equivalent commands that substitute and get objects rather than expressions. That is, they *don't* take an operator's arguments when they replace or get the operator.

FORM

You don't get too far when dealing with rearranging and substituting things in an algebraic object before you realize that you need more detailed control of things.

For instance, COLCT and EXPAN are tremendously useful, but they can't read your mind. They don't know how far you want to go in expanding or collecting nor do they know anything about the order of objects that pleases you most.

Happily, there *is* an algebraic object editor that gives you more control, as you just discovered. It's called FORM.

FORM gives you step-by-step control over the association, distribution and order of objects within an algebraic object – along with some other useful stuff. But it's not a magic algebra box. It allows you to manipulate algebraic objects – just as the HP-28S stack and postfix commands allow you to manipulate general objects – but it can't know what you want to do with them.

Therefore, don't become too discouraged if it takes a few steps to get where you want to go. Also, if it's easier for you to rewrite by hand and then retype an object than it is to use FORM, go for it. It's no sin.

For Example: Type in the expression ' $A^2+2*A*B+B^2$ '. No amount of EXPAN'ding or COLCT'ing will turn this into ' $(A+B)^2$ '. But press **ALGBRA** **EXPAN** **FORM**.

Use **[←]** to move to the third $*$. Press **NEXT** **→** to associate A and B (i.e., move the association – the parentheses – to the right). Now press **ENTER** **[←]** to move to the 2 and press **NEXT** **+1-1**. Both adding and subtracting 1 effectively adds 0 to the expression. Press **↔** to swap the arguments of $-$, and **↔** to associate the $-(1)$ and 2 . Press **ENTER** to get back to the main menu and **[←]** twice to move to the $+$ before the 2 . Press **COLCT**. All this merely breaks the 2 into $1+1$.

Now use **[←]** to move to the second $*$ (i.e., press it 7 times) and press **NEXT** **↔** to distribute $A*B$ over $1+1$. Get back to the main menu by pressing **ENTER**. Collect the 1 's by moving the cursor over the $*$'s and pressing **COLCT**.

Now move to the $+$ between $A*A$ and $A*B$ and press **NEXT** **↔** to associate them. Use **→** on the $+$ between $A*B$ and $B*B$ to associate them (remember to press **ENTER** to get back to the main menu and the cursor keys).

Move to the first $+$ and press **NEXT** **↔** to merge the terms. Move to the last $+$ and press **M→** to merge its arguments. While the cursor is still on the second $*$, press **↔** to swap the order of its arguments.

Move to the middle $+$ and use **M→**. Finally, with the cursor still on the $*$, press **COLCT**. **ATTN** (**ON**) leaves FORM and puts the expression back on the stack.

Yes, it's quite a lengthy and detailed process ("Whoah, you don't say!").

But in most cases you won't need to take an object all the way through from one form to another. You just use FORM to alter a few features to make other manipulations easier. Just pop in for a minute to do a specific distribution or collection and then leave again.

The really good news is that FORM doesn't allow you to perform an invalid operation – one that changes the value of the object. It will only allow you to move between equivalent forms of an object.

Want another look at that last example? All right, here's a little more detail about what you did:

First, you have this object that looks like a binomial expansion – i.e., it's something that you recognize as having a simpler form. That's good. You should always have an idea of where you're going and what you want to do when entering FORM. You know that you need to basically "undistribute" the expression.

So, you expand it with EXPAN. This turns all the x^2 terms into $x*x$ terms – all the better to pull them apart. Then you invoke FORM, which takes the Level-1 object and displays it for you to edit. Notice that it put in all the implied parentheses so that you know *exactly* the order of things.

FORM has its own cursor keys and cursor to allow you to move around within the expression. Notice that you can only move left and right and that the cursor automatically skips parentheses.

You have two $A*B$'s and you want to separate them and give one to $A*A$ and one to $B*B$. So you move to the $((2*A)*B)$ and notice that the association is not what you want. It's $2A \times B$ and not $2 \times AB$. So you change it. You want to shift the association to the right around the second multiplication, so you move there and use **RR** to move the association to the right. You are left with $(2*(A*B))$.

You want to turn $2*(A*B)$ into $(A*B)+(A*B)$. To do so, you must turn 2 into $1+1$ and distribute $A*B$ over the addition. Thus:

$(2*(A*B))$	+1-1	$(((2+1) - 1) * (A*B))$
$(((2+1) - 1) * (A*B))$	++	$((- (1) + (2+1)) * (A*B))$
$((- (1) + (2+1)) * (A*B))$	RA	$(((- (1) + 2) + 1) * (A*B))$
$(((- (1) + 2) + 1) * (A*B))$	COLCT	$((1+1) * (A*B))$
$((1+1) * (A*B))$	ED	$((1*(A*B)) + (1*(A*B)))$
$((1*(A*B)) + (1*(A*B)))$	COLCT	$((A*B) + (1*(A*B)))$
$((A*B) + (1*(A*B)))$	COLCT	$((A*B) + (A*B))$

Next, you reassociate so that one $A*B$ is associated with $A*A$ and the other with $B*B$. You do this by moving to the $+$'s on either side of $((A*B)+(A*B))$ and using **RA** on the left one and **RR** on the right one.

Now you have $(((A*A) + (A*B)) + ((A*B) + (B*B)))$. You basically want to undistribute the A 's from the left-hand pair of terms and the B 's from the right-hand pair. Moving to the $+$'s between the terms to undistribute, you use **RM** when the common term is on the left (i.e., the A) and **MR** when the common term is on the right (i.e., the B). Thus:

$((A*A) + (A*B))$	RM	$(A*(A+B))$
$((A*B) + (B*B))$	MR	$((A+B)*B)$

You next order the terms so that $(A+B)$ is on the right for both multiplications (although you could move it to the left just as well). Then you merge around the central $+$.

$((A+B) \times B)$	↔	$(B \times (A+B))$
$((A \times (A+B)) + (B \times (A+B)))$	→	$((A+B) \times (A+B))$

This is really very close to being done. You simply collect like terms around the \times with **COLCT** and that's it.

$((A+A) \times (A+B))$	COLCT	$((A+B)^2)$
------------------------	--------------	-------------

ATTN leaves FORM and returns the object to the stack.

(whew!)

Less Basic Algebra Problems

1. You've already seen (on page 272) how to combine (or substitute)

$$\begin{array}{ll} & v = v_0 + at \\ \text{and} & x = x_0 + .5(v_0 + v)t \\ \\ \text{to get} & x = x_0 + v_0 t + .5at^2 \end{array}$$

Using the same two equations, show that $v^2 = v_0^2 + 2a(x - x_0)$.

2. Convert $1/(x/2+y)$ to $2/(x+2y)$. These are equivalent expressions.
3. Given that $v_2 = v_1(r_1/r_2)$, $v_1 = w_1 r_1$ and $v_2 = w_2 r_2$, show that $w_2 = w_1(r_1/r_2)^2$.
4. Convert $1 - ((m_1 - m_2)/(m_1 + m_2))^2$ to $4m_1 m_2 / (m_1 + m_2)^2$.
5. Given that $GMm/(R+r)^2 = mw^2 r$, R is much less than r , and $w = 2\pi/T$, show that $GM = 4\pi^2 r^3 / T^2$.

Less Basic Algebra Solutions

1. First press $\{\} \text{V} \text{' } \text{V0} \text{' } \text{A} \text{' } \text{T} \text{' } \text{X} \text{' } \text{X0} \text{ [PURGE]}$ to purge all the names you'll to be using. Then key-in ' $V = V0 + A * T$ '. Solve the equation for T (i.e., move T to one side of the equation by itself) by pressing $\text{V0} \text{ [ALGBRA] [COLT] [A] [÷] [COLT]}$. The result is ' $(V - V0) / A = T$ '.

Key in ' $X0 + .5 * (V0 + V) * T = X$ '. Notice that you've moved the $=$ so that the expression with T is on the left-hand side of the equation. You did it so that when you isolate T it will be on the left-hand side of the equation. Press

$\text{X0} \text{ [ALGBRA] [COLT] [0.5] [÷] [COLT] \text{V0} \text{ [ENTER] } \text{V} \text{ [÷] [COLT]}$.

You'll have ' $T = 2 * (X - X0) / (V + V0)$ '.

Combine the two with $[+]$ to get ' $(V - V0) / A + T = T + 2 * (X - X0) / (V + V0)$ '. Get rid of the T 's by pressing $\text{T} \text{ [ALGBRA] [COLT]}$.

Multiply by both the denominators (i.e., press $\text{A} \text{ [X] [COLT] } \text{V} \text{ [ENTER] } \text{V0} \text{ [÷] [X] [COLT]}$). The equation is now ' $(V - V0) * (V + V0) = 2 * (X - X0) * A$ '.

Expand the left-hand side by pressing FORM , moving to $*$ with $[↔]$ and pressing EXEC . You're now out of the FORM menu and ' $(V - V0) * (V + V0)$ ' is on the stack. Press EXPAN EXPAN to completely distribute the multiplication and then COLT to collect like terms. Press EXEC to put this expression back into the equation. You now have ' $V^2 - V0^2 = 2 * (X - X0) * A$ '.

Now add $V0^2$ to the equation ($\text{V0} \text{ [ENTER] } 2 \text{ [^] [÷] [COLT]}$) and you're done.

2. Key in ' $1/(X/2+Y)$ ' and press **ALGBRA** **FORM**. Move the cursor over the $+$ and press **NEXT** **NEXT** **MF**. AF (Add Fractions) puts the two arguments of $+$ over a common denominator. You get ' $1/((X+(2*Y))/2)$ ' which is 1 over a fraction. Press **ATTN** to leave FORM and press **COLCT**.

Another more intuitive method would be to multiply the original expression by $2/2$. You might do it this way:

Key-in ' $1/(X/2+Y)$ ' and press **2** **X** **2** **÷** to get ' $1/(X/2+Y)*2/2$ '. Don't collect. Press **FORM**. Move to the $*$ and press **NEXT** **÷÷** **÷H** to associate the first 2 with the numerator. Move to the first $*$ and press **COLCT** to collect the numerator. You'll have ' $((2/(X/2+Y))/2$ '.

Move to the last $/$ and use **H÷** to associate the last 2 with the denominator. Move now to the $*$ and press **÷D** to distribute the 2 over the denominator. At this point, the cursor is over the $+$, so press **COLCT** to collect the denominator. Press **ATTN** to leave FORM and you're done.

As you can see, the intuitively obvious method is not necessarily the easiest to implement.

3. Key in ' $V2=V1*(R1/R2)$ '. Store ' $W1*R1$ ' in ' $V1$ ' (i.e., press $\boxed{W1}\boxed{\times}\boxed{R1}\boxed{=}\boxed{V1}\boxed{STO}$) and ' $W2*R2$ ' in ' $V2$ '. Evaluate the equation (via \boxed{EVAL}) to get ' $W2*R2=W1*R1*(R1/R2)$ '. Divide the equation by ' $R2$ ' and collect to get ' $W2=R1^2*R2^{(-2)}*W1$ '.

To associate the two squared terms, use FORM:

Move to the second \wedge and press \boxed{NEXT} $\boxed{1/O}$. You now have $(W2=((R1^2)*W1/(R2^2))*W1))$. Move back one to the $*$ and press \boxed{NEXT} $\boxed{\leftrightarrow}$ $\boxed{\leftrightarrow}$ to turn $*INV$ into $/$. Press \boxed{NEXT} $\boxed{M\rightarrow}$ to merge the two under 2 . Move to $*$ and press \boxed{NEXT} $\boxed{\leftrightarrow}$ to make it pretty. You now have $(W2=(W1*((R1/R2)^2)))$. Leave FORM by pressing \boxed{ATTN} .

4. First, key in ' $1-((M1-M2)/(M1+M2))^2$ ', and then press \boxed{ALGBRA} \boxed{FORM} . Next distribute 2 over the numerator and denominator by moving to \wedge and pressing \boxed{NEXT} $\boxed{\div D}$. Next, create a single fraction with a common denominator by moving to the first $-$ and pressing $\boxed{NEXT}\boxed{NEXT}$ \boxed{MF} . You should have

$$(((1*((M1+M2)^2))-(M1-M2)^2)/(M1+M2)^2).$$

Collect the leading 1 by moving to the first $*$ and pressing \boxed{COLCT} . Then expand the squared terms of the numerator by moving to each of the two \wedge 's and pressing \boxed{EXPAN} . Finally, move to the first $-$ and press \boxed{COLCT} and \boxed{ATTN} to leave.

5. Key in ' $G*M*m/(R+r)^2=m*w^2*r$ '.

Since R is insignificant, eliminate it by storing 0 into it. Viz., $\boxed{0}\boxed{)}\boxed{R}\boxed{STO}\boxed{EVAL}$.

The equation is now ' $G*M*m/r^2=m*w^2*r$ '.

To get rid of the left-hand denominator, multiply by ' r^2 '. Since m is on both sides of the equation, divide by ' m '. Remember to collect after each step.

The equation is now ' $G*M=r^3*w^2$ '.

Since $w=2\pi/T$, store ' $2*\pi/T$ ' in ' w '. Evaluate the equation.

The equation is now ' $G*M=r^3*(2*\pi/T)^2$ '.

Go into the FORM menu and distribute 2 over $2*\pi/T$. I.e., move the cursor over the final $^$ and press $\boxed{\div}$. Next distribute 2 over $2*\pi$.

Now $((G*M)=((r^3)*((2^2)*(\pi^2))/(T^2))))$.

Finally, collect 2^2 by moving the cursor over its $^$ and pressing \boxed{COLCT} . Press \boxed{ATTN} .

The equation is now ' $G*M=r^3*(4*\pi^2/T^2)$ '.

Some Short-Cuts

Solving an equation is, in many cases, a matter of isolating the name of interest on one side of the $=$, with everything else on the other, and you've had some practice doing that in the preceding problems. But while solving those problems, you probably were wishing for some short-cuts.

Well, there are some:

The ISOL (ISOLate) command will automatically perform all the transformations necessary to an algebraic object to isolate the specified name.

For example, if you had the equation ' $A=B*C$ ' in Level 1 and typed $\boxed{\boxed{C}}\boxed{\boxed{ISOL}}\boxed{\boxed{ENTER}}$, the result would be ' A/B '.

You asked the HP-28S to isolate ' C ' for you, and it did so. You can see by inspection that if you were to divide the original equation by ' B ', then ' C ' would indeed be equal to ' A/B '. Notice that there is no $C=$. This is so that the resulting object can be stored in ' C ' so that you can perform substitutions if you like. If you want the $C=$, you can always add it.

Be careful: While isolation of a name in this way can be very useful when a name occurs only *once* in an expression or equation, if there's more than one occurrence of the name, the resulting expression will also contain the name. In other words, you will not have achieved very much by using ISOL.

Viz.: $\boxed{\boxed{A \times B = A + C}}\boxed{\boxed{ENTER}}\boxed{\boxed{A}}\boxed{\boxed{ALGBRA}}\boxed{\boxed{NEXT}}\boxed{\boxed{ISOL}}$ gives ' $(A+C)/B$ ' rather than ' $C/(B-1)$ '.

There's another method of solving an algebraic object if that object is specifically in the form of a quadratic expression or equation. QUAD will take an algebraic object from Level 2 and the name of the variable for which the expression is quadratic from Level 1. The result will be an expression for one or both roots of the quadratic expression.

But if QUAD returns only one object, how does it show both roots? The answer is an unequivocal "It depends." It depends on whether flag 34, the *principal value flag*, is set or clear.

Both ISOL and QUAD are capable of solving for multiple roots. In other words, if an expression (like a quadratic) has more than one solution, all solutions can be found and represented.

To see how, first set flag 34 ($\boxed{3}\boxed{4}\boxed{'}\boxed{S}\boxed{F}\boxed{\text{ENTER}}$) and purge 'A', 'B', 'C' and 'X'. Next, key in the expression 'A*X^2+B*X+C' and put a copy in both Levels 1 and 2. Now key in $\boxed{X}\boxed{\text{ENTER}}\boxed{\blacksquare}\boxed{\text{ALGBRA}}\boxed{\text{NEXT}}\boxed{\text{QUAD}}$.

The result is ' $(-B+\sqrt{B^2-4*(A*2/2)*C})/(2*(A*2/2))$ '.

This is clearly *one* of the roots, but because you specified that you wanted only the principal root (you set the principal value flag), you were given only one root.

On the other hand, type $\boxed{3}\boxed{4}\boxed{'}\boxed{C}\boxed{F}\boxed{\text{ENTER}}\boxed{\text{DROP}}\boxed{X}\boxed{\blacksquare}\boxed{\text{QUAD}}$. The result is:

$'(-B+s1*\sqrt{B^2-4*(A*2/2)*C})/(2*(A*2/2))'$.

This object is different from the first by the inclusion of the name 's1'. You will recall that the general solution is $-b \pm \sqrt{\dots}$ et cetera. Well, 's1' functions as the \pm here. Since there are two solutions and an algebraic object can only return one result, QUAD gives you the option of choosing one, the other or both of the possible solutions.

Here's how:

If you were to associate the value 1 with the name '**s1**' (in the usual manner – with **STO**) and then evaluate the expression, the result would be '**-b+√**....

On the other hand, if you were to associate -1 with '**s1**' and evaluate the expression, the result would be '**-b-√**....

'**s1**', therefore, stands for *sign1* and should be interpreted when reading the object as \pm . (If there were more than one \pm possible in the object, you would see '**s2**', '**s3**', '**s4**', et cetera.)

This convenience is fine for solutions to a quadratic equation, but what about objects that have more than two roots?

Let's find out. Take a periodic function, like sine, for which there are infinitely many roots.

Do this: Press **DEG****ENTER** **X****ENTER** **SIN****ENTER** **X****ENTER** **ISOL****ENTER**.

The result is '**180*n1**'. Here, '**n1**' stands for any integer, indicating that the sine of any integral multiple of 180° is zero. (Again, if there were more integral multiples in the expression, you would see '**n2**', '**n3**', '**n4**') Realize that if flag 34 were set, the result would be simply 180 (the principle value).

One final thing to note with QUAD is that if you want strictly symbolic results, take care to PURGE any names that the expression uses. If you don't, the names will be evaluated and replaced by their referent objects. This precaution isn't necessary with ISOL.

Another time-saver is the TAYLR command. TAYLR performs a TAYLoR Series expansion of the algebraic object you supply. Though you may not even know what a Taylor series is, you don't need to in order to see its usefulness in the following case.

Case: Say that you have the binomial $(x+1)^5$ and you want to expand it to the equivalent polynomial. How do you do it?

Well, you could EXPAND it repeatedly until all the distributions had been performed and then use COLCT to collect the terms. But this is a trifle time consuming (you should try this at least once – when you have 15-30 minutes to kill).

Or, you could use TAYLR as follows:

Type `'X'ENTER 1 + 5 ^ 'X'ENTER 5  ALGBRA NEXT TAYLR.`

Here 5 is the degree of the polynomial, and X is the variable in which it is in that degree.

Your HP-28S will think for a little while, and then return the polynomial (in order!) from low order term to high order term. Viz.:

`'1+5*X+10*X^2+10*X^3+5*X^4+X^5'`

Quite the deal, eh? A real time-saver with long expansions!

The Equation Solver

You've seen that you can consider algebraic objects to be programs written in algebraic syntax, and as such, you can use them to solve for numerical results of algebraic expressions. In such cases, the algebraic object's internal names ("variables") refer to numeric objects.

You've also used algebraic objects as algebraic expressions to be manipulated algebraically and that, through substitution (storing expressions in the names), these can be transformed into other expressions.

But you've also seen that assigning values to these names can be tedious.

SOLV, the equation-solver menu, exists to aid you with just such problems. The SOLV menu allows you both to conveniently load an algebraic object (via the command called STEQ, or STore EQUation) and then use SOLVR to fill in the values of its variables.

Try This: Press $\{ \} X ' Y$ \blacksquare **PURGE** \blacksquare **CLEAR** $' 3 X X = 2 5 - 4 X Y$ **SOLV** **SOLVR** .



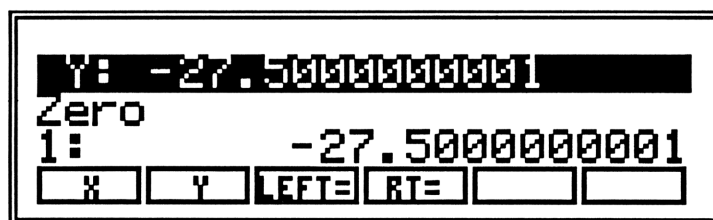
SOLVR finds all of the names within the object and generates a *storage menu* (remember page 224?), including **LEFT=** and **RT=** which evaluate the expressions on the left and right sides of the equal sign, respectively!

SOLVR makes it easy to associate values with names in an algebraic object.

But wait – there's more!

SOLVR is also a sophisticated numerical root finder. That is, if you give all but one of the names in the algebraic object real values, SOLVR will find the value of the final name that satisfies (solves) the expression! In the case of equations, this means that a value is found such that the left and right sides are equal. In the case of expressions (i.e., objects with no $=$) a value is found such that the expression evaluates to 0 – just as if there were an invisible $=0$ after the object.

In This Case: Press $\boxed{4}\boxed{5}\boxed{X}\blacksquare\boxed{Y}$.



When $X = 45$, $Y = -27.5$. Pressing \blacksquare before the menu key tells SOLVR that you want to solve for that name.

Press $\boxed{1}\boxed{3}\boxed{Y}\blacksquare\boxed{X}$. SOLVR tells you that $X = -9$.

Although this in itself is terribly convenient – especially if you want to play with different values to see how the expression acts – there are some important ramifications that radically increase the usefulness of SOLVR.

First, *it doesn't matter which value is the unknown*. Normally, you would need to rewrite an algebraic expression so that the name of the unknown is on one side of the $=$ and an expression of known values is on the other. SOLVR does this rewriting for you, automatically!

Second, there are expressions and equations for which it is impossible or virtually impossible to explicitly isolate a particular unknown in this way. For such problems, the only solution may be an approximation to the value of the unknown. But SOLVR knows this, and will find this solution automatically!

Unfortunately, in this latter case, there may be no unique solution; that is, there may be more than one answer that will satisfy the expression. You must be able to recognize expressions that are likely to behave this way, because although SOLVR can find a result, it may not be the only one – or the best one.

So you may supply a guess or guesses as to the value of the unknown before solving for it. SOLVR will start with these guesses as it looks for a solution and therefore find a result (if any) that's relatively close to one of your guesses.

In reality, SOLV and SOLVR are actually friendly ways to use the actual numeric root-finder program, ROOT. They are excellent for interactive problem solving and algebraic object manipulation, but it's quite possible that you'll have a more sophisticated problem, one where a solution to an algebraic object is only half the battle. In that case, you may opt to use ROOT itself in a postfix program.

ROOT takes either a name (most commonly the name of an algebraic function, expression, equation or postfix program), an algebraic object or a postfix program from Level 3, the name of the unknown from Level 2, and one or more guesses from Level 1 of the stack. It returns a numeric result to Level 1.

For Example: Press `'X` `PURGE` `'X` `^2+X-2` `ENTER` `ENTER` `'X` `ENTER` `5` `SOLV` `NEXT` and `ROOT`.

Now press `DROP` `'X` `ENTER` `5` `CHS` `ROOT`.

The result is 1 when you guess 5, and it's -2 when you guess -5.

Unfortunately, a good discussion of how and why ROOT comes to its results would require a good discussion of the nature of algebraic expressions, equations and functions, as well as the nature of numerical approximations. These are clearly beyond the scope of this book, but there are some such discussions in the HP-28S Reference Manual.

Still More Algebra Problems

1. If $1/l + 1/O = 1/F$, $O = 9$, and $F = 24$, what is the value of l ?
2. Given $v = v_0 + at$ and $x = x_0 + .5(v + v_0)t$
 - a. Find x when $x_0 = 100$, $v_0 = 0$, $a = -32$, $t = 0, 1, 2$, and 3 .
 - b. Find x_0 when $x = 50$, $v_0 = 0$, $a = -32$ and $t = 2$.
 - c. Find v_0 when $x_0 = 0$, $x = 50$, $t = 2$ and $a = -32$.
 - d. Find t when $a = -32$, $v_0 = 0$, $x_0 = 100$, $x = 90, 80, 70$, and 60 .
3. $\text{Det}(\mathbf{A} - l\lambda) = 0$, where \mathbf{A} is a square array, \mathbf{I} is the identity matrix of the same degree as \mathbf{A} , λ is an eigenvalue for the system, and Det is the Determinant operation.

Find the corresponding eigenvalues (there are three distinct values) when $\mathbf{A} = \begin{bmatrix} 16 & -24 & 18 \\ 3 & -2 & 0 \\ -9 & 18 & -17 \end{bmatrix}$.

(You don't need to understand all of this to be able to solve the problem, though both this problem and its solution introduce some new and powerful things.)

S. M. A. P. Answers

1. Create the object ' $1/I+1/O=1/F$ ' and press **SOLV** **STOR** **SOLVR**.

Store 9 in ' O ' and 24 in ' F ' (i.e., press **9** **STOR** **O** **24** **STOR** **F**). If you press **SOLV** **I** now, you'll find that the solver will run for quite some time and then quit and tell you that F is some huge number. The reason is simple (the x -axis is an asymptote of the function; it never reaches 0), but it's not obvious, because it depends on the nature of the root finder and its limitations.

Fortunately, it's easy to get around, because you can use **ISOL** to solve the equation for I . So press **SOLV** **RCER** (ReCall EQuation) **I** **ISOL** **EVAL**.

The answer is: $I = -14.4$.

Note that you could have isolated I first thing, thrown the resulting expression into **SOLVR**, stored the values of O and F and pressed **EXPR=**. **EXPR=** replaces **LEFT=** and **RT=** in the **SOLVR** menu when you are dealing with an expression rather than an equation.

2. Create the object ' $X=X_0+.5*(V+V_0)*T$ '. Press **SOLV** **STEP** **SOLVE**. Create the object ' V_0+A*T ' and press **V**. Notice that the name V goes away and the name A appears in the menu.

a. Press **1****0****0** **X0** **0** **V0** **3****2****CHS** **A**

0 **T** **X** $x = 100;$

1 **T** **X** $x = 84;$

2 **T** **X** $x = 36;$

3 **T** **X** $x = -44.$

b. Press **5****0** **X** **2** **T** everything else is the same as before.
X0 $x_0 = 114.$

c. Press **0** **X0** everything else is the same as before.
V0 $v_0 = 57.$

d. Press **1****0****0** **X0** **0** **V0**

9**0** **X** **T** $t = .790569415044;$

8**0** **X** **T** $t = 1.11803398875;$

7**0** **X** **T** $t = 1.36930639376;$

6**0** **X** **T** $t = 1.58113883009.$

3. Create the object $\text{A DUP IDN L * - DET}$, where A and L are names, and IDN and DET are HP-28S array commands. IDN will convert the Level 1 array into an identity matrix. DET calculates the determinant of the Level 1 array (matrix).

Press $\boxed{\text{'}}\boxed{\text{L}}\boxed{\text{PURGE}}\boxed{\text{SOLV}}\boxed{\text{STO}}\boxed{\text{SOLVR}}$. Key in the array

$\boxed{\text{[[16 -24 18] [3 -2 0] [-9 18 -17]]}$

and press $\boxed{\text{R}}\boxed{\text{L}}$. Now press $\boxed{\text{R}}\boxed{\text{L}}$. The result is 1. This is one of the eigenvalues.

To find the others, you must supply some guesses. First, try less than 1:

Press $\boxed{\text{'}}\boxed{\text{1}}\boxed{\text{0}}\boxed{\text{0}}\boxed{\text{CHS}}\boxed{\text{'}}\boxed{\text{0}}\boxed{\text{L}}\boxed{\text{L}}$. The result is -8. The list you stored in L contained the range in which you wanted SOLVR to work at finding an answer. That's two out of three answers, now try greater than 1.


Press $\boxed{\text{'}}\boxed{\text{1}}\boxed{\text{0}}\boxed{\text{0}}\boxed{\text{L}}\boxed{\text{L}}$. The result is 4.00000000005. The 100 you stored in L was your guess.












Now, you can't use SOLVR to solve for A. SOLVR works only when finding real-number solutions, but as you've seen, this doesn't mean that the other names in the expression must contain real numbers.

Notice that you can use a postfix program as the "equation" in SOLVR. The program must not take values from the stack for use in its calculation and it may either return one value to the stack (as an algebraic expression would when evaluated) or store its result in a name.

Calculus

Differential calculus is well represented and easy to use on the HP-28S.

You need only place an algebraic object in stack Level 2, the name of the variable of differentiation in Level 1, and press . The resultant derivative is left in Level 1.

For Example:           
yields : '1+INV(X)'.

Representing the world of Integral calculus on the HP-28S are commands for *symbolic* integration of polynomials and *numeric* integration (i.e., definite integration with a numeric result) of any expression.

To symbolically integrate a polynomial, you must put the polynomial in Level 3, the name of the variable of integration (this is also the name of the variable for which the expression is a polynomial) in Level 2, the degree of the polynomial in Level 1 and press $\blacksquare \int$.

Like So: $\boxed{1}\boxed{3}\boxed{X}\boxed{X}\boxed{\blacksquare}\boxed{\wedge}\boxed{2}\boxed{-}\boxed{2}\boxed{X}\boxed{X}\boxed{+}\boxed{.}\boxed{3}\boxed{\text{ENTER}}\boxed{1}\boxed{X}\boxed{\text{ENTER}}\boxed{2}\boxed{\blacksquare}\boxed{\int}$

yields $\text{'}.3*X-X^2+X^3'$.

To numerically integrate an object, you must put the object in Level 3, a list containing the name of the variable of integration and the limits of integration in Level 2, the accuracy of the result in Level 1 and press $\blacksquare \int$.

Like So: $\boxed{1}\boxed{E}\boxed{X}\boxed{P}\boxed{(}\boxed{X}\boxed{\text{ENTER}}\boxed{\{}\boxed{X}\boxed{,}\boxed{0}\boxed{,}\boxed{1}\boxed{\text{ENTER}}\boxed{E}\boxed{E}\boxed{X}\boxed{5}\boxed{C}\boxed{H}\boxed{S}\boxed{\blacksquare}\boxed{\int}$

gives the results 1.71828182875 ,

and $1.71814388996E-5$.

The first number is the result of the integration and the second is the accuracy.

Symbolic Constants and Symbolic Function Evaluation

Symbolic calculation is great, but of course, it's not the only way to calculate. Users of most other calculators and computers will recognize this readily.

Most calculation tools are intended to help you generate numeric results. After all, you usually want a numeric value when you add 1 and 1, not a symbolic expression. Most calculators and computers do this as a matter of course – both because it's commonly what you want and because they just can't do it any other way.

The HP-28S gives you the *option* of exclusively evaluating objects down to numeric results – if it's most convenient.

You simply clear flag 36 (press **3****6****,****C****F****ENTER**).

All evaluations after this point will attempt to generate non-name objects (real numbers, vectors, et cetera). If it cannot find such an associated object for the name it's trying to operate on, it generates an error : **Undefined Name**.

In this mode, the HP-28S acts most like calculators and computers that you have used before. It assumes that you want the name to refer to something, or you wouldn't be trying to calculate with it. In this mode the HP-28S is most like a *calculator* and least like a *symbolic manipulator*.

Of course, even when in symbolic evaluation mode (you can choose this by setting flag 36: **3****6****,****S****F****ENTER**), you can always *force* numeric evaluation for individual cases by using \rightarrow NUM. But it's good to know you can choose.

Finally, the HP-28S sports a set of data objects called *symbolic constants*. This is a set of three commonly used mathematical constants and two machine-specific constants. They are:

e	Euler's constant, the base of the natural logarithm;
π	the ratio of the circumference of a circle to its diameter;
i	the square root of -1 ;
MINR	the HP-28S's smallest representable number; and
MAXR	the HP-28S's largest representable number.

Each of these can be considered a numeric function because each can be made to yield a number. But they are in fact constants because they never change.

They're *symbolic* constants, partly because they have names and partly because in symbolic constant mode (flags 35 and 36 set), they resist conversion to their respective numeric values. In that mode, they will remain symbols unless forced to become numbers by \rightarrow NUM.

This stubborn resistance to change can be extremely useful, especially in the simplification of expressions, because you're often interested in a result that is a function of e , π , or i . If the result were strictly a number, you would potentially lose some information.

For example, which is more meaningful: 2π or 6.28318530718 radians?

And which is more exact? The HP-28S (in symbolic constant mode) has certain functions that recognize these constants – noting the fact that any numeric representations of e and π are only approximations.

Thus, $\text{SIN}(\pi)$ under symbolic evaluation mode (and radians mode) is 0 , but under numeric evaluation mode, it's **-2.06761537357E-13**.

A Final Visit with Algebra/Calculus

1. Find the expressions for v and a when $x = A\cos(\omega t + b)$, $v = dx/dt$ and $a = dv/dt$.

2. Find $\int (x+1)^6 dx$.

3. Find the derivatives (with respect to x) of each of the following:

a. $2x^3 - 3x^2 - 12x + 3$

b. $2^{\sqrt{x}}$

c. $\log(e^x)$

d. $\pi^x + x^\pi + \pi^\pi$

e. π^{x^3}

4. Find the following integrals between the specified limits:

a. $\int 3^{2x} dx$ between 1 and 5.

b. $\int x(10^{-x^2}) dx$ between 1 and 2.

c. $\int \tan(x+\pi) dx$ between 0 and $\pi/2$.

Final Answers

1. First, purge 'A', 'W', 'T', 'B' and 'X' so that no unexpected substitutions occur. Also, set the angular mode to radians (MODE RAD) to simplify the resulting expressions.

Key in the expression 'A* $\cos(W*T+B)$ ' and then press $\frac{d}{dx}$. You get: 'A*(- $\sin(W*T+B)*W$)'. A little messing about with FORM will give you '-(A*W)* $\sin(T*W+B)$ '. This is v.

Now press $\frac{d}{dx}$ and get '-(A*W*($\cos(T*W+B)*W$))'. Using FORM again gives you a: '-(A*W^2)* $\cos(T*W+B)$ '.

You could have started with an equation instead. Since x is a function of t 'X(T)=A* $\cos(W*T+B)$ '. Differentiating this in the same manner as before will give you 'derX(T,1)...'. The HP-28S creates the name **derX** because it cannot find a function it knows is the derivative to substitute for X. It also tries to pass this non-existent function both T (since **derX** is a function of T) and $\frac{dT}{dT}$ which is 1. This action can be useful since it effectively labels the expression on the right of the = as dx/dt.

Then differentiating a second time will yield 'derderX(T,1,1,0)...' indicating the second derivative and appending dt/dt and $\frac{d^2t}{dt^2}$ (the derivatives of the original parameter list) to the parameter list.

2. First, purge 'X'. Then key in the expression ' $(X+1)^6$ ' and press $\boxed{\text{X}}\boxed{+}\boxed{1}\boxed{\wedge}\boxed{6}\boxed{=}$. You get: ' $X+3*X^2+5*X^3+5*X^4+3*X^5+X^6+.142857142857*X^7$ '.

Yes the polynomial was expanded. It is the same as if you had used TAYLR to expand the original binomial (see page 289) and then integrated the polynomial. As a matter of fact, since $\boxed{\text{=}}\boxed{\text{J}}$ will only *symbolically* integrate a polynomial, that's exactly what it did. Notice that $\boxed{\text{=}}\boxed{\text{J}}$ doesn't add the arbitrary constant C to the result.

If you prefer the result in its unexpanded form, you must do some mucking about first. You could perform a *u-substitution* where $u = x+1$ and $du/dx = 1$. Key in the ' $(X+1)^6$ ' and substitute 'U' for ' $X+1$ ' by one of the methods you know.

Now, integrate the expression with respect to u (i.e., press $\boxed{\text{'}}\boxed{\text{U}}\boxed{\text{ENTER}}\boxed{6}\boxed{\text{=}}\boxed{\text{J}}$) to get ' $.142857142857*U^7$ '. Now, resubstitute ' $X+1$ ' for 'U' and you have your result. Notice that this result differs from the first one by the addition of .142857142857 (i.e., $1/7$).

Notice also that this method was easy to do because $du/dx = 1$. Other cases of u-substitution will be more difficult when this isn't true.

3. Purge 'X' before doing the following:

- a. Key in ' $2X^3-3X^2-12X+3$ ' and press $\boxed{X} \boxed{\blacksquare} \boxed{d/dx}$ to get ' $2*(3X^2)-3*(2X)-12$ '. COLCT gives ' $-12+6X^2-6X$ '.
- b. Key in ' $2^{(\sqrt{X})}$ ' and press $\boxed{X} \boxed{\blacksquare} \boxed{d/dx}$ to get ' $.69314718056/(2\sqrt{X}) * 2^{(\sqrt{X})}$ '. .69314718056 is $\ln(2)$.
- c. Key in ' $\text{LOG}(\text{EXP}(X))$ ' and press $\boxed{X} \boxed{\blacksquare} \boxed{d/dx}$ to get ' $\text{EXP}(X)/(\text{EXP}(X)*2.30258509299)$ '.

COLCT gives .434294481904. Notice that LOG is common logarithm (base 10) and that the final result is $1/\ln(10)$.

- d. Key in ' $\pi^X + X^\pi + \pi^\pi$ ' and press $\boxed{X} \boxed{\blacksquare} \boxed{d/dx}$ to get ' $\text{LN}(\pi) * \pi^X + \pi * X^{(\pi-1)}$ '.
- e. Key in ' $\pi^{(X^3)}$ ' and press $\boxed{X} \boxed{\blacksquare} \boxed{d/dx}$ to get ' $\text{LN}(\pi) * (3X^2) * \pi^{(X^3)}$ '.

Notice that in these last two problems, if π were to be converted to a decimal number (such as by $\boxed{3} \boxed{5} \boxed{'} \boxed{C} \boxed{F} \boxed{\text{EVAL}}$), you would get an awful mess of unrecognizable numbers.

4. a. Key in ' $3^{(2*X)}$ ' and { X 1 5 }. Choose an accuracy of about 10^{-5} (i.e., press $\boxed{\text{EEX}} \boxed{5} \boxed{\text{CHS}} \boxed{\text{ENTER}}$) and press $\boxed{\text{F}} \boxed{7}$. You get a result of 26870.2622426 with an error of .268479611498 – that is, accurate to 5 decimal places.
- b. Put ' $X*10^{(-X^2)}$ ' (or ' $X*\text{ALOG}(-X^2)$ '), { X 1 2 } and an accuracy of .000000001 (i.e., 10^{-8}) on the stack and press $\boxed{\text{F}} \boxed{7}$. You get $2.16930093711 \times 10^{-2}$ with an accuracy of $2.16912752397 \times 10^{-10}$.
- c. Put the calculator in radians mode ($\boxed{\text{MODE}} \boxed{\text{RAD}}$, you'd like π radians not π degrees, thank you). Key in ' $\text{SIN}(X+\pi)$ ', { X 0 ' $\pi/2$ ' } and .000000000001 (i.e., 10^{-10}) and press $\boxed{\text{F}} \boxed{7}$. You get a result of -1 with an accuracy of $1.00001310044 \times 10^{-10}$.

Notice that, in general, the greater the accuracy, the longer the calculation time. Also notice that the limits of integration can be expressions (as in c).

Plotting

Information comes in many forms. You've seen numbers, letters, bits, and various and sundry compound information types built from them. Each of these forms has advantages based on how it's used and what you want to know.

Graphs are, in a sense, pictures of numbers. Such pictures give you easy access to (1) trends in collected data; (2) peaks and valleys in the output of functions; (3) comparison between different functions; (4) function zeros; et cetera, etc.

In short, graphs give you information about information.

And the HP-28S gives you the ability to generate graphical pictures of numeric information.

The DRAW function in the PLOT menu is basically a program written to automate the process of graphing real-valued *functions*.

A function, in this case, is anything that maps one real value onto another. As far as the HP-28S is concerned, therefore, the function can be an algebraic object, a postfix program, and even a constant or a name.

In the case of the postfix program, the function must be written so that it takes no values from the stack (i.e., it refers to values via names) and so that it leaves only one.

The first and second pages of the PLOT menu contain the operations you will need to set up and plot the function(s) of your choice.

Some examples of a postfix program and an algebraic object used to plot X^2+X-2 :

`'X^2+X-2'`

`« X DUP SQ + 2 - »`

The DRAW command assumes that the function contains only one undefined name. That is, since the plot will be two-dimensional, it must have one and only one name (the independent variable – the "x-value").

You may explicitly select the independent variable using INDEP. If you do not explicitly choose the independent variable, DRAW will scan the object and use the first name it finds as the independent variable.

The y-axis is used to indicate the value of the function given the current value of the independent variable. This does not mean that the algebraic object cannot have more than one name in it. It does mean that DRAW will only vary one of them as it successively evaluates the function. Thus, every other name had better have a value attached to it, otherwise DRAW will generate an error.

Scaling

Plotting is not always as simple as storing a function (i.e., via STEQ) and invoking DRAW.

Sometimes, in order to get the clearest picture of the function, you must have some idea of the scale of the plot and therefore the domain over which you want to plot (the x-values) and the range of the function (the y-values).

If you don't, it's possible that what you'll see may be so little of the curve that you can't get much information from it, or so much of the curve that you don't see important detail.

Try this: Press `'X` `^` `2` `+` `X` `-` `2` `▢` `PLOT` `▢` `STEQ` `DRAW`. The resulting plot has its "bottom" cut off. You know that this expression "bottoms out" when X is -0.5, but you can't see it.

Scale is established primarily by setting the minimum and maximum values of the domain and the range. You do this by making a complex number (x,y) out of the maximum x- and y-coordinates and then using the command PMAX. Then you do the same for setting the minimum values of the domain and range, except that you would use PMIN.

To correct the plot you just generated, press `▢` `ATTN` to leave the plot. Then press `(` `6` `.` `8` `CHS` `,` `2` `.` `2` `5` `CHS` `▢` `PMIN` `DRAW`.

Looking in the second level of the menu (via `▢` `ATTN` `▢` `NEXT`), you can increase or decrease the width of the plot with `▢` `W` (which multiplies the plot width by a constant). Numbers less than 1 will decrease the width (reduce the domain), while numbers greater than one will increase the width (enlarge the domain).

`▢` `H` works similarly with the plot height (the range, i.e. the y-values).

Digitizing

Once you have stored the function, established an initial scale, possibly selected the independent variable, and invoked DRAW, the HP-28S will plot the function. Depending on the function, this process may take some time. You'll know when it's done, because the busy annunciator will be turned off (but if you become impatient, ATTN will interrupt the plot).

When it's done, the plot is left for your inspection. Then you also have the option of digitizing some points. That is, you may move a special cursor (it resembles a +) around the plot to points of interest by using the cursor keys. Once you've found a likely spot, you can record its coordinates (put them on the stack) by pressing **[INS]**. Don't worry that you can't see the recorded value. It will be in the stack when you leave the plot display.

One reason to digitize points is to zero in on an interesting bit of the curve. You may pick two new points on either side of the interesting portion to be your new PMIN and PMAX. By digitizing those points, you'll have them on the stack and thus available for **PMIN** and/or **PMAX** and then replotting with a new scale.

Another good idea: You can use the digitized points as guesses for use in SOLVR.

One feature of DRAW that's not obvious is what it does with algebraic equations. Since an algebraic equation is essentially a pair of algebraic expressions separated by **=**, the DRAW command *plots both at the same time*.



The advantage to this is that the point(s) at which the two curves cross (if they cross) is the point at which the two expressions are equal (go after it with that **[INS]** key's digitizing capabilities!).


Keyboard Error Recovery

Anyone who's attempted to do any time-consuming thing is grateful for a method of recovering from false starts. No matter how careful you are, there will be times that you'll want to redo, undo, or throw away and restart whatever it is you're working on. You'll be happy to know, then, that the HP-28S has a set of "ways out" from false starts and blunders.

COMMAND : The Command Stack

You've already been introduced to the command stack (page 65), and after reading this far you've probably gained some appreciation for its utility.


To cover this ground again, the command stack contains copies of the last four command lines that you **ENTER**'ed. Repeatedly pressing  **COMMAND** recalls successively older command line copies to the active command line for you to edit and/or re**ENTER**. Pressing  **COMMAND** a fifth time cycles back to the most recent command line copy.


The advantage here is two-fold. First, if you've made a keystroke error in a long or involved command line,  **COMMAND** will allow you to recall that command line if it's not too old, then correct it and re-enter it (note that re-entering simple command lines is often easier than using the command stack).

Second (as in the quiz solution on page 109), you can use the command stack to repeat lengthy and redundant commands. It's also a convenient way to enter a series of slightly different commands (see page 82). And notice that, since immediate-execution commands are not normally recorded in the command line, you may need to do some planning ahead and use **⌘** for this kind of command repetition.

UNDO : UNDOing a Command

Sometimes you will find that you will need to UNDO whatever it is you just did. It may be that you did something that you didn't intend to do, or perhaps that last command ate your only copy of some important datum. Never fear, you've a way out.

Any time you press **ENTER**, or any time you press an immediate-execution key since it effectively "presses **ENTER** on itself" (see page 33), the HP-28S makes a secret copy of any stack Levels that are changed by the invoked commands. The advantage of this is that if you need to undo something, you can then press  **UNDO** and the following things will happen:

1. Any and all results of the last command will be dropped from the stack.
2. The stack contents eaten by the last command will be replaced – pushed back onto the stack.
3. UNDO will have amnesia. I.e., pressing  **UNDO** a second time will not undo the UNDO, nor will it repeat its action. It will become active again only after another **ENTER**-pressing command has been invoked.

Things will then be as if you had never invoked that last command.

As you can see, this is tremendously convenient. In fact, UNDO will probably be the most commonly used error-recovery mechanism in your arsenal.

■ **LAST** : Recalling the Stack as It Was Before the LAST Command

LAST is a slightly different flavor of UNDO. Here's a list of its actions, so you can compare it with UNDO:

1. Any and all *results* of the last command will be *left on* the stack.
2. The stack contents eaten by the last command will be replaced; pushed back onto the stack.
3. LAST *won't* have amnesia. I.e., pressing ■ **LAST** a several times will repeat its action. You'll get several copies of the remembered stack Levels.

The main use for LAST is when you have an especially gnarly object on the stack and you need to do several operations on it. You don't need to re-**ENTER** it. Rather, you can simply press ■ **LAST** between operations, to recall the original gnarly object for re-use. Meanwhile, the results of the different operations would be pushed onto higher Levels of the stack.

Enabling and Disabling Error Recovery

A final point of interest about error recovery: you can turn it off. On the second page of the MODE menu are commands for turning on and off each of these error recovery schemes.

But why would you want to turn them off? You never know when you might need them.

The answer is: to conserve memory.

Since each of these mechanisms works by remembering (storing) something in case you want it again, there will be times when it's just plain wasteful to take up memory with something that's only *potentially* useful.

Consider also the case of especially large objects: there may simply not be enough memory to remember the last large object *and* put the next one on the stack, too.

Editorial

Well now...having dug deep into the nitties and the gritties of the HP-28S, you should probably poke your head back into the fresh air and catch a new perspective (or maybe re-catch an old one). It's too easy to get lost in the details and how-to's and forget the big picture. It'd be a shame to lose sight of exciting potentials while mired in the mundanity of getting the basics under your belt.

So in case you've forgotten, it's time you were reminded how much you have to be excited about. If you're a serious problem-solver, by now you should be feeling like a kid in a candy store, or maybe like an auto connoisseur at a new car show. There's so much here, so much you can do, and so many new ways to do it that the mind delights – and maybe boggles a little bit too. That's okay. It's all part of the excitement.

Just remember to be excited.

What you have in the HP-28S is more than the Cadillac of calculators. It's more like being on the freeway at rush hour and finding out that your vehicle can fly. You're no longer bound to the pavement. You don't need to go where everyone else is going before going where you want to go. You have a whole new way to travel. Not only can you get there (wherever there is) faster, easier and more directly than anyone else, you can also go places that they can't.

If you haven't caught the drift by now, here it is: This machine is "radibolical."

You just ain't never saw nothin' like it nohow nowhere before, but you can bet you'll be seeing more of it. It's just too useful and too good an idea not to catch on, and if it doesn't, it's because we aren't ready for it – like di Vinci's helicopter.

Don't be cowed by its power and flexibility. Take it slow and get to know the most capable problem-solver you've ever met. He's a little short on small talk, but in his element, he's "dynobitchin'."

Can you tell we like this machine? And actually, we've realized its flexibility even more during the writing of this book. Back on page 9, we called the HP-28S a problem-solving tool, but you can see now that it's really a *collection* of different tools and attachments – more like a full toolbox, actually.

You can also see how hopeless it would have been to try to cover everything in this book, so, true to our early warnings, we didn't try to tell you how to build a house. The design of your house is your job (but once you decide where and how the boards ought to go together, do we have a toolbox for you)!

Just remember – with all the real satisfactions you should get from such great tools – you'll be wasting them if you build more house than you need.

This seems to be true of a lot of modern inventions. Two related questions come up over and over again: What peaks of performance can you squeeze out of them? *Should* you push them that far?

Like most machines, the HP-28S answers these two questions very differently, and being so representative of the machine age, it therefore gives you a chance to begin asking better questions about our technology. Instead of idly wondering "How many neat-nifty-awesome-but-useless things can I make this little box do?" we hope you'll ask "What better things can I do with the time and energy these tools save me?"

For without sorting out the advisable from the possible, you'll be no better off than before you ever had these tools. Your time and talents will have gone merely to "gee-whiz" tinkering; you and the world will be the poorer for it. A sophisticated machine may be the subject of a course, but it's not the object of the game.


Thanks for listening...and happy hammering!

A handwritten signature in cursive script, appearing to read "John W. Jones".A handwritten signature in cursive script, appearing to read "Chris Coffey".

February, 1988

By the way, if you liked this book, here's a full list of books that you or someone you know might enjoy:

- An Easy Course in Using the HP-27S
- An Easy Course in Using the HP-17B
- An Easy Course in Using the HP-19B
- An Easy Course in Using the HP-28S
- An Easy Course in Using the HP-28C
- An Easy Course in Using the HP-16C
- Computer Science on Your HP-41 (Using the HP Advantage ROM)
- The HP Business Consultant (HP-18C) Training Guide
- Statics For Students (Using the HP Advantage ROM)
- The HP-12C Pocket Guide
- An Easy Course in Programming the HP-11C and HP-15C
- An Easy Course in Using the HP-12C
- An Easy Course in Programming the HP-41

You can use this handy set of order forms here 

Or, you can contact us for further information on the books and where you can buy them locally:

Grapevine Publications, Inc.

P.O. Box 118

Corvallis, Oregon 97339-0118

U.S.A.

Phone: 1-800-338-4331

(Within Oregon: 754-0583 Outside US: 503-754-0583)

ORDER FORM (*Impress a Friend!*)

"Yes!" Please send:

_____ copies of <i>An Easy Course in Using the HP-28S</i>	@ \$22 ea.= \$_____
_____ copies of <i>An Easy Course in Using the HP-19B</i>	@ \$22 ea.= \$_____
_____ copies of <i>An Easy Course in Using the HP-17B</i>	@ \$22 ea.= \$_____
_____ copies of <i>The HP Business Consultant (HP-18C) Training Guide</i>	@ \$22 ea.= \$_____
_____ copies of <i>An Easy Course in Using the HP-12C</i>	@ \$22 ea.= \$_____
_____ copies of <i>The HP-12C Pocket Guide (Just In Case)</i>	@ \$ 5 ea.= \$_____
_____ copies of <i>An Easy Course in Using the HP-27S</i>	@ \$22 ea.= \$_____
_____ copies of <i>An Easy Course in Using the HP-28C</i>	@ \$22 ea.= \$_____
_____ copies of <i>An Easy Course in Programming the HP-11C and HP-15C</i>	@ \$22 ea.= \$_____
_____ copies of <i>An Easy Course in Programming the HP-41</i>	@ \$22 ea.= \$_____
_____ copies of <i>Computer Science on Your HP-41 (Using the Advantage Module)</i>	@ \$15 ea.= \$_____
_____ copies of <i>Using Your HP-41 Advantage: Statics For Students</i>	@ \$12 ea.= \$_____
_____ copies of <i>An Easy Course in Using the HP-16C</i>	@ \$22 ea.= \$_____

(Prices and availability may change without notice.)

Subtotal = \$=====

SHIPPING INFORMATION:

For orders less than \$10.00

ADD \$1.00 \$_____

For all other orders- **Choose one:**

Post Office shipping & handling ADD \$2.00 \$_____
(allow 3 weeks for delivery)

or
UPS shipping & handling ADD \$3.50 \$_____
(allow 7-10 days for delivery)

TOTAL AMOUNT: -----> \$

plus
↓
\$_____

\$

PAYMENT:

Your personal check is welcome. Please make it out to **Grapevine Publications, Inc.** or:

Your VISA or MasterCard #: _____ Exp. date: _____

Your signature: _____

Please ship my book(s) to:

Name _____

Address _____

Note: UPS will **not** deliver to a P.O. Box! Please give a street address for UPS delivery.

City _____ State _____ Zip _____

OR order by phone with a VISA or MasterCard!

Call **1-800-338-4331** (In Oregon: 754-0583)

Thank You !

Reader Comments

We here at Grapevine love to hear feedback about our publications. It helps us write books tailored to our readers' needs. If you have any specific comments or advice for our authors after reading this book, we'd appreciate hearing them!

Which of our books do you have?

Comments, Advice and Suggestions:

May we use your comments as testimonials?

Your Name:

Profession:

City, State where you live:

How long have you had your HP calculator?

Please send Grapevine Catalogues to the following people:

Name_____

Address_____

City_____State_____Zip_____

and

Name_____

Address_____

City_____State_____Zip_____

Tell 'em You Heard it Through The Grapevine...
(Sorry – we just *had* to use that in here somewhere)

An Easy Course in Using The HP-28S

This cover flap is handy for several different things:

- Tuck it just inside the front cover when you store this book on a shelf. That way, you can see the title on the spine.
- Fold it inside the back cover--out of your way--when you're using the book.
- Use it as a bookmark when you take a break from your reading!



An Easy Course In Using The HP-28S

If you're looking for a clear, straightforward explanation of the powerful HP-28S, then this is your book! Authors Loux and Coffin sort through the myriad features of this machine, giving you the pictures and the practice you need to make the HP-28S your favorite calculating tool.

The first several chapters bring you up to speed on the mechanics of operation – what keys you need to press to control and command the display, the stack, and the menus. You'll get lots of practice problems and explanations designed to get your fingers trained for action.

Then you go straight to the heart of the machine, exploring all the different information "objects" and how you can manipulate them, combine them, name them and (best of all) think about them. You'll see how HP's well-known stack-oriented (postfix) arithmetic becomes the engine behind all this math power, and soon you'll be harnessing it for yourself!

Then in separate discussions, this Easy Course touches upon specialized topics, such as symbolic algebra, calculus, plotting, and programming.

It's all in Grapevine's familiar Easy Course format – a book full of examples, review questions, and quizzes, designed to let you work at your own speed (and your own speed will soon amaze you)! It's a very pleasant surprise to find that learning about a calculator can be this satisfying – when the right explanation transforms a mysterious machine into a friendly and powerful tool.



**FROM THE PRESS AT
GRAPEVINE PUBLICATIONS, INC.**

P.O. Box 118 • Corvallis, Oregon 97339-0118 • U.S.A. • (503) 754-0583

ISBN 0-931011-18-3



HP Part # 92236D