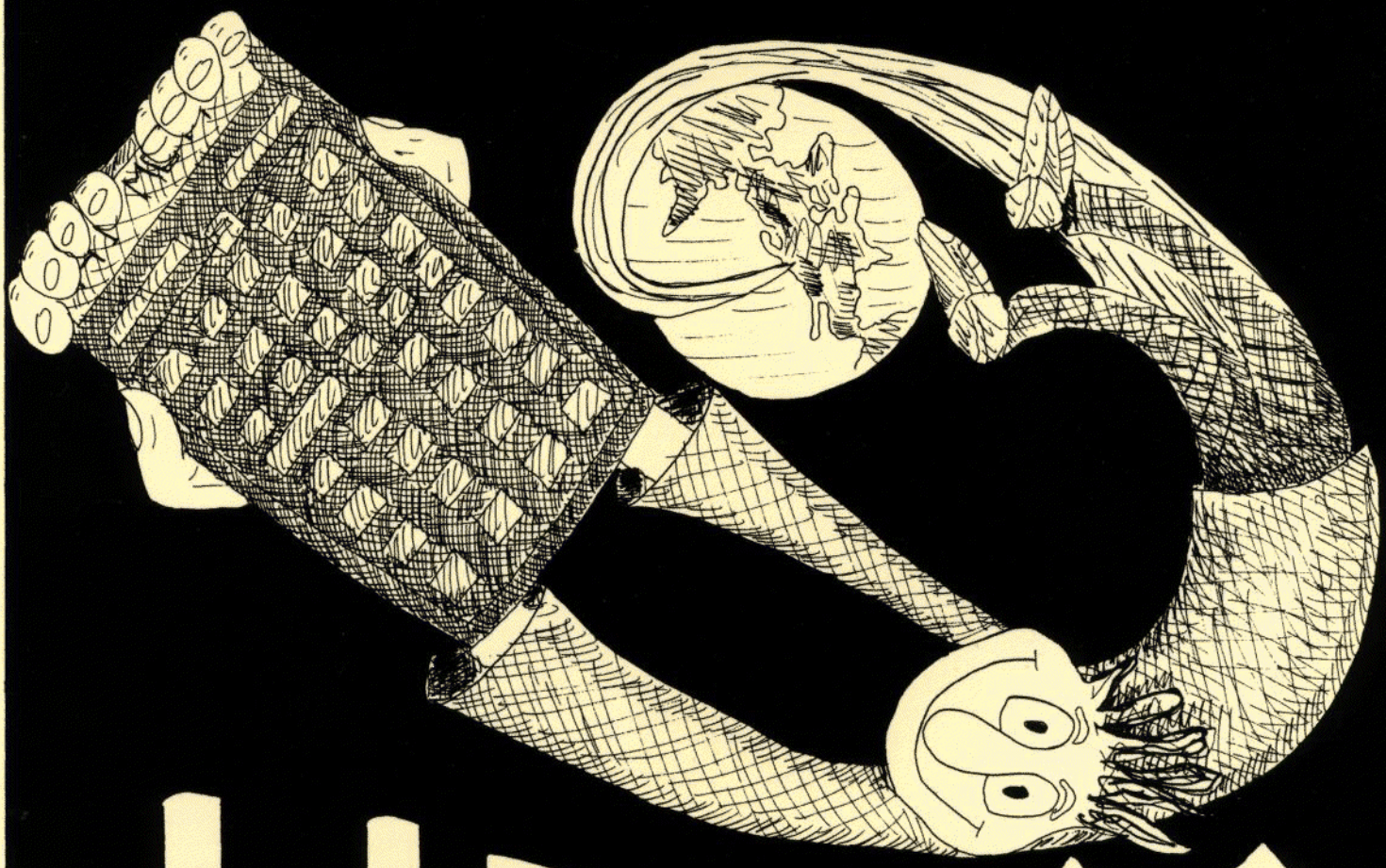
An Easy Course In Programming The



HP-41

By Ted Wadman and Chris Coffin
Illustrated by Robert Bloch

AN EASY COURSE
IN
PROGRAMMING THE HP-41

By Ted Wadman and Chris Coffin

Illustrated by Robert Bloch

Grapevine Publications, Inc.
P.O. Box 118
Corvallis, Oregon 97339
USA

Ninth Printing - March, 1989

For reasons of brevity and clarity, the name "HP-41" has been used throughout this course to denote the "HP-41C," "HP-41CV," or the "HP-41CX," which are the complete names of the handheld computers made by:

The Hewlett-Packard Company.

We extend our thanks to Hewlett-Packard for producing such top-quality products and documentation.

You have an HP-41!

Now... what are you going to do with it?

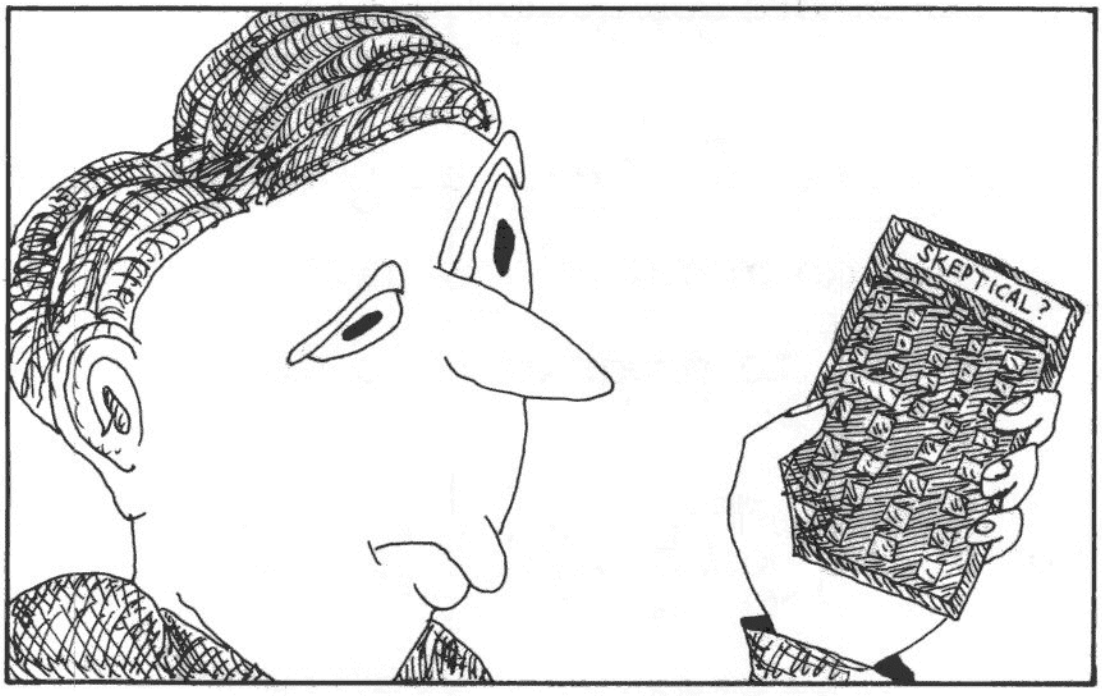
You bought an HP-41 because you were sure that it could do many things. But you are not quite sure how to program it to do those things.

That's why you bought this book ... ⟶

Well, you're right. The HP-41 can do many, many things.

That's what is meant when it's called a powerful machine. That doesn't mean it can leap tall buildings in a single bound, or predict the future, or appreciate music. The HP-41 is not a magic box. It is only a tool to help you solve problems, and the reason it can be used to solve so many different problems is that it is very...

SIMPLE (REALLY) →



Think about it this way: you can build many different things with a pile of bricks, but the bricks themselves are very simple. They are the fundamental building blocks you use to build anything you want.

That's exactly what a computer like the HP-41 really is: just a pile of bricks, so to speak.

By giving the computer instructions on how to manipulate numbers and letters, you can build the solution you are looking for, step-by-step, brick-by-brick.

Now, when a bricklayer is finished, the building looks large and complex and very impressive ... and it is. The completed building is very useful as well. But there's no mystery about how it was done. You saw the bricklayer putting the bricks together, one-by-one.

That's how it is with the HP-41. It's just one simple brick after another. You just need to know how to put the bricks together.... So, let's mix some mortar. →

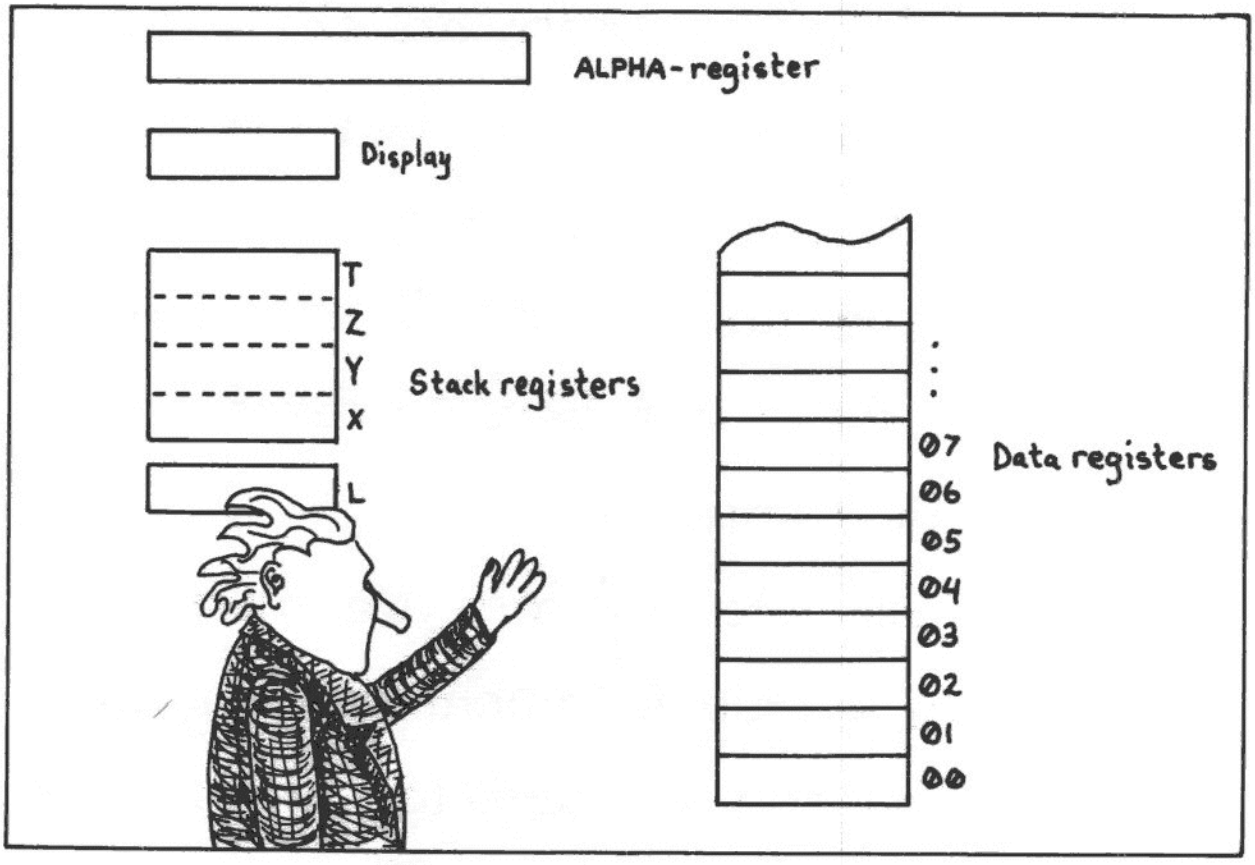
(Mix, Mix, Mix, ...)

HOW DO THE BRICKS GO TOGETHER?

How does the HP-41 manipulate numbers and letters for you?

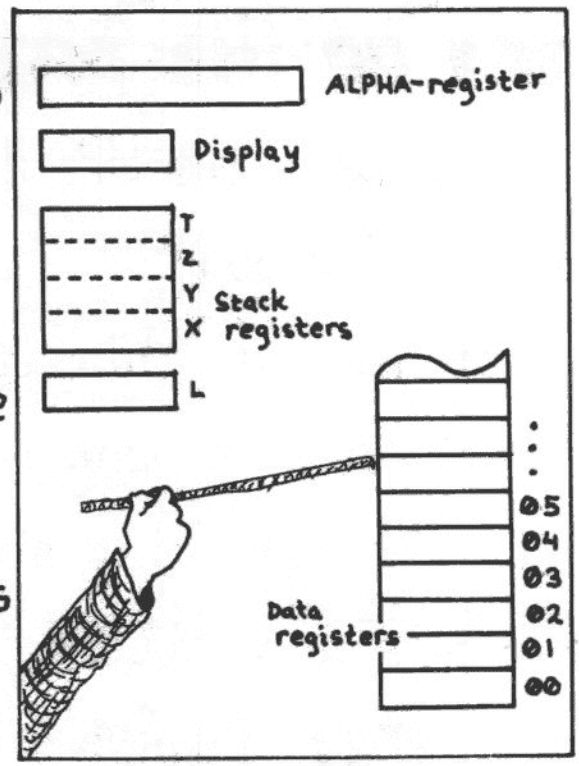
In learning this, the first thing you need to have is a good way to picture in your mind just <u>how</u> the computer stores and keeps track of numbers and letters in its continuous memory.

PICTURE IT LIKE THIS:



If you know all about this already then go on ahead to page 18 . Otherwise ... ⟶

DATA REGISTERS



Just for convenience, we call numbers and letters "data," and so it makes sense that a data register is a place to store data - numbers and letters.

All of the data registers are the same size. Each can hold one number or up to six letters (but not both at the same time).

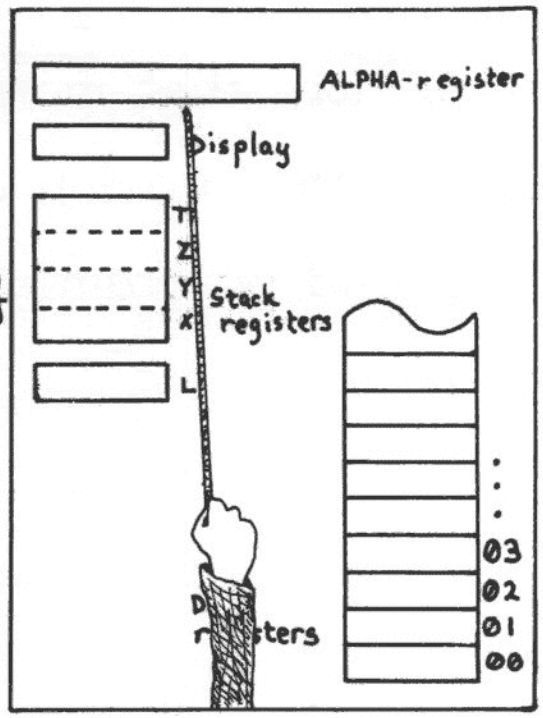
Data registers are numbered, starting at 00. You'll use these numbers to refer to the registers in the computer.

So how many data registers are there?

Well, there's only so much memory space in the HP-41. You can use some (or all) of it for data registers - to store data. Then you can use the remainder for storing programs - the

instructions that manipulate this data. You can change the location of the boundary between data and program memory anytime, as you'll see later.

ALPHA-REGISTER



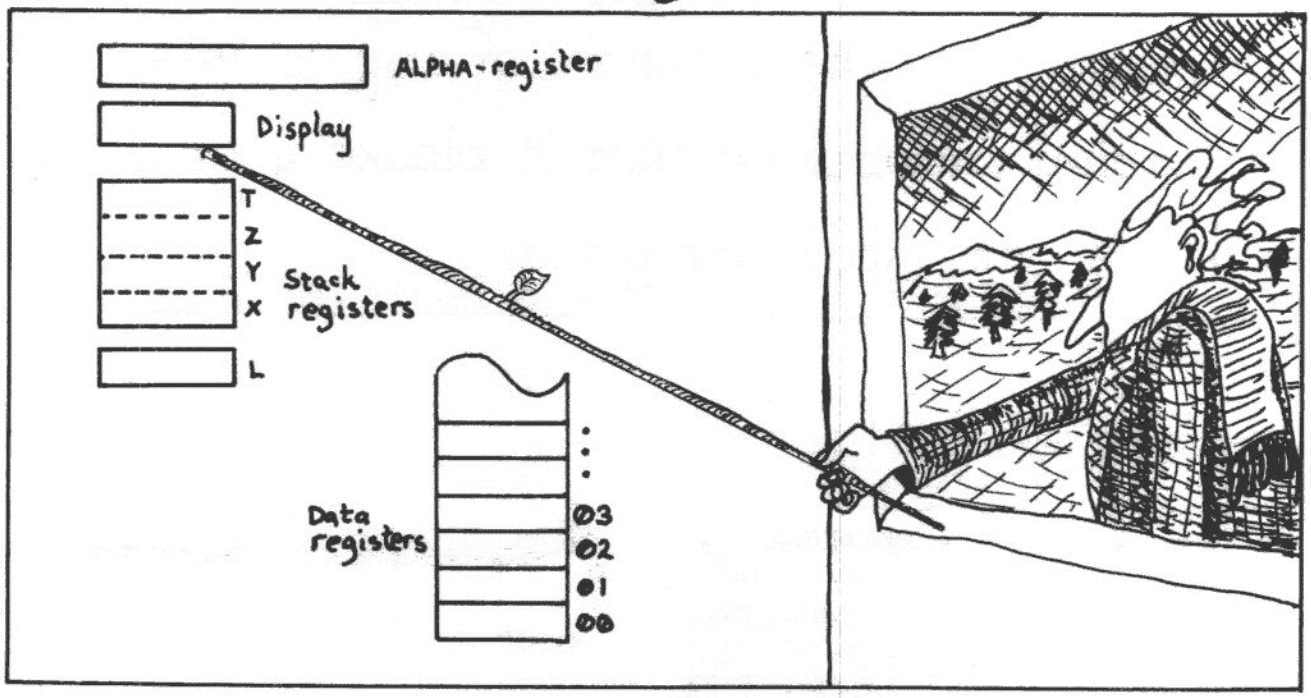
This special register is what helps to make the HP-41 an <u>especially</u> powerful tool. With the help of this register, the computer can store and manipulate <u>letters</u> as well as numbers.

Now, when we say letters, we really mean more than just A to Z. Actually, we are referring to most of the characters you can produce with a standard typewriter. And, for this reason, we should start <u>calling</u> them characters – ALPHA characters.

The ALPHA-register can hold only ALPHA characters. It can hold up to 24 of these. BUT IT CAN <u>NEVER</u> HOLD ANY NUMBERS.

If you see "12" in the ALPHA-register, this is simply the numeral "1" sitting next to the numeral "2". It may as well be "?K". They're all just characters. There's no number twelve in sight. And if you ever try to get the HP-41 to do arithmetic with ALPHA characters, it will refuse to do it.

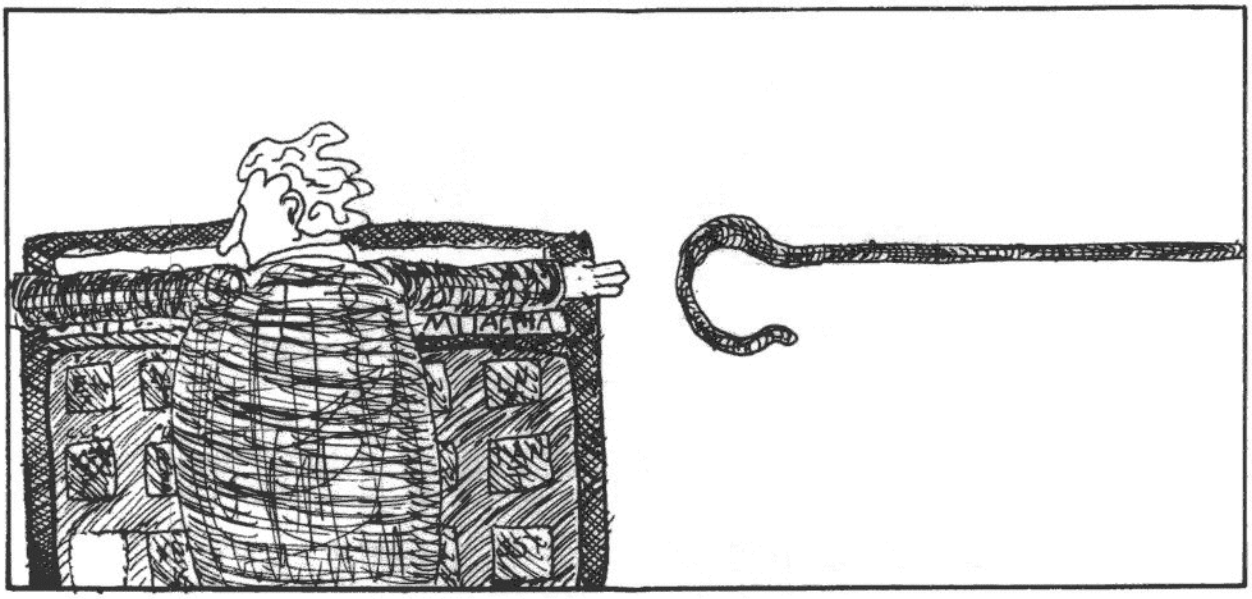
THE DISPLAY



You can think of the display as a window. When you look at your computer and there is nothing in front of this window, then you can look through this window to see the contents of some register.

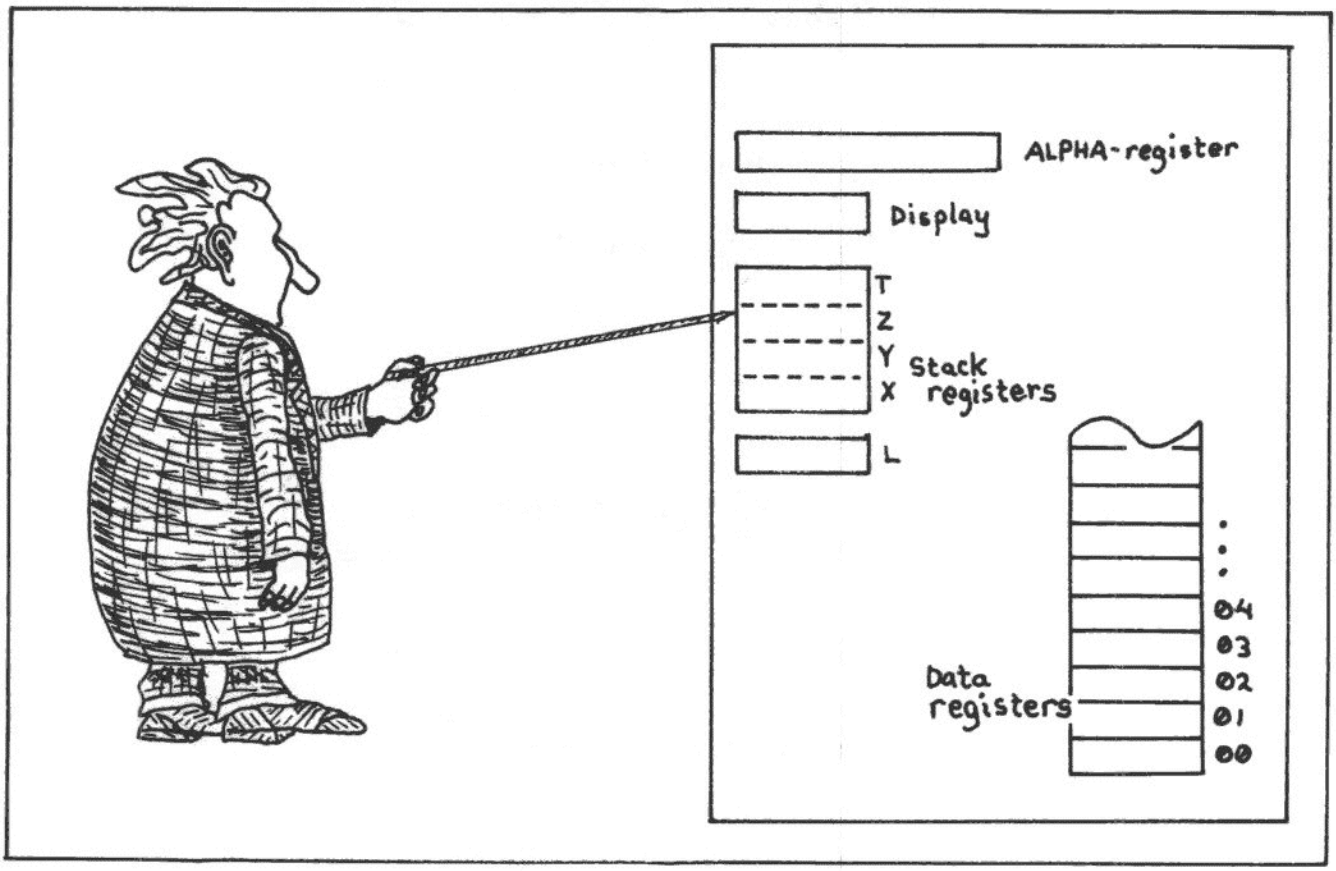
This window we call the display can be "located" over either the X-register or the ALPHA-register. Then, if the window is clear, you see the contents of one of those registers. But if there's something "in front" of this window, then <u>that</u> is what you see when you look at the display. Usually you see the contents of the X-register or the ALPHA-

register <u>through</u> the display window. But it's possible to move the contents of any register into the display window so that it blocks your view of the X- or ALPHA-register.



When you are looking at a number in, say, the X-register, you can use the display to "block out" part of your view of that register by fixing the decimal places that you wish to see. The <u>display</u> does this by rounding the number. But <u>remember</u>, the display does this screening just to <u>show</u> you the part of the number you want to see. It does <u>not</u> change the actual number as it sits in the X-register.

STACK REGISTERS



The stack registers are really just data registers. They are the same size as the numbered data registers, but they are named with letters (rather than numbers):

X, Y, Z, T, and L.

The reason they are called "the stack" is because they are stacked on top of one another.

The stack registers are linked together in a special way, and they are the most frequently used registers in the HP-41.

POP QUIZ

1. What's a data register?

2. How many data registers are there?

3. What kind of register is the Z-register?

4. How many numbers can the ALPHA-register hold? How many ALPHA-characters?

5. If you looked at your computer and it read "HELLO," what register would you be looking at?

GIVE UP? ⟶

POP ANSWERS

1. A data register is a place to hold data. It can hold one number or up to six ALPHA characters, but not both at the same time (Review page 11).

2. The number of data registers depends on the location of that boundary between data and program memory. That boundary is something that you can adjust (page 9).

3. The Z-register is one of the stack registers. It is a data register, but it is named with a letter to remind you that it belongs to the stack - that collection of specially linked data registers (page 14).

4. The ALPHA-register cannot hold <u>any</u> numbers. That's why it's called "ALPHA" - it can hold only ALPHA characters. It can hold up to 24 ALPHA characters (page 11).

5. You may not be looking at <u>any</u> register. It's entirely possible to have "HELLO" in the display and nowhere else (page 13).

How Did You Do?

Do you feel that you have a good "mental picture" of the different parts of the memory and what they're used for?

If so, then... go for it →

If not, then go back and let it soak in a little bit more.

Re-read the pages that bounced off the first time. It's worth it, even if it takes a couple extra minutes.

Now that you know where the computer stores data, it's time to learn how to get data into it in the first place.

TOGGLE KEYS

The first thing to do, of course, is to turn on the HP-41. Press the [ON] key at the upper left.

Question: Where's the [OFF] key?

Nice try, right?

Answer: There is no [OFF] key. To turn off the HP-41, just press [ON] once again.

This kind of key is called a <u>toggle</u> key. If you know all about toggle keys, move to the next page.

A toggle key is any key whose meaning alternates between two opposites (such as ON and OFF, or RUN and STOP, etc.).

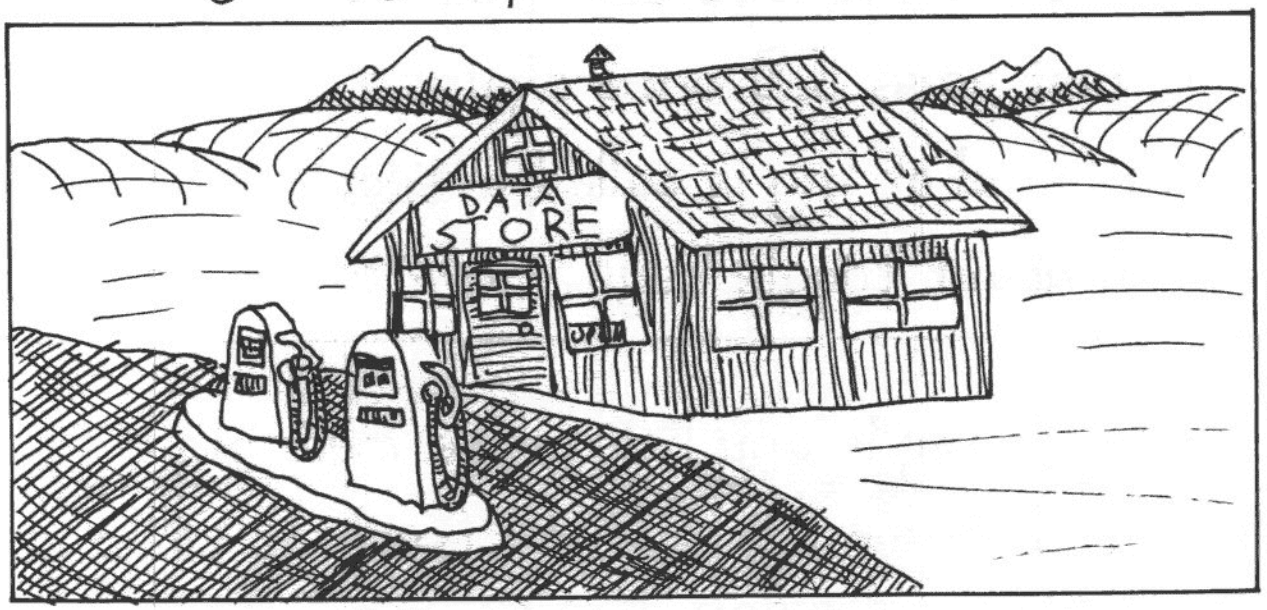
Thus, if your computer is off, then the [ON] key means "turn on," but if your computer is already on, then the [ON] key means "turn off."

TO STORE SOME DATA

There are only two ways to key data into the HP-41.

1. To enter numeric data (numbers), you just key them in and they're stored in the X-register. (Remember the X-register? It's one of the stack registers.) All numeric data (all numbers) go into the X-register when you first key them in.

For example, if you want to store the number -1.2345 in the X-register, just press [1] [.] [2] [3] [4] [5] [CHS].



2. All ALPHA data (all characters) go into the ALPHA-register when you first key them in.

Easy to remember, right? Let's see...

Before you go on, if the little word USER appears in your display, press the [USER] key.

Challenge: Enter the characters ABCDE into the ALPHA-register.

Solution: Press [ALPHA] [A] [B] [C] [D] [E] [ALPHA].

SATISFIED? 

NO?

Think about it this way:

Anytime you press the [ALPHA] key before you press any other key, you are ready to enter characters into the ALPHA-register.

When you're finished with this entry, just press the [ALPHA] key once again to restore the computer to its normal readiness for numeric entry.

Notice that the [ALPHA] key is a toggle key.

FUNCTIONS

Now that you can picture how the HP-41 stores data and how you can get it into the computer to start with, it's time to start thinking about how it combines data and moves data around from one register to another.

Let's agree to call any of these "combining" or "moving-around" actions by one name:

FUNCTIONS

A function is any <u>action</u> that the computer takes ... period. If this is clear, go on. ⟶

Look, for example, at the <u>function</u> called X<>Y. (Say "X exchange Y.")

As the name implies, this function simply exchanges the contents of the X- and Y-registers. That's it. X<>Y doesn't do any arithmetic at all.

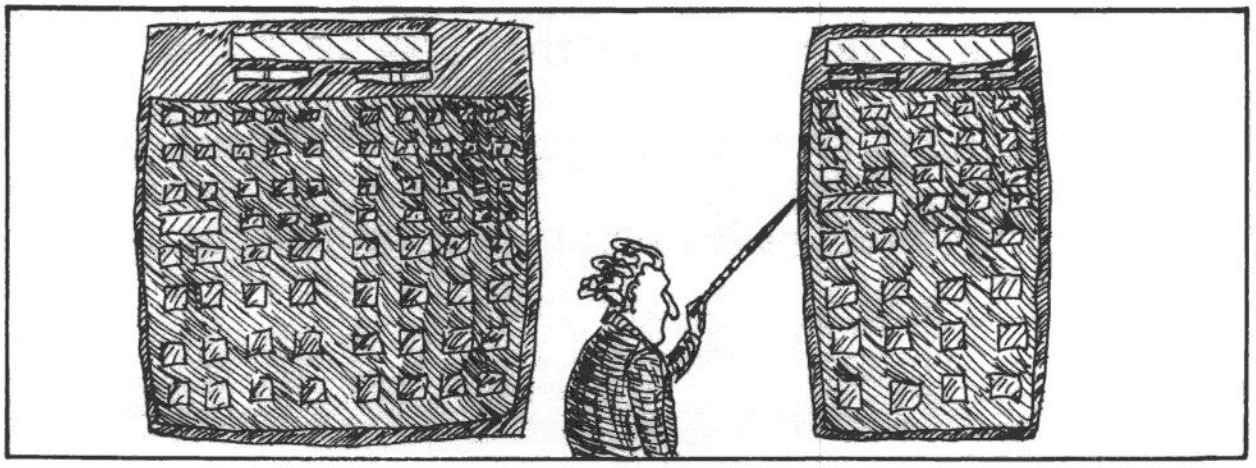
Now, the function + also does something to the X- and Y-registers, but it <u>combines</u> numbers as well as moving them around. It puts the sum of these two registers into the X-register.

All told, the HP-41 has well over 100 different functions you can choose from.

Question: There are only about 39 keys on the keyboard. How can we possibly use over 100 different functions?

If you know the answer, move on. 

First, that gold key (called the "shift" key) allows you to change the meaning of any key, if you press the gold key just prior to pressing the desired key. You don't have to hold down this shift key while you press the desired key. Just think of the *shift key* as a "prefix" key. This little feature virtually doubles the number of keys as far as the computer is concerned, just as a typewriter shift key allows each key to print either upper- or lower-case.



Challenge: Use the BEEP function to make the computer beep at you.

Solution: [shift] [BEEP]

NO PROBLEMS? →

Think about it this way:

The [4] key means just that: when you press this key, you key the number 4 into the X-register (or it's part of another number you're keying in).

BUT, if you press the shift [shift] key before pressing the [4] key, then the computer will beep. And, sure enough, written in gold just above the [4] key is the word BEEP. Pressing the gold key will cause all the keys to change their meanings to the functions written above them in gold.

Notice the little word SHIFT that appears in your display whenever the keys have switched their meanings. Notice also that the shift key is a toggle key.

YES, BUT...

Even with the shift key, we've accounted for less than 80 functions on the keyboard.

What about the rest of them? How can we tell the computer to perform them?

Well, we spell them. This is one of the great things about the HP-41. Its ALPHA capabilities allow you to spell out your instructions.

Challenge: Tell the computer to BEEP at you by spelling out the name of the function. Don't use [shift] [BEEP].

Solution: [XEQ] [ALPHA] BEEP [ALPHA]

First, find the [XEQ] key. That key is the "execute" key. (If you say the letters X-E-Q, they sound like the word "execute," and XEQ is short enough to fit on a key.)

Whenever you press the [XEQ] key before any other key, the computer is immediately alerted. It has just

been told that it is about to be asked to do something.

TRY IT NOW: XEQ

To let you know that it is listening, the HP-41 responds by putting the letters XEQ into the display (and only the display - not into the ALPHA-register or anywhere else). So those letters are now in front of that "window," as we called it, and any letters that you key in to follow XEQ will go into only the display.

Now you should see XEQ __ in your display. The computer is ready and waiting for your command. To spell anything, you must be "IN ALPHA MODE." So press the [ALPHA] key and be sure that the little word ALPHA appears in the display. Whenever this ALPHA annunciator appears in the display, the keys all change their meanings - just as they did with the [Shift] key - only this time all the keys become letter keys. Each key corresponds to a character, and these characters are printed in blue on the front faces of the keys. Also, the

display window moves from in front of the X-register to in front of the ALPHA-register.

The computer now knows that you want to "execute" a function and that you are now going to spell it out.

So you're going to key in the word BEEP? But, wait a minute! Why would you ever spell it out when there is a key waiting to be used for BEEP?

In this case, of course, you'll spell it because we told you to spell it. True, it is more convenient to use one of the keys, if you can. For this reason, you'll find that the functions you use most often do have keys of their own.

But the point is, when you press a key you are telling

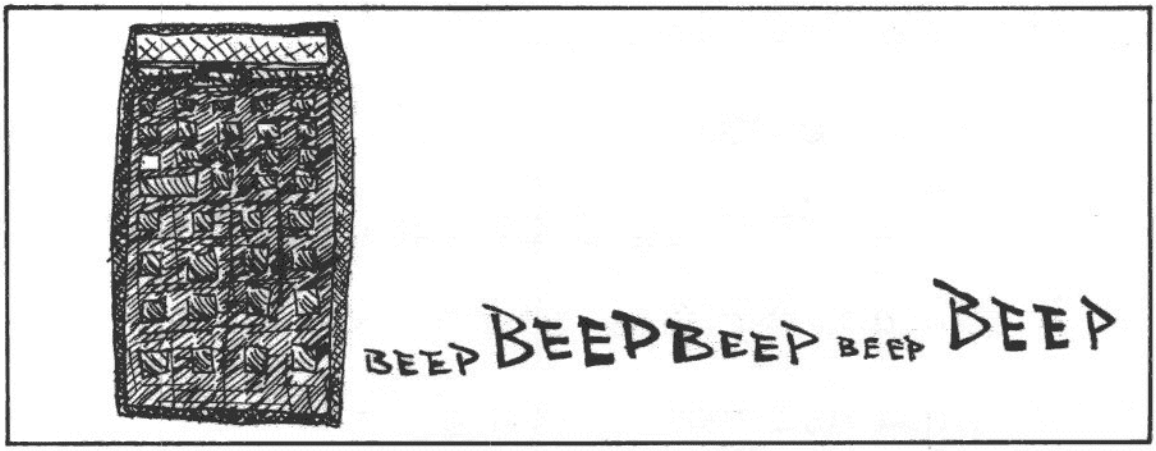
the HP-41 to execute a function - just as if you had spelled out that function name using the [XEQ] [ALPHA] ... procedure. And the computer will respond exactly the same in either case.

So go ahead and key in BEEP.

Your display should now be: XEQ BEEP_, with the ALPHA annunciator still on. Turn that annunciator off by pressing the [ALPHA] toggle key once again.

IT BEEPS!

It's supposed to beep. Whenever you finish the [XEQ] [ALPHA] ... procedure by turning off the ALPHA annunciator, you are telling the computer "Hey, HP-41! I'm finished telling you what to do - so do it!"



AND IT DOES IT

Suppose now that you have some message in the display and you want to clear that "window" so that you can look through it and see the X-register or the ALPHA-register (but you don't want to destroy the contents of that register).

Question: How do you do it?

Suppose, then, that you wish to clear the contents of that register (X or ALPHA).

Question: How do you do it?

Or what if you are keying in a number to the X-register or a string of ALPHA characters to the ALPHA-register and you make a mistake (key the wrong number or letter)? How do you correct the mistake without clearing the entire contents of the register and starting over?

Answer: The answer to all three of these questions:

USE THE ← KEY

If all this is review, proceed to page 31.

As you may have noticed by now, the only two registers that the display "window" can "cover" (the X-register and the ALPHA-register) are also the only two into which you can directly key data. (Hmmm...)

So the [←] (backarrow) key must be allowed to do several things.

1. If you have something (a message) in front of the display window so that it is blocking your view of the X- or ALPHA- register, and you want to clear that message (and only <u>that</u> message), then press [←].

Under these circumstances, the [←] means "clear the display only."

2. If there's nothing "in front" of your display window, so that you have a clear view of either the X- or ALPHA-register, then the [←] key means:

"Clear the X-register back to zero" if you're looking at the X-register.

OR

"Clear the ALPHA-register" if you're looking at the ALPHA-register.

3. If you're in the process of keying in the digits of a number or the characters of an ALPHA string, then the [←] key means: "Delete the last digit or character keyed in." In other words,

BACKSPACE!

Three different uses: Not bad for one little key!

Also, notice that the gold function written above the [←] key is CLX/A (clear X/clear ALPHA).

This means that [shift] [←] will either clear the X-register or the ALPHA-register, depending on where the display is located at the moment. [shift] [←] will <u>always</u> clear a register as well as the display, whereas just [←] may not clear a register-as in cases 1 and 3 above.

So, on with more and more functions ....

How can anyone remember them all? And how to spell them correctly?

Suppose you forget that X<>Y has a key of its own, and you want to spell it using the [XEQ] [ALPHA] ... procedure, but you can't remember how to spell the name of this function, and you left your Owner's Handbook in Siberia?

Question: What do you do?

a. Lump it

b. Cry

c. Panic

d. None, or all, of the above

Answer: The answer is d with the "none of the above" option. Instead, use [shift] [CATALOG] 3 (CAT 3), and then stop the listing so that X<>Y shows in the display.

Now, if you already know all about this little convenience, move ahead to page 34.

Otherwise, ... attend:

The CATALOG function tells the computer to run briefly through an internal list. The third such list, CATALOG 3, is a list of all the functions. One-by-one, in alphabetical order, each function name is brought, very briefly, in front of the display window.

Now, anytime the HP-41 is busily doing something automatically, and you want to stop it, chances are that you can do so by pressing [R/S]. This key is the "run-stop" key, and it's called that because it's a toggle key that alternates meanings between "run" and "stop."

If you've just executed CAT 3 (CATALOG 3) and the computer is ripping down the list, and you're watching breathlessly for X<>Y, relax.

First, you don't need an acute sense of timing to get to the proper entry. If you <u>do</u> choke under the pressure and press [R/S] to soon, the list will stop, of course. But pressing [R/S] again will re-start it.

Second, once you've stopped the list, you can use the [SST] (single-step) and [BST] (back-step) keys to

step forward or backward, one function name at a time, through the list. You can get anywhere, using this method, in any CATALOG.

Challenge: Adjust the computer so that you will see 9 digits after the decimal point, wherever possible.

Solution: [SHIFT] [FIX] 9

OK?

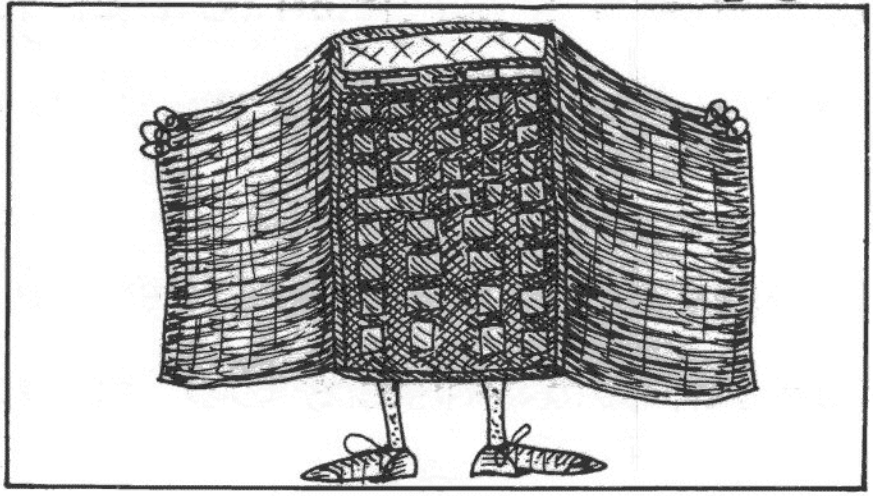
Next page. →

NO?

Remember, this is all that FIX does. You can FIX from 0 to 9 decimal places, as you wish.

Also, remember that no matter how many decimal places you actually <u>see</u>, the computer still "knows about" and works with all possible digits (10 altogether).

The HP-41 merely uses the display (<u>not</u> any register) to round as you specify when <u>showing you</u> a number.



Now, remember when we promised that you would be able to decide how many registers the computer would have?

It's time to learn how to do this.

Challenge: Adjust the total number of data registers in your computer to 20.

Solution: [XEQ] [ALPHA] SIZE [ALPHA] 020.

OK? Move ahead 

Well then.... SIZE is just another function—one that demands a three-digit <u>argument</u>, rather than the usual 2.

All that SIZE does is move the partition between the last data register and the .END. of program memory. It moves this partition so that you end up with exactly the number of data registers you requested.

So, after you ask for 20 data registers, you have 20 of them, namely, data registers 00 through 19.

Challenge: Recall the contents of data register 01 to the X-register.

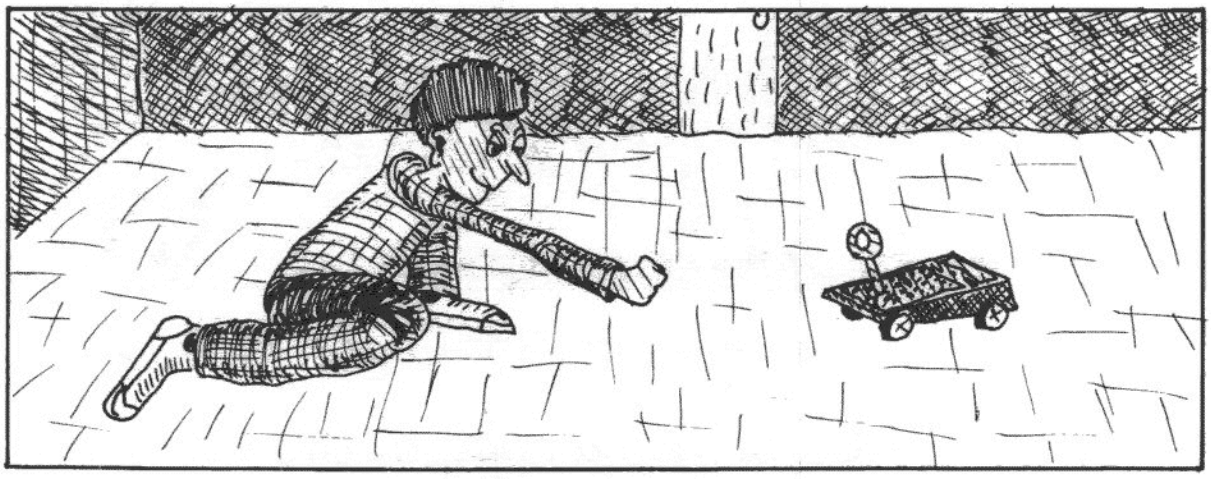
Solution: [RCL] 01

OK?

Move to the next page.

TROUBLES?

[RCL] is the recall key. When you press it, you tell the HP-41 to recall something to the X-register. The computer responds by placing the name of the function (RCL __) in the display. The two blanks by RCL are the HP-41's way of asking you "what register?" So, you key in [0][1].



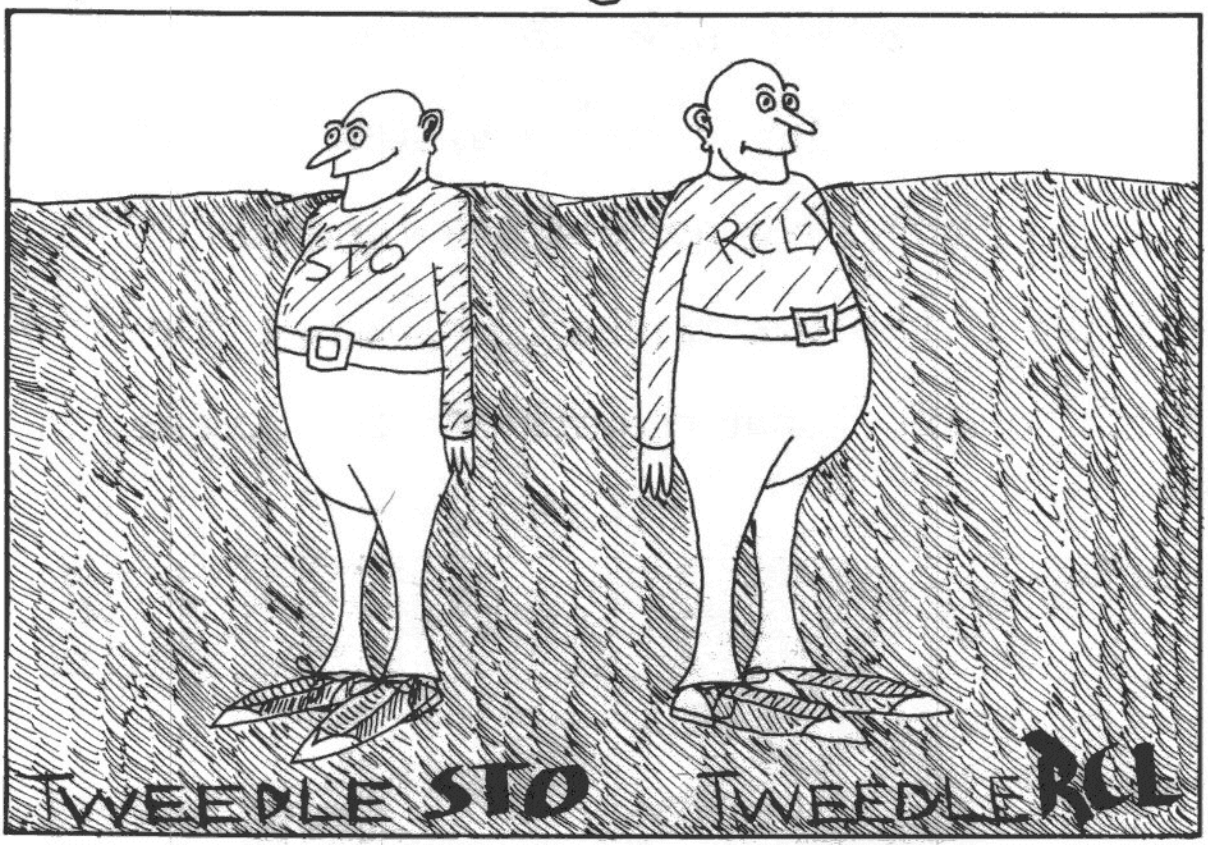
As soon as you have given it two digits, it takes off and does the job.

Challenge: Store the contents of the X-register into data register 02.

Solution: [STO] 02

TOO EASY? →

This is the same idea as RCL – exactly – except that you use the [STO] key (STO means "store").



Notice that, in many ways, STO and RCL are a "matched set." RCL brings a copy of the contents of the specified register to the X-register. STO sends a copy of the contents of the X-register to the specified register.

Challenge: Recall the contents of the Z-register to the X-register.

Solution: RCL ⊡ Z

YAWNING? Move ahead 

NOT SO FAST?

The RCL part is no problem, right? But then you have to tell the machine that the register number it's expecting (two digits) is not going to appear. You're going to give it a stack register instead.

To do this, press the ⊡ key. This changes the display to RCL ST_. Now the computer is expecting a STack register name—a single letter.

Just press the key with the Z on it.

(Don't go into ALPHA mode to get the Z. The computer has already told you with "ST" that it is expecting a letter.)

Challenge: Recall the contents of data register 01 to the X-register, but this time don't use the [RCL] key.

Solution: [XEQ] [ALPHA] RCL [ALPHA] 01

TRIVIAL? Right this way. 

Notice that this procedure uses the exact same format as when the [RCL] key is used.

First, tell the computer the name of the function to execute.

It then acknowledges your request by placing that name in the display. Then it prompts <u>you</u> for the register (i.e., it prompts you for the <u>argument</u> of the function).

PRESSING THE [RCL] KEY IS EXACTLY LIKE PRESSING [XEQ] [ALPHA] RCL [ALPHA].

Challenge: There is a [R↓] (RDN: roll-down) function on a key in the second row. There's also a R↑ (roll-up) function, but it's not on the keyboard. Execute the function R↑.

Solution: [XEQ] [ALPHA] R↑ [ALPHA]

NO PROBLEMS? Move on ⟶

The ↑ character isn't printed in blue on any of the keys, is it? Ah, but turn the HP-41 over.

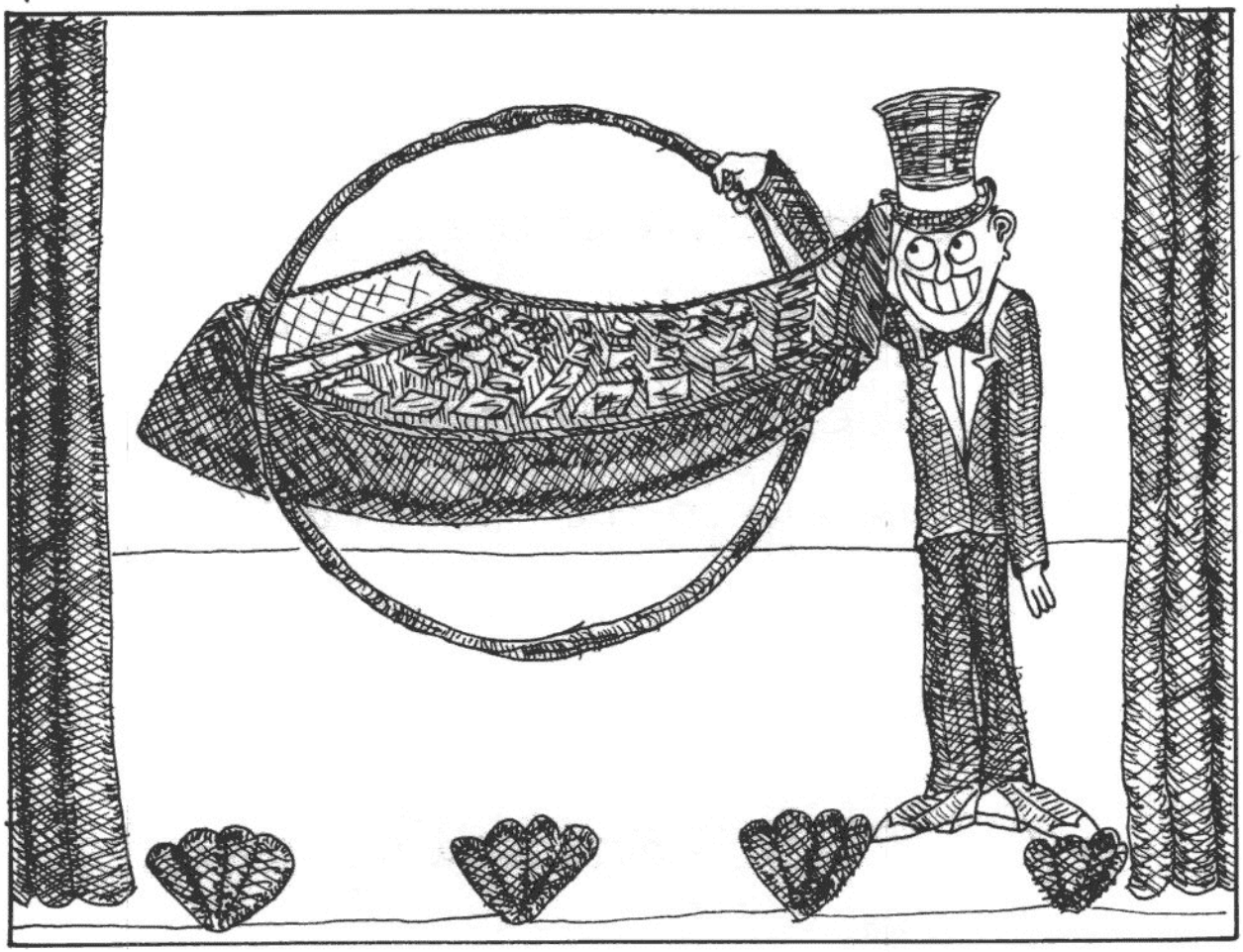
That little gold picture of the keyboard was put there to help you remember what the entire ALPHA keyboard looks like. This is what every key and every <u>shifted</u> key means when the ALPHA annunciator is on.

The ↑ character is accessed by using the shifted [N] key.

Challenge: Execute X<>Y without pressing the [X<>Y] key.

Solution: [XEQ] [ALPHA] X<>Y [ALPHA]

Did you manage to find the < and > characters? Notice that the computer doesn't pause to prompt you for an argument after you have spelled the name. It knows exactly what action to take and what registers to perform it upon.



Challenge: Execute X<>L

Solution: [XEQ] [ALPHA] X<> [ALPHA] [⊡] L

If you know all this, move ahead. 

MURKY WATERS?

This function's name is just X<>, not X<>L (there's no X<>L in CAT 3).

So you must give the computer the function name and let it go through its routine where it puts the name in the display and asks you for the argument. All it knows is that it is supposed to exchange the contents of the X-register with that of some other register. But, just like STO and RCL, it doesn't know which register until you supply the argument ([⊡] L in this case).

The X<> function is totally different from X<>Y. The X<>Y function was provided because it is used so often. (Yes, you can use the X<> function and specify STack Y ([⊡] Y) as the argument. But why bother?)

Challenge: Execute the function called X≤∅? (X less than or equal to zero ?).

Solution: XEQ ALPHA X < = ∅ ? ALPHA

SMOOTH GOING?

Go to page 45. ➡➡

HEAVY SEAS?

Chances are that if you had problems with this one, they were:

1. The ≤ notation (how to key it in ?).
2. How to key in a ∅ character.
3. Forgetting the necessary "?".

1. Even if you checked the handy-dandy ALPHA keyboard on the back of your computer, you couldn't find a character that looks like ≤, could you? BUT, if you checked CAT 3 you found X<=∅? (in its proper alphabetical order).

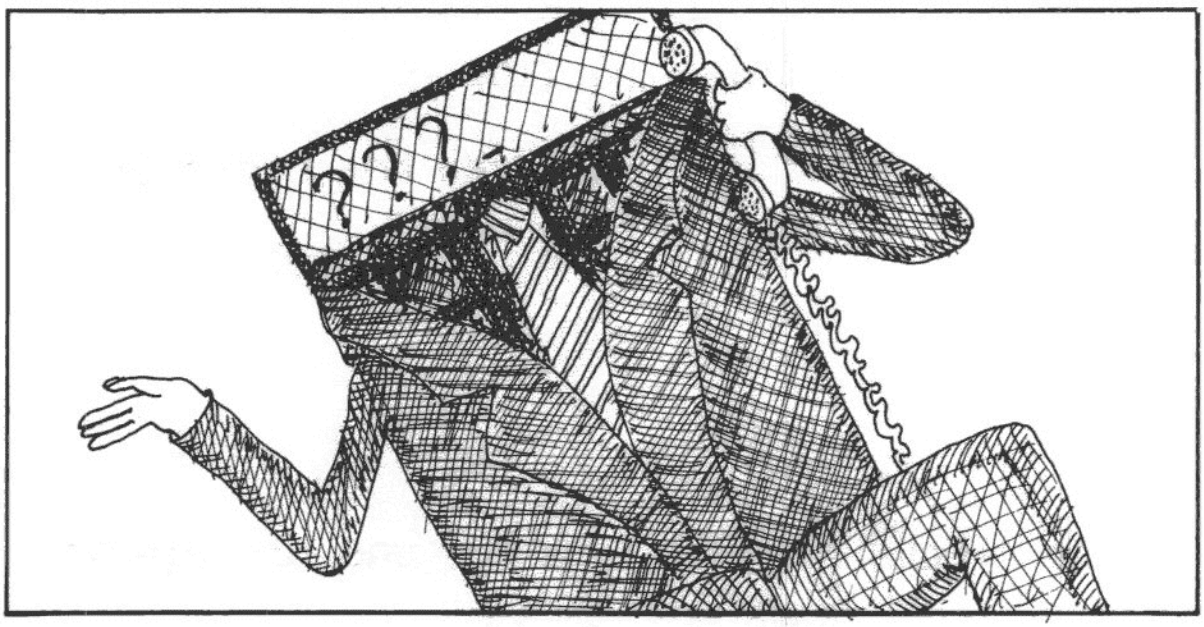
Notice that you also find X≤∅?, and this does look similar in your display to X≤∅?, but X≤∅? means X≠∅? (X not equal to zero?). The ≠ (not

equal to) sign <u>is</u> on the keyboard.

2. Any of the numerals 0 to 9, the decimal point, and the symbols +, −, *, and / can be obtained by pressing (in ALPHA mode, of course) the shift key (gold key) and then the corresponding key. The ALPHA keyboard on the back shows this clearly.

3. If a question mark were not a part of the function name, then the ? would not appear in the CAT 3 entry for that function.

Don't forget – the HP-41 will not recognize any function name that doesn't <u>exactly</u> match the function's name as it appears in CAT 3.



Challenge: Execute the function FS?C 22.

Solution: [XEQ] [ALPHA] FS?C [ALPHA] 22

NO SWEAT? Next page →

SWEAT? (Even mild dampness?)

You probably got into trouble with:

1. The "?". Don't neglect it.
2. The argument, 22: When to key it in?

REMEMBER! The name of the function is what you tell the HP-41. Then it asks you for the argument.

The name of this function is FS?C, as CAT 3 will show. So that is all you tell the computer at first.

Then it will prompt you for the argument (22 in this case) with two cursors.

Challenge: Compute the sine of π radians. (If you've never had a trigonometry course, feel free to skip ahead to page 49.)

Solution: [XEQ] [ALPHA] RAD [ALPHA] [π] [SIN]

NO PROBLEMS?

Move to page 48. →→

The HP-41 makes assumptions about the numbers it's asked to work on. In this case, the assumption is about the angle it works on with the SIN function.

If you see the little RAD (radians annunciator) in the display, that means the number in the X-register is in radians as far as the HP-41 is concerned. If you see GRAD, likewise, the computer assumes the number is in grads. If you don't see any annunciator there, then the computer is assuming degrees.

You can change what your computer assumes by executing the appropriate function—RAD, GRAD, or DEG. For this problem, you execute RAD. Then you bring π into the X-register with the PI function ([π] key). Now, if you press [SIN] you should get SIN(π), which is zero.

BUT WAIT! You didn't get zero!?!

What's wrong? SIN(π) is zero.

The problem here is that you didn't take the sine of π (Say what?). You took the sine of 3.141592654. That is not π. That's almost π. That's the first 10 digits of π.

But π has an infinite number of digits (according to the latest information). There is no way that any computer can ever take the sine of exactly π, because a computer can only work with so many digits.

The point of all this is not to quibble about π, but to remind you that the HP-41 keeps 10 digits (which is usually more than enough) of any number.

So, no matter how exact you know an answer should be (mathematically speaking), the computer uses 10 digits of each number involved, and therefore, the last digit of the final answer may vary from your expectations. This limitation is characteristic of all computers, but you will seldom need to consider this at all.

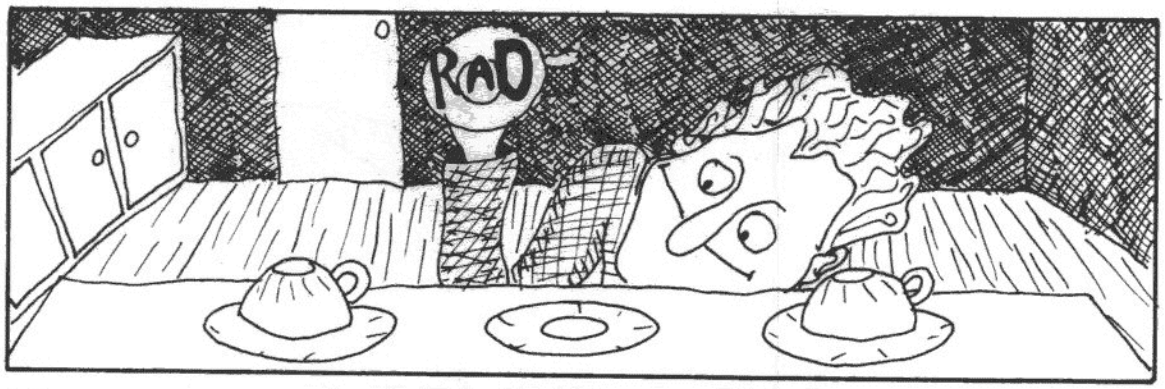
Challenge: Take the inverse cosine (or "arc-cosine") of the number you got as a result for SIN(π). Then tell the computer to return its assumption to DEGrees.

Solution: [shift] [COS⁻¹] (or [XEQ] [ALPHA] ACOS [ALPHA]), then [XEQ] [ALPHA] DEG [ALPHA]

Satisfied? Next page. →

Notice that the name of this function (as it appears in CAT 3) is different than the symbol used for it on the key.

Notice the answer you got: 1.570796327. This is "almost ½ π," which is correct, because the cosine of "almost ½ π" radians is "almost zero," and you had "almost zero" to begin with.



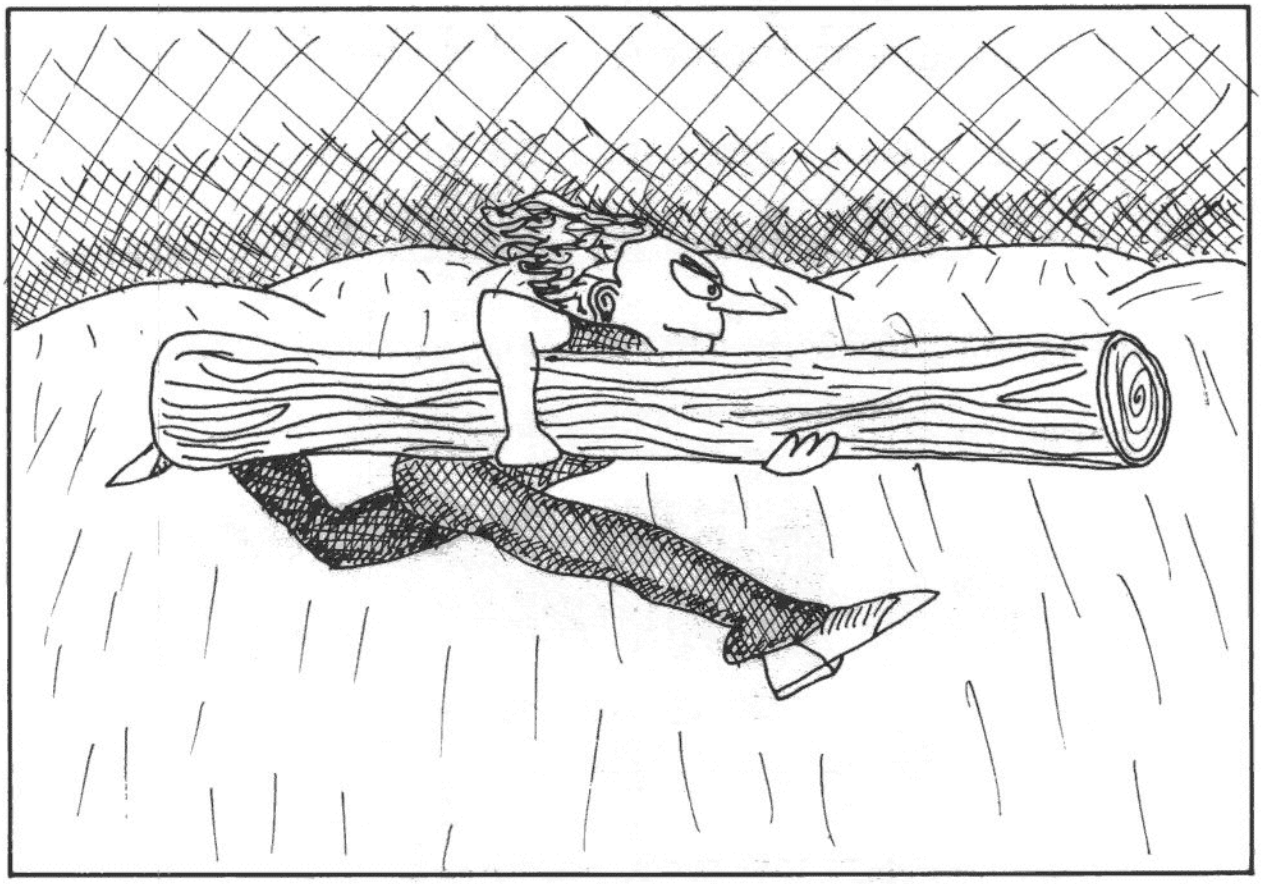
Notice how the radians annunciator disappeared when you executed DEG. The calculation was done while the HP-41 was still assuming radians, but now it is assuming degrees.

Challenge: Put "TAKE 5" into the ALPHA-register.

Solution: [ALPHA] TAKE [SPACE] 5 [ALPHA]

Just checking to see if you remember how to put ALPHA data into the ALPHA-register (as opposed to spelling a function name in the display).

If you're hazy at all, go back and review page 20. (mark your place here).



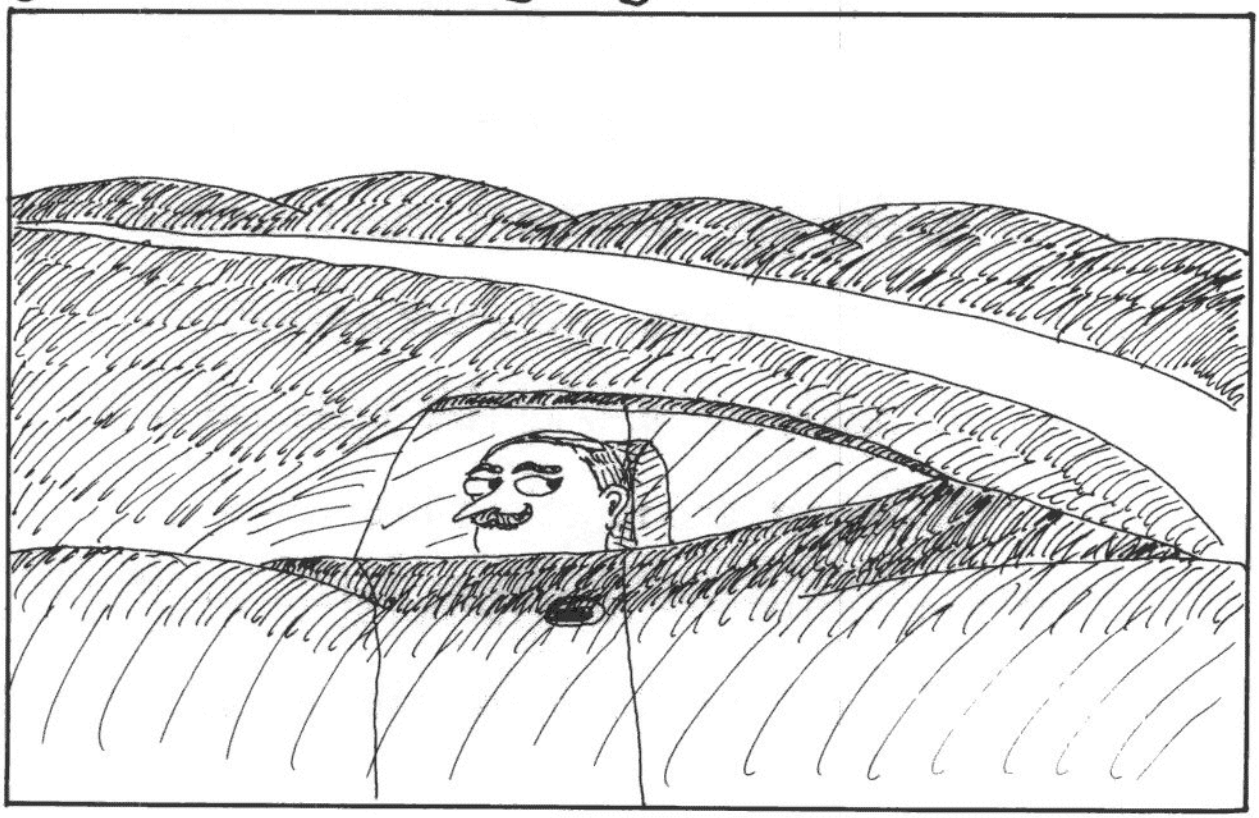
THEN BLAST ON AHEAD

Challenge: Look at the contents of data register 02 without using the RCL function at all.

Solution: [Shift] [VIEW] 02

CLEAR? Step this way. ⟶

MUD? Well, VIEW is a "clever" function. It puts a copy of the contents of the indicated register into the display only.



In other words, VIEW puts those contents "in front" of the display "window," so you can no longer see "through" the "window" into the X-register.

Challenge: Put the contents of data register 03 into the ALPHA-register.

Solution: [ALPHA] [Shift] [ARCL] 03 [ALPHA]

IF THIS IS "OLD HAT," move on →

"NEW HAT?"

It happens that when you are in ALPHA mode (that is - when the computer is in ALPHA mode), not all of the keys change their meanings to ALPHA characters. Some of them change to different functions, and some of them don't change at all. The handy-dandy ALPHA keyboard on the back of the HP-41 shows these functions in white.

The function you just executed is "ALPHA-recall," ARCL. Now, put the HP-41 back into ALPHA mode to see what happened.

Notice that the contents of register 03 were added to the characters that were already in the ALPHA-register (namely, TAKE 5).

Challenge: Recall the contents of the X-register to the ALPHA-register.

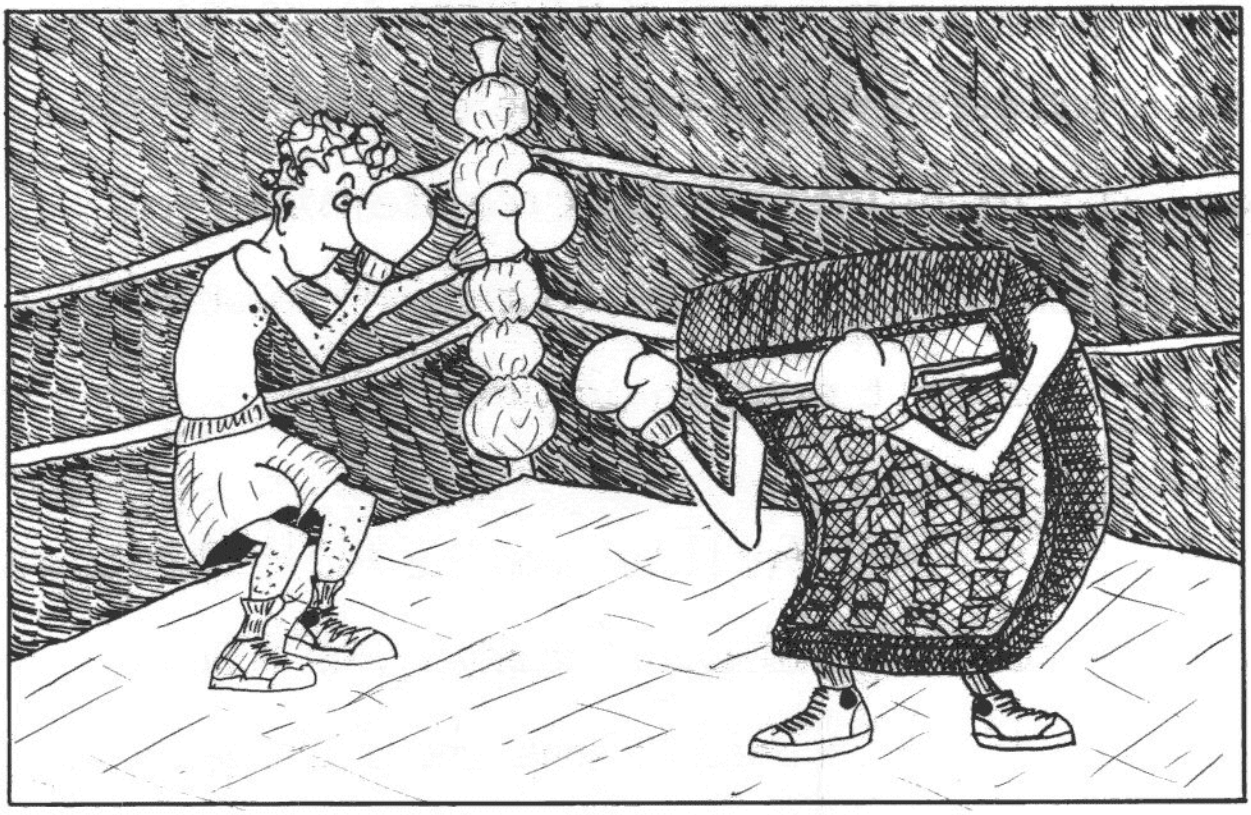
Solution: [ALPHA] [Shift] [ARCL] [.] X [ALPHA]

OK? ⟶

NO?

Remember, to specify a stack register, use the decimal point [.].

ARCL needs an argument, just like many other functions. You treat it the same way whenever you see the computer prompting you for an argument.



Challenge: Arrange things so that you are looking at the contents of the ALPHA-register... without being in ALPHA mode.

Solution: [XEQ] [ALPHA] AVIEW [ALPHA]

QUESTIONS? No?⟶

YES!

Just like the VIEW function, AVIEW fills the display so that the register behind the "window" is no longer visible. The only difference between VIEW and AVIEW is that AVIEW uses the contents of the ALPHA-register (VIEW uses the contents of data registers – including the stack registers).

Because there is no doubt about which register AVIEW is using, it doesn't need an argument.

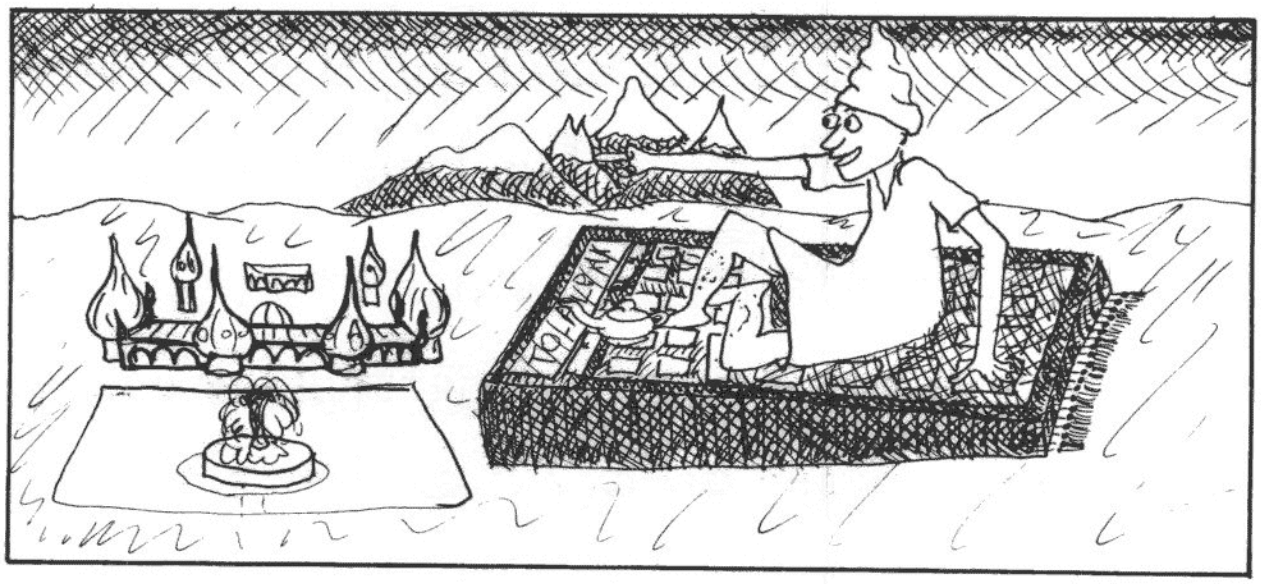
Also, notice that AVIEW is on the ALPHA keyboard. You can execute it that way. (However, the keyboard function could not be used to solve the challenge as we posed it. Try it and you'll see why.)

Challenge: Add, that is, <u>append</u> the characters "ABC" to those already in the ALPHA-register.

Solution: [ALPHA] [↗] [APPEND] ABC [ALPHA]

APPEND is a key on the ALPHA keyboard (because it pertains to the ALPHA-register). When you press the APPEND key, the HP-41 is instructed to "pretend" that you are suddenly in the middle of an ALPHA data entry — as if you had just entered the current contents of the ALPHA-register.

Then, you can add characters simply by spelling them out, or you can delete characters, one-by-one, using the [←] key.



Challenge: Key the number 1,000,000 into the X-register without using the [1] or [0] (zero) keys.

Solution: [EEX] [6]

[EEX] means "enter exponent," and it is used to express numbers in powers of 10.

So, $10^6 = 1 \times 10^6 = 1{,}000{,}000$ (note the six zeros), and this represents the amount of money, in dollars, that we plan on making by writing this book.

For another example, $1.35 \times 10^{-3} = 0.00135$, and you would key it in as [1] [.] [3] [5] [EEX] [3] [CHS]

Notice that when you press [EEX] before any number key (as in the above solution), the computer puts a one (1) in for you. The keystrokes [1] [EEX] [6] are the same as just [EEX] [6].

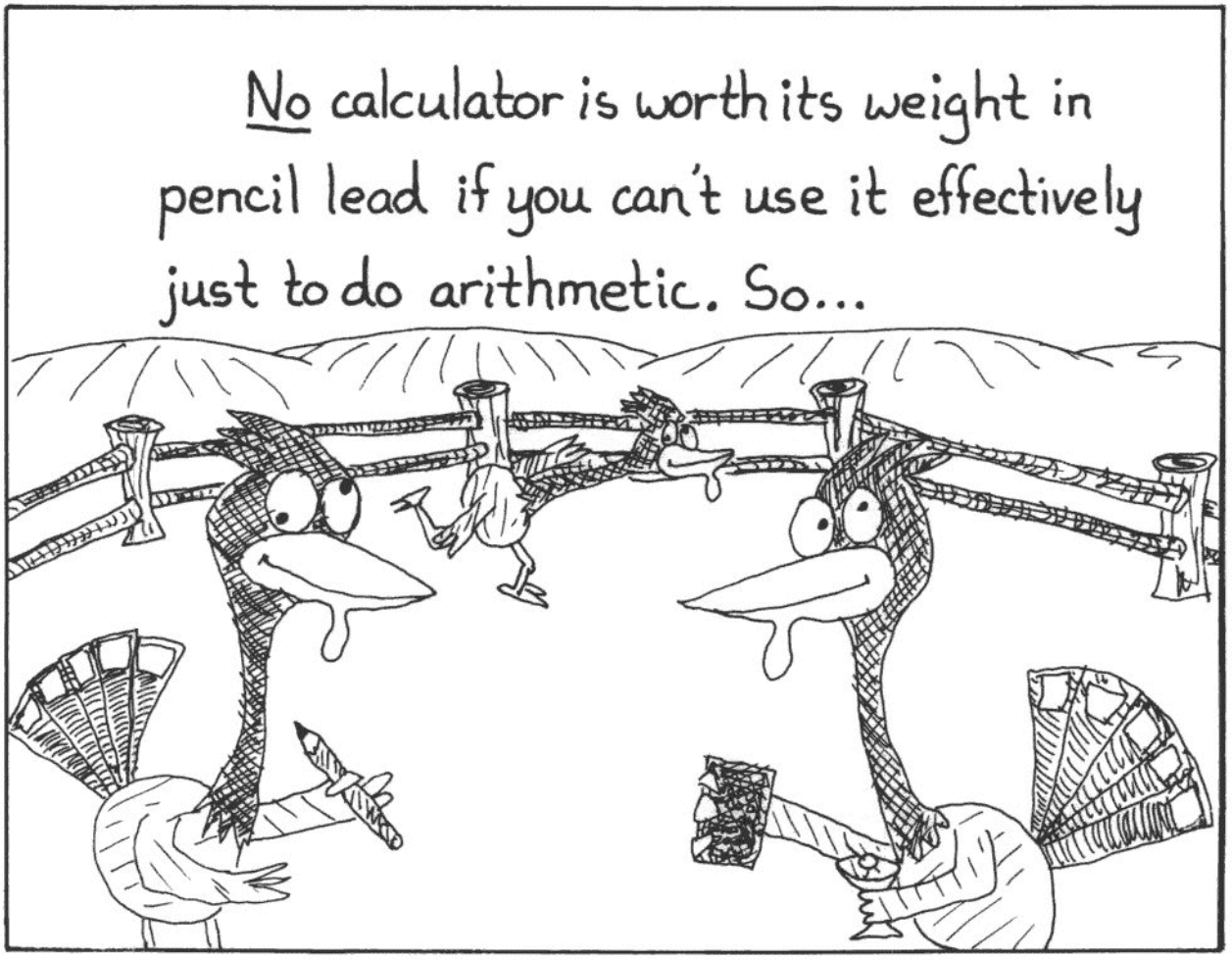
One more thing: You may sometimes see a number like 8.2×10^8 written as 8.2 E8. (The E stands for "exponent.")

Notes

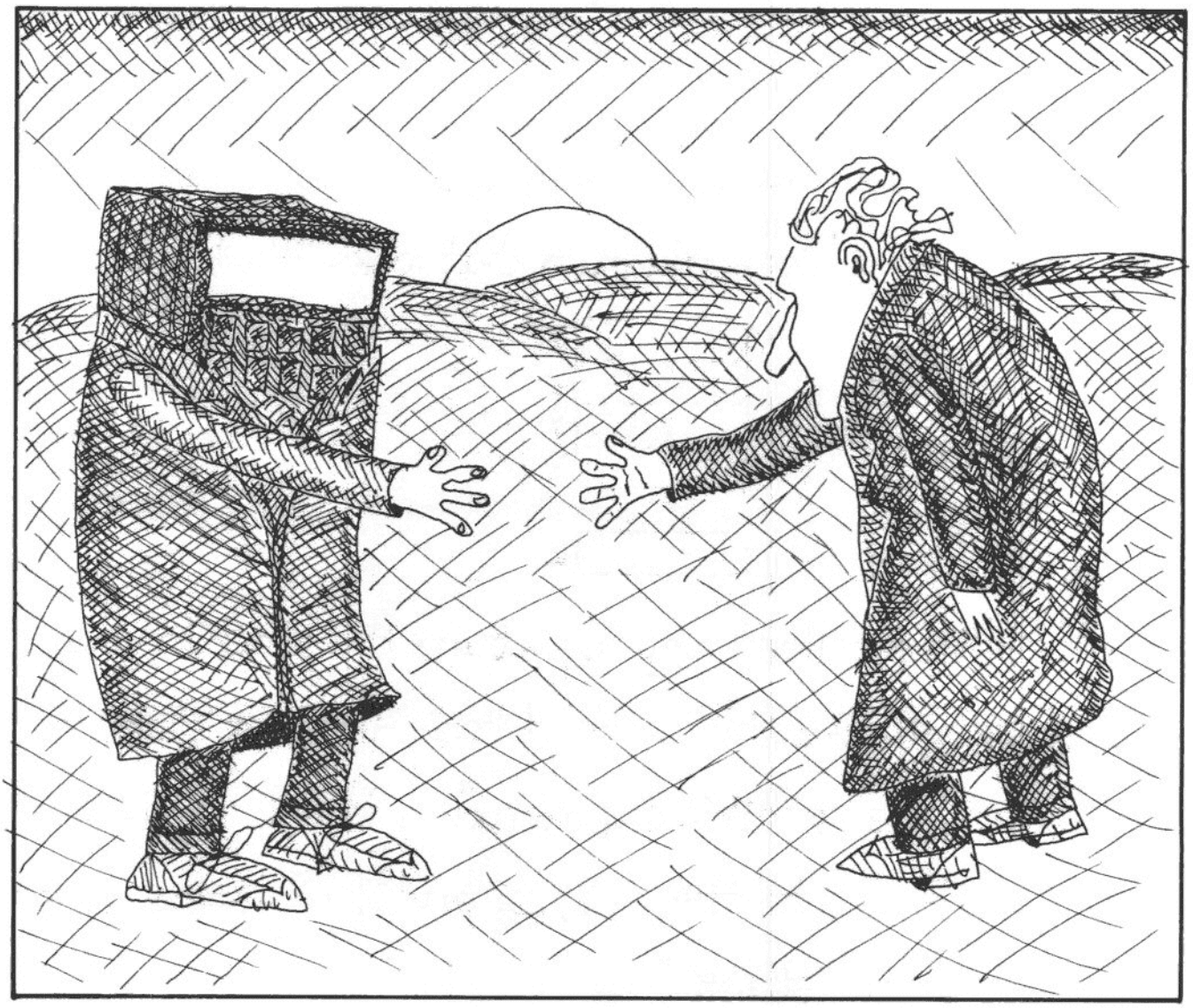
1. The only way to tell the HP-41 that you are specifying a function name - rather than simply keying in a bunch of characters to the ALPHA-register - is to use the [XEQ] key.

So, you've seen some keystroke combinations, and the keys are already getting some individual personalities.

But let's talk turkey, here.

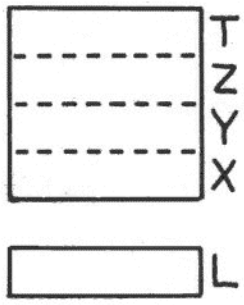


YOU'VE GOT TO KNOW YOUR STACK



The stack is a special set of five data registers that is found on most HP calculators. This stack is what makes HP calculators so much easier to use than the other leading brands. The main reason HP calculators are easier to use is that when you're doing lengthy calculations, your intermediate results get saved for you automatically; thus you aren't forced to use parentheses to grind through a big, fat, hairy equation.

So, how does it work? Well, the Owner's Handbook often refers to a block diagram that looks like this:



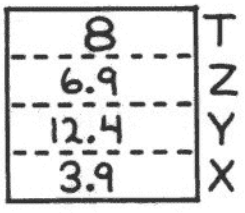
This is an excellent way to picture the stack in your mind, so we will use this method, too. Also, let's agree that, unless we specifically mention the L-register, we will be referring to the X-, Y-, Z-, and T-registers when we use the word "stack."

Now, as you remember (from page 19), WHENEVER you key in a number, you are keying it into the X-register.

ALL NUMBERS ARE KEYED INTO THE X-REGISTER

Once you've put a number into the X-register, THEN you can store it or add it, etc.

Challenge: Set up the X-, Y-, Z-, and T-registers as follows.



Solution: 8 ENTER 6.9 ENTER 12.4 ENTER 3.9

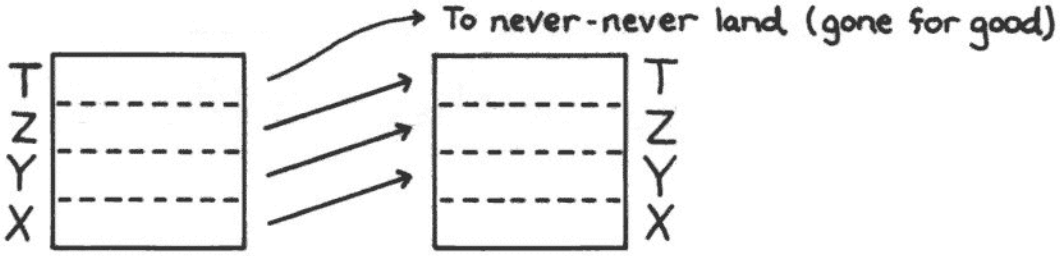
Now, if you thoroughly understand this, and you truly want to skip a fine discussion of ENTER and CLX, then go ahead to page 67.

OTHERWISE →

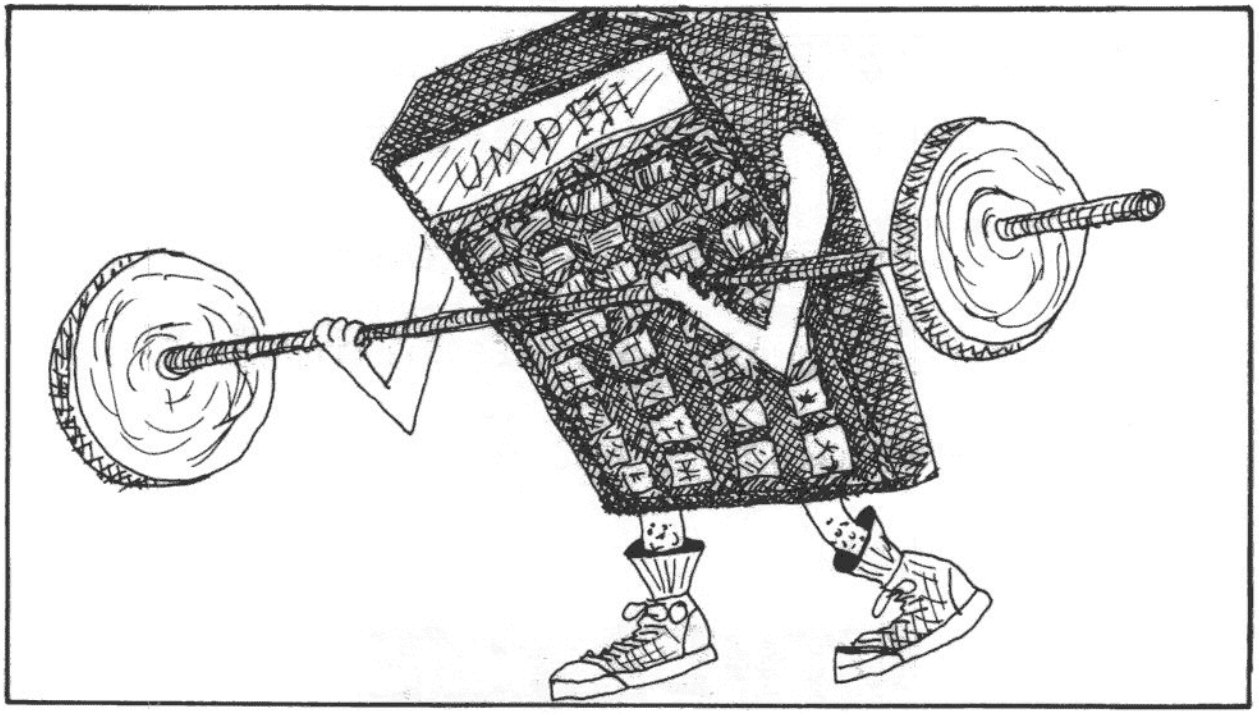
So, what is the famous ENTER key? Well, before we get to that, there is a certain phrase we'll need to use:

STACK-LIFT

Stack-lift is the process by which each of the values in the stack get lifted one notch.



Notice that the original value in the T-register is gone for good after a stack-lift.



"But when does this stack-lift happen? How do I know whether it's going to happen when I key in a number?"

Well, if the stack-lift does occur when you key in a number, that means it was ready-and-able to do so. We say stack-lift was "enabled."

But, if the stack doesn't lift when you key in a number, then we say stack-lift was "disabled."

So, the question you're really asking is: "How do I know when stack-lift is enabled and when it is disabled (i.e., when the stack is ready to lift and when it's not)?"

You should use this rule: There are only two things you will commonly do to the computer to DISable stack-lift. Those are:

press [ENTER]

or press [CLX].

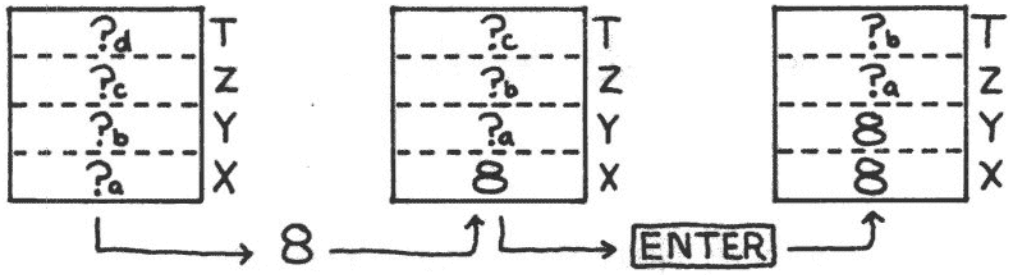
Now we're ready to discuss [ENTER]. →

"So, the ENTER function leaves stack-lift DISabled, right? What else does it do?"

It does two things (in this order):

1. First, it performs a stack-lift.
2. Then, it disables stack-lift.

Look at the first two steps in our solution.

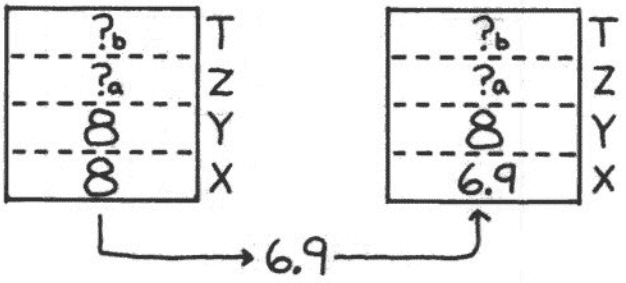


(? means: "We don't know what's in there, and we don't care")

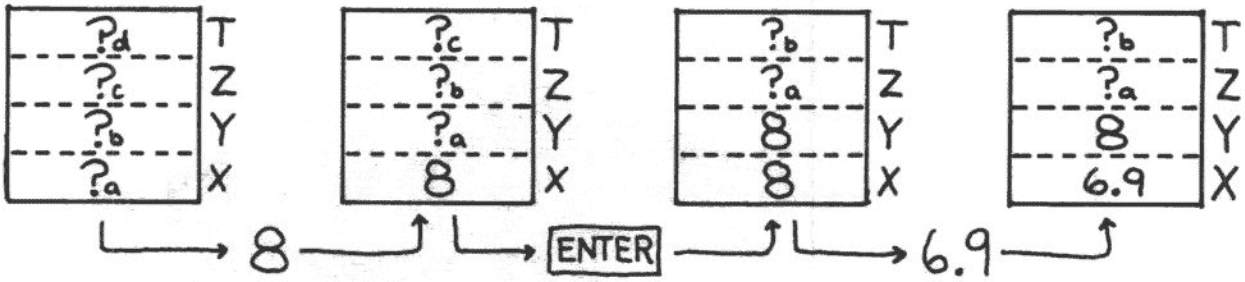
The first step is to key-in the number 8. No mystery, right? It goes into the X-register.

Now, when we press ENTER, a stack-lift is performed (regardless of whether stack-lift was enabled or disabled). So our stack is "lifted." That is, a copy of the 8 is sent to the Y-register, and the other values are bumped up one notch.

Now, ENTER also disables stack-lift, so that if the next step is one where a number is recalled or keyed in to the X-register, the stack won't lift. The value formerly in the X-register just gets replaced by the new value, and the other registers aren't touched.

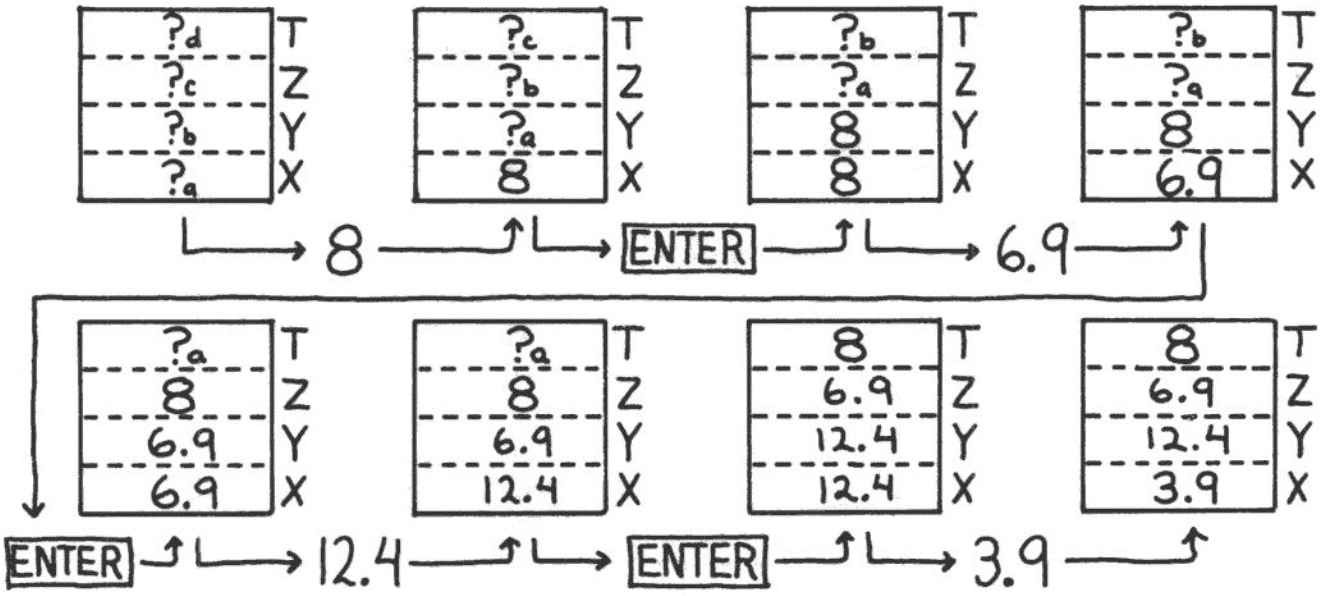


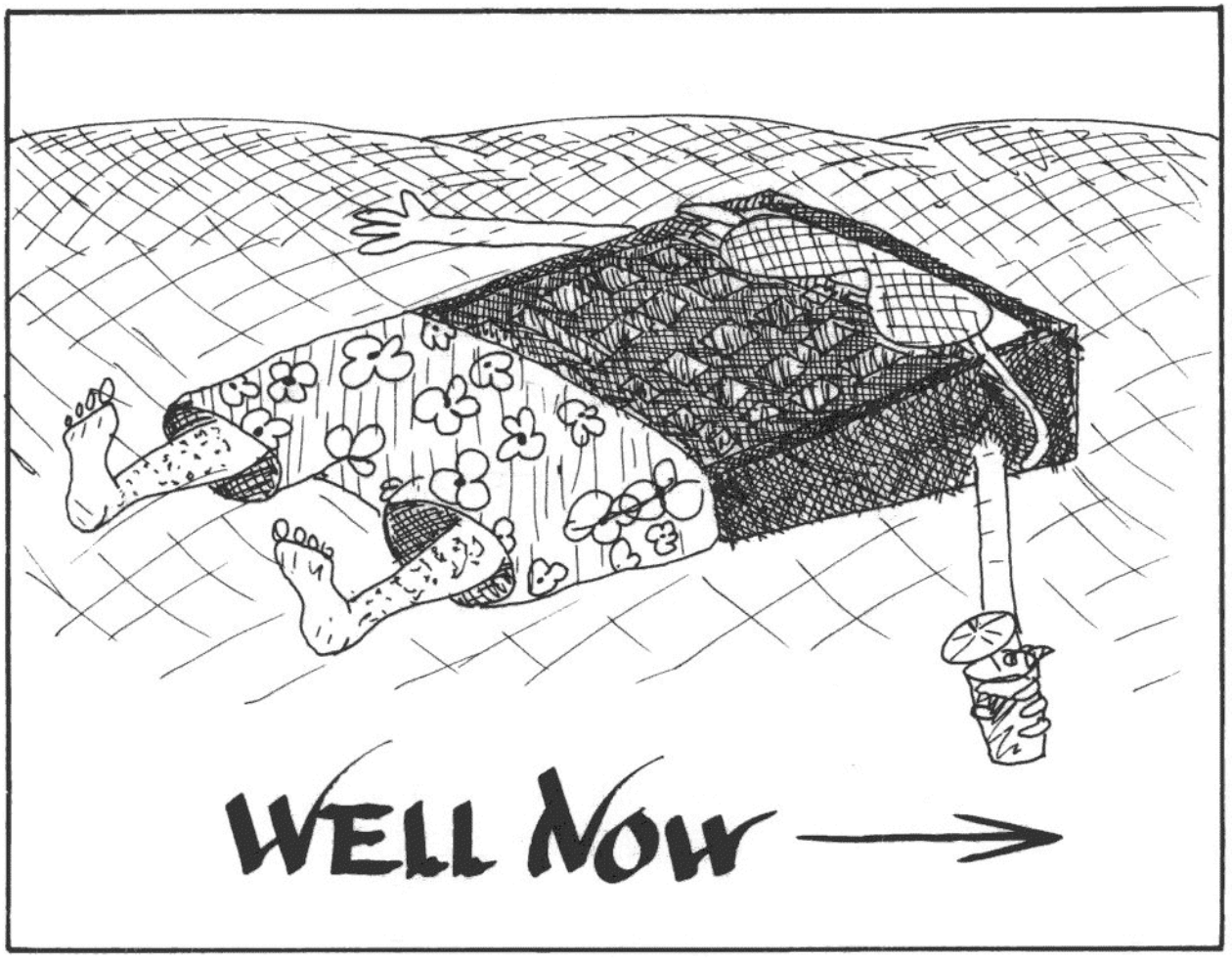
Watch once again:



Think about it this way: ENTER makes a copy of the X-register in the Y-register and bumps everything else up one notch (lifts the stack). But that value in the X-register is a sitting duck if the next step performed is a recall (RCL) or the keying-in of another number.

Here is a complete diagram of the solution to the last challenge. Study it until you're comfortable with the ENTER function.





While we're on the subject of disabling stack-lift, let's mention [CLX].

CLX also does two things.

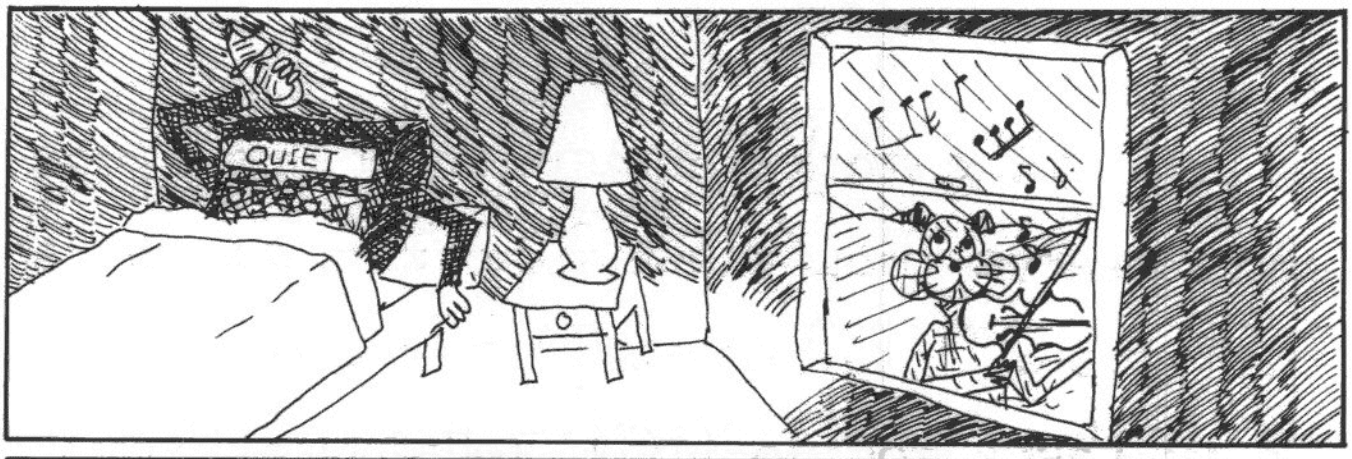
1. It replaces the number in the X-register with a zero – without disturbing anything else.

2. It disables stack-lift.

So, both ENTER and CLX <u>leave</u> stack-lift disabled. But they both do very different things before that:

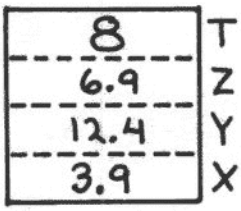
ENTER disturbs the whole stack.

CLX disturbs only the X-register.



ENTER and CLX are the only two commonly used functions that leave stack-lift disabled.

Your stack should look like this:



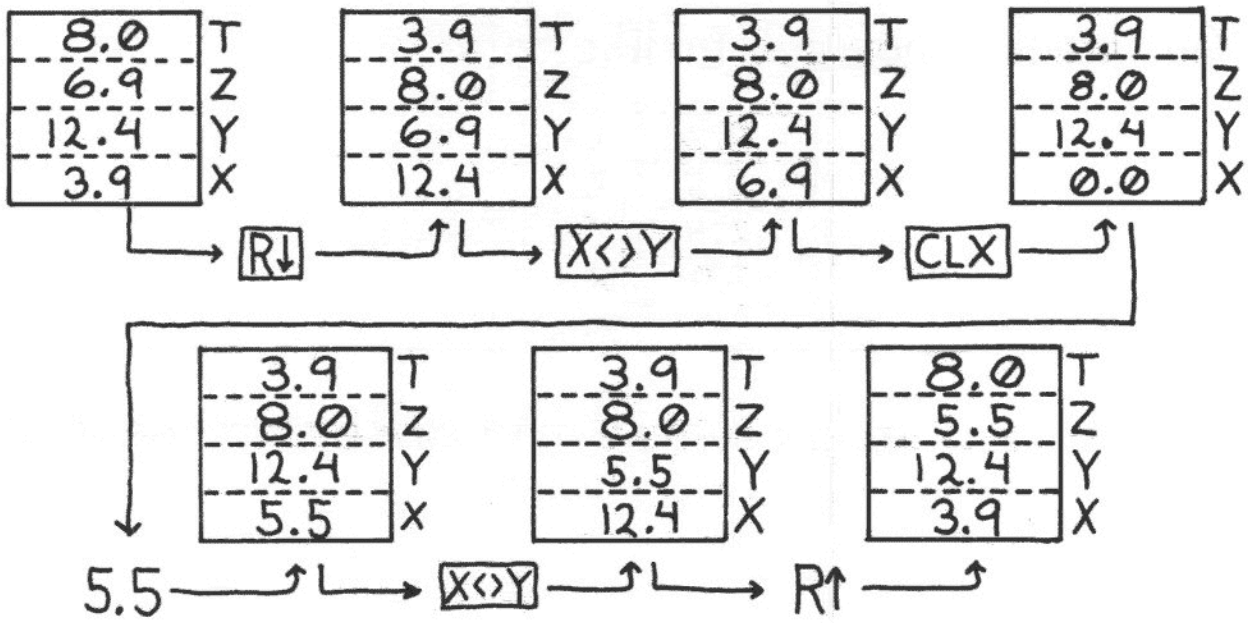
Challenge: Replace the 6.9 in the Z-register with a 5.5.

Solution: Here's one of many solutions: [R↓] [X<>Y] [CLX] 5.5 [X<>Y] [XEQ] [ALPHA] R↑ [ALPHA]

PIECE OF CAKE? Try a bigger piece (page 69).

So here are more useful functions for manipulating the stack, right?

Look at the stack diagrams on the next page and observe how these functions accomplish their task. It will be easy for you to see why we call them "roll-up"(R↑), "roll-down"(R↓) , and "X exchange Y" (X<>Y).



Remember, R↑ is not on the keyboard, so you use the [XEQ] [ALPHA] ... procedure.

Now, what would happen if you pressed [STO] 02 after [CLX]? What would the stack look like after you keyed in 5.5?

Well, pressing [STO] 02 would store 0.0 in register 02 and leave stack-lift <u>enabled</u>. After keying in 5.5, the stack would look like this:

8.0	T
12.4	Z
0.0	Y
5.5	X

Since [STO] 02 leaves stack-lift <u>enabled</u>, the stack would lift when you keyed in 5.5. Every common function except ENTER and CLX leaves stack-lift <u>enabled</u>.

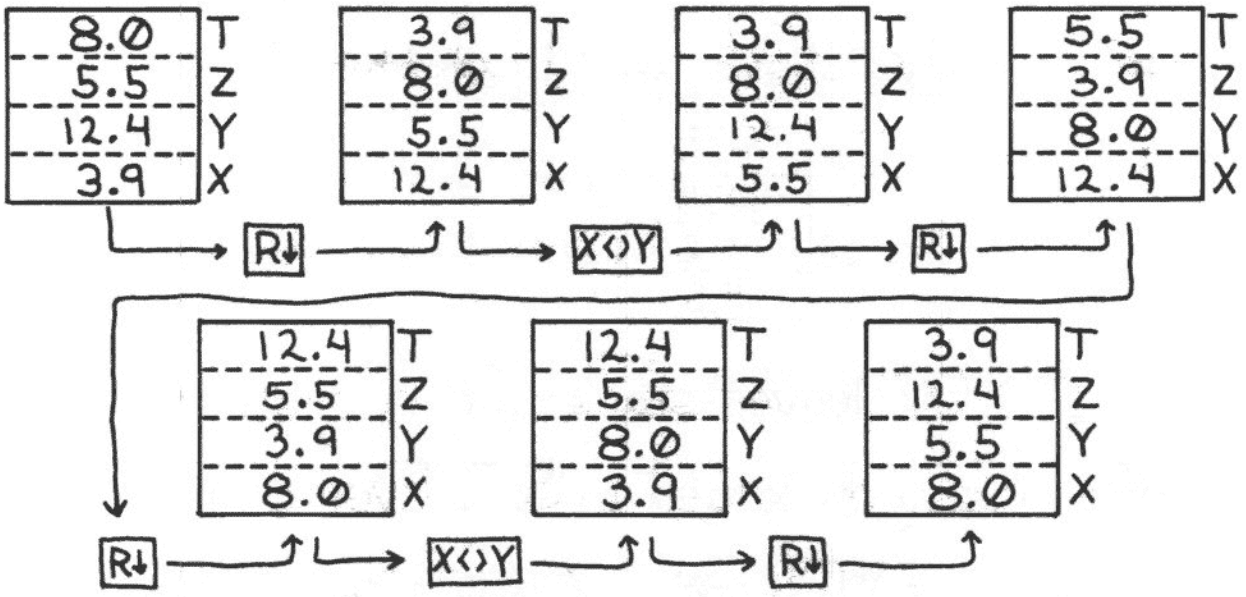
Your stack is now set up like this:

8.0	T
5.5	Z
12.4	Y
3.9	X

Challenge: Let's reverse the order of the values in the stack.

Solution: There are many ways to accomplish this, the most straightforward of which is probably: R↓ X<>Y R↓ R↓ X<>Y R↓.

If this settles well with you, move on ahead. Otherwise, follow the step-by-step solution below; then take a break. When you return, re-read from a couple pages back.



Your stack is set-up like this:

3.9	T
12.4	Z
5.5	Y
8.0	X

Question: What is the value in the Y-register after you press ⊞? The T-register?

Answer: After you press ⊞, the stack looks like this:

3.9	T
3.9	Z
12.4	Y
13.5	X

OK? GO TO PAGE 72

The functions +, −, *, ÷, and y^x operate on the X-and Y-registers. That is, they take the value in the X-register and the value in the Y-register and combine them as specified by the function.

The result stays in the X-register and the stack "drops." The value in the Z-register drops to the Y-register, and the value in the T-register drops to the Z-register.

Note that the value in the T-register stays the same.

The y^x function is named so that it tells you what it's doing. From the name, you can see that it raises the value in the Y-register to the power of the value in the X-register. If 2 is the value in the Y-register, and 3 is in the X-register, then pressing [y^x] will return an 8 (which is 2^3) to the X-register.

It would be nice if [−] and [÷] were named in a similar manner, because in these functions, the order of y and x is important. The [−] function could be named Y−X, and the divide function could be named Y/X, because these names would be more descriptive. But that isn't how they're named, so just remember:

[−] means Y−X (the number in the Y-register minus the number in the X-register);

[÷] means Y/X.

Your stack is set-up like this:

3.9	T
3.9	Z
12.4	Y
13.5	X

and you've just come from the last challenge (a couple pages back).

Challenge: Without touching any of the number keys, get the 8.0 (that used to be in the X-register) back into the X-register.

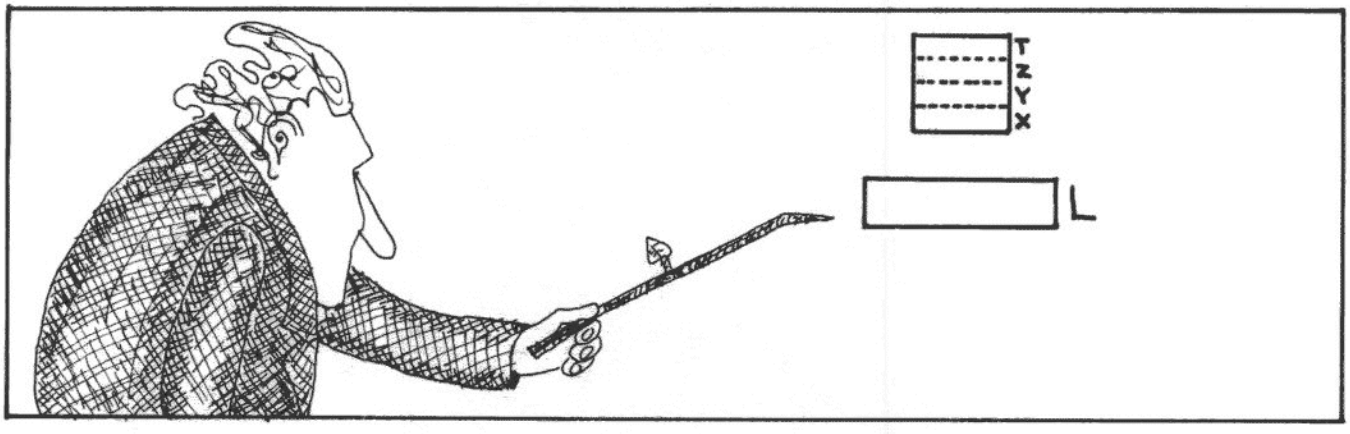
Solution: [Shift] [LASTX]

Are you comfortable with the L-register?

Go to page 74.

Are you thinking, "What-in-L is the L-register?"

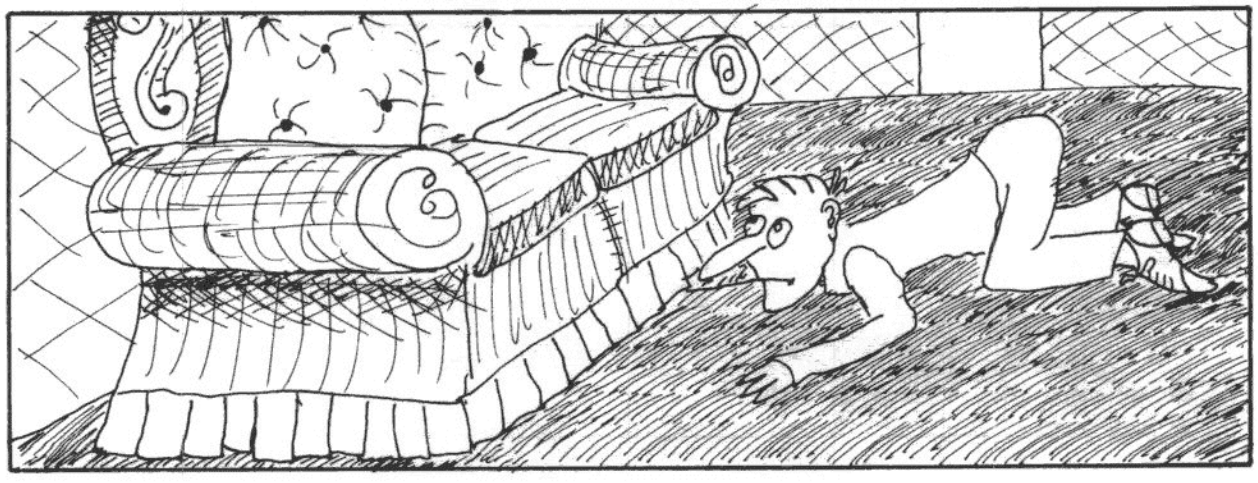
Well, allow us to introduce you to this unique member of the stack.



The L-register is where the last X-value is automatically saved after the execution of many functions that operate on (or change) X. So, if you wish to recover the last X-value, just press [shift] [LASTX].

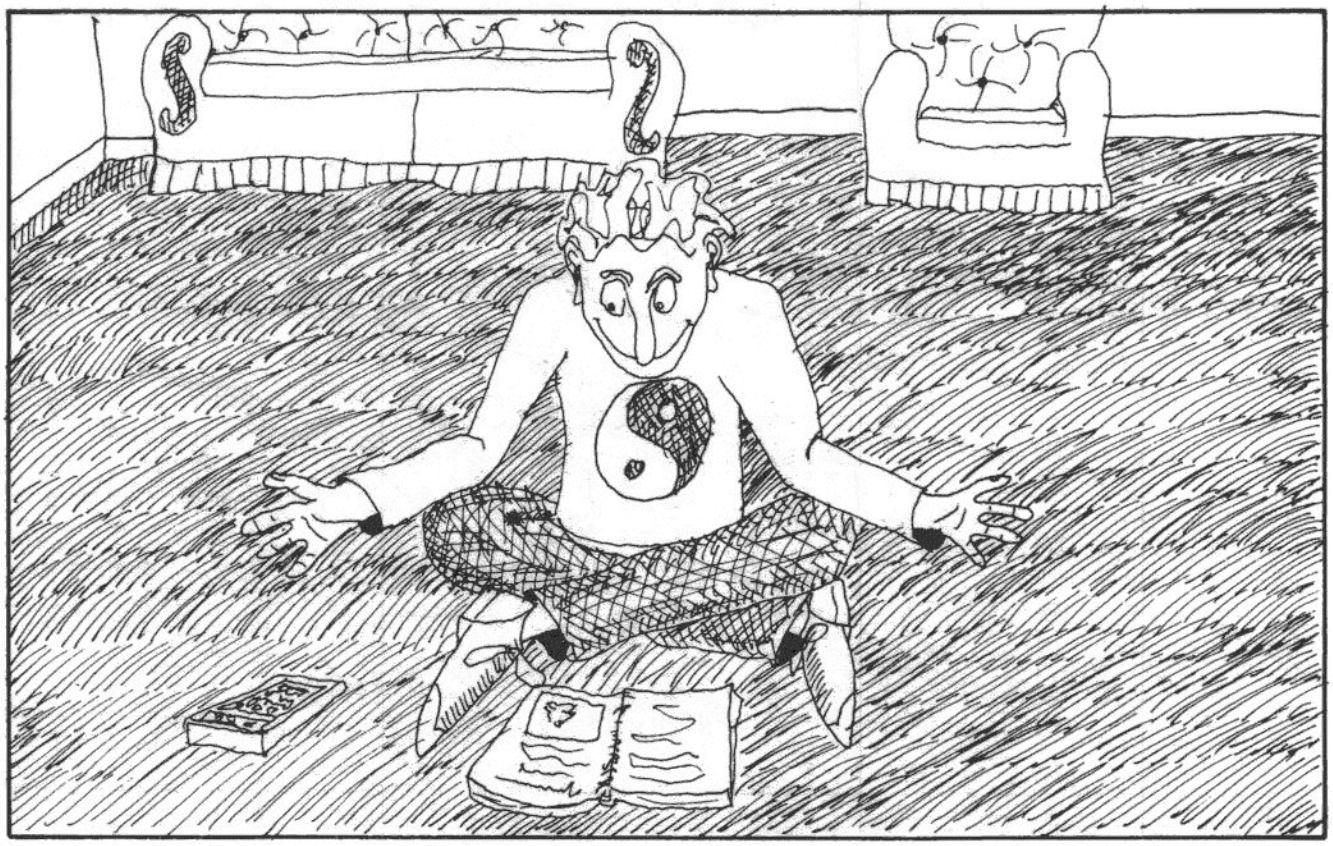
Most functions that change numbers (like +, $1/x$, COS, y^x, ...) save x in the L-register. But functions that move numbers around (like STO, RCL, X<>Y, R↑, ...) do <u>not</u> save x in the L-register.

In the Owner's Handbook, around page 250, there's a list of the functions which tells whether they save x (the number in the X-register) in the L-register. If you're wondering about a particular function, that's the place to look.



Now we are going to supply you with a few more challenges to help you develop a "feel" for the stack. It will take a little time before all the motions become automatic. But now you know when to use ENTER and when ENTER isn't necessary. Plus, you know that many intermediate answers are automatically saved in the stack.

Take a few deep breaths, and relax before you proceed. (A lotus position, somewhere close to the center of your livingroom, may be appropriate while working the next few pages.)



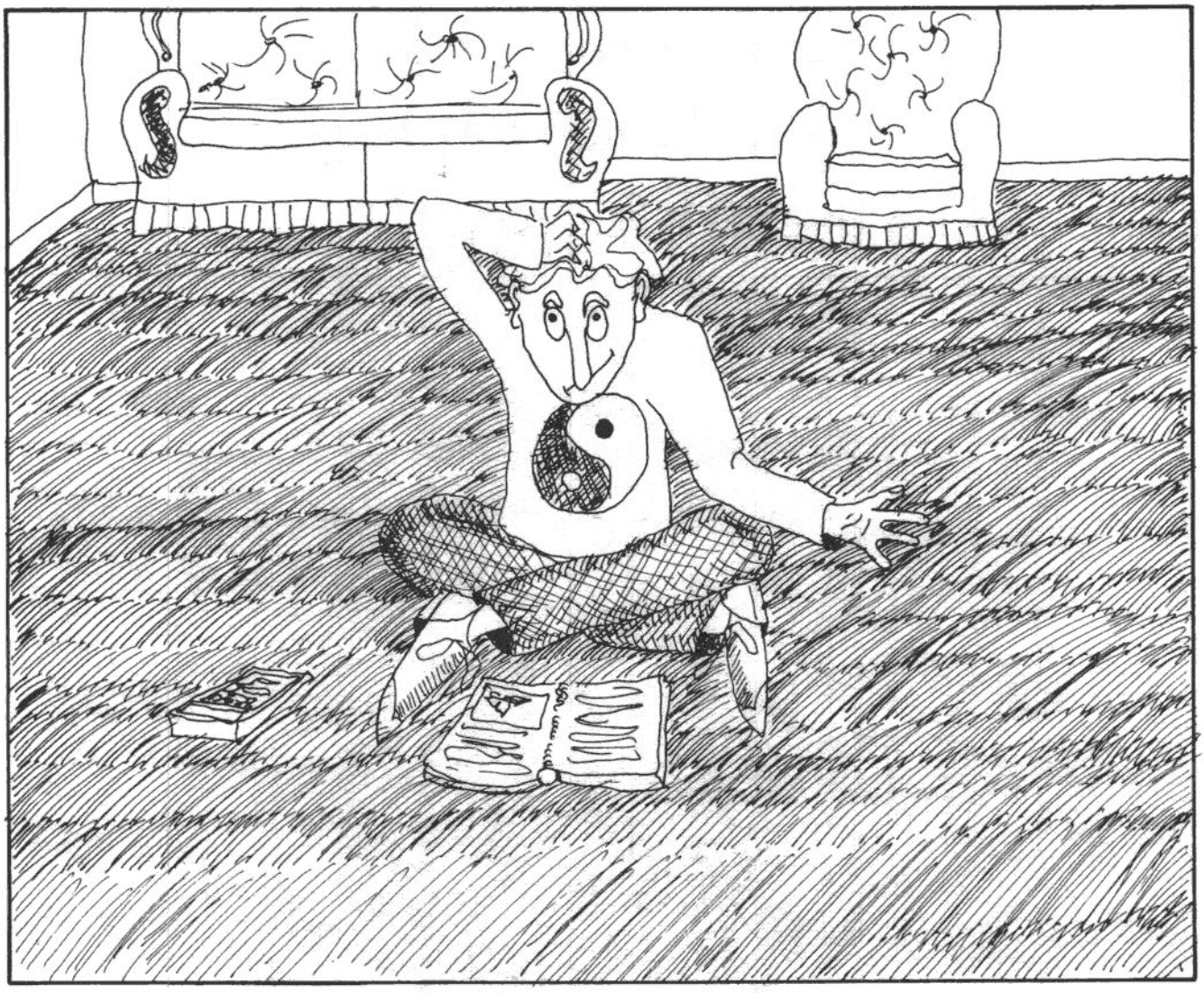
Challenge: Using only the stack (no paper or numbered data registers), calculate the answer:

$$4 + \frac{\sqrt{3+(5*7)} - 72}{9*(13^2-5)}$$

(Looks tough, doesn't it?)

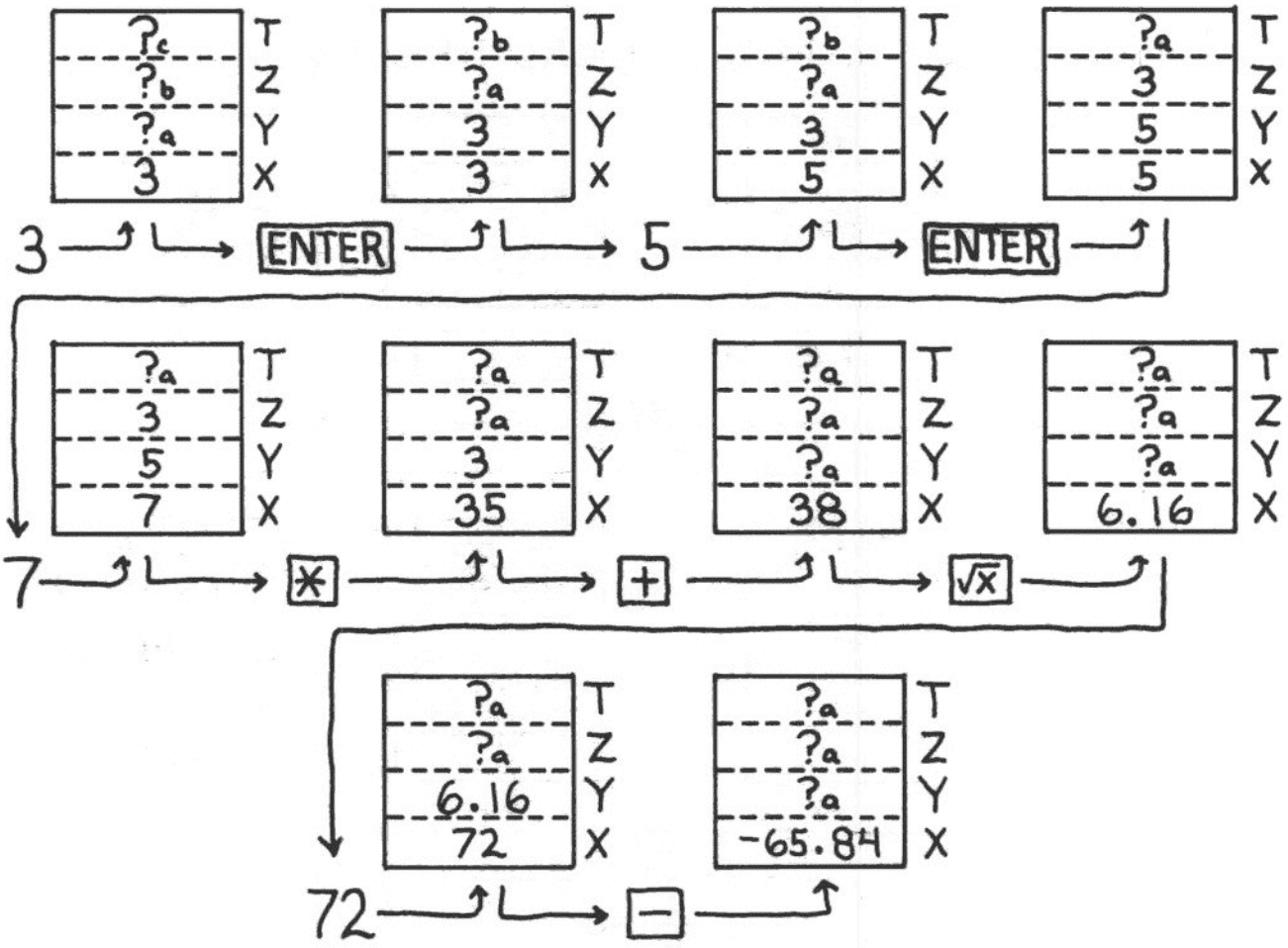
Solution: 3.955395944

If you got that answer the first time and you don't want to look at the step-by-step solution to this problem, then skip ahead to page 79.



Challenge: Evaluate $\sqrt{3+(5*7)} - 72$

Solution: 3 [ENTER] 5 [ENTER] 7 [*] [+] [√x] 72 [−], and the stack looks like this, step-by-step.



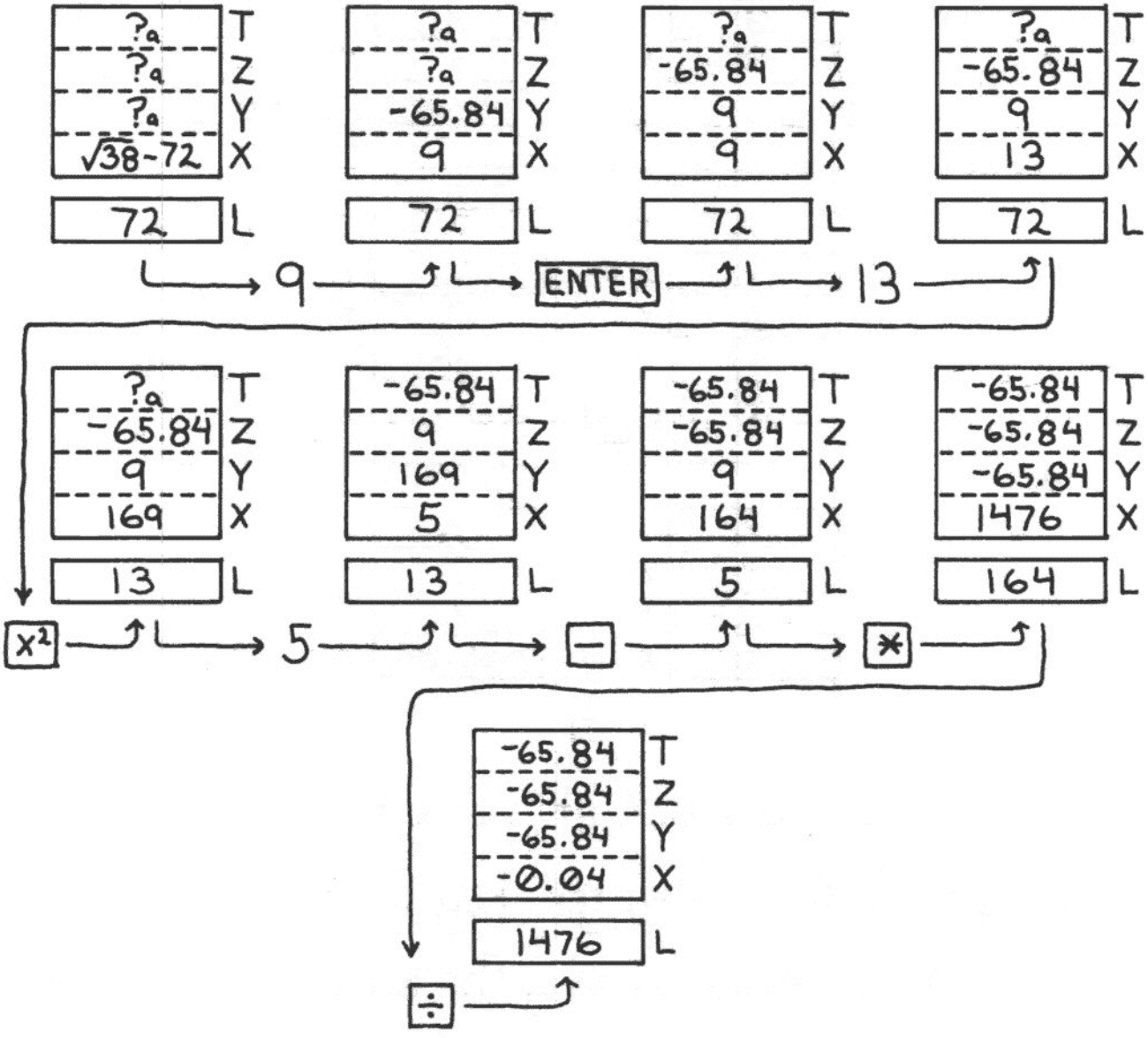
Notice that the stack lifts when you key in 72, but it doesn't lift when you key in the 5 or the 7. If you don't understand why the stack lifts in one case and doesn't in the other, review page 66.

Challenge: Coming from the last page, evaluate

$$\frac{\sqrt{38}-72}{9*(13^2-5)}$$

Solution: 9 [ENTER] 13 [SHIFT] [x^2] 5 [−] [×] [÷]

And the stack? Well, it looks like this:



We're showing "rounded-off" versions of the numbers. There's really 10 digits in each register.

Answer: Well, if you just did the challenge on this page, then the L-register contains 4.000000000. However, if you are just skipping around for the bonus points, you are cheating! And we have no idea what's in your L-register. You could try VIEW L ([Shift] [VIEW] ⊡ L), then press ⇐ to clear the display "window." Now, move on to the next page (except you cheaters who are skipping around).

Bonus Question: What's in the L-Register?

Challenge: Now add 4 to the last result.

Solution: 4 ⊞, and the answer is 3.955395944.

Now, go back to page 75 and work that challenge.

Challenge: Calculate A. $(3/5)^{1/3}$

B. $(3/5)^3$

Solution: A. 3 [ENTER] 5 [÷] 3 [1/x] [Shift] [y^x] gives 0.843

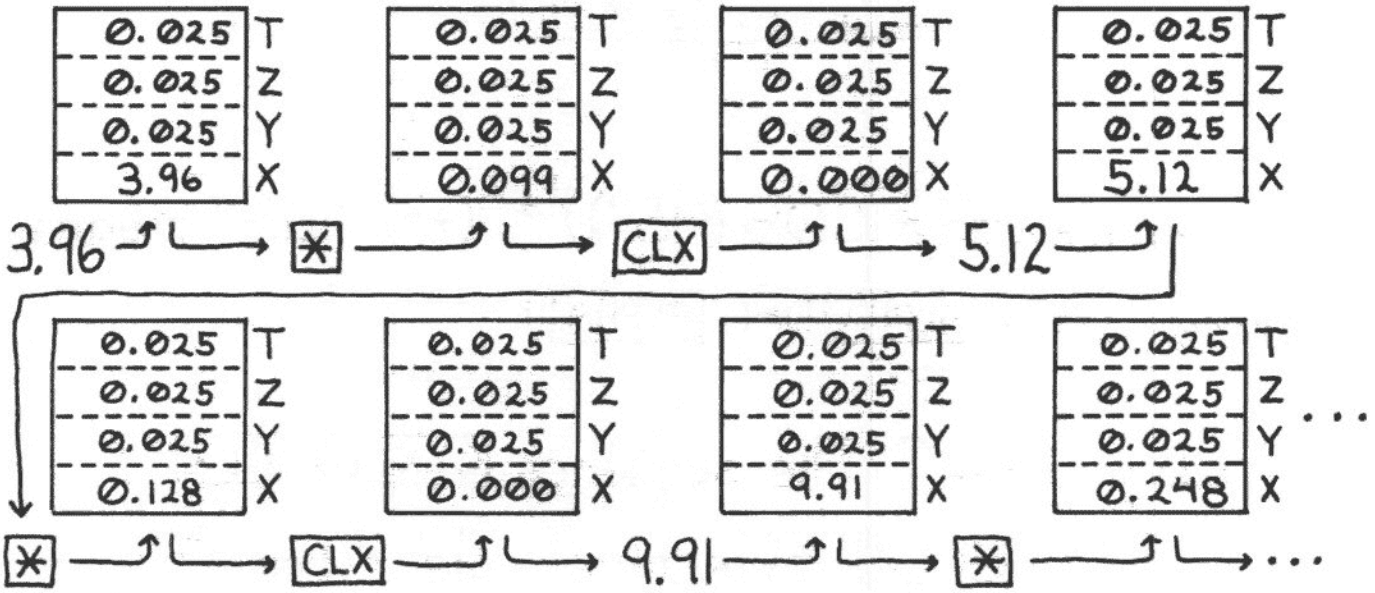
B. 3 [ENTER] 5 [÷] 3 [Shift] [y^x] gives 0.216

Why don't we press [ENTER] after [÷]? Would it make a difference if we did press [ENTER]?

The reason we don't press [ENTER] here is because stack-lift is left enabled by the [÷] function. The only difference that would result by pressing [ENTER] here is that we would have pressed an unnecessary key. Thus, we would have wasted an essential fraction-of-a-second of our lives (not good). [ENTER] leaves stack-lift disabled, so the extra [ENTER] would have no effect on the result.

Challenge: Let's say that you have a list of 100 numbers that starts out 3.96, 5.12, 9.91, 10.67,..., and you need to divide each of these numbers by 40 (the same as multiplying by 0.025). Making use of the fact that the T-register doesn't change when the stack "drops," outline an easy way to accomplish this task.

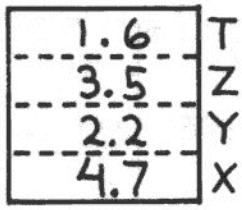
Solution: First, fill the stack with 0.025 (40 [1/x] [ENTER] [ENTER] [ENTER]), then use this sequence:



Similarly, if you had a list of numbers that you wanted to subtract 3 from, you could fill up the stack with -3's and use [+].

TEST

1. Without using the [ENTER] key, configure the stack as such:



2. Using the numbers in the stack from the above problem (don't key in any numbers), compute:

$$\frac{(3.5-2.2)^2+4.7}{1.6}$$

3. Compute $(45 * \text{COS}(45°))^{1/3}$ (the little ° means degrees - [XEQ] [ALPHA] DEG [ALPHA])

4. True or false? The sequence [CLX] ∅ [CLX] ∅ [CLX] ∅ [CLX] ∅ will effectively clear the stack-registers X, Y, Z, and T.

ANSWERS

1) 1.6 [X<>Y] [X<>Y] 3.5 [XEQ] [ALPHA] RAD [ALPHA] 2.2 [STO] 00 4.7 is one possibility. The trick here is that the execution of almost any function (other than ENTER or CLX) leaves stack-lift enabled. Thus, when you do something like [XEQ] RAD (or even press [XEQ], then [←] to clear it away) the computer assumes you're done keying in the previous number, and stack-lift is enabled.

2) [R↓] [−] [shift] [x²] [XEQ] [ALPHA] R↑ [ALPHA] [+] [X<>Y] [÷] = 3.9938

3) 45 [COS] [shift] [LASTX] [×] 3 [1/x] [shift] [y^x] = 3.16883

4) False; CLX leaves stack-lift disabled, so the zeros are not pushed up into the stack. Though it's rarely necessary, you can clear the stack by executing the function CLST, or you can clear it by pressing 0 [ENTER] [ENTER] [ENTER].

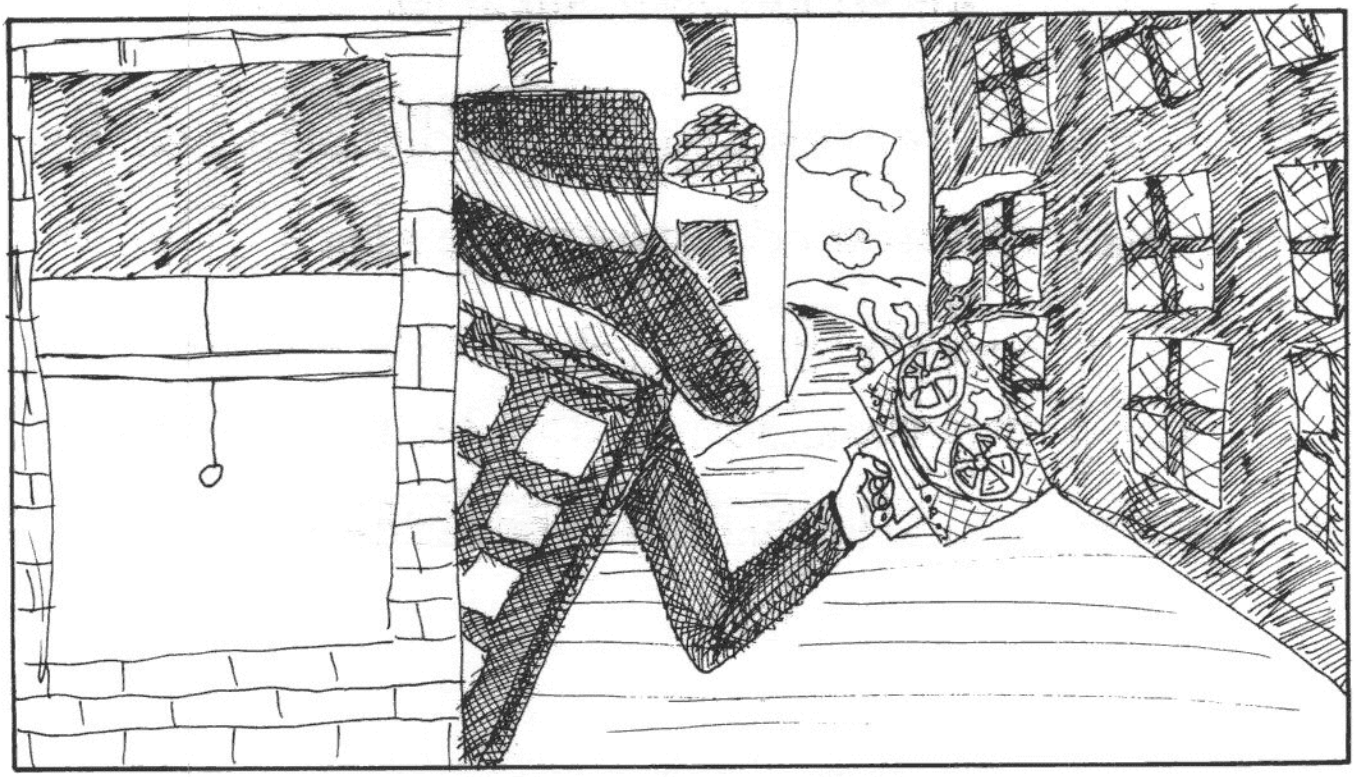
Preview Problem

You are given a list of 10 data shown below, and you know from your previous studies of latin that a <u>datum</u> is 1 piece of <u>data</u>. Your mission, should you choose to accept it, is to run each datum through the formula:

$$32+\left(\frac{9\sqrt{\text{datum}}}{44}\right)^{7}(\text{datum})^{2}.$$

What keystrokes would you use each time?

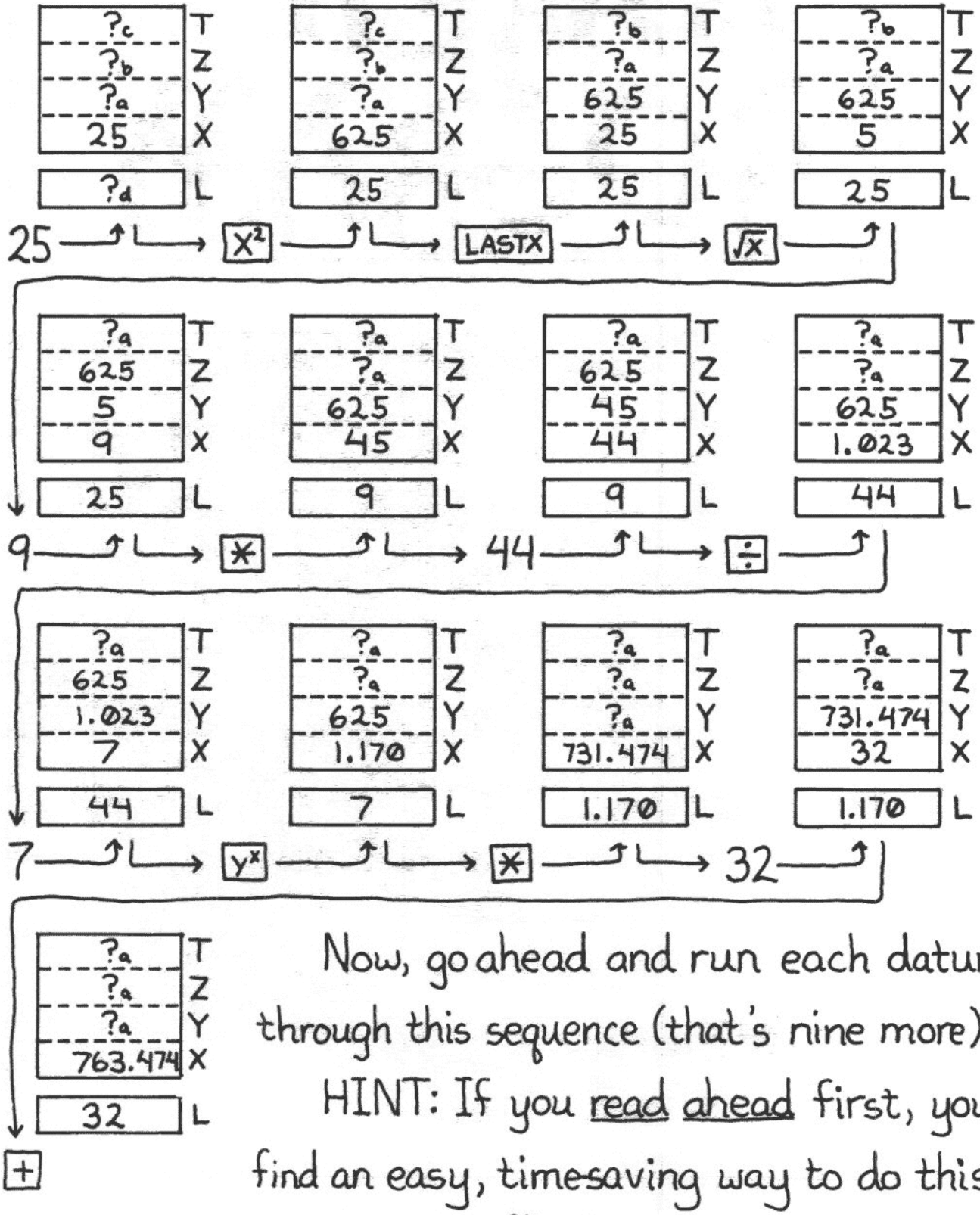
Data: 25, 49, 64, 12, 3, 9, 5, 26, 31, 33



Answer: [Shift] [x^2] [Shift] [LASTX] [$\sqrt{x}$] 9 [×] 44 [÷] 7 [Shift] [y^x] [×] 32 [+]

(Whew!)

If we run the first datum through these keystrokes, the stack looks like this:



Now, go ahead and run each datum through this sequence (that's nine more).

HINT: If you <u>read ahead</u> first, you'll find an easy, timesaving way to do this.

THE NAKED PROGRAM

In the preview problem, you were faced with a list of data that you had to run, piece-by-piece, number-by-number, through a sequence of keystrokes. The result that you got from the keystroke sequence will depend on the original number. In other words, each unique number input to the sequence of keystrokes will return a unique output.

The input is the number that enters a process, and the output is the number that results from that process. In the above case, the process is the sequence of keystrokes.

We could substitute the word "program" for the word "process," because what is a program but a process that is carried out by you and the computer? So the keystroke sequence is a program.

It may come as quite a shock to you when you realize that you have already created a program. The keystroke sequence you developed as a solution to the preview problem is a program.

At the present, this program is recorded in your mind (and a couple pages back). When you work through

the formula:

$$32+\left(\frac{9\sqrt{datum}}{44}\right)^{7}(datum)^{2}\quad,$$

first, you must "call up" the program in your mind and, with the help of your fingers, work through it, step-by-step, starting with [X²] and ending with [+].

BUT...
THERE IS A BETTER WAY.

Why clutter <u>your</u> mind with the numerous keystroke sequences required to resolve your common mathematical problems? Why not <u>store those keystrokes</u> in the continuous memory of the powerful HP-41?

GIVE IT A TRY →

Challenge: Key in a program that you can use to solve the equation:

$$(\text{output}) = 32 + \left(\frac{9\sqrt{\text{input}}}{44}\right)^7 (\text{input})^2$$

Before we spell out the solution to this challenge, we will describe a couple of things that everyone should know about program mode on the HP-41.

PROGRAM POINTER

First, we will describe the program pointer. This pointer is what the computer uses to remember where it is in program memory. There is only one program pointer, and the HP-41 always knows where it is. The HP-41 users, however, are occasionally unsure of the position of the program pointer. But this uncertainty is easily remedied by going into program mode and looking at the display.

When you turn on the HP-41, it wakes up in what's called "RUN mode." To put the HP-41 into program mode, press [PRGM]. To put the computer back into RUN mode, press [PRGM] again. [PRGM] is a toggle key.

Now, when the HP-41 is in program mode, you can move the program pointer one step at a time in either direction - forward or backward - by using the [SST] or [BST] key. SST means "single-step," and BST means "back-step."

The [←] key is the "delete" key in PRGM mode. Pressing this key erases the line you're looking at. DON'T CONFUSE [←] WITH [BST]! Notice also that if you want to key in CLX as a program line, you must use [Shift] [CLX] or [XEQ] [ALPHA] CLX [ALPHA].

In general, to key in any program line, just key it in. It will be <u>inserted</u> right after the line you're looking at, and it becomes the new "line you're looking at" (i.e., the program pointer moves to the new line).

For now, you need to position the program pointer to the end of program memory. The easiest way to do this is just GTO . . The HP-41 will display PACKING for a short while.

SIZE

Next, put the HP-41 into program mode. The display shows: 00 REG nnn, where nnn is some two- or three-digit number. The number nnn shows you how many empty registers that you have allocated for programming. If nnn is 00, you have no room allocated for storing program lines.

Go through this exercize to set the number of program registers to20: First, XEQ "SIZE" 000 (from now on, quote marks (") in keystroke and program listings mean ALPHA, and they correspond to the ᵀ in the display).

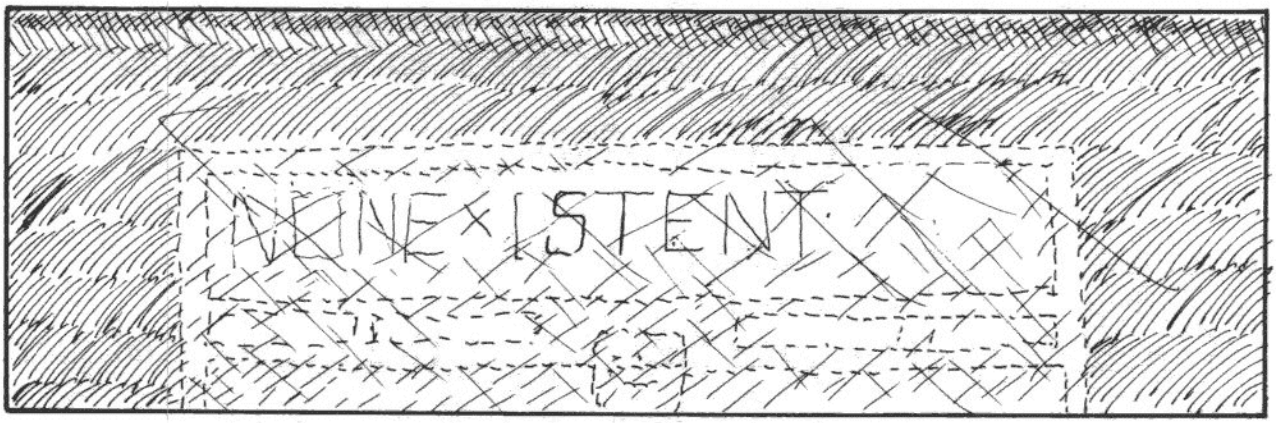
SIZE 000 means that you have no memory allocated to data registers. Get into RUN mode and press STO 00. You will get the message NONEXISTENT. Data register 00 doesn't exist because it is not allocated. Try STO 02. You get another NONEXISTENT. You have

no memory allocated to data registers.

Go into program mode and the display will show 00 REG nnn. Since you have no memory allocated to data registers, all your memory is allocated to program storage. For our purposes, set nnn equal to 20 by the following sequence:

XEQ "SIZE" (nnn-20).

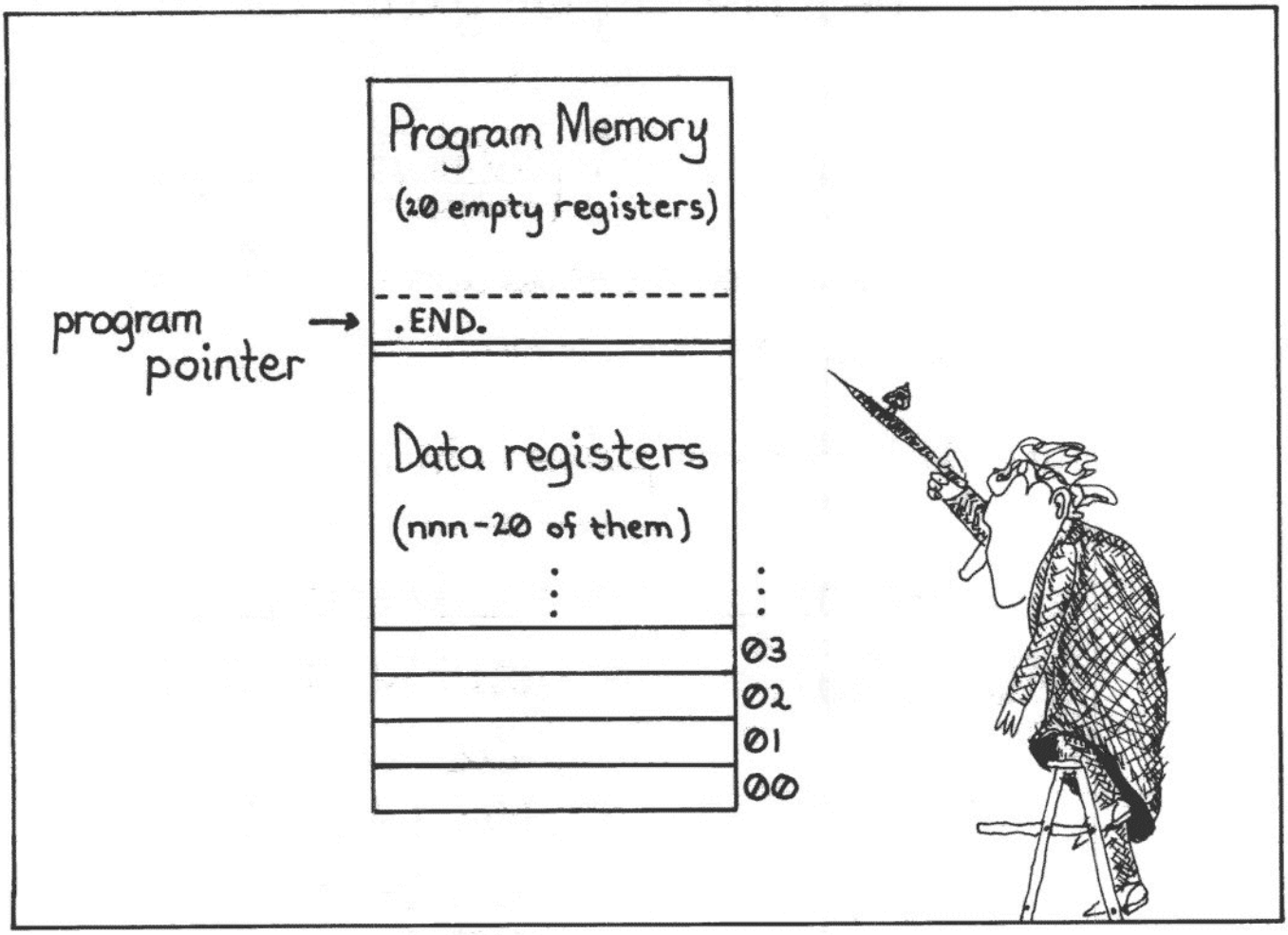
So, if nnn=046, you set the SIZE to 026.



THE PERMANENT .END.

Press the SST key. The display will show .END. REG 20. This is the permanent END of program memory. This is the thing that you move around when you XEQ "SIZE." The .'s preceding and following the permanent .END. distinguish it from a regular program END (keep reading).

Your memory is now set up like this:



Notice that the program pointer is positioned to the permanent .END. of program memory, and this is what appears in the display. So, in addition to the X- and ALPHA-register, the display "window" can be positioned over any line in program memory, as directed by the program pointer.

Now, key in GTO . 000. This moves the program pointer back to line 00. The display shows 00 REG 20.

Here is the solution to the previously posed challenge; this is one program you can use to solve the equation:

$$(\text{output}) = 32 + \left(\frac{9\sqrt{\text{input}}}{44}\right)^7 (\text{input})^2$$

(notice that this is the same equation we referred to in the preview problem).

Be sure that your computer is in program mode!

Key-in	Display
[LBL] "FIRST"	01 LBL"FIRST
[x^2]	02 X↑2
[LASTX]	03 LASTX
[$\sqrt{x}$]	04 SQRT
9	05 9
[×]	06 *
44	07 44
[÷]	08 /
7	09 7
[y^x]	10 Y↑X
[×]	11 *
32	12 32
[+]	13 +

Remember, the quote marks around the FIRST in line 01 mean: press the [ALPHA] key.

So, up to now, we've shown that the only thing you have to do to write a program to solve the equation:

$$(\text{output}) = 32 + \left(\frac{9\sqrt{\text{input}}}{44}\right)^7 * (\text{input})^2$$

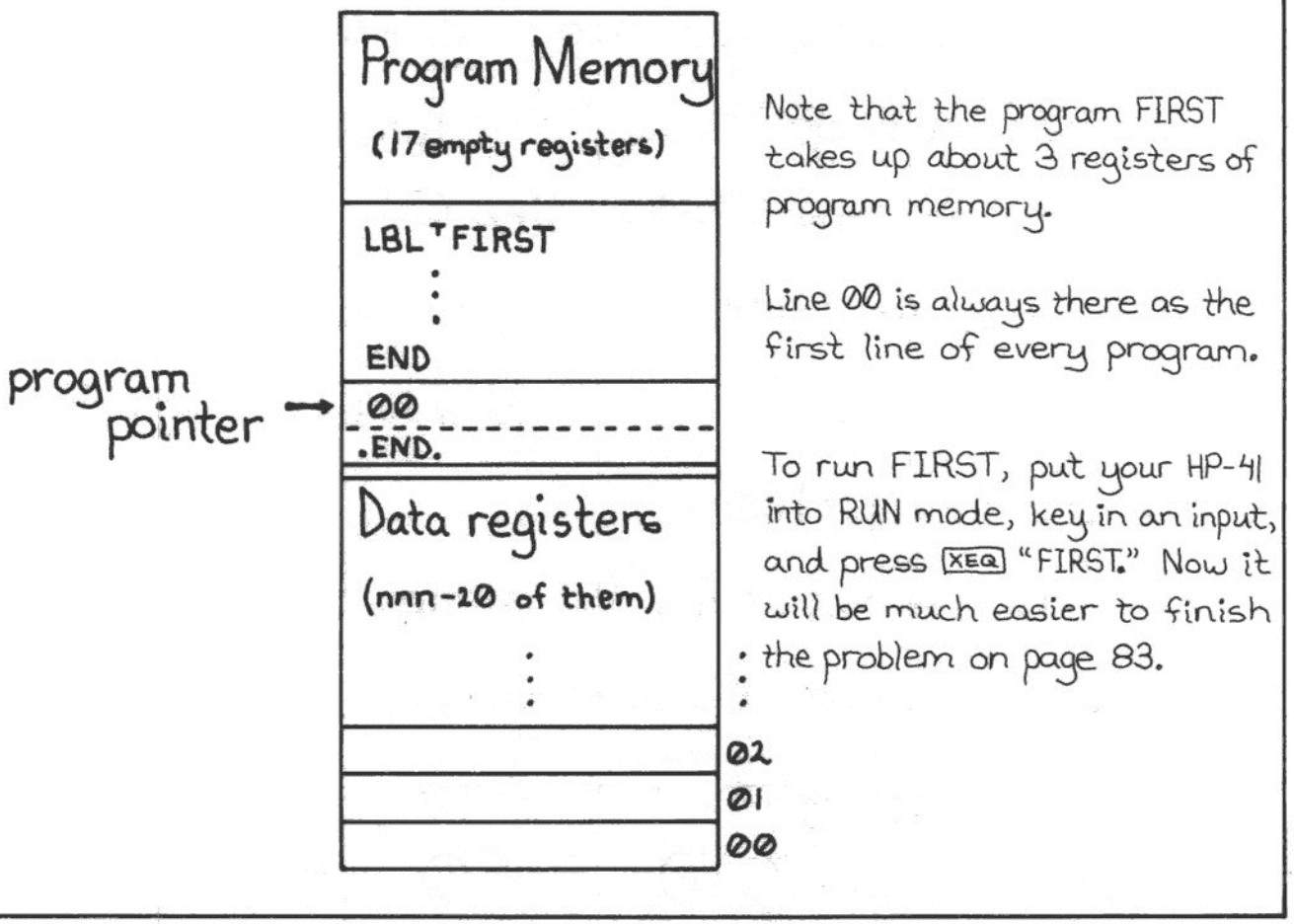
is to figure out the keystrokes necessary to get an answer, put the computer into program mode, key in a label, and then go through the keystrokes. Notice that, except for the label at line 01, the keystrokes for the program FIRST are identical to the keystrokes originally developed to solve this equation in the preview problem.

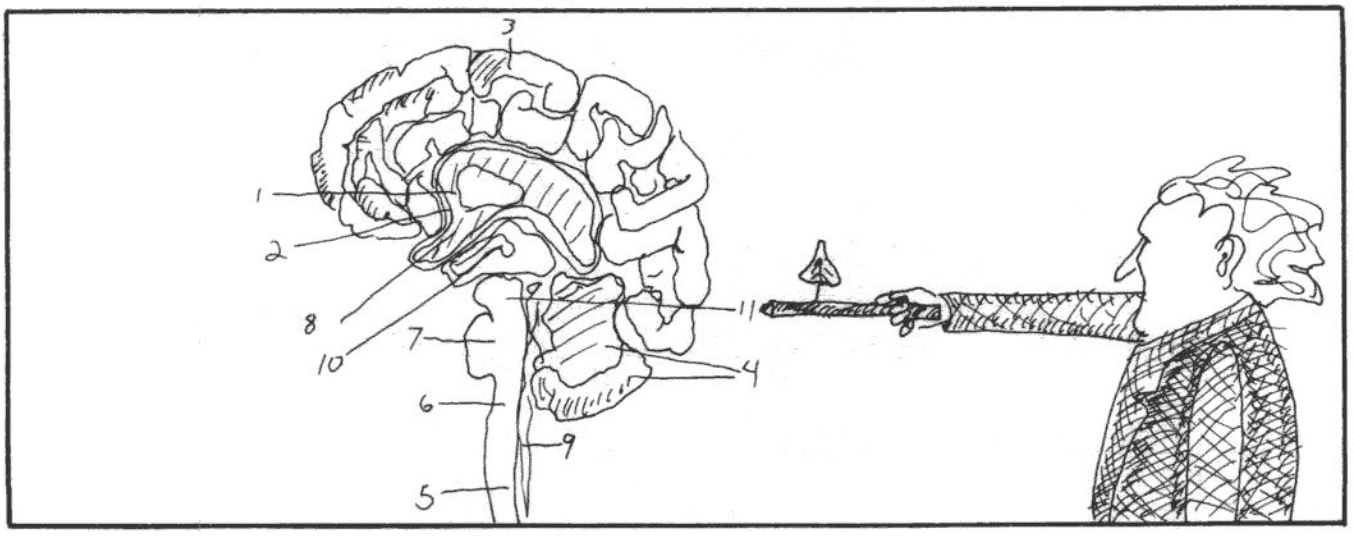
The LBL we put at line 01 is important and it is <u>always</u> good to put a LBL of more than one letter as one line of any program. This allows you to access (call) the program using GTO or XEQ. LBL's are discussed in more detail later. But, for now, remember that a LBL of more than one letter should <u>always</u> be included in a program.

Finish off the program by pressing [GTO] [.] [.]. This places a normal END (different than the

permanent .END.) at the end of the program FIRST. And it positions the program pointer to the end of program memory.

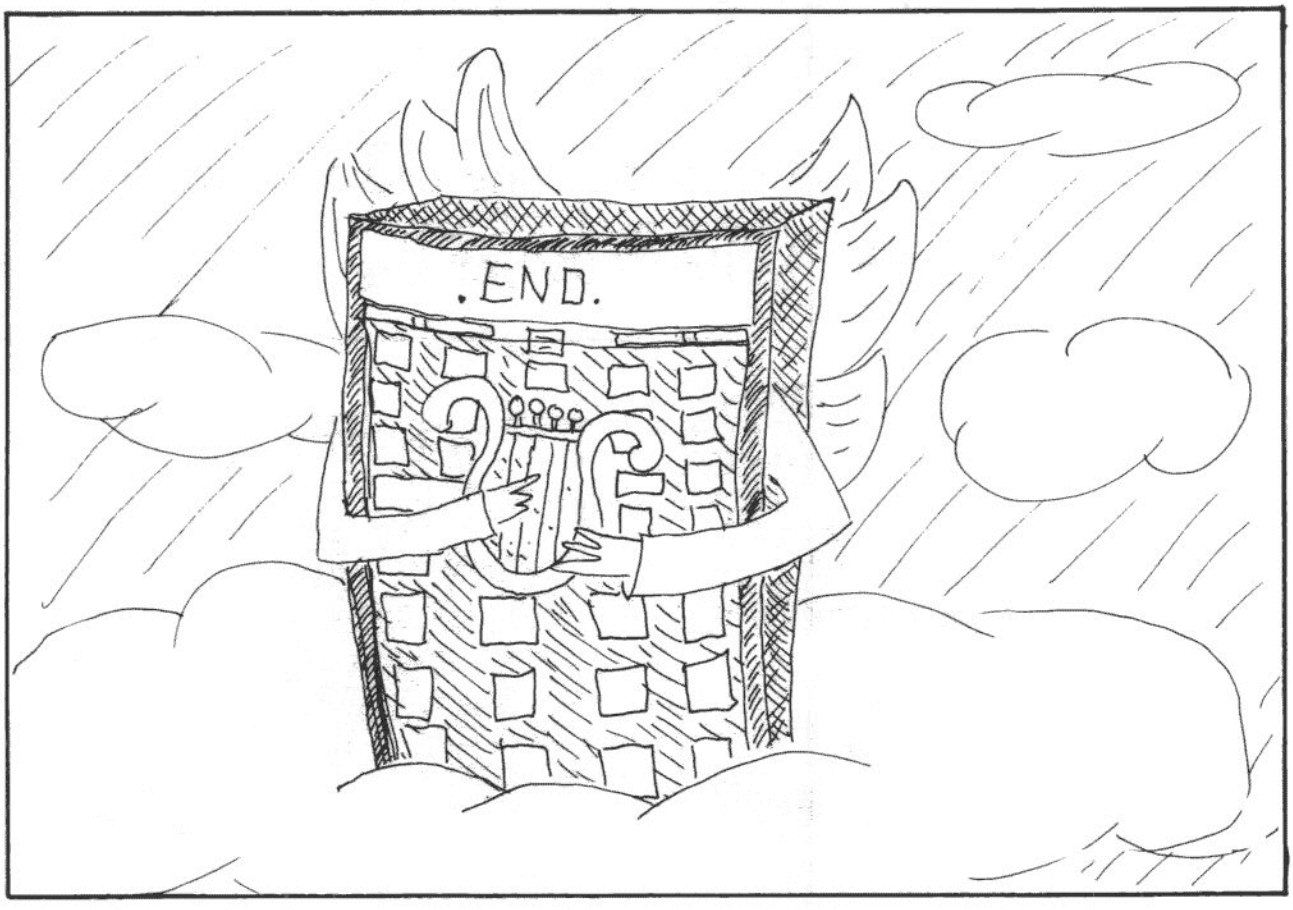
Your memory looks like this:





Whenever you see the permanent .END. in your display, your program pointer is positioned to the last program in memory. There is only one .END. and it appears at the end of the last program in memory.

There can be numerous normal END's in memory. The normal END separates one program from another program. If your program pointer is positioned to one program and you want to jump to another program, you have to call a label in the other program.

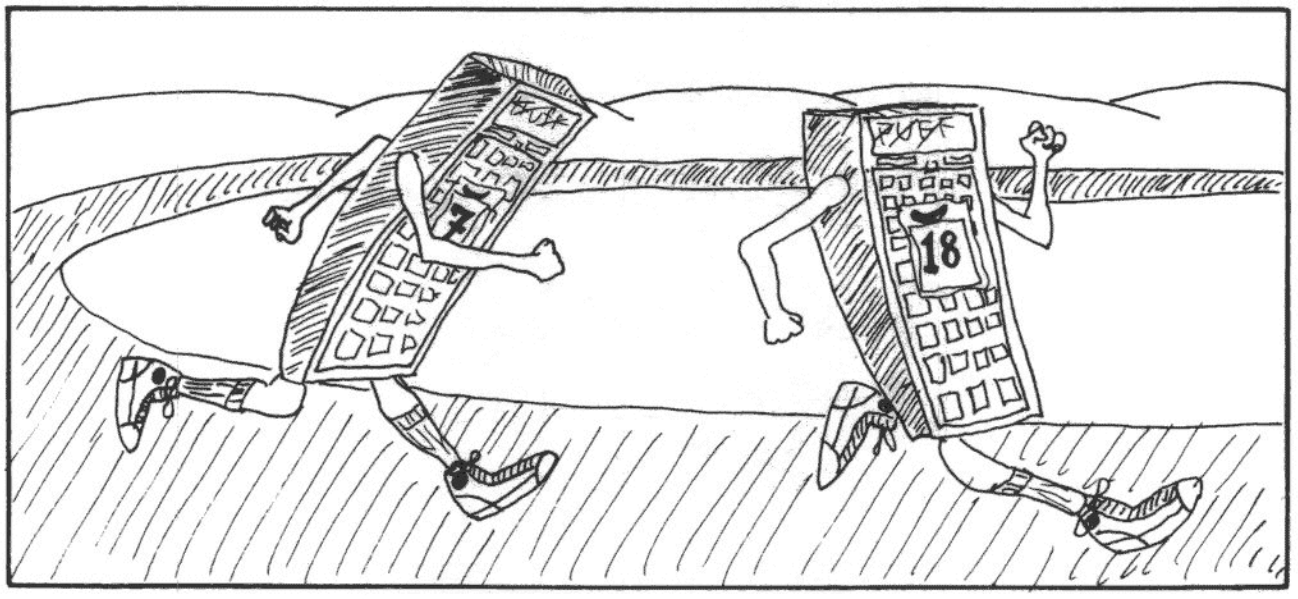


Question: In run mode, what are two ways to position the program pointer to line 00 of the current program?

Answer: [GTO] [.] 000 or [RTN]

When you are positioned to a program, you can always move the pointer to any line of that program by using the [GTO] [.] key sequence. This works regardless of whether you are in program mode or run mode. So, if you want to go to line 20, key [GTO] [.] 020. To go to line 125, key [GTO] [.] 125.

When you are in run mode, the [RTN] function sends the program pointer to line 00 of the current program.



QUIZ

1. What are two ways to determine where the program pointer is in program memory?

2. The formula for the volume of a sphere is $\frac{4}{3}\pi R^3$, where R is the radius of the sphere. Write a program to compute the volume of a sphere given its radius. That is, write a program to solve the equation: output = $\frac{4}{3} \times \pi \times (\text{input})^3$.

3. How many permanent .END.'s are there in program memory?

4. What happens when you are in program mode, you've just finished keying in a program, the END is showing in the display, and you press the [SST] key?

5. What happens when you press [GTO] [.] [.] ?

6. Write a program to take the value in register 03, divide it by the value in register 04, and add that to 4 times the value in register 02.

7. True or false? The display "window" can be positioned only over the X- or ALPHA-register.

ANSWERS

1. The first way to determine where the program pointer is in program memory is to press the [PRGM] key to put the HP-41 into program mode. The line number that is in the display is where the program pointer is.

There is another way that we haven't mentioned yet, and that is to press and hold down either the [SST] key or the [R/S] key. The line to which the program pointer is positioned will appear in the display. Then, after holding the key down for a short while, a NULL will appear in the display, to tell you that the computer will now <u>ignore</u> the command specified by the key you are holding down. So, the HP-41 <u>won't</u> perform the SST or the R/S when you let up the key.

2.

01 LBL"VOL	04 LASTX	07 *
02 3	05 /	08 PI
03 Y↑X	06 4	09 *

Lines 02 and 03 cube R, lines 04 and 05 divide R^3 by 3, lines 06 and 07 multiply by 4, and lines 08 and 09 multiply by PI.

3. One.

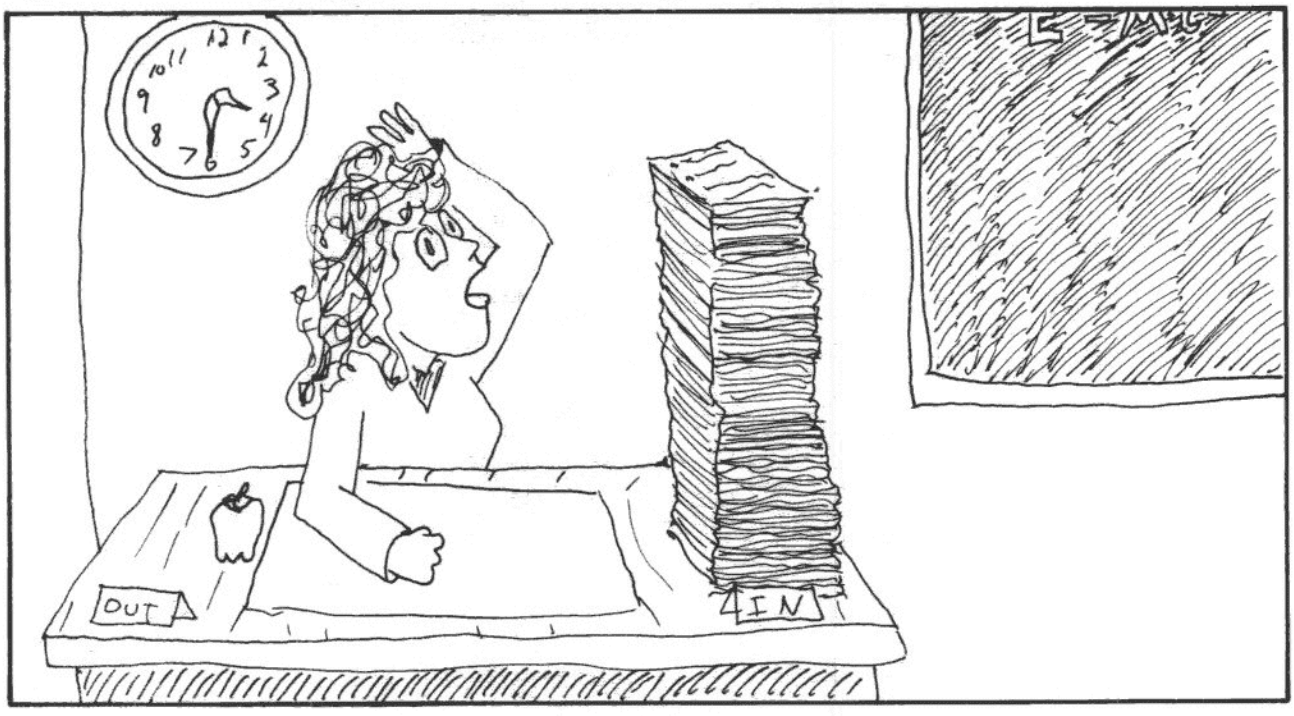
4. The program pointer moves to line 01 of your program (try it).

5. The computer puts a normal END statement on any program in memory that doesn't already have an END. Then the program pointer moves to the .END. of program memory.

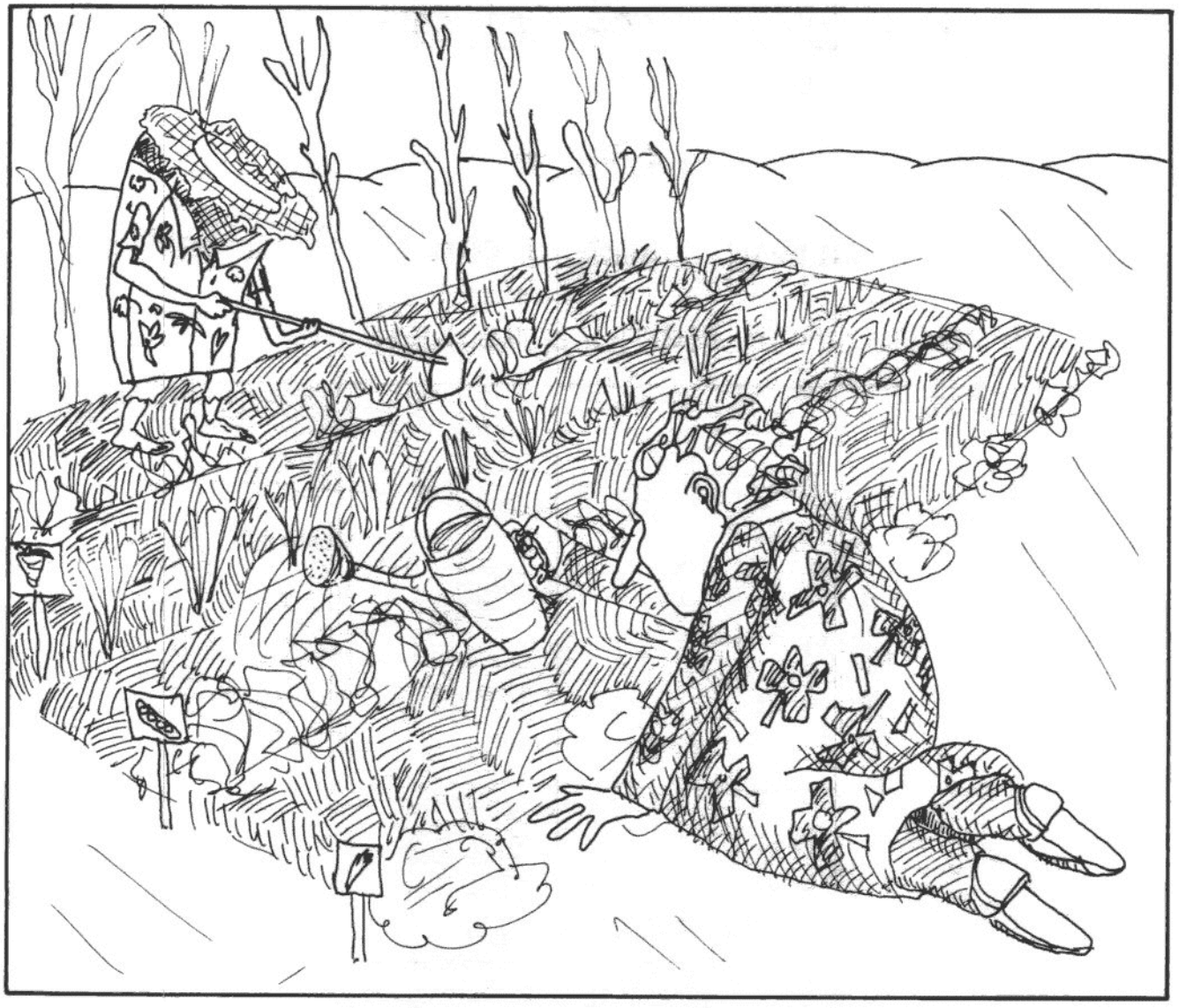
6.

01 LBL "QU	05 4
02 RCL 03	06 RCL 02
03 RCL 04	07 *
04 /	08 +

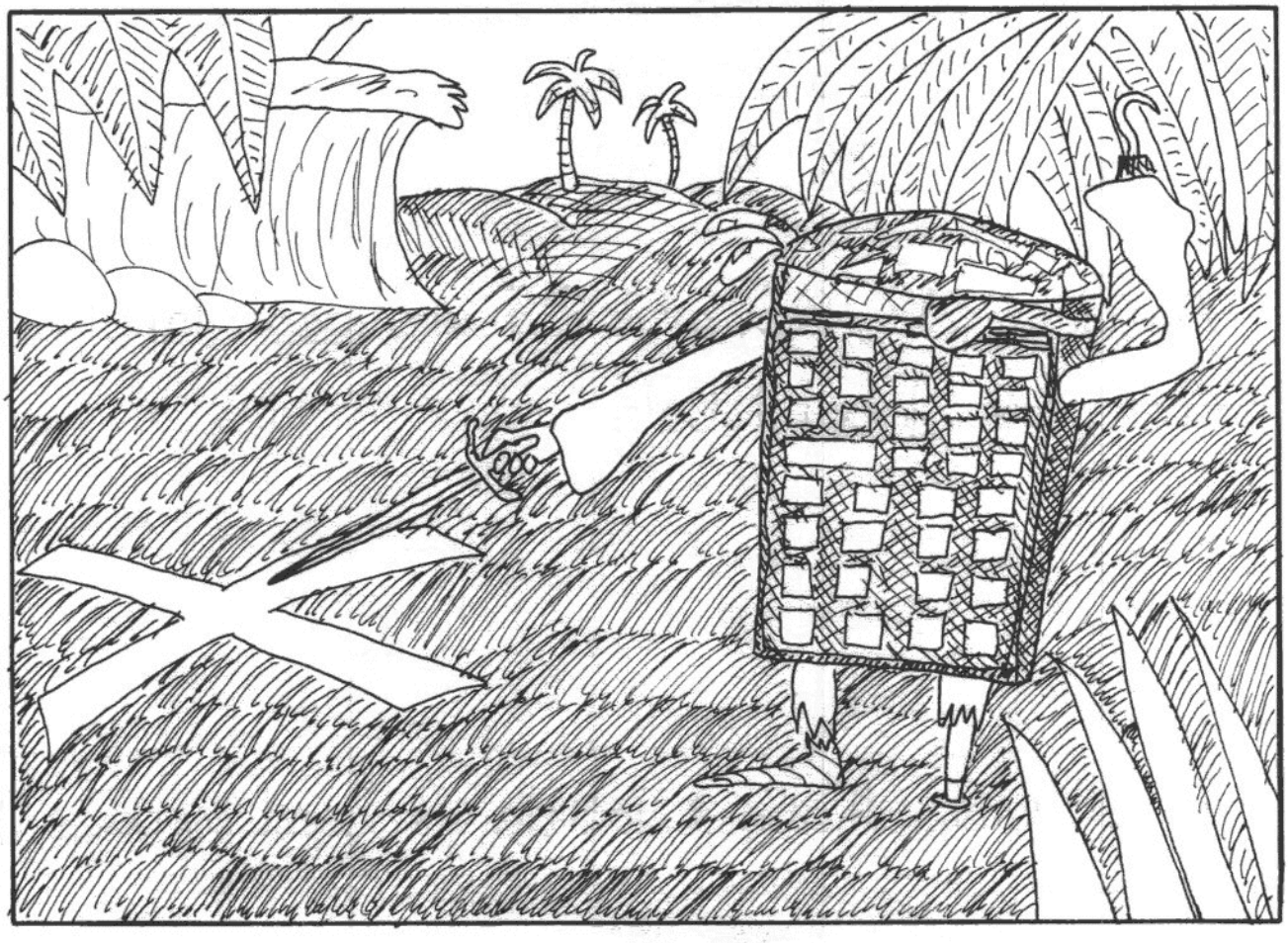
7. False; the display "window" can be situated over any line in program memory, as well.



LABELS AND BRANCHING



The HP-41 uses labels as points of access to a program. So, whenever the program pointer is jumping more than one line in program memory, it is headed for a label. When you want to run the program "FIRST," you key in the command XEQ "FIRST." This causes the program pointer to go zipping through program memory in a systematic fashion, searching for the label "FIRST." When this label is located, the computer begins program execution at that line.



Question: True or false? When they emerge from the womb, most people know the difference between a global label and a local label.

Answer: False, ... next question?

Next Question: Which of the following are local labels and which are global labels?

```
01 LBL"FIRST
02 LBL 01
03 LBL B
04 LBL "Q
05 LBL 99
06 LBL "99
```

Answer: 02, 03, and 05 are <u>local</u> labels. The rest are <u>global</u> labels.

If you know all about labels, proceed to page 111.

LABELS

There are two types of labels – global labels and local labels. Global labels are used for jumping between programs. Local labels are used for jumping within a program. LBL"FIRST is a global label. You can always tell a global label by the little " that shows in the display (" for "text") right before the letters in the label.

GLOBAL LABELS

A global label consists of one to seven ALPHA characters. However, the single letters A through J and a through e are reserved as local labels. But almost every other keyboard character or combination of characters is allowed as a global label.

GLOBAL LABELS CAN BE ACCESSED FROM ANYWHERE IN PROGRAM MEMORY.

LOCAL LABELS

The most important thing to remember about local labels is that they <u>are</u> local. The only time the HP-41 can "see" a local label is when the program pointer is positioned to the program that contains that label. <u>If there is an END statement between the program pointer and a local label, the HP-41 will never find that label.</u>

Challenge: Write a program that will start at zero and count continuously (pausing at each number) until it is stopped by pressing [R/S].

Answer: (GTO..)

01 LBL "COUNT	05 1
02 0	06 +
03 LBL 01	07 GTO 01
04 PSE ← ([XEQ] [ALPHA] PSE [ALPHA])	

With line 07 (GTO 01) in your display, press [SST]. The permanent .END. should show up in your display. This means that the program with

LBL"COUNT is the last program in memory.

LBL"COUNT is a global label. LBL 01 is a local label. Press [shift] [CAT] 1. You will see that LBL"COUNT is the last entry in your program catalog (CAT 1). Only global labels and END statements show-up in this catalog. LBL 01 does not appear in CAT 1 because it is a local label.

Get out of program mode and then press [XEQ] "COUNT." It works! Line 02 loads a zero into the X-register. Line 03 serves as the beginning of the loop. Line 04 momentarily displays the contents of the X-register, lines 05 and 06 add 1 to the X-register, and line 07 sends the program pointer up to line 03 to repeat the loop. This gives the effect of counting.

Stop the program ([R/S]). Key in 5, and restart the program by pressing XEQ 01. The program starts counting at 5.

Now press [R/S] to stop the program; then press [GTO] [.] [.]. Key in 12 to the X-register, and restart the program by pressing [XEQ] 01.

WAIT A MINUTE!

Question: Why did we get a NONEXISTENT when we tried to XEQ 01? LBL 01 existed just a second ago!

Answer: It still exists, even as you are reading this. But pressing [GTO] [.] [.] put an END on the COUNT program and moved the program pointer to the .END. of program memory. Since LBL 01 is a local label, the computer can no longer "see" LBL 01.

WHENEVER THERE IS AN END STATEMENT BETWEEN THE PROGRAM POINTER AND A LOCAL LABEL, THE HP-41 WILL NOT BE ABLE TO FIND THAT LOCAL LABEL.

We can move the program pointer to the global label COUNT by pressing [GTO] "COUNT."

Now there's no END statement between the program pointer and LBL 01, so press XEQ 01. The program starts incrementing the X-register once again (-Clever, these natives).

WHY USE A GLOBAL LABEL?

Use a global label at least once in every program and at any point in a program that you want to be able to access from another program.

You can call a global label from anywhere in program memory, by using GTO or XEQ. If you do call a global label, using a GTO or XEQ statement, and the program pointer doesn't have to jump over an END statement to get to that global label, then you may be using global labels incorrectly.

Global labels take up lots of memory space, so it is best to use them conservatively.

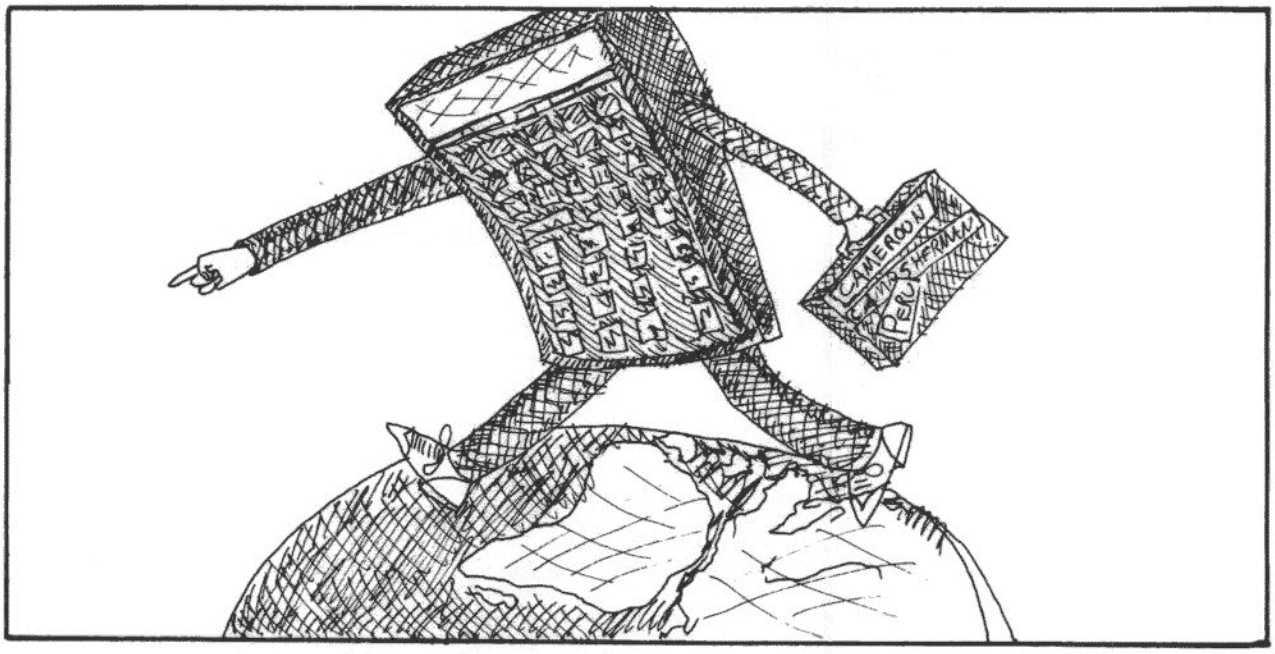
Also, when the HP-41 searches for a global label, it starts at the permanent .END. and searches backwards, one at a time, through your list of global labels. It can take a while to find a global label, especially if your memory is chock-full of global labels.

Also, the HP-41 <u>always</u> has to search for global labels, whereas with local labels, it may already "know" how far to "jump" to get there.

WHY USE A LOCAL LABEL?

Use local labels to make jumps within a program. A program that uses local labels to make internal jumps will run faster and take up less memory than would that same program if it used global labels for internal jumps:

Let's say that at the top of one of your programs you have the global label PRGM1 and, at line 10, you want to jump to the top of the program. Don't use GTO "PRGM1. Instead put a numeric local label (like LBL 03) after LBL "PRGM1 and use GTO 03. You actually save memory by doing this, and the final program will run faster.



ON GTO AND XEQ

We've mentioned GTO and XEQ statements, but we haven't clarified when you would use a GTO in a program and when you would use an XEQ.

GTO (go to) and XEQ (execute) statements are both used for branching to somewhere else in program memory. The difference between GTO and XEQ is best explained by referring to the two programs below:

```
01 LBL"SONG
02 GTO"TN
03 BEEP
04 TONE 0
05 BEEP
06 TONE 4
07 BEEP
08 TONE 8
09 END
```

```
01 LBL"TN
02 TONE 9
03 END
```

Challenge: Key in the two programs on the previous page.

Solution: With your HP-41 in program mode, key the following: [GTO] [.] [.]

Keystrokes	Display
[Shift] [LBL] [ALPHA] SONG [ALPHA]	01 LBL"SONG
[Shift] [GTO] [ALPHA] TN [ALPHA]	02 GTO"TN
[Shift] [BEEP]	03 BEEP
[XEQ] [ALPHA] TONE [ALPHA] 0	04 TONE 0
[Shift] [BEEP]	05 BEEP
[XEQ] [ALPHA] TONE [ALPHA] 4	06 TONE 4
[Shift] [BEEP]	07 BEEP
[XEQ] [ALPHA] TONE [ALPHA] 8	08 TONE 8

(Now press [GTO] [.] [.] or [XEQ] "END.")

[Shift] [LBL] [ALPHA] TN [ALPHA]	01 LBL"TN
[XEQ] [ALPHA] TONE [ALPHA] 9	02 TONE 9
[Shift] [GTO] [.] [.]	
[PRGM] (get out of program mode)	

If you get a NO ROOM or TRY AGAIN message, you'll have to set a smaller SIZE (fewer data registers means more program memory) or clear away some programs, using the CLP function.

Now, run the program SONG (XEQ "SONG"). Note that although you keyed in all those TONE statements, all you get is one note. Put your computer into program mode. The display shows 00 REG nn. So what happened when you ran the program SONG?

Well, what happened is this: The computer started executing at LBL'SONG, but since the second line of that program was GTO'TN, it jumped to LBL'TN and followed those instructions. When it got to the END statement in the TN program, it stopped. So the GTO at line 02 of SONG was like a fork in the road.

That's what a GTO statement is - a fork in the road. It's no temporary sight-seeing trip. It's a heavy commitment.

When the pointer jumps to the specified label, it forgets all about where it jumped from, and it just forges on, following whatever instructions it encounters.

Challenge: Now, go back and change line 02 of the SONG program to "XEQ"TN.

Solution:

[PRGM] (Get into program mode.)

[Shift] [GTO] [.] [ALPHA] SONG [ALPHA] (Move the pointer to the SONG program.)

[SST] (Move to line 02.)

[←] (Delete this line.)

[XEQ] [ALPHA] TN [ALPHA] (Key in the new line 02.)

[PRGM] (Get out of program mode.)

Now execute SONG.

This time, it does a lot more, doesn't it?

And if you listen, you'll hear TONE 9 (high pitch) before the first BEEP. The pointer jumps to the TN program, sounds the TONE 9, and then returns to the point it branched from in the SONG program, to continue on from there.

How did it know to do this?

Well, that's what XEQ really means: "Start

from this point, search for this label, and when you find it, follow your nose through the instructions after it UNTIL you encounter either a RTN statement or the END of a program. From there, you must return directly back to this point (do not pass GO) and continue on from here."

Note that even when you pressed the keys to XEQ "SONG," you were telling the computer this same thing.

But when you gave this instruction, it wasn't doing anything at the time (just sitting around) So, when the program pointer reaches the END statement at line 09, it returns and takes up where it left off- doing nothing. It stops!

Also, you can think of all the functions as having built-in RTN statements. So, after performing them, the HP-41 returns to what it was doing previously, whether it was running a program or just sitting around.

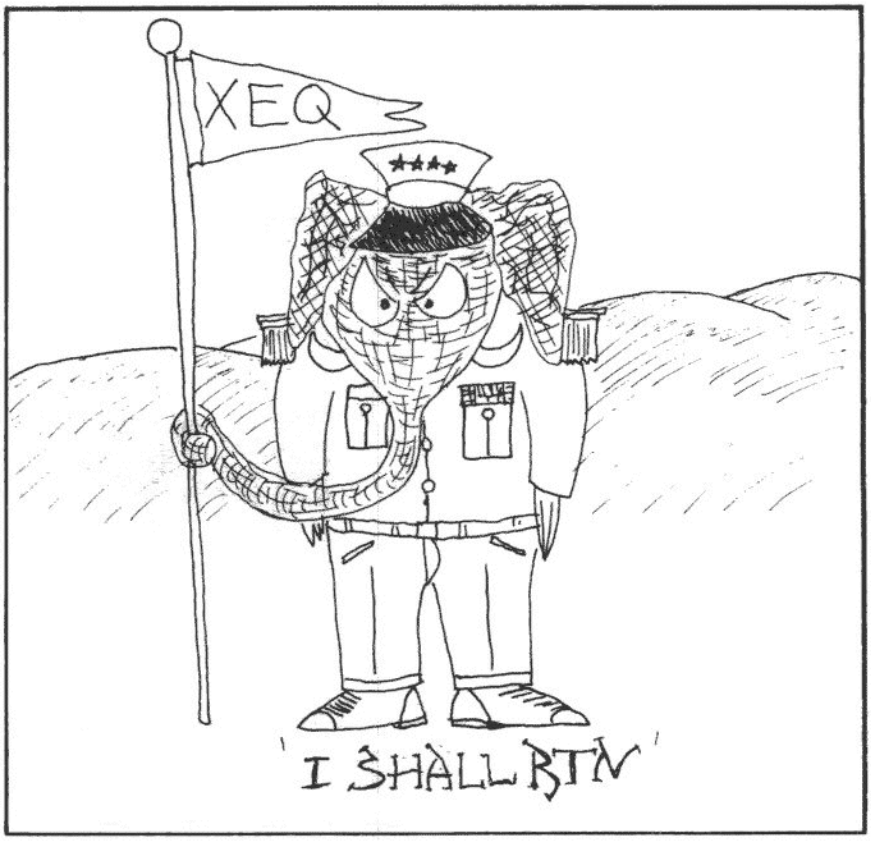
So, all the functions are just one-step side-trips.

In fact, that's what any XEQ really is. It's a side-trip that temporarily branches execution to another place until a RTN or END is encountered.

And, no matter how twisted the path gets, the computer can even remember its path back through 6 "layers" of XEQ's, like this:

```
01 LBL "PILE
02 XEQ 02
03 RTN
04 LBL 02
05 XEQ 03
06 RTN
07 LBL 03
08 XEQ 04
09 RTN
10 LBL 04
11 XEQ 05
12 RTN
13 LBL 05
14 XEQ 06
15 RTN
16 LBL 06
17 BEEP
18 END
```

A mess, right? But the XEQ never forgets!



Dear ALPHA,

Yes, but sometimes I want my programs to branch and sometimes I don't. It all depends on the numbers that come up as results.

What should I do?

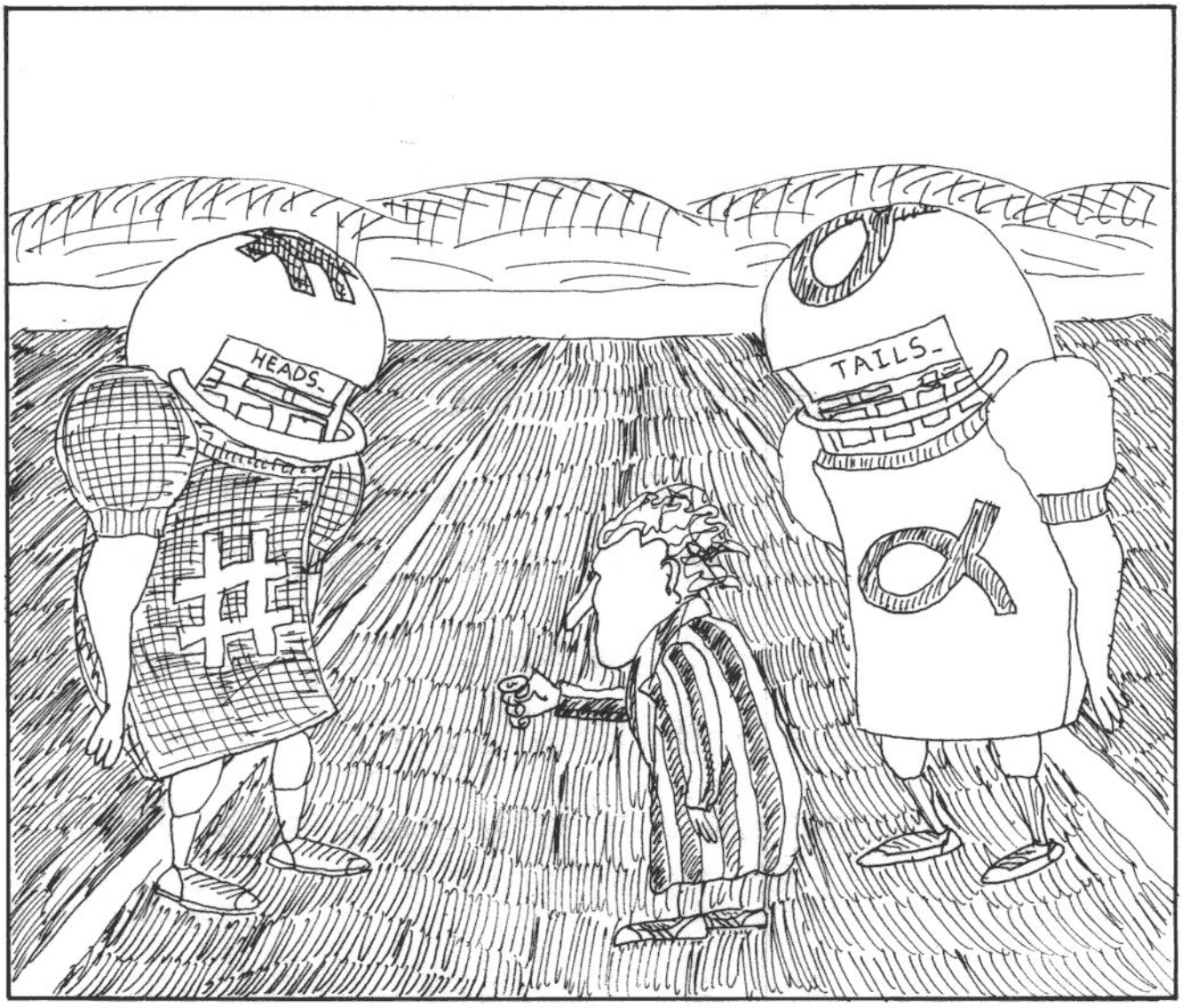
Signed,

Ann Bivalent

Dear Ann B.,

READ ON →

DECISION-MAKING IN PROGRAMS

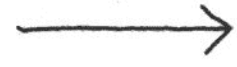


Stop and think for a moment about the design of this book:

We, the authors, wanted to write a self-paced instruction manual on programming the HP-41. The key phrase here is SELF-PACED: We don't know how fast you learn details or concepts. So we had to write a book that would satisfy students of either extreme - slow and methodical or quick and intuitive - and everyone in between.

So we wrote a programmed book.

Review a bit: In the first section, we had to assume that some readers were not yet familiar with the computer's stack logic, ALPHA-register, or function execution. But to allow for those who did know these concepts already, we put a little message here and there, saying, in effect:

"If you already know all this stuff, skip ahead to page xx." 

So a person can find exactly where in the book he or she needs to be in order to start learning- in a very short search- no matter how fast or slow his or her learning is.

This is called solving the "general case," that is, where your learning habits were unknown, but where those very habits could always qualify you for one of the provided options.

But how did we provide those options?

Well, we asked questions of you, the student, and then we gave you directions based upon your answers.

This is called "conditional testing" because we test you with a question, and the instructions we give you afterwards are conditional (they depend) upon your answer to the test.

We have a very powerful tool here in conditional testing. These tests can help make programs (or books) flexible enough to accommodate a wide variety of cases (or students).

The HP-41 has functions that are conditional tests. The conditional test functions are all those that contain question marks (?) as part of their function names. These functions all work similarly:

If the answer to the question asked is "yes," then the computer continues, performing the next program step, etc.

But, if the answer to the question is "no," then the computer skips the step immediately following the conditional test.

Let's try an example.

Remember that little program called "COUNT" (on page 105)? You keyed that in and ran it as a demonstration of how a GTO statement and a label can be used to form a continuously looping program. That version of COUNT will just keep going until you stop it.

But suppose we try this:

Challenge: Use a conditional test to change the program so that it will count up to a certain number (say 10) and then automatically stop.

Solution:

```
01 LBL"COUNT      08 X=Y?
02 10             09 STOP (R/S)
03 STO 00         10 RDN  (R↓)
04 0              11 1
05 LBL 01         12 +
06 PSE            13 GTO 01
07 RCL 00         14 END
```

Plain enough? Move to page 125.

Not ultra-lucid? Study it a bit....

```
01 LBL "COUNT
02 10
03 STO 00
```

After the label, we store the number 10 in register 00. When the computer gets to 10, we want it to stop counting.

```
04 0
```

We want to start the count value at zero.

```
05 LBL 01
06 PSE
07 RCL 00
08 X=Y?
09 STOP
```

After pausing to display the current count value in the X-register, the program recalls a copy of the ending value (10) to the X-register, which bumps the current count value to the Y-register.

Then, at line 08, the conditional test asks the question, "Is the value in the X-register

equal to the value in the Y-register?"

If the answer to this question is "NO" (as long as the current count value in the Y-register hasn't reached 10 yet), then the STOP statement will be skipped. The computer will go on to perform lines 10 through 12, which add 1 to the current counter value and, at line 13, it will go to the top of the loop (LBL 01) and continue.

However, when the count value <u>has</u> reached 10, the answer to the test is "YES," so the STOP statement is performed to halt the program.

Notice something about this program: You can change the length of the count simply by changing lines 02 and 04.

Bonus Question: Suppose you want the program to count up to the number you key in right before you run it. How do you do this?

Bonus Answer: Delete line 02. Then the STO 00 will store whatever value you key in before you run the program.

So, we've used a conditional test to make the COUNT program more flexible. In this case, the test compared the value in the X-register with that in the Y-register.

But there's another kind of conditional test that doesn't have anything to do with the X-register. This test is called a flag conditional test.

RIGHT...

...WHAT'S A FLAG?

Funny you should ask, because that's the next topic.

Simply put, a flag is an indicator which has just two possible values: Set or Clear. (Call it true-or-false, up-or-down, yes-or-no, 1-or-0, us-or-them, whatever you want.)

These indicators - these flags - are stored off by themselves, not in data registers. There are 56 flags in all, and you can check any of them to see if they're set or clear. But there are only 30 that you can change if you don't like what you see. The other 26 are controlled by the HP-41.

And out of the 30 you can control, only 11 (flags 00-10) mean nothing to the computer. The other 19 flags each instruct it to do something.

For example, if you clear flag 26, you are telling the HP-41 to turn off its beeper, so that it will no longer sound BEEP's and TONE's.

So you have 11 flags whose meanings you can determine by the way you use them in your programs.

Challenge: Suppose the one PSE (pause) statement in the latest COUNT program is not long enough for some users, and you want to give them more time to view each number. So you decide to let the user choose one or two PSE's, as follows:

If the user keys in a negative number as the upper limit of the count, this will be taken to mean that two PSE's are desired. Otherwise, if a positive number is keyed in, only one PSE is desired. (In either case, the count will be displayed in positive numbers. The negative sign just means: "two pauses, please.")

GOT THAT?

Use a flag to help you solve this one....

Solution: Here's one way:

```
01 LBL"COUNT      11 PSE
02 CF 00          12 RCL 00
03 X<0?           13 X=Y?
04 SF 00          14 STOP
05 ABS            15 RDN
06 STO 00         16 1
07 0              17 +
08 LBL 01         18 GTO 01
09 PSE            19 END
10 FS? 00
```

SEE HOW THIS WORKS?

Then skip ahead to page 131.

OTHERWISE ... 

```
01 LBL "COUNT
02 CF 00
```

Remember, when you execute COUNT, the value in the X-register tells the HP-41 both how long to keep counting and whether to pause once or twice in each loop.

So, after the label, the first thing to do is to clear the flag we're going to use. Then we know the flag was initially clear; and if we find that it is set later in the program, we know it was set by the program - not prior to running the program. This is called "initializing" the flag.

```
03 X<0?
04 SF 00
```

Next, we test to see if the value in the X-register is negative, using the conditional test X<0? (Is the value in the X-register less than zero?) If YES, <u>do</u> line 04 and continue. If NO, <u>skip</u> line 04 and continue.

So, if the input value is negative (which means

that two pauses are desired), then flag 00 gets set.

```
05 ABS
06 STO 00
```

Here, we store the absolute value of the input number (this is the "positive version" of that number) in register 00. This will be the ending count value – the upper limit.

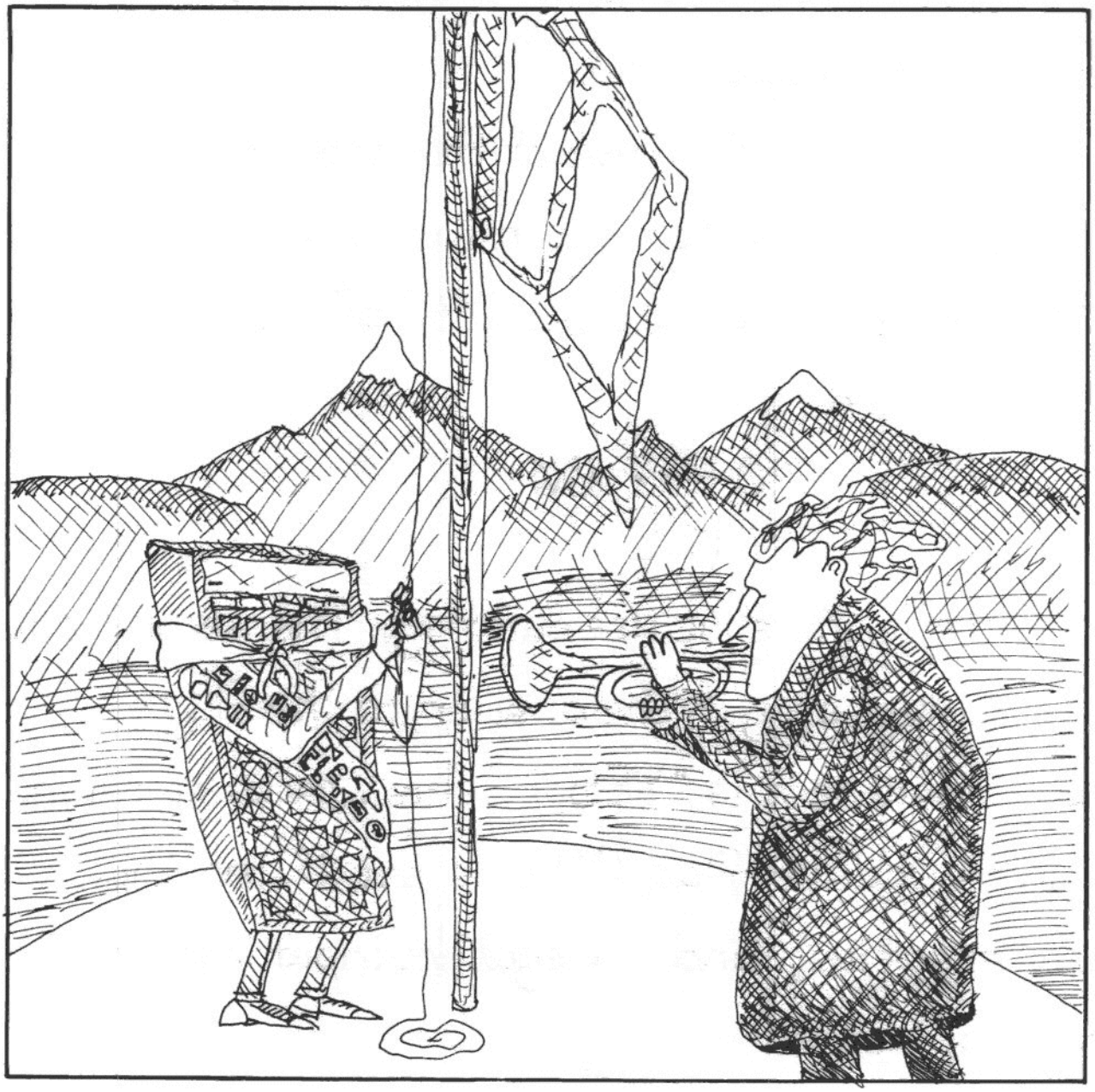
From here, the rest of the program proceeds as usual, except for one minor change. At line 09, we have our pause: 09 PSE, but look at lines 10 and 11.

```
10 FS? 00
11 PSE
```

Here's where we use our flag. The second pause is executed only if the answer to the flag conditional test is "YES." This works out just right because flag 00 was set if the input was negative, and a negative input meant "two pauses, please."

Flags are handy this way. They're a good way to remember a YES-or-NO answer long after the question was asked.

Notice that if flag 00 is set, a little 0 (zero) appears in the display. This convenience also works for flags 01 through 04.



LOOP COUNTERS

Now we're going to take a look at a very convenient pair of functions that can do two useful things at once.

1. They can provide a counter for program loops.
2. They can allow exiting from loops after a given number of cycles.

The two functions are ISG and DSE which mean: "Increment and Skip if Greater than,"

and

"Decrement and Skip if Equal to or less than."

They each demand an argument – a data register number or stack register letter – because they operate on the number contained in the named register.

HERE'S WHAT HAPPENS...

When the computer is told to ISG 02 (for example), it does the following:

1. It looks at the number in register 02, particularly at the integer portion (the portion to the left of the decimal point), and the first five digits of the fractional portion.

Let's suppose the number in register 02 is:

27.05712

2. It takes the digits in the 4th and 5th decimal places (here they are 1 and 2) and makes a number out of them: 12 (twelve).

3. It adds (Increments) this new number to the integer portion of the original number.

27 + 12 = 39, so the new number is:

39.05712 in register 02.

4. Finally, the HP-41 makes a little comparison. It takes the integer portion (39) and compares it to the number that appears in the first three decimal places (057: fifty-seven). Then, if the integer

portion is Greater, the line following ISG 02 would be Skipped.

Thus the name: Increment and Skip if Greater than.

ISG

This all sounds pretty complicated, but let's try some more examples, and you can start thinking about it like this:

Question: "What number should I store in register 00 so that repeated executions of ISG 00 will help me to count from 10 to 27 by 3's?"

Answer: 10.02703

Question: "What number should go there to count from 0 to 9 by 1's?"

Answer: 0.00901 or .00901 or
.00900 or .009

As you can see, because the most common way of counting is by 1's, the increment is assumed to be 1 if the forth and fifth decimal places are zero.

Now, the ISG and DSE functions <u>don't</u> form a program loop all by themselves. They still need labels and GTO's and all that. For example, how would you write a program loop to count from zero to nine, by one's?

```
01 LBL'TRYIT
02 .009
03 LBL 01
04 ISG X   (remember how to key this in?)
05 GTO 01
06 END
```

How about from 0 to 100 by 4's?

```
01 LBL'WORKS
02 .10004
03 STO 02
04 LBL 01
05 ISG 02   (Note: 02 refers to a data register)
06 GTO 01   (Note: 01 refers to a LBL)
07 END
```

How about a loop that counts down from 200 to 29 by 7's?

```
01  LBL 'OKIDOKI
02  200.02907
03  LBL 05
04  DSE X
05  GTO 05
06  END
```

Now, these don't pause to display anything, but they all work the same way. Let's look at the last case: The number that's being decremented is

200.02907.

The first time through the loop, DSE X subtracts 7 from 200 to get 193. Then it compares this 193 to 29. Since 193 is not Equal to (or less than) 29, no skipping takes place. The GTO 05 is performed, and around we go again.

Now the number in the X-register is 193.02907. So DSE X subtracts 7 from 193 to get 186. But 186 is still greater than 29, so no skip. And here we go again, around and around the loop, subtracting

and comparing....

Finally, the value in the X-register has been reduced to 32.02907. This time, when DSE X decrements this X-value by 7, we get:

25.02907.

Well, 25 is less than 29, so the skip takes place. The GTO is skipped, the loop is exited, and that's all she wrote.

Remember:

ISG means Increment (add) and Skip if Greater than.

DSE means Decrement (subtract) and Skip if Equal to or less than.

Well, you knew it was coming, so let's get on with it.... Go for broke!

Challenge: Rewrite the COUNT program just once more (this is the last time – yes, we promise).

Rewrite it so that it uses ISG. The ending value should still be specified in the X-register (negatives still mean an extra pause), but the amount of the increment should be in the Y-register.

So if you want to count to 39 by 3's with <u>two</u> pauses, you'll key in 3 [ENTER] 39 [CHS] [XEQ] "COUNT."

Solution: (one of many possibilities)

```
01 LBL"COUNT    08 X<>Y      15 INT
02 CF 00        09 1 E5      16 PSE
03 X<0?         10 /         17 FS? 00
04 SF 00        11 +         18 PSE
05 ABS          12 STO 00    19 ISG 00
06 1 E3         13 LBL 01    20 GTO 01
07 /            14 RCL 00    21 END
```

See how it works? Then exit, stage right (to page 143).

Want a closer look? OK, next page. ⟶

Now the first five steps of the program are nothing new, right?

```
01 LBL"COUNT
02 CF 00
03 X<0?
04 SF 00
05 ABS
```

We test for a negative number and adjust flag 00 accordingly. If we're counting to 39 by 3's, as stated in the challenge, flag 00 would be set for two pauses.

```
06 1 E3
07 /
```

Now, instead of storing the ending value, we are going to use it to create the index number for controlling the ISG loop. As you recall, the ending value in the ISG loop is the first 3 digits to the right of the decimal point.

So we divide the given ending value, 39, by 1000 (i.e., by 1×10^3 or 1 E3, all the same thing). This moves the 39 over to those 3 decimal places just to the right of the decimal point: $39 \div 1E3 = .039$.

```
08 X<>Y
09 1 E5
10 /
```

Next, we exchange the contents of the X- and Y-registers so that we can horse around a bit with the increment value. Since the increment value in an ISG index number is always found at the fourth and fifth decimal places, we have to divide by 100,000 to move it there. Our example increment was 3. So, $3 \div 1\,E5 = .00003$.

```
11 +
12 STO 00
```

So now we sum the contents of the X- and Y-registers.

$.039 + .00003 = 0.03903$

And, THAT looks like an index number that will help us count from 0 to 39 by 3's. So we store it in register 00, ready to use as our counter.

Now, there's only one page left in this section, and we're doing the best we can to keep you from getting bored. While we're on the subject of boredom, have you

ever tried handlettering an entire book like this? Boy, the things people will do for money!

So, where were we?

```
13 LBL 01        16 PSE
14 RCL 00        17 FS? 00
15 INT           18 PSE
```

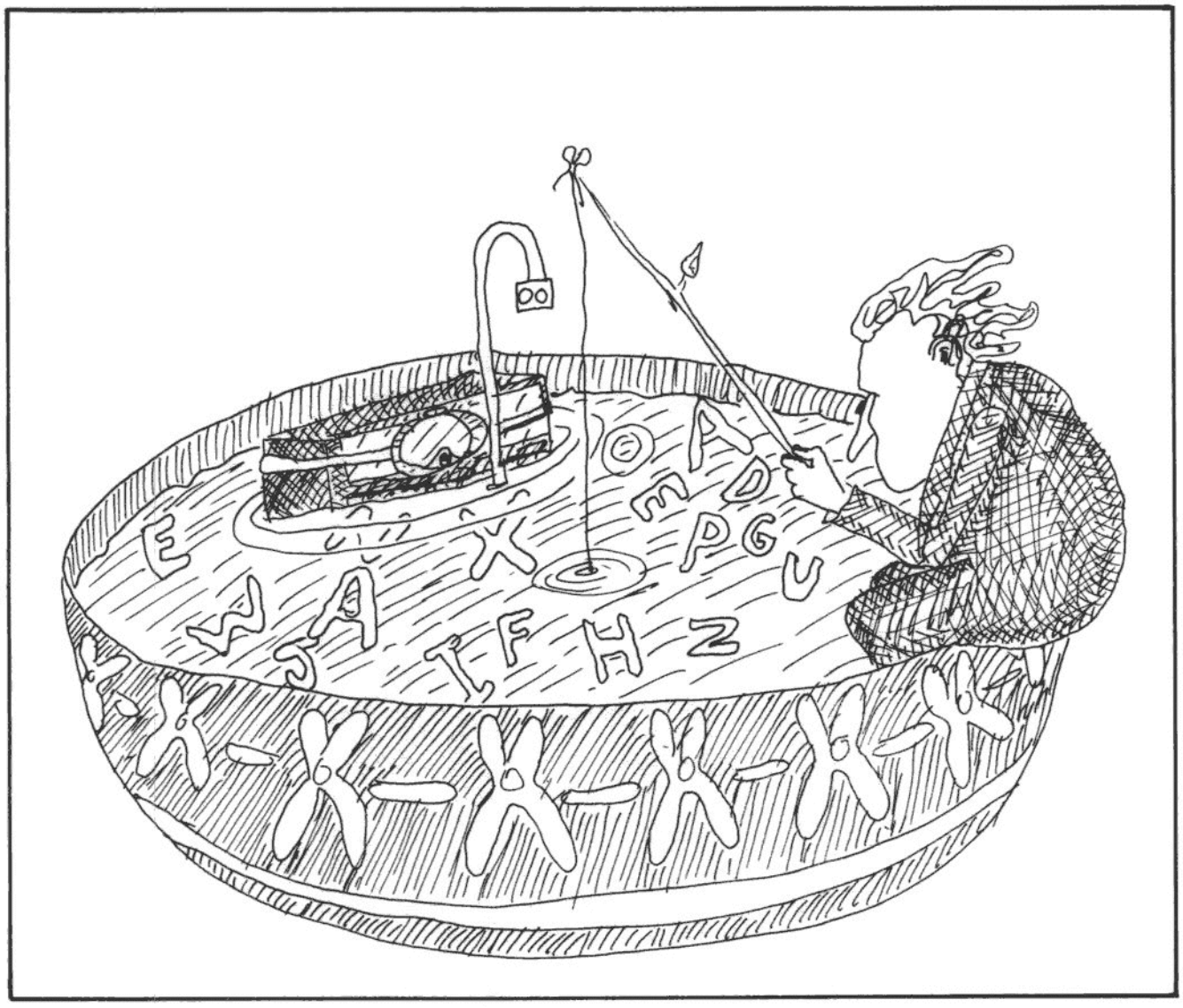
Here, we start the loop with a label. Then, we recall the current counter value, lop off the fractional portion with the INT function, and do the usual charade with the PSE's.

```
19 ISG 00
20 GTO 01
21 END
```

And here's the payoff: That ISG 00 will increment our counter each time through, test it and decide (correctly, of course) when to skip the GTO 01, thus ENDing the count.

Notes

ALPHANUMERICS IN PROGRAMMING



In the "naked program" section, you took a series of keystrokes and turned them into a program. Now, let's see how to make programs more "friendly," via the ALPHA mode.

The HP-41 gives you the capability to make programs user-friendly with a little help from ALPHA strings and built-in functions such as PROMPT, AVIEW, and ARCL. We'll go into gory detail about these built-in functions in a little bit. Now, let's discuss ALPHA strings.

ALPHA strings are messages, or, in some cases, just a collection or "string" of characters- "THIS IS AN ALPHA STRING"; "TIME=?"; "TIME=2PM"; "BLARNEY"- are all ALPHA strings. You can use them in your programs to prompt for input, label your output, or even to tell you what's going on in the program while it's running.

In program mode, your HP-41 will always display ALPHA strings with a little ᵀ preceding them. The only place this little ᵀ (text mark) will

appear is preceding ALPHA strings and global ALPHA labels. Keying in an ALPHA string as a program line is just like keying ALPHA data into the ALPHA-register: press [ALPHA], key the ALPHA string, then press [ALPHA] again. If you have trouble, review pages 39 to 49.

In the "naked program" section, you developed a program to solve the equation:

$$\text{OUTPUT} = 32 + \left(\frac{9\sqrt{\text{INPUT}}}{44}\right)^7 (\text{INPUT})^2.$$

The program to solve it was this:

01 LBL"FIRST	06 *	11 *
02 X↗2	07 44	12 32
03 LASTX	08 /	13 +
04 SQRT	09 7	14 END
05 9	10 Y↗X	

To run the program, you key in a value of INPUT (let's use INPUT=4), then you press [XEQ] [ALPHA] FIRST [ALPHA], and out pops the value for OUTPUT (32.03 if your display is set to FIX 2).

Simple enough to remember, right? But, what if you shelved the program for a while and didn't use it? Then six months down the road, you needed to use it again, but you forgot how it worked. That's where ALPHA strings step in. They make the program help you remember how to use it.

Challenge: Let's put a few ALPHA strings into the program labeled FIRST. Insert a few program lines so that the program will prompt for an input by displaying "INPUT=?" and label the output with "OUTPUT = nn.nnn" where the n's represent the numerical answer.

Solution:

```
01 LBL"FIRST   06 SQRT  11 7     16 "OUTPUT=
02 "INPUT=?    07 9     12 Y↑X   17 FIX 3
03 PROMPT      08 *     13 *     18 ARCL X
04 X↑2         09 44    14 32    19 AVIEW
05 LASTX       10 /     15 +     20 END
```

If you know all about this, turn to page 149.

GORY DETAILS

Okay, it's that time!

PROMPT

PROMPT is used in a program to stop the program and display whatever is in the ALPHA-register at the time. In the program FIRST, it will stop and display "INPUT=?". That's your cue to key in the value for INPUT (try 4). To have the program start running again, press R/S. The program will then use the value 4 just exactly like it did before you put in the prompt. PROMPT does not change the stack or the actual calculation of the program. It just halts the program for some information. The HP-41 goes on its merry way as soon as you press R/S.

ARCL

ARCL recalls from a register the contents of that register and joins it to the end of whatever is already in the ALPHA-register. You can specify either a numbered register or a stack register. (If

you have trouble keying in ARCL X or ARCL Z, turn to page 38.) In the program FIRST, we used ARCL X to add the contents of the X-register to the contents of the ALPHA-register, which was OUTPUT=.

AVIEW

AVIEW displays the contents of the ALPHA-register. Then, in the program FIRST, the END stops the program. So, the final display (for INPUT= 4) looks like this: "OUTPUT= 32.031"

Now, anytime you want to run the program FIRST, all you need to do is load it into your HP-41, (if it's not already in there) and press [XEQ] [ALPHA] F IRST [ALPHA]. The program will tell you what it needs; then it will label and display the answer for you.

How much friendlier can you get?

GORY DETAILS II

In order to be an expert with ALPHA mode on the HP-41, you have to know a few more things. You have to know when to use CLA and when it isn't necessary to use CLA. You have to know how to add things to the ALPHA-register without destroying what's in there already. You have to know how to programmatically turn ALPHA mode on and off. And you have to know how to store and retrieve part or all of the ALPHA-register.

If you already know all these things, then skip ahead to page 154.

CLA, ARCL, AND +

You know that when you're in RUN mode and you press the [ALPHA] key, then press a letter key, whatever was in the ALPHA-register prior to pressing the letter key is cleared away. So, whenever you just click into ALPHA mode, it isn't necessary to clear the contents of the ALPHA-register before you start

typing. It's cleared automatically.

The same is true in programs. That is, unless you make an effort to preserve the contents of the ALPHA-register, a program line that is an ALPHA string will completely replace what's in the ALPHA-register. So it's not necessary to use CLA before an ALPHA string.

BUT:

If you want to preserve the contents of the ALPHA-register and add a character or string of characters to those contents, then you have to use the APPEND character (⊢) as the first character of the addition.

The ARCL function is a kind of "appending" function. The program line: ARCL 14 is like saying "take the contents of register 14 and append it to the contents of ALPHA." ARCL <u>always</u> adds onto whatever is in the ALPHA-register at the time. So, if you want the ALPHA-register clear before you ARCL something, you have to use CLA.

Let's clear the fog with some challenges.

Challenge: Write a program that will prompt for an input with the message "AMOUNT?", then display that input in the following format: $amount CREDIT. So if you input a 5, it will display $5.00 CREDIT. And if you input a 12, it will display $12.00 CREDIT.

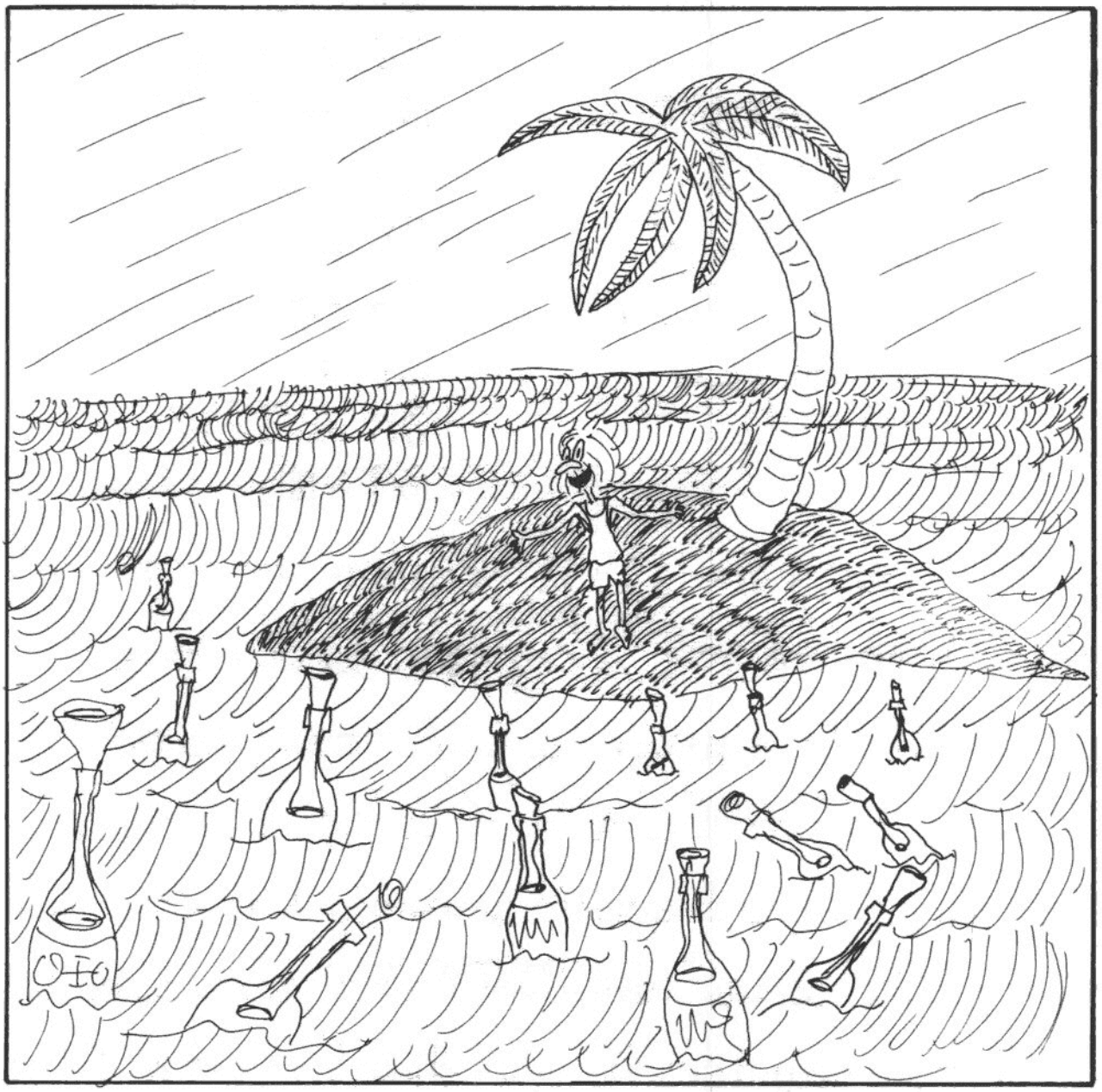
Solution:

```
01 LBL"PR          06 ARCL X
02 "AMOUNT?        07 "⊢ CREDIT
03 PROMPT          08 AVIEW
04 "$              09 END
05 FIX 2
```

Line 02 replaces whatever is in the ALPHA-register with the ALPHA string "AMOUNT?". Line 03 halts the program and displays the contents of the ALPHA-register.

After you key in the amount (to the X-register of course) and press R/S, line 04 replaces the string "AMOUNT?" with the string "$". Lines 05 and

06 add the contents of the X-register to the ALPHA-register in the FIX 2 format. Line 07 adds a space and the word "CREDIT" to the ALPHA-register (the APPEND character is found on the shifted K key in ALPHA mode). And line 08 displays the message.



Challenge: Change the program PR so that if you input a 5 at the prompt "AMOUNT?", it will display "5.00 FEET."

Solution:

WRONG

```
01 LBL"PR
02 "AMOUNT?
03 PROMPT
04 FIX 2
05 ARCL X
06 "⊢ FEET
07 AVIEW
08 END
```

RIGHT

```
01 LBL"PR
02 "AMOUNT?
03 PROMPT
04 FIX 2
05 CLA
06 ARCL X
07 "⊢ FEET
08 AVIEW
09 END
```

The point here is that, because ARCL X adds the contents of the X-register to whatever is in the ALPHA-register, a CLA needs to be inserted before the ARCL X to get the right display. In the previous version, line 04 ("$) replaced the string "AMOUNT?" that was already in the ALPHA-register. So you didn't need a CLA.

ASTO and ASHF

ASTO stores the first six characters in the ALPHA-register into a specified register. ASHF shifts everything in the ALPHA-register six spaces to the left (it lops off the first six characters).

Challenge: Write a program that fills the ALPHA-register with the letters A through X (24 letters), then stores these letters in registers 01 to 04, then prompts the user for an input ("INPUT=?"), the stores the square-root of that input in register 00, then restores the letters A through X to the ALPHA-register. (This is only a drill.)

Solution:

```
01 LBL "FUTILE
02 "ABCDEFGHIJKLMNO
03 "P-PQRSTUVWX
04 ASTO 01
05 ASHF
06 ASTO 02
07 ASHF
08 ASTO 03
09 ASHF
10 ASTO 04
11 "INPUT=?
12 PROMPT
13 SQRT
14 STO 00
```

→

```
15  ARCL 01        18  ARCL 04
16  ARCL 02        19  AVIEW
17  ARCL 03        20  END
```

GOT IT? NEXT PAGE →

NEED SOME EXPLANATION?

The first 10 lines store the letters A through X in data registers 01 through 04 (six letters per register). Don't forget the APPEND character (⊢) in line 03.

Lines 11 through 14 prompt for an input and store its square-root in register 00. And the rest of the program brings the letters A through X back into the ALPHA-register and displays them.

Bonus Question: Why don't we need a CLA between lines 14 and 15?

Bonus Answer: We don't need a CLA because we know each of our 4 ARCL's will add 6 letters; a total of twenty-four characters will be brought into the ALPHA-register. That completely fills the ALPHA-register and pushes out anything already in there.

AON AND AOFF

AON in a program will turn on ALPHA mode. AOFF turns it off.

If a program stops with ALPHA-mode on, then the display "window" will be sitting over the ALPHA-register rather than the X-register. So, as long as this window is clear, you will see the contents of the ALPHA-register.

AON and AOFF, along with STOP, are handy functions when prompting for alphanumeric input. With these functions, for example, a program can ask you a yes-or-no question, and you can respond with Y or N for an answer.

Now, let's take a <u>close</u> look at the AVIEW function.

AVIEW AND FLAG 21

In programs FIRST and FUTILE (pages 146 & 154), when the computer encountered the AVIEW instruction, it displayed the contents of the ALPHA-register and stopped. Actually, in both programs, there wasn't much choice, since AVIEW was the last executable statement. But suppose you had AVIEW's scattered throughout the program? Would it stop at every AVIEW encountered?

(QUESTIONS, QUESTIONS!)

In a program, when an AVIEW is encountered, the status of flag 21 determines whether the program halts or continues. If flag 21 is set, AVIEW will cause the running program to stop and display what's in the ALPHA-register. Then [R/S] has to be pressed to continue the program. If flag 21 is clear, AVIEW will cause the contents of the ALPHA-register to be displayed while program execution continues.

If you have a printer, you'll want to study

more about flag 21, flag 55, and AVIEW. But since we are dealing only with the computer itself (no extra attachments), the above explanation is all we can offer.

When flag 21 is set, AVIEW acts like PROMPT. Heavy seas? Calm them with this example:

```
01 LBL"CALM          08 TONE 2
02 "THIS IS A VERY   09 TONE 0
03 "⊢ SHORT ONE      10 CLD
04 AVIEW             11 25
05 TONE 9            12 SQRT
06 TONE 9            13 END
07 TONE 5
```

With program CALM keyed in to your HP-41, set flag 21 (SF 21), clear the X-register, and run the program (XEQ "CALM"). You should see the display: THIS IS A VERY SHORT ONE. But what happened to the song? And where is the 5 that should be in the display when the program stops?

Well, because flag 21 was <u>set</u>, the HP-41

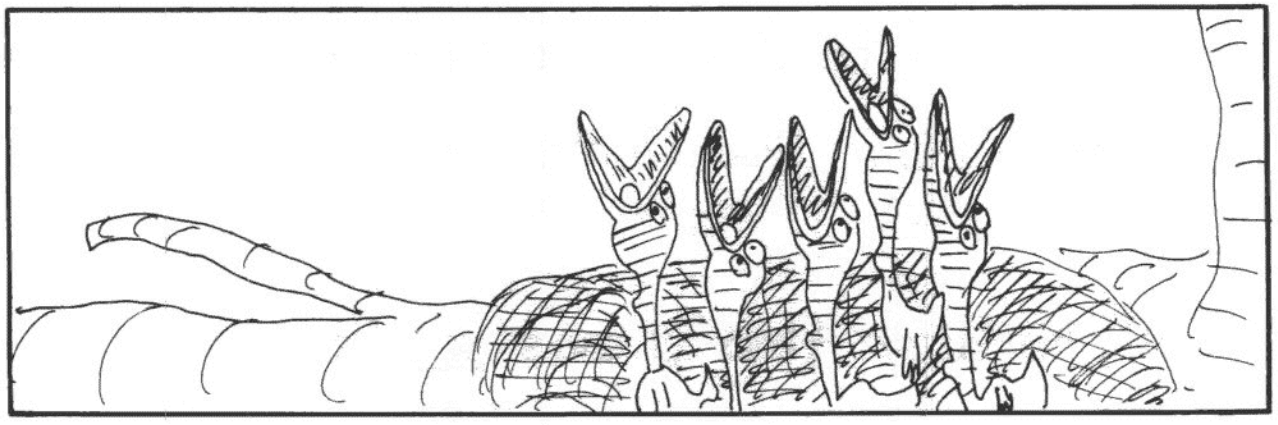
stopped when it encountered the AVIEW instruction at line 04. Put your HP-41 into program mode, and you'll see that it's waiting patiently at line 05.

Now, with flag 21 cleared (CF 21), run CALM. You'll see the ALPHA display: THIS IS A VERY SHORT ONE. Then you'll hear the tune, and then the program will finish with a 5 in the X-register.

To borrow a noble phrase and twist it for our purposes: To set or not to set flag 21? That is the question.

You can write programs so that their actions depend on the status of flag 21.

Incidentally, the VIEW function behaves the same as AVIEW in its dependence on flag 21.



Notes

INDIRECT ADDRESSING

So, what is all <u>this</u> stuff – "indirect addressing?" What's wrong with direct addressing (whatever <u>that</u> is)?

Well, here's how it works:

Picture a bashful young man who is going on a blind date. Of course, he called a dating service that caters especially to shy people. Therefore, he was not given the address of the young lady, because this is much too personal to divulge to just anyone. Instead, he was told the address of her brother's home. There (if he meets with the brother's approval, of course), he would be given her actual address.

The dating service has told him <u>indirectly</u> how to get to his date's home: He was told what address to go to in order to <u>obtain</u> her address.

Now, at the risk of removing a bit of romance, the instruction from the dating service is like a RCL IND instruction – like RCL IND 00, for example.

The brother's address is 00. (He "lives" in register 00.) The number in register 00 is the address of the young lady.

Question: But how do you key in RCL IND 01 ?

Answer: [RCL] [Shift] 01

Whenever you want to specify an <u>indirect</u> address, press the [Shift] key <u>before</u> the address.

Challenge: Set up your HP-41 so that the value in the X-register is 29.7 and the value in register 01 is 3. Now, if you execute the function:

STO IND 01 , what happens?

Answer: The computer will store the number 29.7 into register 03.

EASY? **OK →**

A little shaky? (That's O K - blind dates tend to be a bit shaky at first.)

Here's what happens:

STO IND 01 says: "STORE the value in the X-register into the register whose address is indicated by the contents of register 01."

So, the HP-41 takes a copy of the 29.7 that's sitting in the X-register and goes to look at register 01. There it finds the number 3. Then it knows where to put the 29.7- into register 03.

Let's try another...

Challenge: Starting with the results from the first try and using <u>indirect</u> addressing only, store the number 29.7 into register 00.

Solution: 0
RCL IND 01
STO IND Y (STO [.] Y)

NO PROBLEMS? →

PROBLEMS? WATCH:

First, we keyed a 0 into the X-register.

Then we performed RCL IND 01 which places 29.7 into the X-register (lifting the 0 into the Y-register). Here, the indirect addressing works exactly as it did in the first example – only it does a RCL instead of a STO.

Then comes STO IND Y. A copy of the contents of the X-register is stored into the register indicated by the Y-register. That indicated register is 00.

GETTING THE HANG OF IT?

Also, keep in mind that indirect addressing works for other functions besides STO and RCL.

In fact, it works for all the functions listed on pages 197 and 198 of the Owner's Handbook.

For example, suppose you have a program with 26 sections (each with a numeric label and a RTN statement) – one for each letter of the alphabet. Each section is supposed to do some calculation about how much space and ink each letter requires on a certain type of printer. The program might look like this:

```
01 LBL "LETTERS
   02 LBL 01
      :
      RTN
      LBL 02
      :
      RTN
      LBL 03
      :
      :
      LBL 25
      :
      RTN
      LBL 26
      :
      END
```

Each of these sections will do a different calculation, but for each input (each letter), there's only one section that applies.

Question: What's an easy way to tell the HP-41 to choose the proper section for a given letter?

Answer: Key in the number of the letter you want (1 for A, 2 for B, ..., 26 for Z) and then use:

XEQ IND X.

UNDERSTAND ? →

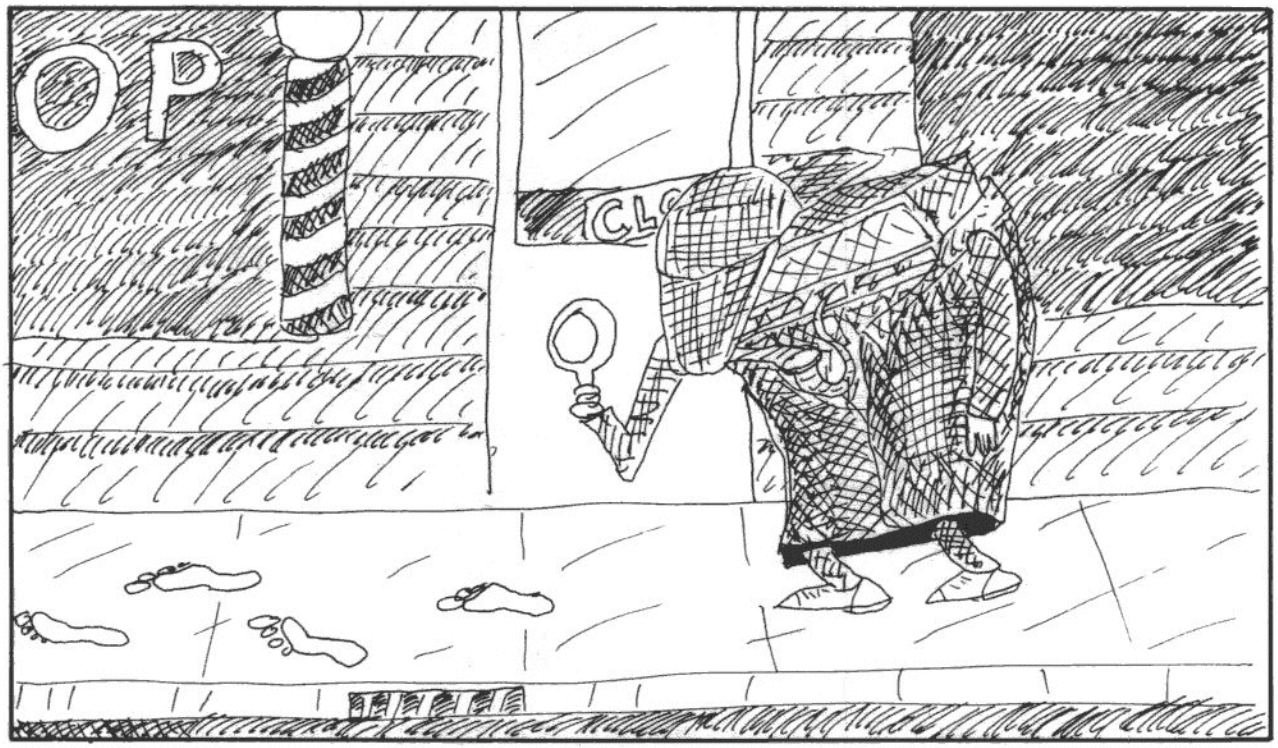
No? Watch what happens:

Let's say you want to calculate all this good stuff for the letter K.

You would key in the number 11 (K is the eleventh letter, right?).

Now, when you use XEQ IND X this tells the HP-41 to "execute the section whose label is indicated in the X-register." In the X-register, it finds the number 11, so it executes the section with LBL 11.

Remember that the numeric labels in the last example are <u>local</u> labels, so the program pointer would have to be positioned to the program LETTERS before the HP-41 could find them.



Also, you can execute ALPHA labels indirectly by ASTOring the characters of the label in a data or stack register and then using XEQ IND, with that register as the argument. The ALPHA labels here are limited to a maximum of six characters because that's all you can store in a register.

(Unfortunately, there is no XEQ IND ALPHA function, so you <u>must</u> store the label in a register first.)

As a good final example, here's a ...

Challenge: Write a short program (10 steps or less) that stores the integers 0 through 19 in registers 0 through 19, respectively.

Solution: Here's one way to do it:

```
01 LBL"FILL        06 STO IND X
02 .0 19           07 RDN
03 LBL 06          08 ISG X
04 ENTER           09 GTO 06
05 INT             10 END
```

If you have the SIZE set below 020 data registers, you'll get a NONEXISTENT if you run this program, because you will not have enough data registers. Change line 02 or change the SIZE, and it should work.

If you were able to write this program without looking at the solution, go to page 173.

If it was a struggle, go to the next page ⟶

01 LBL 'FILL
02 .019

First, after the label, we put the number .019 into the X-register. This is going to be the index number for the ISG loop, and it's also going to act as an index for indirect storage. (Remember, .019 is the same as 0.019 and, as an ISG index, it also acts the same as 0.019<u>01</u>.)

---

03 LBL 06

Now we put in a local label at the head of the repeating loop. Notice that everytime the label is reached, the index number must be in the X-register.

---

04 ENTER

Inside the loop, the first thing to do is to make a copy of the index number in the Y-register.

---

05 INT

Next, we extract the integer portion of the index number in the X-register. For example, if that number

is 0.019, the INT function produces the number 0 in the X-register.

---

06 STO IND X

Here's the elegant part: STO IND X says "store a copy of the value in the X-register into the register indicated by the X-register." If that number is 0 then 0 gets stored in register 00. If it's 19, it gets stored in register 19. So the very number you want to store is also the one that tells the computer where to store it!

Bonus Question: What would happen if we used STO IND Y at line 06?

Bonus Answer: The program would still do exactly the same thing. The X-register contains a copy of the integer portion of what's in the Y-register. But indirect addressing considers only the integer portion, anyway (and the positive version, at that). As an indirect address, $5 = (-5) = (-5.922) = 5.013$.

Now to wrap things up:

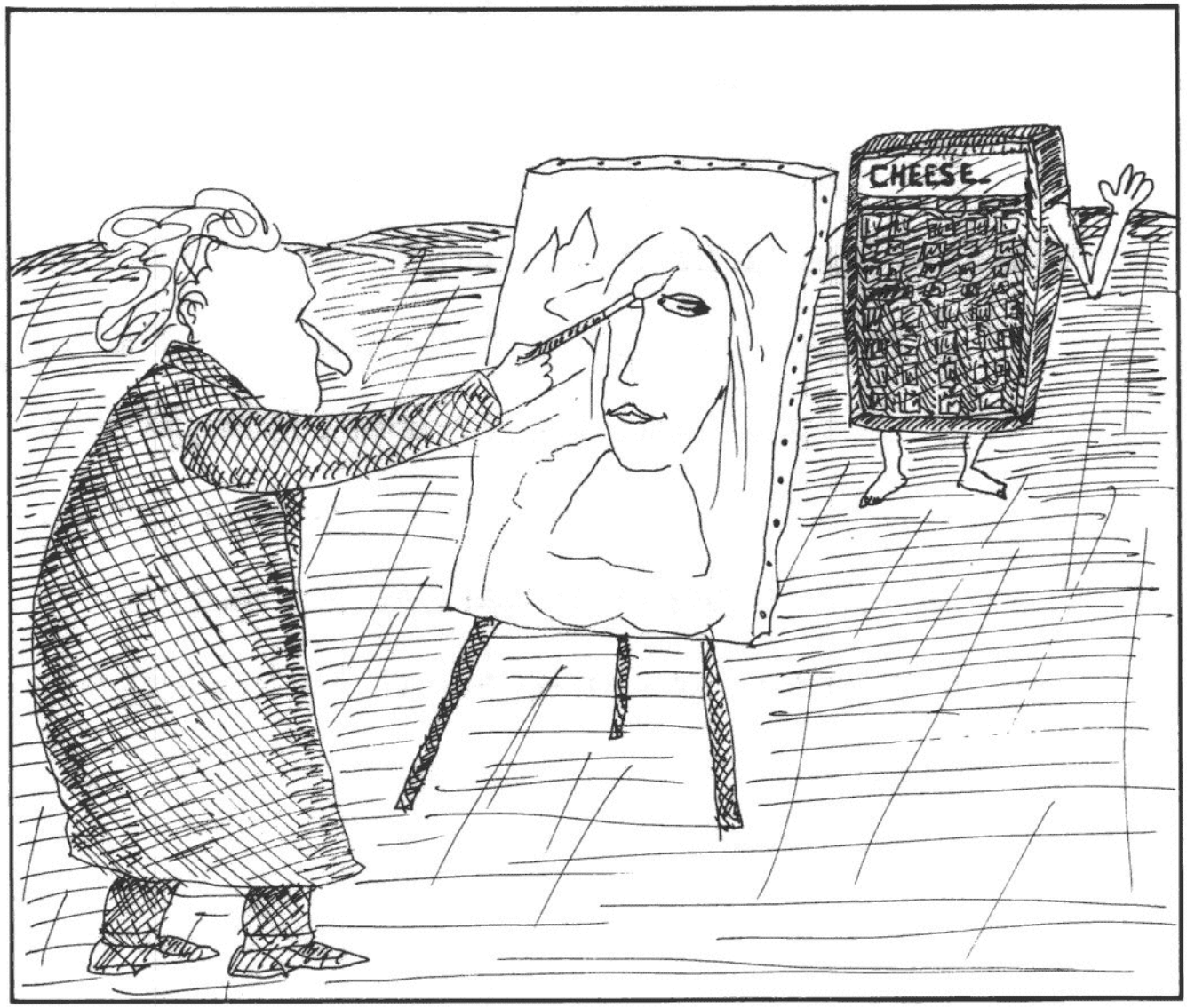
```
07 RDN .
```

We bring the unmolested version of the index back down from the Y-register to the X-register.

```
08 ISG X
09 GTO 06
10 END
```

Now we increment this index and, as you remember from the "Decisionmaking in Programs" section, the new integer portion of the index is compared to the first three digits of the fractional portion. If the integer portion is greater, then GTO 06 is skipped, and we're done. But, if the integer portion is less, then there's no skipping, and GTO 06 sends the program pointer back to LBL 06, with the newly-incremented index sitting in the X-register.

PROGRAM DEVELOPMENT



At this point in the easy programming course, you know the following:

1. You know how to execute any function given the name of that function. You are no longer confused by the "name" of a function and the "argument" of a function (page 45).

2. You know how to picture in your mind the memory structure (data registers, program memory, stack registers, display, ALPHA-register) of the HP-41 (page 8).

3. You know how the stack works (pages 58-80).

4. You know the difference between keying ALPHA data into the ALPHA-register and keying a function name into the display. Both of these things are done in ALPHA mode, but they are completely different (pages 25, 56, 144-145).

5. You know that when we're discussing programming and we use quote marks in a keystroke listing, those quote marks mean: press the [ALPHA] key.

6. You know <u>when</u> your computer is in RUN mode, and <u>when</u> it's in program mode. You know how to get from one mode to the other (page 88).

7. You know (with a little contemplation) how to go from an <u>equation</u> to a sequence of <u>keystrokes</u> to a simple, linear (one time through) <u>program</u>. You know that you can use a program like this to run great hordes of data through an equation by simply entering the data (as input) and pressing the [R/S] key (page 93).

8. You know about ALPHA prompts and AVIEW and using ALPHA data in a program (pages 147-159).

9. You know that certain functions are "yes or no" questions (like X=0?) and that if the answer to these questions is <u>no</u>, the following line in the program will <u>not</u> be executed (page 121).

10. You know that you can use these conditional test functions as a basis for making decisions in a program (pages 122-125).

11. You know about program branching. You know when to use a GTO statement and when to use an XEQ statement. You know when to use a global label and when to use a local label (pages 102 to 116).

12. You know what a "loop" is in a program and how to conditionally branch out of a loop (pages 135 to 141).

13. You know how to move around in program memory. You know how to use the CAT 1 function to position the program pointer to any global label or END in program memory. You know what RTN does in RUN mode. You know how to use the GTO [.] function and you know what GTO [.][.] does (pages 33, 106, 97-100).

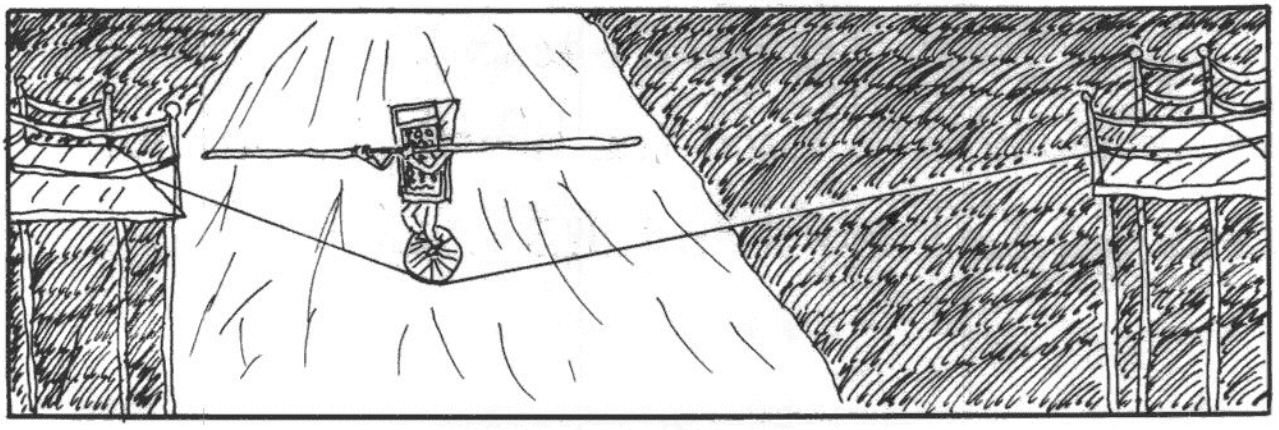
14. You have a good feeling for what flags are and how to use them (page 126 to 131).

15. You know how to turn the HP-41 off without pressing the [ON] key (Hint: [XEQ] "OFF")

16. You know that nowhere in the balcony scene does Juliet wonder <u>where</u> Romeo is (Act II, scene ii)

Well, anyway, you know all these things plus a few others. So it's time to delve into the purpose of this book... program development (You didn't know you were still reading the introduction, did you?).

Our first challenge will be a fairly common application. Checkbook balancing is something that most of us have had experience with (at least the <u>attempt</u> is familiar). So together, we can develop a fairly extensive checkbook-balancing program to incorporate many aspects of HP-41 programming.



Challenge: There are three major steps to balancing your checkbook. Write down these three steps.

Solution:

1. Find the balance from the last time you balanced your checkbook.
2. Add all the deposits and interest since that time. Record the balance resulting from each addition.
3. Subtract all checks and charges. Record the balance from each check or charge.

After performing the above three steps, your checkbook will be balanced. Basically, what we've done here is defined the process by which we balance our checkbooks, in terms that are easy for <u>us</u> to understand.

You could take this list of three checkbook balancing steps, hand it to one of your friends, and they could follow it with no problem.

In a sense, our checkbook-balancing program is complete. We understand the problem, and with these three steps, we've developed a solution, or program, to handle the problem.

WELL, THAT WAS EASY!

Question: What would happen if you were to explain a recipe to a small child (4 to 7 years old) in the following manner?

"Get a cup of flour, ⅓ cup shortening, ½ teaspoon salt, a tablespoon of sugar. Then cut the shortening into a mixture of the flour, salt, and sugar, until the chunks are about the size of peas. Next, add a couple tablespoons of cold water and blend with a fork (not too much). Shape the dough into a ball, and roll it out and VOILÁ! Pie crust (π)."

Answer: The child would be dumbfounded.

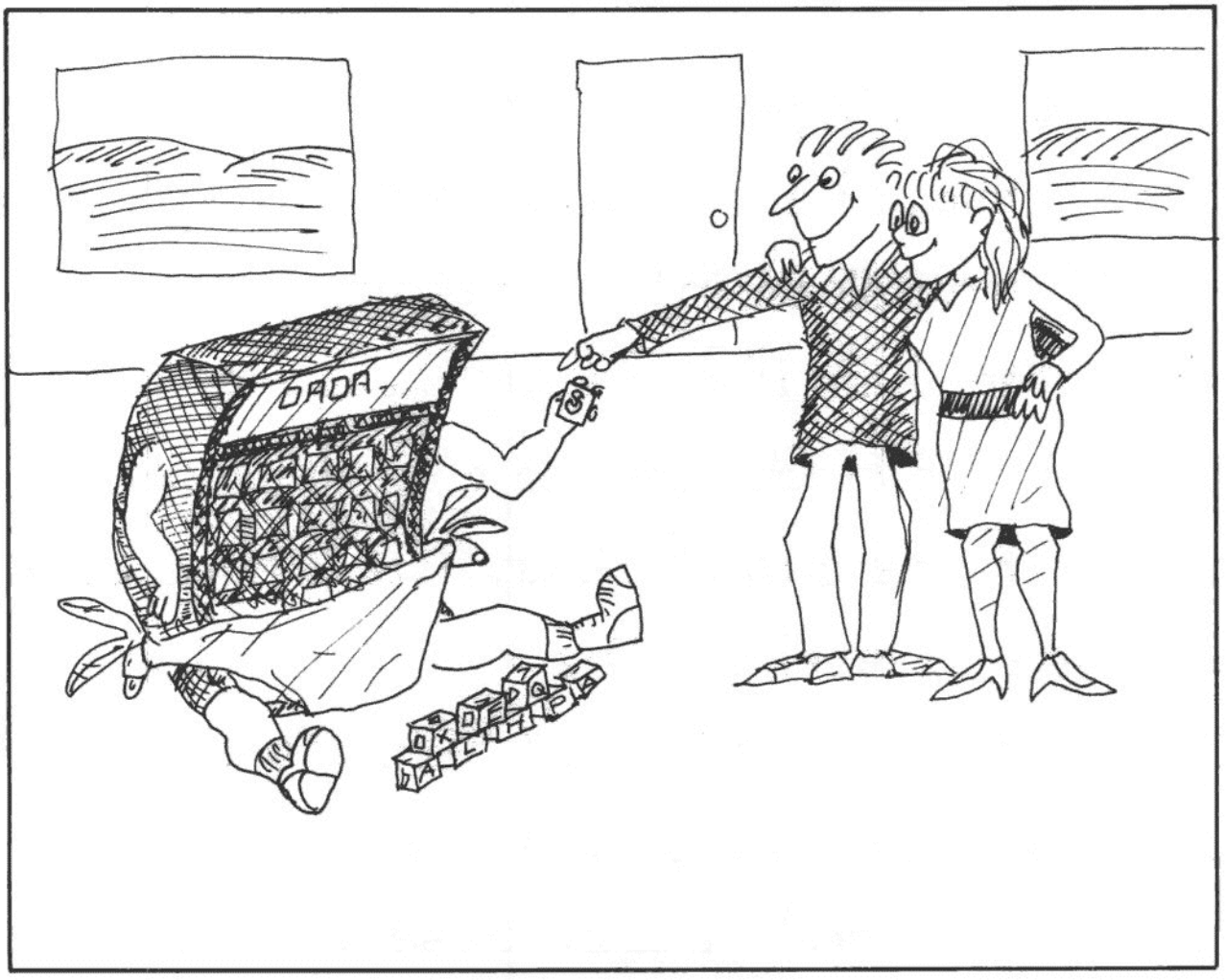
Nevertheless, this is the way we think. As we develop and gain experience in life, many of the details of day-to-day tasks become automatic - they require no conscious thought. Driving a car is a prime example of details becoming automatic (especially a car with a standard transmission).

It takes patience and effort to explain something to a small child. We actually have to slow down our thinking process and analyze each step.

If you were explaining our pie crust recipe to a small child, the first step of the recipe would probably translate into something like this: "Now listen, here is a one-cup measure (note the visual aid). Go over to the flour can and scoop out one cup of flour. You'll have to use this knife to level off the top so you have exactly one cup of flour." So you see, it takes more words and more effort to explain a process to a small child, because a child's

thinking process is less complex than ours.

The "thinking process" of the HP-41 (or of any computer) is far less complex than even that of a small child. In order to program the HP-41 to do a task, we first have to expand the steps by which <u>we</u> do things into many more steps, each much simpler. In other words, we have to translate our complex thinking process into simple terms that the HP-41 understands.



Challenge: Rewrite the three checkbook-balancing steps in terms that are closer to the way the HP-41 "thinks."

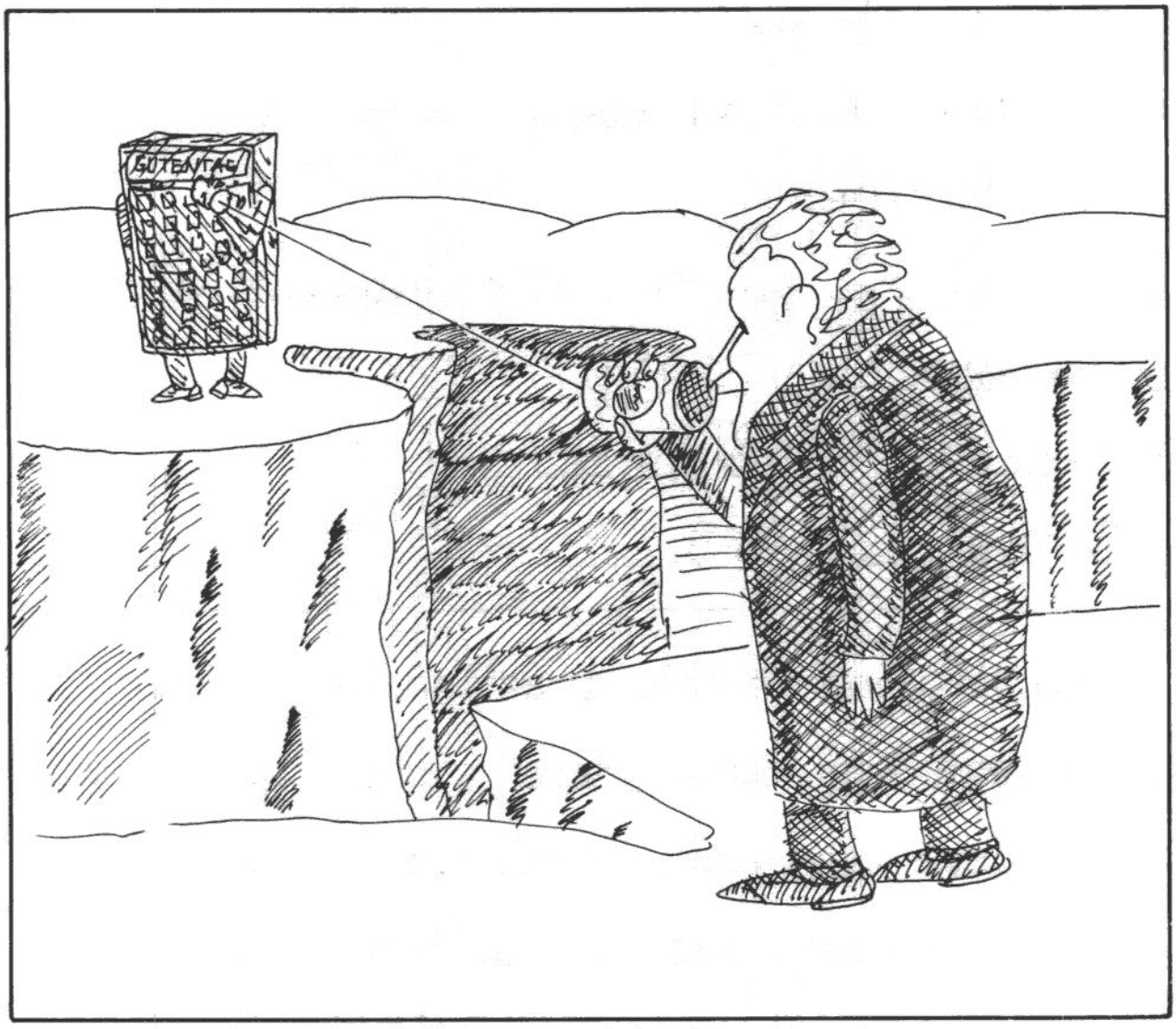
Solution:

1. Prompt for initial balance
2. Store initial balance in a numbered register.
3. Prompt for the input of either a check, charge, deposit, or interest.
4. If the input is a check or charge, subtract it from the balance.
5. If the input is a deposit or interest, add it to the balance.
6. Display the new balance.
7. Go back to step 3.

Your solution to this question may vary considerably from ours. Here is where you start to develop your own programming technique. Everyone will approach a solution in a slightly different manner. However, for this first program,

it may be easiest for you to follow fairly closely to our development technique.

You can see that what was three general steps has evolved into <u>seven</u> more detailed steps. Each of these seven steps carries a simpler concept than each of the original three. And each of these seven steps "sounds closer" to the language of the HP-41.



Challenge: Equipped with the previous seven steps, develop an HP-41 program to balance your checkbook.

Solution:

```
01 LBL"CHKBK        11 CHS
02 FIX 2            12 LBL D
03 SF 27            13 ST+ 00
04 "INITIAL BAL?    14 "BAL=$
05 PROMPT           15 ARCL 00
06 STO 00           16 PROMPT
07 LBL 01           17 GTO 01
08 "AMNT?C,D,E      18 LBL E
09 PROMPT           19 RCL 00
10 LBL C            20 CF 27
                    21 END
```

If this make sense to you, head to page 197.

If you're even slightly confused, that's good. A completely unexplained list of HP-41 code should be confusing.

What follows is an explanation of the thought process that is required to go from the seven

steps in English on page 182 to the 21 lines of HP-41 code that appear on page 184.

First, don't expect to start at the top of the list. The order of the steps will reflect the order that things are done in the completed program. But that doesn't mean you're going to start developing the program at step 1.

The first thing to do is to search through the list for the steps that are most significant to the program. Basically, what you're looking for are the steps that look like they will require the most work. You will develop these steps first and then design the rest of the program around them.

In our list of seven steps, steps 4 and 5 are the only steps that will require some type of calculation and some type of decisionmaking:

4. If the input is a check or charge, subtract it from the balance.

5. If the input is a deposit or interest, add it to the balance.

Looking at steps 4 and 5, you can see that the HP-41 will have to treat an input in one of two ways, depending on whether it is a check or a deposit. Now, it boils down to this: one way or another, you are going to have to tell the HP-41 whether you are inputting a check or a deposit.

There are many ways to tell this to the HP-41. We feel the easiest way is to have one key to press for a check or charge and another key for a deposit or interest.

We notice (by observing some of the programs in the Standard Applications book) that it is possible, by using local-ALPHA labels and USER mode, to design our program so that one key (say the [C] key) can mean a check or charge, and another key (the [D] key) can mean a deposit or interest.

So, when the program is complete, we want to be able to key in an amount, press the [C] key, and have the HP-41 treat that amount like a check

(that is, subtract that amount from our balance). Likewise, we want to be able to key in an amount, press the [D] key, and have the HP-41 treat that amount like a deposit.

The only difference between the way a check is treated and the way a deposit is treated is that a check is subtracted from the balance, while a deposit is added to the balance. Other than that, the program should treat a check the same as a deposit.

Now everyone remembers that adding the negative of a number to something is just like subtracting that number. That is,

$$a-b=a+(-b)=(-b)+a=-b+a\,.$$

So, when we press the [C] key, if the HP-41 just puts a negative sign on the amount we keyed in, then treats it like a deposit, that should do the trick!

With all this in mind, we can sketch out a routine to handle steps 4 and 5 of our list:

```
LBL C
CHS
LBL D
RCL BALANCE
+
STO NEW BALANCE
```

Of course, there are no "RCL BALANCE" or "STO NEW BALANCE" functions on the HP-41. But we haven't designated a storage register for keeping the balance yet. By using this method, we can organize our thoughts in a language that is <u>close</u> to what the HP-41 uses but is still easily understood by us.

The six lines above will just about take care of steps 4 and 5 of our list. Let's put those steps on the back burner for a second, and look at the others. →

Challenge: Translate step 1 (prompt for initial balance) into HP-41'eze.

Solution: 'INITIAL BAL?
PROMPT

That was easy. The line 'INITIAL BAL? will place those ALPHA characters into the ALPHA-register. Then, as you know, PROMPT will halt program execution after filling the display with the message in the ALPHA-register.

Step 2 (store initial balance) is also an easy one to translate. It just requires that you pick a number (we chose 00) and use that register. So step 2 becomes: STO 00.

Step 3 translates much like step 1. All we need to do is think of an effective message (preferably less than 12 characters, so the display doesn't <u>scroll</u>), and put a PROMPT after it. So step 3 becomes:

'AMNT? C,D
PROMPT

The message that we chose reminds the user that an amount needs to be keyed in and that either the [C] key (check) or the [D] key (deposit) should be pressed.

Challenge: Translate step 6 into HP-41 code.

Solution:

```
"BAL = $
ARCL 00
PROMPT
```

The first line is just a text message. The second line appends the number in register 00 to the contents of the ALPHA-register. And, as you know, the PROMPT causes program execution to halt with the contents of the ALPHA-register in the display.

Challenge: Go back and look at the routine we sketched out to handle steps 4 and 5 of our list. Rewrite these six lines into HP-41'eze. (Remember, register 00 contains the balance.)

Solution:

```
LBL C
CHS
LBL D
RCL 00
+
STO 00
```

or

```
LBL C
CHS
LBL D
ST+ 00
```

Since we are concerned that an updated balance be maintained only in register 00, we can use the ST+ (store-plus) function to add the deposit (or the negative amount of the check) to register 00.

The sequence RCL 00; +; STO 00 would do almost the same thing, except it would leave the updated balance in the X-register as well.

Up to now, the program looks like this:

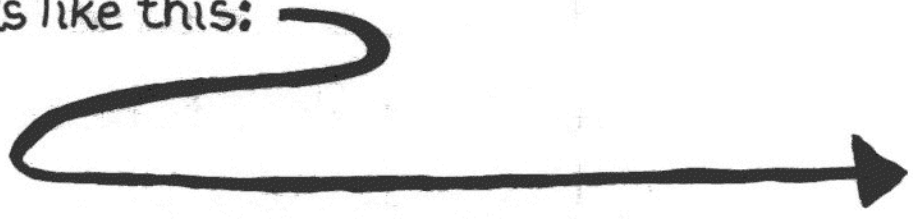
1. {"INITIAL BAL?
   {PROMPT
2. {STO 00
3. {"AMNT? C,D
   {PROMPT

4 and 5 {LBL C
        {CHS
        {LBL D
        {ST+ 00

6. {"BAL=$
   {ARCL 00
   {PROMPT

7. Go back to step 3.

We've translated six of the seven steps into HP-41 code. The seventh step is obviously going to be a GTO statement. But first we have to insert a LBL at the top of step 3, so the final routine looks like this:



```
01 LBL "CHKBK        10 LBL D
02 "INITIAL BAL?     11 ST+ 00
03 PROMPT            12 "BAL=$
04 STO 00            13 ARCL 00
05 LBL 01            14 PROMPT
06 "AMNT? C,D        15 GTO 01
07 PROMPT            16 END
08 LBL C
09 CHS
```

So we've developed our first complete program, right? Well, not really. Once we've finished coding a problem for the HP-41, we should take a look at it to see if there's anything that we have assumed about the status of the computer (flags, display status, etc.)- things we should establish at the beginning of the program.

In other words, we have to <u>initialize</u> the status of the HP-41 so that the program will always run correctly. If we use some flags in the program, and we need those flags to be clear initially, we should have the program clear them.

In the CHKBK program, we use local ALPHA-labels. So, when the computer <u>is</u> in USER mode and the program pointer is positioned to this program, those labels will be assigned to their respective keys. The [C] key becomes XEQ C, for example. Therefore, if the computer is <u>not</u> in USER mode, the program will not work. So we initialize USER mode by inserting a SF 27 after the first line of the program. "Set flag 27" means "turn on USER mode." Now the program will turn on USER mode.

```
01 LBL "CHKBK
02 SF 27
03 "INITIAL BAL?
04 PROMPT
05 STO 00
06 LBL 01
07 "AMNT? C,D
08 PROMPT
09 LBL C
10 CHS
11 LBL D
12 ST+ 00
13 "BAL=$
14 ARCL 00
15 PROMPT
16 GTO 01
17 END
```

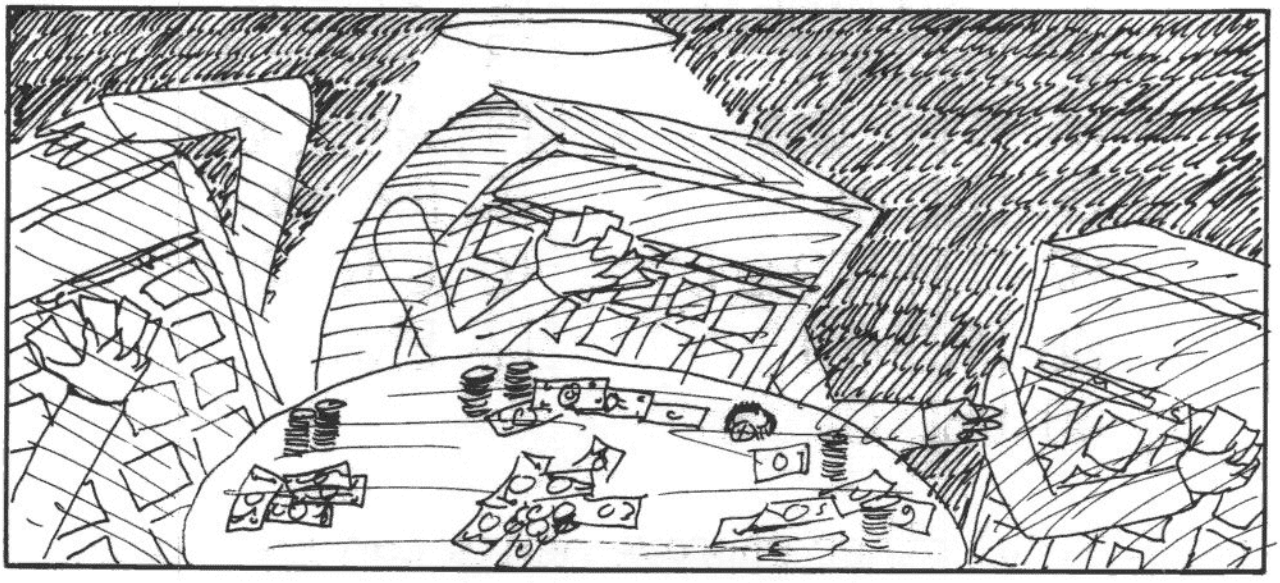
Now, what other improvements can we make?

How about this?

Since we're dealing with dollars and cents, the program should set the display to show only two decimal places.

Also, besides the check ([C]) key and the deposit ([D]) key, it would be nice if there was an exit ([E]) key by means of a local ALPHA-label. Then, when you're all done balancing your checking book, you could press the [E] key to RCL the final balance to the X-register and send the program pointer to the END of the program.

Challenge: Make the above improvements to the program.



Solution:

```
   01 LBL "CHKBK
 * 02 FIX 2
   03 SF 27
   04 "INITIAL BAL?
   05 PROMPT
   06 STO 00
   07 LBL 01
   08 "AMNT? C,D,E
   09 PROMPT
   10 LBL C
   11 CHS
   12 LBL D
   13 ST+ 00
   14 "BAL=$
   15 ARCL 00
   16 PROMPT
   17 GTO 01
** 18 LBL E
   19 RCL 00
   20 CF 27
   21 END
```

* This sets the display to show two decimal places.

** LBL E just recalls the final balance into the X-register and turns off USER mode. (We forgot to mention turning off USER mode.) When you're all done balancing your checkbook, press the [E] key.

Challenge: Let's say that every check you write carries a $0.25 charge. Modify the program to take care of this charge.

Solution:

```
01 LBL "CHKBK
02 FIX 2
03 SF 27
04 "INITIAL BAL?
05 PROMPT
06 STO 00
07 LBL 01
08 "AMNT? C,D,E
09 PROMPT
10 LBL C
* 11 .25
* 12 +
13 CHS
14 LBL D
15 ST+ 00
16 "BAL=$
17 ARCL 00
18 PROMPT
19 GTO 01
20 LBL E
21 RCL 00
22 CF 27
23 END
```

* Lines 11 and 12 make up the modification. There's a problem with this, however. Now the [C] key can handle only checks. If your bank charges you, say, $3.00 a month, plus 25 cents for each check, there's no straightforward way to deduct the $3.00 charge. If you key in 3 and press [C], the

program will deduct $3.25 from your balance.

To remedy this problem, we can just insert a LBL c (little "c") after line 12. Then for checks, you can press the [C] key, and for straight charges, you can press [Shift] [C].

So the complete program looks like this:

```
01 LBL "CHKBK
02 FIX 2
03 SF 27
04 "INITIAL BAL?
05 PROMPT
06 STO 00
07 LBL 01
08 "AMNT? C,D,E
09 PROMPT
10 LBL C
11 .25
12 +
13 LBL c
14 CHS
15 LBL D
16 ST+ 00
17 "BAL=$
18 ARCL 00
19 PROMPT
20 GTO 01
21 LBL E
22 RCL 00
23 CF 27
24 END
```

The next program we will develop is a program to convert feet, inches, and sixteenths of an inch into feet and decimal fractions of feet, and vice versa. For example, 1 foot 6 3/16 inches will be converted to 1.515625 feet, and 1.515625 feet will be converted to 1 foot 6 3/16 inches.

The calculations involved in this program will be pretty simple. The main emphasis of this program will be the <u>format</u> of the <u>input</u> and <u>output</u>. We have to develop a reasonable way to input feet, inches, and sixteenths of an inch!

Now, it would be best if we could key in one number to represent all three units (feet, inches, 16THS). Let's develop a format for input of three different things with one number. ⟶

Challenge: Convert 3 hours, 26 minutes, and 14 seconds into hours and decimal fractions of hours.

Solution: 3.2614 XEQ "HR" displays 3.43722

So, 3 hours, 26 minutes, and 14 seconds is equal to 3.43722 hours. The point here is that for certain functions on the HP-41, one number can represent different things. The HR (HOUR) function looks at the number in the X-register as meaning hours, minutes, and seconds in the form HH.MMSS. The HH means "number of hours," the MM means "number of minutes," and the SS means "number of seconds." There are only two places reserved for minutes, because the number of minutes will never exceed 60 (that's an hour), and fractions of minutes are expressed in seconds. (However, there are actually more than two places reserved for seconds. If you want to key in 57 3/4 seconds, you would key in .005775, because 3/4 = 0.75. The fraction of a second is keyed in after the whole seconds.)

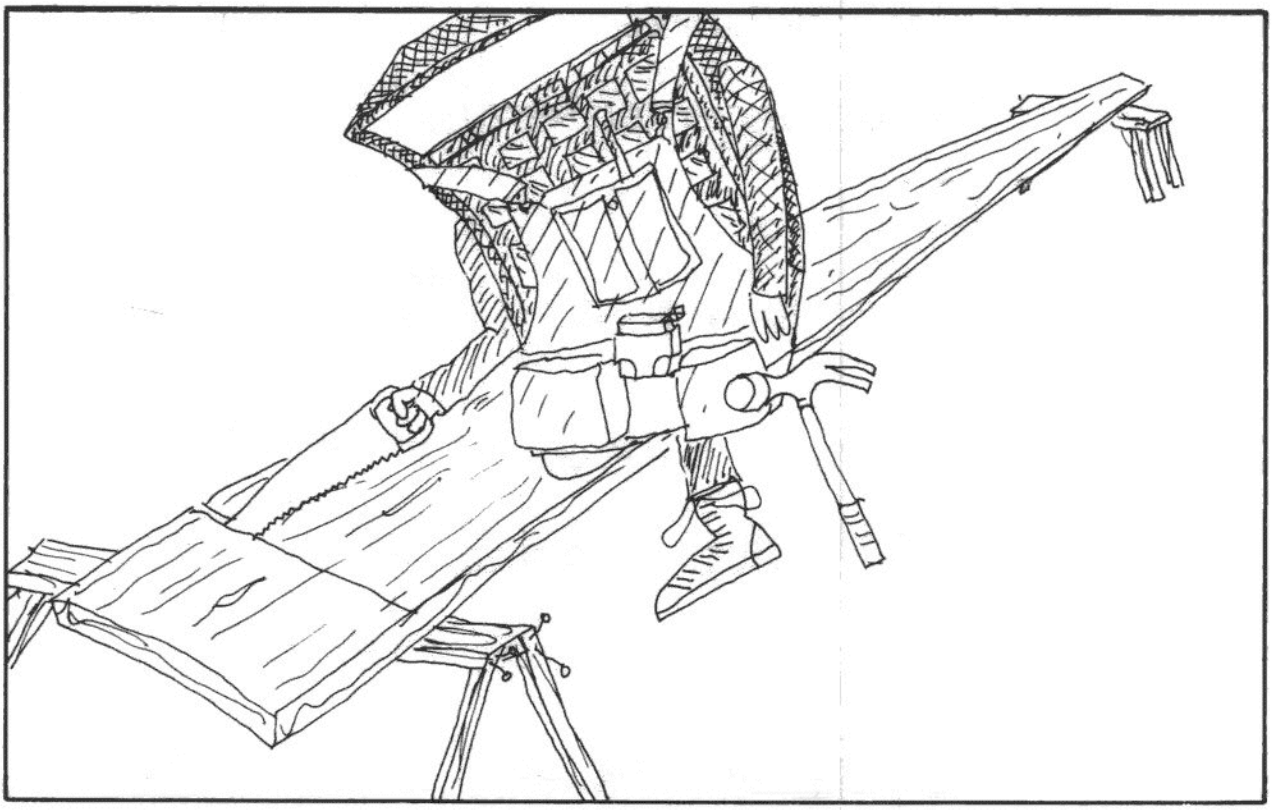
In the last challenge, one number represents hours, minutes, and seconds. In the program we're developing, we want one number to represent feet, inches, and sixteenths of an inch.

Challenge: Express 4 feet, 8 9/16 inches in a format similar to the HH.MMSS format.

Solution: 4.0809 (FF.IISS)

This is the input format that we'll use in our conversion program. We will key in feet, inches, and sixteenths of an inch, using the form FF.IISS. Here, FF means "number of feet," II means "number of inches," and SS means "number of sixteenths." Two places each are reserved for the inches and the sixteenths. This makes good sense, because there will be, at most, 11 and 15/16 inches (.1115) in any fraction of a foot.

Challenge: Sketch down the general steps required to convert an input of the form FF.IISS (feet, inches, sixteenths) to feet and decimal fraction of a foot (we'll represent the output format by FF.ffff....)



Solution:

"Visual Aid"

1. Get the input

FF.IISS

2. Take the integer portion and save it. This is the number of whole feet. Also, save the fractional portion.

FF | .IISS

3. Multiply the fractional portion by 100.

FF | II.SS

4. The integer portion of that result is the number of whole inches, and the fractional portion is the number of sixteenths divided by 100.

FF | II | .SS

5. Divide the fractional part by 1.92

FF | II | $\frac{.SS}{1.92} = \frac{SS}{192}$

6. Divide the number of inches by 12, and add up all the parts.

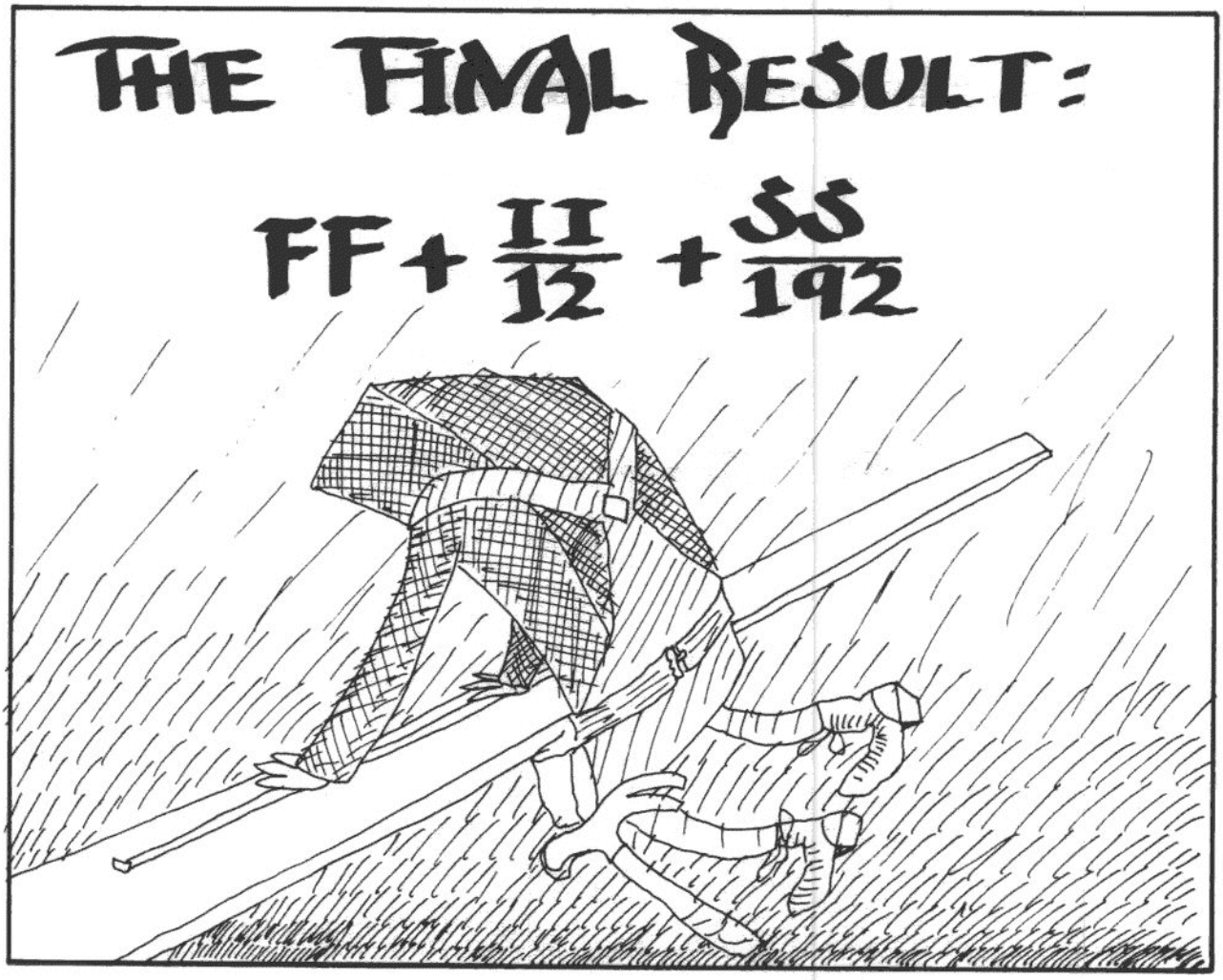
FF | $\frac{II}{12}$ | $\frac{SS}{192}$

7. The result equals the sum $FF + \frac{II}{12} + \frac{SS}{192}$

What we're shooting for in the solution is this:

First, we want to save the FF part of the input because this is the number of whole feet, and we don't need to change this in the final answer.

Next, we need to get the number of inches (II) and divide this by 12 (there are 12 inches in a foot). Then we have to get the number of sixteenths and divide it by 192 (12*16), since there's 192 sixteenths-of-an-inch in a foot.



Challenge: Sketch down the steps required to take an input of feet and decimal fractions of feet (FF.fffff...) and convert it to feet, inches, and sixteenths of an inch.

Solution:

1. Get input of FF.ffff....
2. Save the integer portion, FF.
3. Multiply the fractional portion by 12 and save the integer portion of the result, II.
4. Multiply the remaining fraction by 16, SS. (Let's leave fractions of sixteenths just like fractions of seconds.)
5. Save the result in the X-register in the form FF.IISS ($FF + \frac{II}{100} + \frac{SS}{10,000}$).

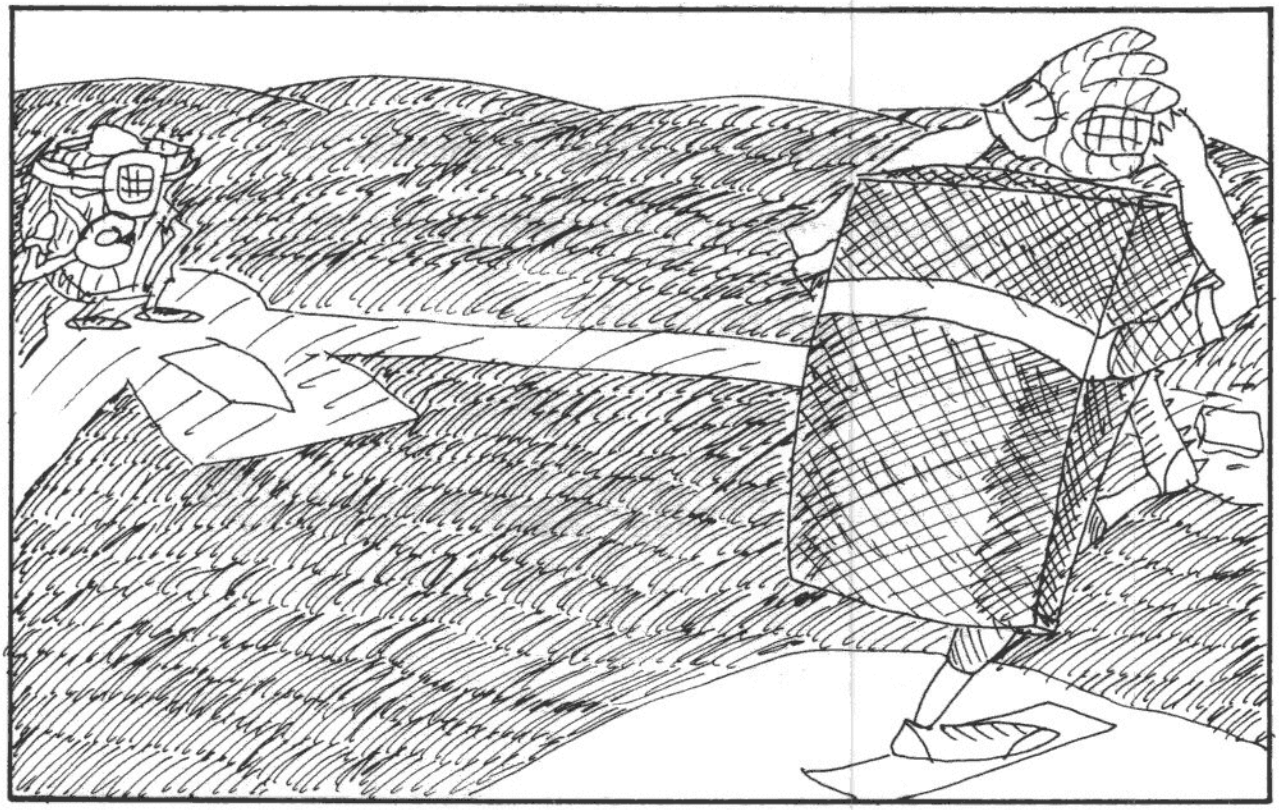
Let's run an arbitrary input through these steps to see if they work. Try, for example, 14.9 feet. First, save the integer portion, 14. Then, multiply .9 by 12 to get 10.8 and, again, save the integer portion. So far we have 14 feet, 10 inches, and 8 tenths of an inch. To convert the 8 tenths to sixteenths, multiply the .8 by 16. This gives 12.80 so the final result is 14feet, 10 $^{12.80}/_{16}$ inches. So our general idea seems to work.

From the way things are working out, it looks like we are going to develop two independent programs. One of these programs will take an input of the form FF.IISS and return feet and decimal fractions of feet. The other program will take an input in the form FF.ffff··· and return it in the form FF.IISS. These two programs will be a matched pair of functions much like the HR (hour) and HMS (hours, minutes, seconds) functions.

Let's call this pair of functions FT and FIS.

1. LBL ᵀFT
2. Get input of FF.IISS
3. Save the integer portion: [FF].
4. Multiply the fractional portion by 100: [II.SS].
5. Save the integer portion: [II].
6. Divide the fractional portion by 1.92: $\left[\frac{SS}{192}\right]$.
7. Divide the inches by 12: $\left[\frac{II}{12}\right]$.
8. Result = $FF + \frac{II}{12} + \frac{SS}{192}$
9. END of FT
10. LBL ᵀFIS
11. Get input of FF.ffff....
12. Save the whole feet: [FF]
13. Multiply the fractional portion by 12: [II]
14. Multiply the remaining fraction by 16: [SS]
15. Result = $FF + \frac{II}{100} + \frac{SS}{10,000}$
16. END of FIS

Challenge: At this point, you should be able to take the list of sixteen steps from the previous page and convert them to HP-41 program lines. Give it a try and see if you can come up with a workable program. It's possible to do all the calculations using only the stack registers.



Solution:

```
01 LBL"FT   - Assume the X-register contains FF.IISS
02 INT      - Separate the whole feet
03 LASTX    - Get back FF.IISS
04 FRC      - Take .IISS
05 1 E2     -)
06 *        -) Multiply by 100: II.SS
07 INT      - Separate II
08 LASTX    - Get back II.SS
09 FRC      - Separate .SS
10 1.92     -)
11 /        -) .SS/1.92 = SS/192 : Convert sixteenths to decimal.
12 X<>Y     - Get II
13 12       -)
14 /        -) II/12 : Convert inches to decimal
15 +        -)
16 +        -) Result = FF + II/12 + SS/192
17 RTN      - END of FT routine
18 LBL"FIS  - Assume X-register contains FF.ffff...
19 INT      - Separate whole feet: FF
20 LASTX    - Get back FF.fff...
21 FRC      - Separate fractional feet: .ffff...
22 12       -)
23 *        -) Multiply by 12
24 INT      - Separate the number of inches: II
25 LASTX    -)
26 FRC      -) Get remaining fraction
27 16       -)
28 *        -) Multiply by 16: SS
29 1 E2     -)
30 /        -)
31 +        -)
32 1 E2     -) Result = FF + II/100 + SS/10,000 = FF + (II + SS/100)/100
33 /        -)
34 +        -)
35 END      - END of FIS
```

The program we've developed here is the "no frills" version. There are no ALPHA prompts or displays, no beeps or tones. For conversion routines, the "no frills" version is usually the best way to go.

To convert, say, 13 feet 9 4/16 inches to feet and decimal fractions of feet, all you have to do is key in 13.0904 XEQ "FT". The result comes back in the X-register.

The RTN statement at line 17 just means "stop" (i.e., return to the keyboard), in this case. But, using this RTN instead of a STOP allows you to call this routine as a subroutine from other programs.

To convert 12.77673 feet to feet, inches, and sixteenths of an inch, just key in 12.77673 XEQ "FIS". The END at line 35 will cause program execution to halt in this case, but it will act as a RTN if the routine is called as a subroutine.

By the way, you would have to have an extremely accurate ruler to determine that something was exactly 12.77673 feet long!

We could snaz up this routine a bit by adding some ALPHA messages and making use of the numeric entry flag (flag 22) for repetitive entries. The snazzed-up version could look something like this:

```
01 LBL "FT
02 "INPUT FF.IISS
03 PROMPT
04 CF 22
05 LBL 02
06 FIX 4
07 INT
08 LASTX
09 FRC
10 1E2 (EEX 2)
11 *
12 INT
13 LASTX
14 FRC
15 1.92
16 /
17 X<>Y
18 12
19 /
20 +
21 +
22 CLA
23 ARCL X
24 "⊢FEET
25 AVIEW
26 RTN
27 FS?C 22
28 GTO 02
29 LBL "FIS
30 FIX 0
31 CF 29
32 "INPUT FEET
33 PROMPT
34 CF22
35 LBL 03
36 INT
37 LASTX
38 FRC
39 12
40 *
41 INT
42 LASTX
43 FRC
44 16
45 *
46 CLA
47 ARCL Z
48 "⊢FT
49 ARCL Y
50 "⊢+
51 ARCL X
52 "⊢/16 IN
53 1E2
54 /
55 +
56 1E2
57 /
58 +
59 AVIEW
60 RTN
61 FS?C 22
62 GTO 03
63 END
```

That CF 29 at line 31 clears away the decimal point in the FIX 0 format. (CF 29 also suppresses commas that divide up big numbers.)

Don't forget the ⊢ signs at lines 24, 48, 50, and 52.

As you can see, the snazzed-up version of the program is considerably longer. Whether this version is easier to use depends on several things. If you only use a program once in a blue moon, then it's best to keep the longer, "friendlier" version around. But if you use a program frequently, then it's best to streamline the program so that it's just a matter of keying in a number or two and executing the right label. ALPHA prompts and elaborate output methods in short, frequently used routines just become redundant and inconvenient. (We've marked these additions with brackets.)

A neat feature of the snazzed-up version that doesn't appear in the original routine is the capability for repeating the same routine by keying in a new input and pressing [R/S]. For example, let's say we have six entries of feet, inches, and sixteenths that we want to convert to feet. ⟶

(1.) 4 feet 11 5/16 inches
(2.) 5 feet 2 inches
(3.) 9 feet 4 1/2 inches
(4.) 11 feet 11 13/16 inches
(5.) 9 3/4 inches
(6.) 7 feet 7/8 inches

The keystroke solution using the first routine would look something like this:

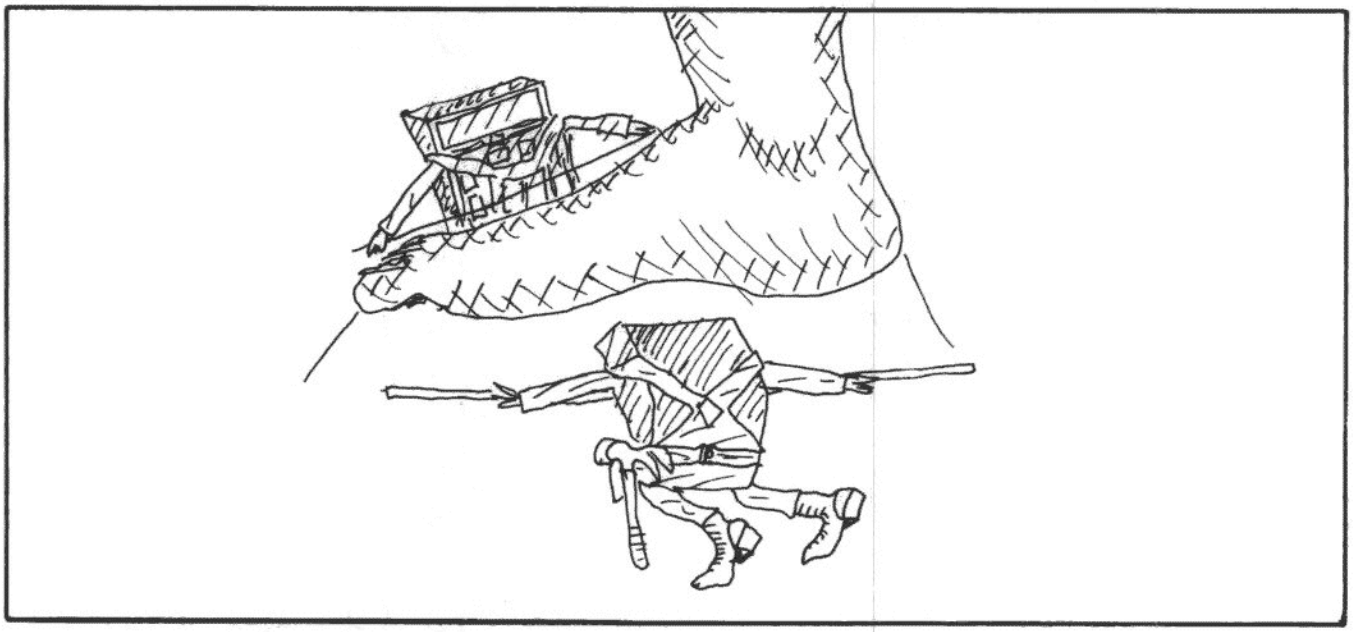
KEYSTROKES	DISPLAY
4.1105 [XEQ] "FT"	4.9427
5.02 [XEQ] "FT"	5.1667
9.0408 [XEQ] "FT"	9.3750
11.1113 [XEQ] "FT"	11.9844
.0912 [XEQ] "FT"	0.8125
7.0014 [XEQ] "FT"	7.0729

Whereas, using the second routine, the keystroke solution would look like this:

[XEQ] "FT"	INPUT FF.IISS
4.1105 [R/S]	4.9427 FEET
5.02 [R/S]	5.1667 FEET
9.0408 [R/S]	9.3750 FEET
11.1113 [R/S]	11.9844 FEET
.0912 [R/S]	0.8125 FEET
7.0014 [R/S]	7.0729 FEET

The second routine requires fewer keystrokes for repetitive conversions. When you key in a number, flag 22 is <u>set</u>. So when you <u>then</u> press [R/S], the GTO 02 is executed and the FT conversion is repeated.

Also, notice that if you <u>don't</u> key in a number before pressing [R/S], the previous result will be converted back into feet, inches, and sixteenths.



Challenge: Incorporate the feature of easy repetition from the steps on page 211 into the streamlined" version on page 209. You will have to make use of flag 22. The end result will be a quick, easy-to-use routine with no ALPHA prompts or displays.

Solution:

```
01 LBL'FT
02 LBL 01
03 INT
04 LASTX
05 FRC
06 1 E2
07 *
08 INT
09 LASTX
10 FRC
11 1.92
12 /
13 X<>Y
14 12
15 /
16 +
17 +
18 CF 22
19 RTN
20 FS?C 22
21 GTO 01
22 LBL'FIS
23 LBL 02
24 INT
25 LASTX
26 FRC
27 12
28 *
29 INT
30 LASTX
31 FRC
32 16
33 *
34 1 E2
35 /
36 +
37 1 E2
38 /
39 +
40 CF 22
41 RTN
42 FS?C 22
43 GTO 02
44 GTO 01
45 END
```

The final program that we will develop is a program to evaluate a table of X-and Y-values for an equation that you program into the HP-41.

Most of us have done this before in an old algebra class. Back in algebra, they called it "graphing" an equation or "plotting" an equation. We start with some equation like:

$$Y = X^2 + 5X.$$

Now these Y's and X's don't have anything to do with the X- and Y-registers on the HP-41. We could easily rewrite this equation as:

$B = A^2 + 5A$ or $(\text{OUTPUT}) = (\text{INPUT})^2 + 5*(\text{INPUT})$.

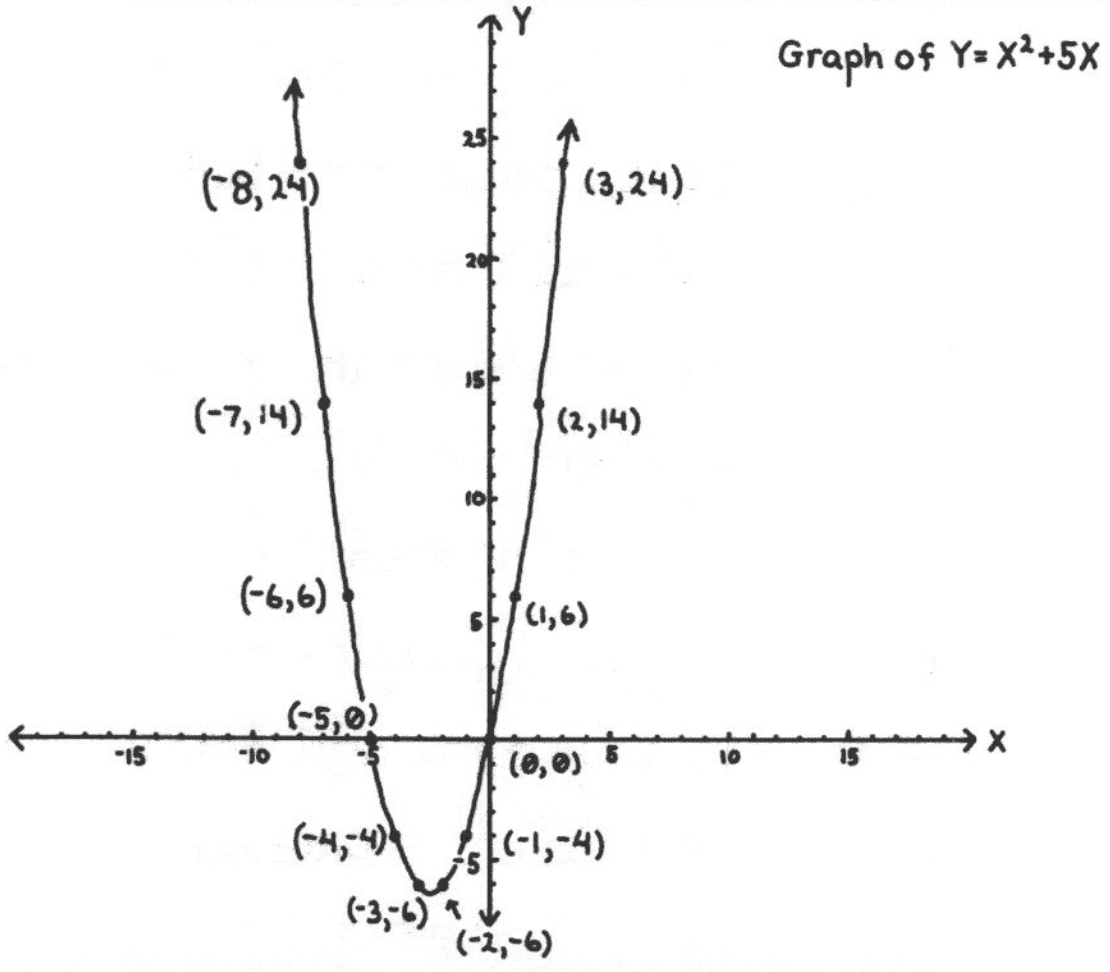
But, because it's common to talk about equations graphed on the X and Y axis, we'll use the original $Y = X^2 + 5X$. Whenever we mean the X-register, we'll write "the X-register." Otherwise, X means the variable number X in the equation.

Anyway, to plot the equation $Y = X^2 + 5X$, we would start at, say, zero and plug in a bunch of values for X to see what Y-value each X gives.

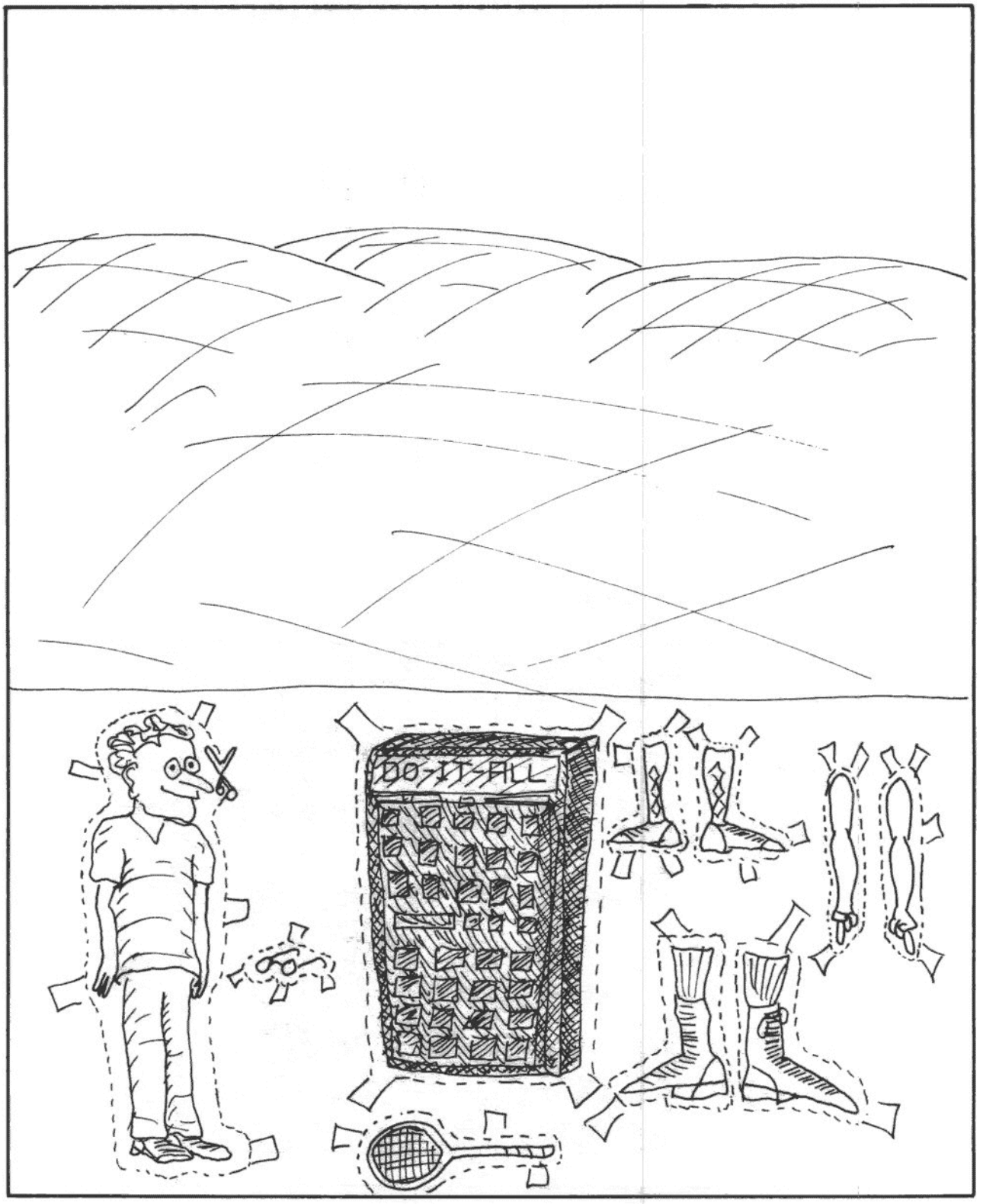
We would generate a table that looks like this:

For X equal to	Y equals	For X equal to	Y equals
0	0	-3	-6
1	6	-4	-4
2	14	-5	0
3	24	-6	6
-1	-4	-7	14
-2	-6	-8	24

Then we could take these pairs and plot them on a graph with the X-value plotted horizontally and the Y-value plotted vertically.



The program we develop will generate the X, Y pairs, so that we can plot the graph of any equation we program into the HP-41.



Challenge: Write a routine to do the equation:

$$Y = 4X^3 - 12X + 5$$

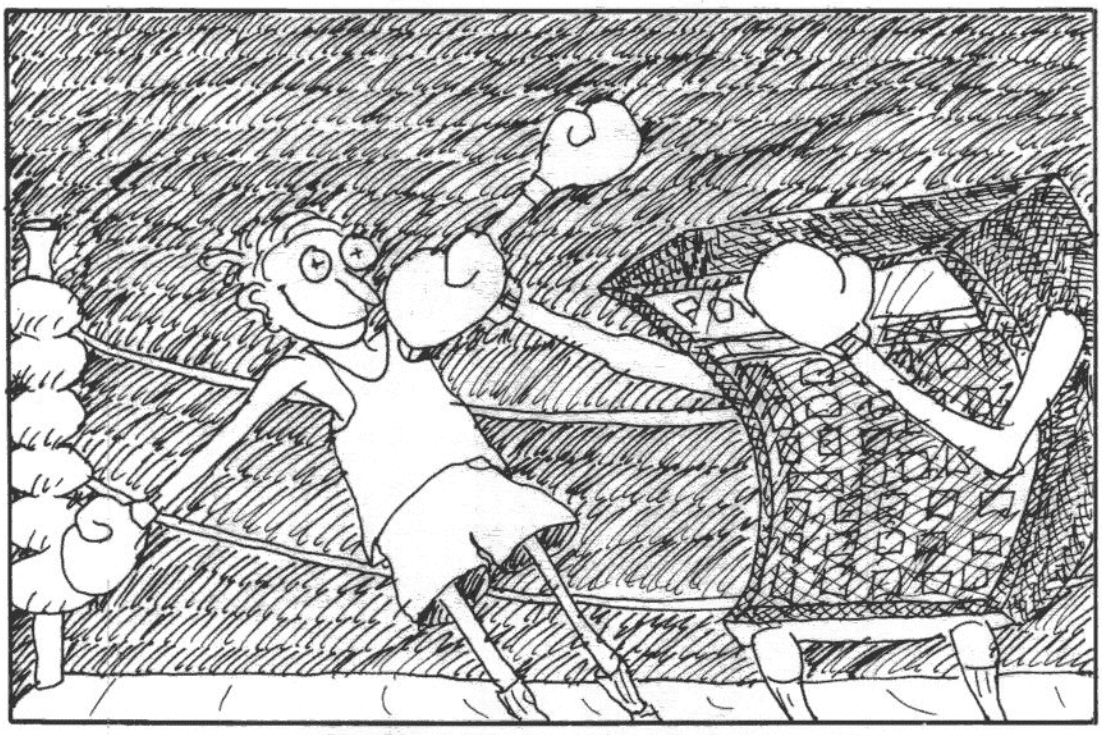
or

$$OUTPUT = 4*(INPUT)^3 - 12*(INPUT) + 5.$$

Solution:

```
01 LBL "FUNK1        08 12
02 STO 10            09 *
03 3                 10 -
04 Y↑X               11 5
05 4                 12 +
06 *                 13 END
07 RCL 10
```

If this "throws you for a loop," review pages 88-92.



Challenge: Using the FUNK1 routine, calculate a set of X, Y pairs, starting at X=0 and increasing by 0.25 each time, up to 2 (that's 9 pairs). Pay close attention to the motions that you go through, because the program you develop is going to do the same thing.

Solution:

For X equal to	Y equals	For X equal to	Y equals
0	5	1.25	-2.1875
0.25	2.0625	1.50	0.5000
0.50	-0.5000	1.75	5.4375
0.75	-2.3125	2.00	13.0000
1.00	-3.0000		

OK?

All it amounts to is this: To get the first Y-value, key in 0 XEQ "FUNK1," and the HP-41 will return 5.0000. So the first X, Y pair is 0, 5. To get the second Y-value, key in .25 XEQ "FUNK1" and the HP-41 will return 2.0625, etc.

Challenge: Write out, step by step, what you just did on the last page, as if you were explaining the process to a friend.

Solution: Well, let's see....

1. I read the problem and saw that I was supposed to use the FUNK1 routine to generate a table of X, Y pairs. I was to begin at X=0, end at X=2, and increase by 0.25 each time. So I was told the name of the program with which to evaluate the Y-values, the beginning X-value, the ending X-value, and the increment.

2. I started at the beginning X-value.

3. I ran this current X-value through the FUNK1 routine to get Y.

4. I wrote down this X, Y pair.

5. I added the increment (0.25) to X.

6. I checked to see if this new current X-value was greater than 2. If it wasn't, I repeated steps 3 through 6. If it was, I stopped.

Question: What are the six steps that a program would have to perform to do what you did on page 220 (two pages back)?

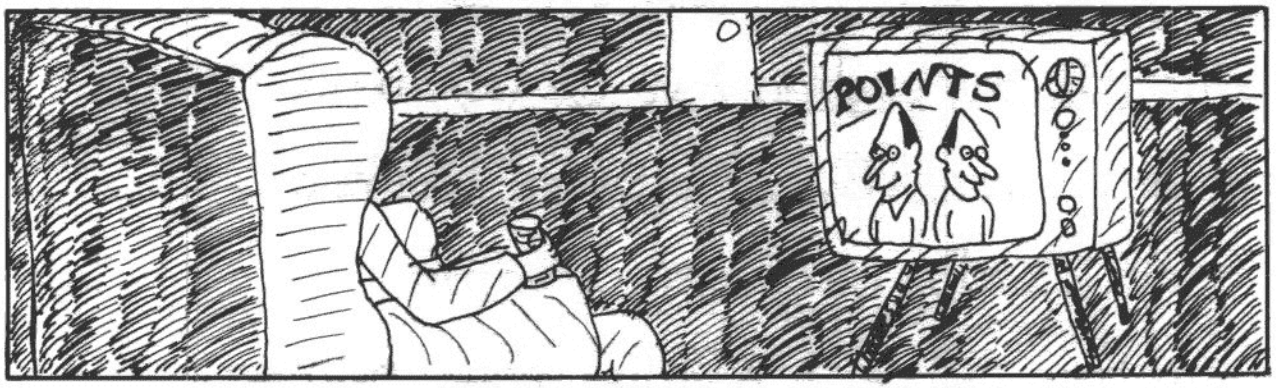
Answer:

1. Prompt for the name of the program for which you're generating X, Y pairs (FUNK1), the beginning X-value (0), the ending X-value (2), and the increment value (0.25). Store them all.
2. Start at the beginning X-value.
3. Run this current X-value through the named program (FUNK1) to get Y.
4. Display the current X, Y pair.
5. Add the increment to the current X-value to get a new current X-value.
6. Check to see if this new current X value is greater than the ending X-value. If it isn't, repeat steps 3 through 6. If it is, stop.

See what we're getting at? These six steps are nearly the same as those on the previous page.

The general steps that you take to complete a process are quite similar to the steps a program has to take to complete the same process.

Since the program we're developing serves to generate points for plotting the graph of an equation, let's agree to call this program "POINTS."



Now, it's just a matter of expanding the general steps on the last page into HP-41 program lines. It's probably best, in this POINTS program, to start expanding at step 1.

Let's agree to store the required data as follows:

register 00 - Name of function to plot
register 01 - Beginning X-value
register 02 - Ending X-value
register 03 - Increment value

Challenge: Expand step 1 on page 222 into HP-41 program lines.

Solution:

```
"NAME?
AON
STOP
AOFF
ASTO 00
"BEGIN X?
PROMPT
STO 01
"END X?
PROMPT
STO 02
"INCREMENT?
PROMPT
STO 03
```

Notice the method we use to prompt for ALPHA data. This method stops the program with the computer in ALPHA mode. So, all you have to do is spell the name and press [R/S].

Also, remember that a data register can hold a maximum of <u>six</u> characters. So the global label of the program you're plotting can be a maximum of <u>six</u> characters long.

Step 2 merely reminds us that, the first time through, the beginning X-value will be the current X-value. So, let's always maintain the current X-value in register 01.

Challenge: Expand step 3 (page 222)

Solution:
```
LBL 01
RCL 01 (current X value)
XEQ IND 00
```

NO SWEAT ? 

Not clear?

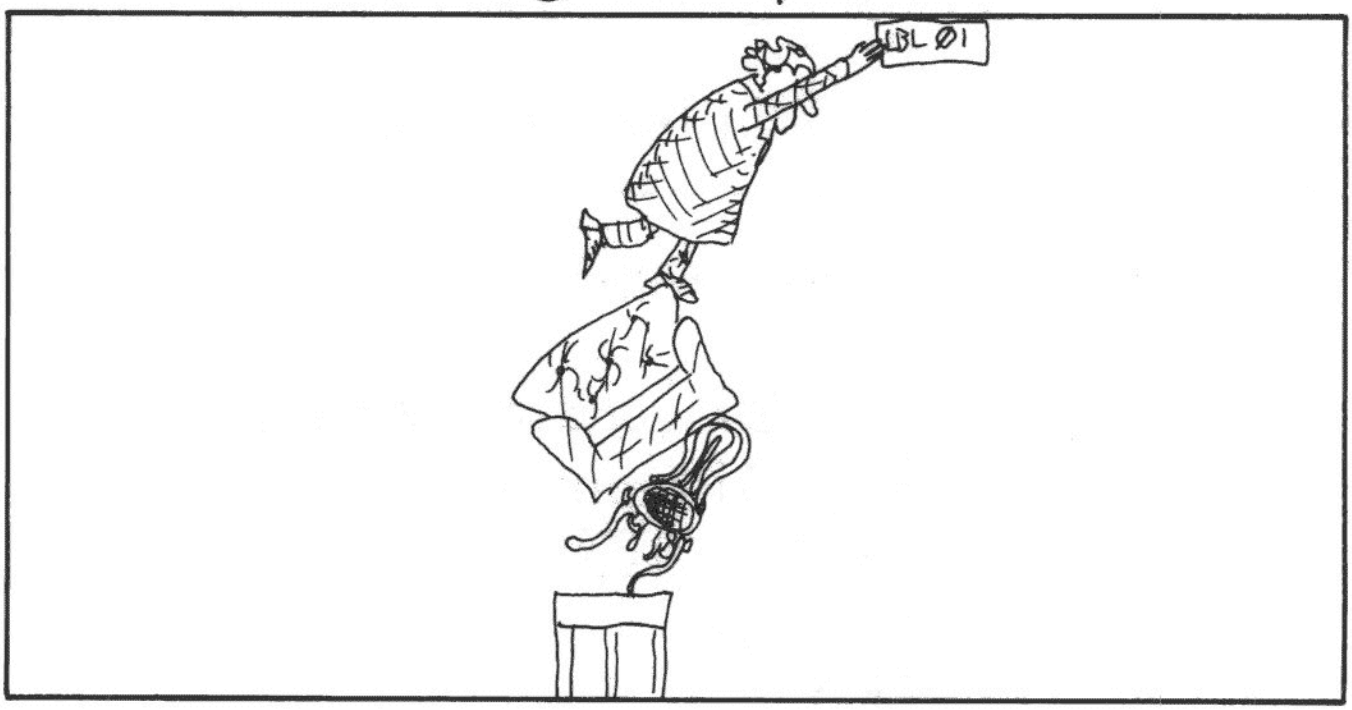
Look at it this way: When you were generating the table of X, Y pairs manually, you would establish a current X-value in the X-register, and then you would XEQ "FUNK1" to get the Y-value. The POINTS program has to go through this same process.

RCL 01 brings the current X-value into the X-register.

XEQ IND 00 says "look at the program name

in register 00 and XEQ that program."

We put the LBL 01 at the top because step 6 mentions returning to step 3.



Challenge: Expand step 4.

Solution:

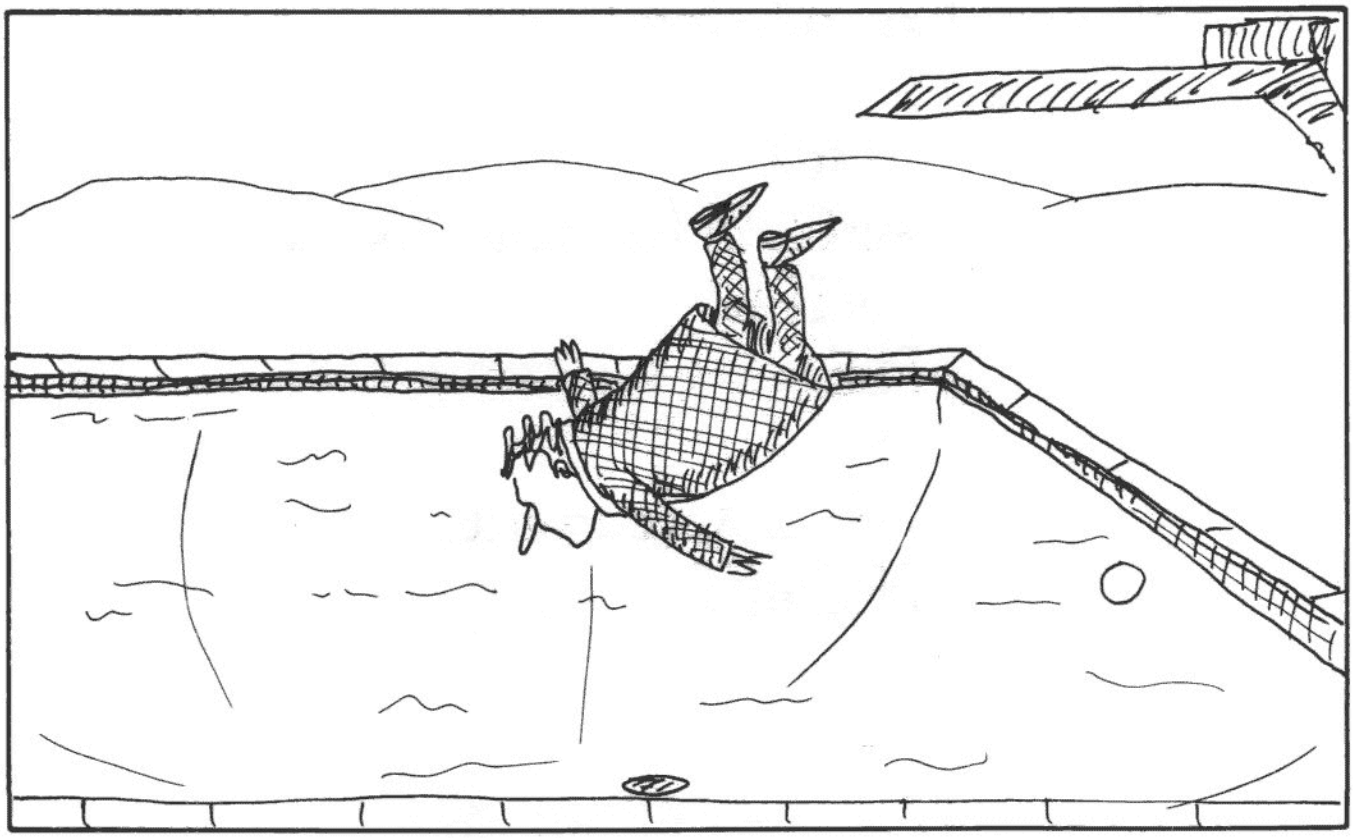
```
"FOR X=
ARCL 01 (current X-value)
AVIEW
PSE
"Y=
ARCL X (current Y-value)
PROMPT
```

Challenge: Expand step 5 into HP-41 code.

Solution:
```
RCL 03 (increment)
RCL 01 (current X-value)
+
STO 01 (new X-value)
```

Challenge: Expand step 6.

Solution:
```
RCL 02 (ending X-value)
RCL 01 (current X-value)
X<=Y? (X≤Y?)
GTO 01
END
```



Challenge: Put it all together under the global label "POINTS."

Solution:

```
01 LBL"POINTS
02 "NAME?
03 AON
04 STOP
05 AOFF
06 ASTO 00
07 "BEGIN X?
08 PROMPT
09 STO 01
10 "END X?
11 PROMPT
12 STO 02
13 "INCREMENT?
14 PROMPT
15 STO 03
16 LBL 01
17 RCL 01
18 XEQ IND 00
19 "FOR X=
20 ARCL 01
21 AVIEW
22 PSE
23 "Y=
24 ARCL X
25 PROMPT
26 RCL 03
27 RCL 01
28 +
29 STO 01
30 RCL 02
31 RCL 01
32 X<=Y?
33 GTO 01
34 END
```

Now, is the program finished?

Have we used any flags? Have we assumed anything about the machine prior to running the program that may not be true?

Well, at line 21 we've used an AVIEW statement, and because we use a PSE statement after it, we have assumed that AVIEW is not going to halt program execution. But remember, if flag 21 is <u>set</u>, AVIEW <u>will</u> stop program execution.

So, we should insert CF 21 early in the program:

```
01 LBL"POINTS   13 "INCREMENT?  25 ARCL X
02 "NAME        14 PROMPT       26 PROMPT
03 AON          15 STO 03       27 RCL 03
04 STOP        *16 CF 21        28 RCL 01
05 AOFF         17 LBL 01       29 +
06 ASTO 00      18 RCL 01       30 STO 01
07 "BEGIN X?    19 XEQ IND 00   31 RCL 02
08 PROMPT       20"FOR X=       32 RCL 01
09 STO 01       21 ARCL 01      33 X<=Y?
10 "END X?      22 AVIEW        34 GTO 01
11 PROMPT       23 PSE          35 END
12 STO 02       24"Y=
```

Your final mission (should you choose to accept it - and you'd better! You've come too far to give up now): Generate a table of X, Y pairs for the equation: $Y = 4X^3 - 12X + 5$, with X beginning at -2.2, ending at 0.4, and increasing by 0.2 each time.

Solution:

For X equal to	Y equals	For X equal to	Y equals
-2.2	-11.1920	-0.8	12.5520
-2.0	-3.0000	-0.6	11.3360
-1.8	3.2720	-0.4	9.5540
-1.6	7.8160	-0.2	7.3680
-1.4	10.8240	0.0	5.0000
-1.2	12.4880	0.2	2.6320
-1.0	13.0000	0.4	0.4560

Keystrokes	Display
XEQ "POINTS"	NAME?
FUNK1 R/S	BEGIN X?
2.2 CHS R/S	END X?
0.4 R/S	INCREMENT?
.2 R/S	FOR X=-2.2000
	Y=-11.1920
R/S	FOR X=-2.0000
	Y=-3.0000
R/S	FOR X=-1.8000
⋮	⋮

Nothing to it, right?

This is what's meant by a friendly program: It leaves little doubt about what to do.

It looks like you are now a friendly programmer!

(Did you save any of the champagne from page 4?)

.END.

APPENDIX

ON USING THE MANUALS

- Diagnosis of Problems
- Statistical Functions
- Fun facts to know and tell

The manuals you received with your HP-41 are some of the best you'll find anywhere. But like most books, their benefits are most obvious when you read them, thoroughly, cover to cover, including the examples and their solutions.

Yes, but that can get pretty tedious, right? True, few people really enjoy reading technical manuals. So part of the aim of this book was to provide a basic tutorial guide so that you would be free to use the provided manuals more as references than as instructional materials.

Obviously, we couldn't cover all the details contained in the HP-41 manuals. So now we'll give you a few tips on using those books as reference manuals:

Got a problem? Is the computer doing something weird? Maybe an error message? Or are you looking for a certain function or a certain way to do something? Where do you look for the answers?

Well, what part of the computer does the problem involve?

If pressing certain keys gives unexpected results, look in the index under "keys," "keyboard," or "key assignments." If all else fails, look in Appendix B of the owner's handbook, under "Maintenance and Service."

Be glad when you get an error message. The message will always lead you quickly to the problem:

• First, if the message appears during a running program, you can determine what caused the error by switching into program mode: The program line in the display is the one that caused the error.

• If the message is one that you've never seen before, you can look it up in the index. It will appear in blue letters in the index. Also, you can always look under "Errors" or "Error messages."

• If you need to find out about a certain function, the best place to look is in the shaded function index immediately following the main index. There

you will find, in alphabetical order, a brief description-and a page reference-of each standard function.

Now, if you're writing a program, and you need to do some kind of calculation or manipulation, and you don't know what functions to use:

- Is it an ALPHA manipulation? (Look under ALPHA.)
- Is it a stack manipulation? (Look under stack.)
- Is it a conditional test, or a flag manipulation.
- Does it involve program looping, or maybe you want to use indirect addressing, or branching?
- Are you confused with any procedure done in program mode (PRGM)?
- Do you need information on subroutines?
- Are you having trouble using labels effectively?

Not surprisingly, the rule of thumb is to look up the answer according to the nature of the problem. All we're doing here is listing some of

the key words and phrases you can look up in the manual's index.

As another (perhaps quicker) alternative, once you're fairly familiar with your computer, you can refer to the "Operating Manual - A Guide For the Experienced User."

This booklet summarizes, in brief, the main points you need to know to program your HP-41. Some of the fine points not covered in the main manual are here, too - certain keystroke shortcuts and programming hints.

Also, one of the handiest parts of this book is the function index (starting on page 52), which lists each function, its memory usage, and its effect upon the stack (very important).

Finally, we come to the "Standard Applications" book: This is a collection of programs for you to key in and run - mainly to give you practice at keying in programs, and to provide you with examples of working programs.

Be sure to read the introductory pages before starting in on the programs. These pages contain good information, and they tend to answer the questions you would otherwise have upon encountering certain program lines.

By the way, when we refer to page numbers in the owner's handbook, we are referring to the "HP-41C/41CV Owner's Handbook and Programming Guide" that was printed by HP in September of 1980. If the page numbers that we've referred to send you on a wild goose chase in your manual, you'll have to rely on that manual's index for direction.

THE STATISTICAL FUNCTIONS

If at some time you have the urge to use statistics to try to prove something, you will find pages 99-104 of the Owner's Handbook helpful.

Here are a few added comments:

1. REMEMBER, the statistical registers are <u>movable</u>. They are not guaranteed to be registers 11-16, except after a MEMORY LOST. <u>You</u> adjust the location of this block by using the ΣREG function to specify the location of the first register in this block.

2. All 6 of these registers must be allocated data registers. Otherwise, the execution of any statistical function will result in a NONEXISTENT.

3. Σ+ and Σ- are functions that <u>disable</u> stack lift.

4. The statistical functions Σ+, Σ-, MEAN, and SDEV <u>always</u> operate on both X and Y data, regardless of whether you are interested in just the X-values.

For this reason, you may encounter an OUT OF

RANGE error when you attempt SDEV, if you haven't been paying attention to the Y-register at all. When this happens, it's because the computer has been faithfully accumulating Y-data right along with your X-data. And it is possible to have data that produces an error with SDEV (see page 256 of the Owner's Handbook).

The cure is simple: When you use CLΣ to start your accumulation, press 0 ENTER. Then just proceed as usual.

5. When you accumulate a set of X, Y data pairs, you can fit a straight line to them according to the formula:

$$Y = A + BX.$$

You can compute the values A and B by using the values in the statistical registers and these formulae:

$$B = \frac{n\Sigma XY - \Sigma X \Sigma Y}{n\Sigma X^2 - (\Sigma X)^2} \quad \text{and} \quad A = \frac{\Sigma Y - B\Sigma X}{n}$$

Notice that page 42 of the Standard Applications Book shows the relationship between A, B, X, and Y for computing the equations of 4 different types of curves.

FUN FACTS TO KNOW AND TELL

There are some functions in catalog 3 that seem clever enough, but they seem to lack practical uses... not so!

It's just that these uses may not appear obvious to the casual observer. So, as a public service, we thought we'd point them out - along with a few other handy tidbits.

To wit:

1. If you want to write a program with several different sections, where these sections may be used in any order, chances are good that you will find it convenient to use local-label key assignments.

Study, for example, the Financial Calculations

program on page 33 of the Standard Applications Handbook.

See also: pages 33 and 34 of the "Guide for the Experienced User."

---

2. Notice the section on page 8 of the "Guide for the Experienced User," called "Single Key Parameter Specification."

All this means is that anytime you are keying in a function that demands a numerical argument from 1 to 10 (inclusive), you can enter that argument with a single key.

This is not a big deal. But, it <u>is</u> convenient because you'll soon find that you are constantly using registers 1 to 10, flags 1 to 10, LBL's 1 to 10, and even SIZE's from 1 to 10.

---

3. "I need to find the seventh root of a number, but there's only a square-root key! What should I do?"

Well, the first thing to do in any such emergency is to remain calm.

Now, it happens that when you take the n^{th} root of a number, what you're really doing is raising that number to the $1/n$ power.

$$\sqrt[7]{11} = (11)^{1/7}$$

To compute this:

11 [ENTER] 7 [1/x] [shift] [y^x] ⟶ 1.41

So, the seventh root of 11 is just 11 raised to the $1/7$ power.

Remember that [y^x] key!

---

4. LN; LOG

Just what are logarithms?

Simply put, the <u>logarithm</u> of a number, N, is the <u>power</u> to which you must raise another number (called the base, B) to arrive at N itself:

If you say "L is the log in base B of the number N," you are really saying:

$$B^L = N$$

The LN function uses the base B=2.718281828, which is the first 10 digits of a number that some people call "e."

The LOG function uses the base B=10.

Notice the shifted functions on the [LN] and [LOG] keys are [e^x] and [10^x], respectively. These functions are the antilogarithms that correspond to the logarithms on the unshifted keys.

---

5. MOD

What's this good for?

Well, suppose you are trying to figure out what time of day your astronaut friend will will re-enter the earth's atmosphere if he/she reached orbit at 1400 hours, to begin a 65-hour orbit.

First, you add: 14 [ENTER] 65 [+] ⟿ 79.

So, re-entry will occur at 7900 hours after midnight of the launch date.

But, what time of day is that?

Use the MOD function and the fact that there's 24 hours in a day to determine the time of re-entry:

24 [XEQ] [ALPHA] MOD [ALPHA] ⟿ 7

Your friend will re-enter the atmosphere at 700 hours (7 a.m.).

So, MOD is handy for finding the remainder of divisions. It's great for problems involving regular cyclic intervals of time or position.

Take note of how it works with negative numbers...!

---

6. FRC; INT

What good are these?

Well, as an example, suppose you have more numbers to store than you have registers to put them in.

If you know how many digits these numbers will have, then you can solve the storage problem by

combining two numbers as one.

For example, you could combine the numbers 32 and 6 by using 32.06, or combine the numbers 1 and 17 to get 1.17.

Here, the integer portion of the number would represent the first of the pair, and the fractional portion would represent the second.

To retrieve the original pair of numbers from the "hybrid" number, you would use:

FRC and INT.

Can you see how?

Using INT on 32.06 produces 32.00.
Using FRC on 32.06 produces 0.06.
Then multiplying by 100 gives 6.00.

AHA!

(Remember our FT and FIS routines, too - page 209).

7. Remember!

You'll never get the HP-41 to tell you numerically that the sine of π radians is zero or that $(\sqrt{2})^2 = 2.000000000$.

This is because it can never work with π or with $\sqrt{2}$ or with any other number with an infinite number of digits (only 10 digits, <u>always</u>).

The best it can do (which is <u>quite</u> adequate) is $SIN(3.141592654) = -4.100000000 \times 10^{-10}$ and $(1.414213562)^2 = 1.999999999$. This is accurate and consistent.

---

8. HMS; HR; HMS+; HMS-

These 4 functions are used for calculations involving <u>H</u>ours, <u>M</u>inutes, and <u>S</u>econds, or decimal Hou<u>R</u>s (i.e., for <u>time</u>). But notice that they work just as well with <u>degrees</u>, <u>minutes</u>, and <u>seconds</u> (i.e., angles).

---

9. Flags: Certain flags are useful in writing nicely polished programs: ⟶

FLAG 21: Although this is used mainly to control a printer, it can determine when the VIEW and AVIEW functions halt program execution (and when they don't).

FLAGS 22 and 23: You can use these to write programs where the user has a choice whether to respond to a prompt with a data input, or with just a [R/S], or with some other command. (For an example, see the Financial Calculations program in the Standard Applications book.)

FLAGS 24 and 25: Error-ignore flags are handy in preventing unforeseen inputs from stopping execution due to computation errors.

FLAG 27: This lets you turn on (or off) USER mode in a running program.

FLAGS 28 and 29: With these flags, you can get European notation (CF 28), and remove the decimal point and digit separators if, for example, you want to ARCL integers only.

Here are two other great books on your HP-41 You won't want to miss them!

Using Your HP-41 Advantage: Statics for Students

This is a book written especially for all you engineering students out there (and it was written by engineering graduates)! You know how tedious it can be to solve all those 2-dimensional free-body static equilibrium problems, right? Well, what would you say to a program that will do it for you on your HP-41? You'd say, "Tell me more!"

All right, picture this:

- You start with the problem you're trying to solve, drawn, as always, with some 2-dimensional coordinate system established.
- You identify and key in all known forces and moments – and their directions (in your choice of rectangular or polar coordinates).
- You ask the calculator to solve for up to three unknowns (the sum or resultants of the total X- and Y- Forces and the Z-Moment).

And it's easy to build and add on to your description of your structure, with the points and Knowns editor. So don't waste any more of your time trying to figure out where you dropped a digit; let the HP-41 and the Advantage module do your crunching for you. This book comes with complete program listings and bar code for both the HP-41CV and the HP-41CX, so it's ready when you are!

Computer Science On Your HP-41 Using the Advantage Module

Here's an ingenious idea: You take one of the most popular programmable calculators of all time (the HP-41), and you program it to imitate another calculator – so you don't need to carry both!

In a program that he likes to call his "16E," Computer Science instructor Ed Keefe has achieved a remarkable emulation of Hewlett-Packard's HP-16C "Computer Scientist" calculator. So if you need the kind of digital "bit-crunching" power that the HP-16C delivers – but you have only an HP-41 – then here's your answer (and if you do happen to have an HP-16C, then see below for the book written specifically for that, too)!

The program in this book will let the HP-41 do virtually anything that the HP-16C can do. You'll learn about:

- **Word Size and how to set it**
- **Integer arithmetic**
- **1's and 2's Complements & Unsigned formats**
- **Logical Operators** • **Masks**
- **Bit clearing, setting, shifting and rotating**
- **Programming**

And you'll see how to customize the HP-41 keyboard to make the emulator as easy to use as possible. So solve your digital logic problems right there on your trusty HP-41. This is a program that really does it.

Here's a list of all our other books:

- An Easy Course In Using The **HP-42S**
- An Easy Course In Using The **HP-14B**
- The **HP-14B Pocket Guide**: Just In Case
- An Easy Course In Using The **HP-32S**
- An Easy Course In Using The **HP-22S**
- An Easy Course In Using The **HP-28S**
- An Easy Course In Using The **HP-27S**
- An Easy Course In Using The **HP-17B**
- The **HP-17B Pocket Guide**: Just In Case
- An Easy Course In Using The **HP-19B**
- The **HP-19B** Pocket Guide: Just In Case
- An Easy Course In Using The **HP-16C**
- The HP Business Consultant Training Guide
- An Easy Course In Programming The **HP-11C And HP-15C**
- An Easy Course In Using The **HP-12C**
- The **HP-12C Pocket Guide**: Just In Case

(Use next page to order any of these books.)

Grapevine Publications, Inc.
P.O. Box 118
Corvallis, OR 97339-0118

To Order:

Call our Toll-Free Line for the location of the GPI dealer nearest you, OR charge the books to VISA or Mastercard, OR

Fill out this Order Form and return it to: **Grapevine Publications, P.O. Box 118, Corvallis, OR 97339**

_____ copies of	An Easy Course In Using The **HP-42S**	@ $22.00 ea.=	$_____
_____ copies of	An Easy Course In Using The **HP-14B**	@ $22.00 ea.=	$_____
_____ copies of	The **HP-14B Pocket Guide**: Just In Case	@ $ 5.00 ea.=	$_____
_____ copies of	An Easy Course In Using The **HP-32S**	@ $22.00 ea.=	$_____
_____ copies of	An Easy Course In Using The **HP-22S**	@ $22.00 ea.=	$_____
_____ copies of	An Easy Course In Using The **HP-19B**	@ $22.00 ea.=	$_____
_____ copies of	The **HP-19B Pocket Guide**: Just In Case	@ $ 5.00 ea.=	$_____
_____ copies of	An Easy Course In Using The **HP-17B**	@ $22.00 ea.=	$_____
_____ copies of	The **HP-17B Pocket Guide**: Just In Case	@ $ 5.00 ea.=	$_____
_____ copies of	The HP Business Consultant **(HP-18C)** Training Guide	@ $22.00 ea.=	$_____
_____ copies of	An Easy Course In Using The **HP-12C**	@ $22.00 ea.=	$_____
_____ copies of	The **HP-12C Pocket Guide**: Just In Case	@ $ 5.00 ea.=	$_____
_____ copies of	An Easy Course In Using The **HP-28C**	@ $22.00 ea.=	$_____
_____ copies of	An Easy Course In Using The **HP-28S**	@ $22.00 ea.=	$_____
_____ copies of	**HP-28S** Software Power Tools: **Utilities**	@ $18.00 ea.=	$_____
_____ copies of	**HP-28S** Software Power Tools: **Electrical Circuits**	@ $18.00 ea.=	$_____
_____ copies of	An Easy Course In Using The **HP-27S**	@ $22.00 ea.=	$_____
_____ copies of	An Easy Course In Programming The **HP-41**	@ $22.00 ea.=	$_____
_____ copies of	Computer Science on Your **HP-41** (Using the Advantage Module)	@ $15.00 ea.=	$_____
_____ copies of	Using Your **HP-41 Advantage**: Statics For Students	@ $12.00 ea.=	$_____
_____ copies of	An Easy Course In Programming The **HP-11C** and **HP-15C**	@ $22.00 ea.=	$_____
_____ copies of	An Easy Course In Using The **HP-16C**	@ $22.00 ea.=	$_____

(Prices valid through February 5, 1989) **Subtotal = $_____**

SHIPPING INFORMATION: ***plus***

For orders **less than $16.00** _____ **ADD $ 1.00** $_____

or

For all other orders – **Choose one:** **Post Office shipping and handling** **ADD $ 2.00** $_____
(allow 2 weeks for delivery)

or

UPS shipping and handling **ADD $ 3.50** $_____
(allow 7-10 days for delivery)

TOTAL AMOUNT: ——————————> $_____

PAYMENT:

Your personal check is welcome. Please make it out to Grapevine Publications, Inc. **Or:**

Your VISA or MasterCard #:_____ Exp. date:_____

Your signature:_____ Phone: ( ) _____

Name _____

Shipping Address _____

Note: UPS will not deliver to a P.O. Box! Please give a street address for UPS delivery.

City _____ **State** _____ **Zip**_____

Call Our 24-Hr. Toll-Free Number:

1-800-338-4331

(In Oregon: 754-0583)

- EDITORIAL -

We hope you liked the design of this book. We have used simple language, cartoons, and an "un-compact" format, because this is still the easiest way for anyone to learn from a book. There is no need to resort to a lot of technical jargon to explain programming.

If you have any comments or have noted any errors (although there's no possible way that we made any), please write to us.

In the making of this book, we certainly wanted to help you learn about the HP-41. It is an excellent machine and you will appreciate it even more as you learn more about it.

But beyond helping you learn, we wanted to reassure you that computer programming is not some cult of technobabble, explicable only to child prodigies, sci-fi buffs, engineers, mathematicians, and persons

surrounded by mist and butterflies. Anybody who can learn to read and write can learn about programming.

We have watched, with growing concern, the crescendo of media hype over "high tech" and "personal computers," and it seems much too easy for teachers, parents, and other mortals to be led to believe that they or their children will somehow be left behind if they don't join the mad rush to "get into" computers, either vocationally or avocationally.

Computers are indeed very useful tools, and they are becoming a common part of our lives. But, this doesn't mean that the world is rushing you by just because you don't own one. People have quite enough worries for themselves and their children without the addition of such an artificial burden of guilt.

And above all, when you find yourself in the midst of some over-inflated, buzz-word egos,

maintain your confidence. It is no crime to prefer plain English, or to take more time in learning to write a program. In fact, there is no crime in being more interested in, say, gardening tools than in computing tools. Each tool has its place and purpose. And, with the help of good sense and patience, each will bear fruit.

Ted Wadman

Chris Goffinet

Robert Bloch

June 1983

TABLE OF CONTENTS

(JUST IN CASE YOU WANT TO BRUSH UP)

CONTENTS PAGE

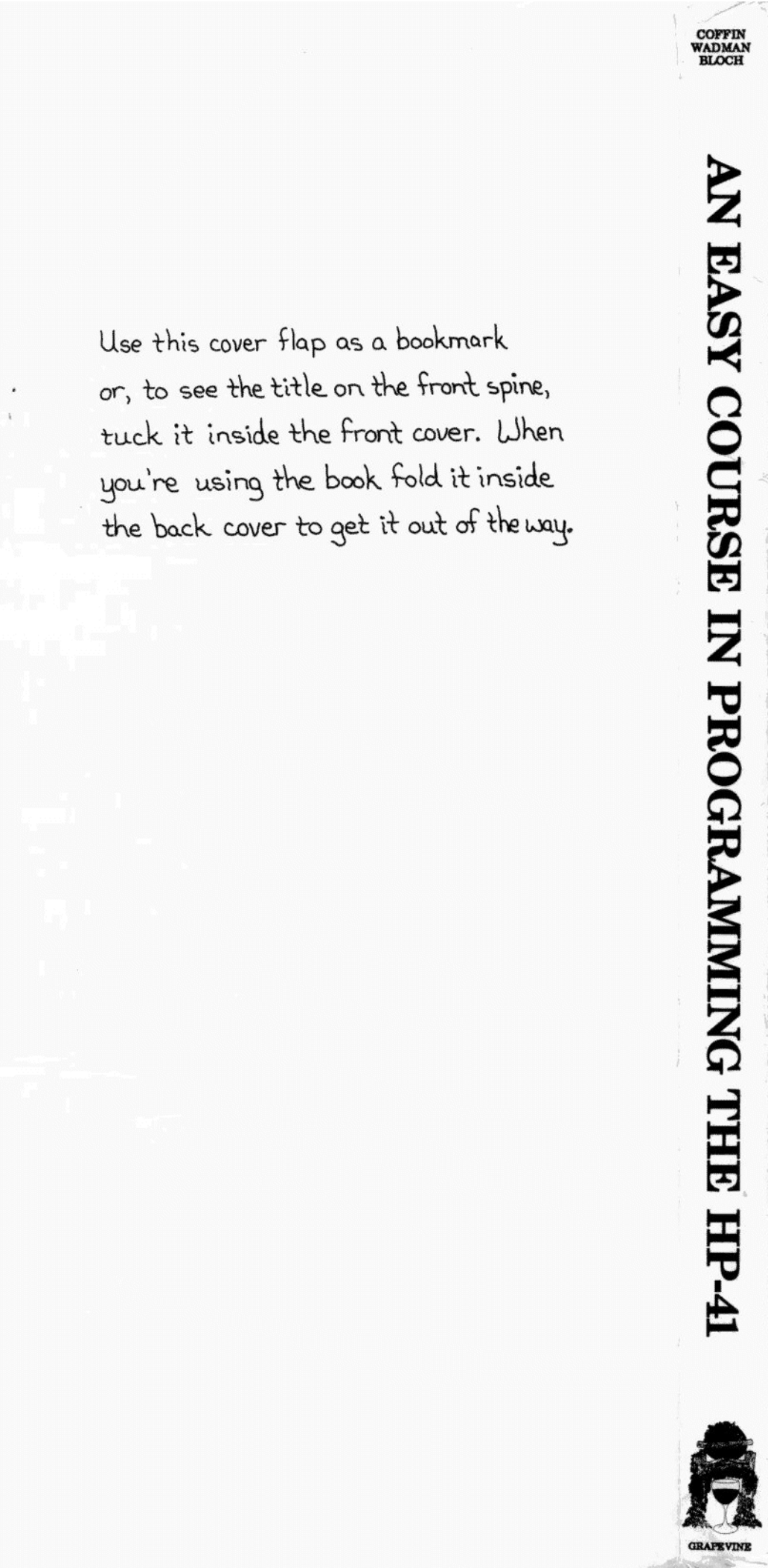
CONTENTS PAGE

CONTENTS PAGE

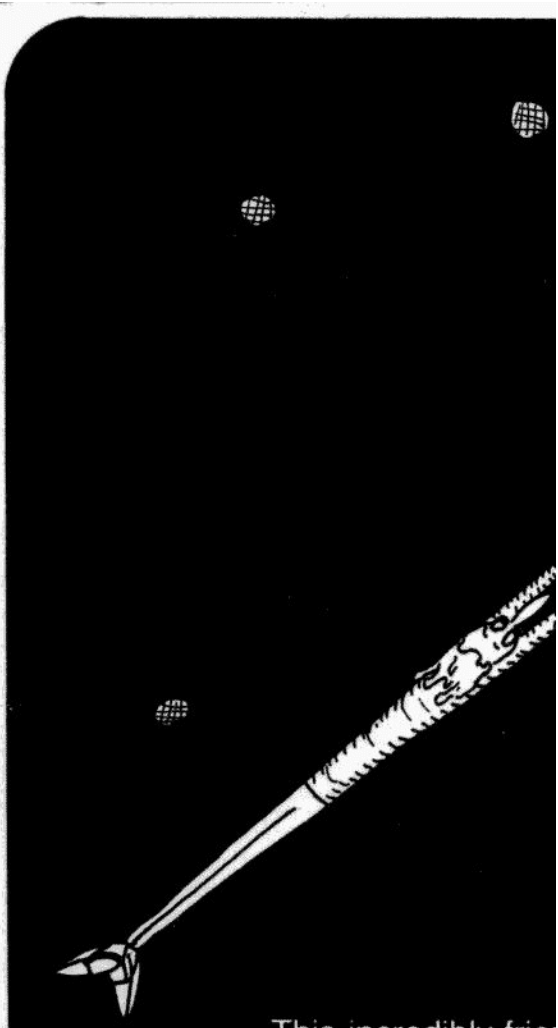
Use this cover flap as a bookmark or, to see the title on the front spine, tuck it inside the front cover. When you're using the book fold it inside the back cover to get it out of the way.

COFFIN
WADMAN
BLOCH

AN EASY COURSE IN PROGRAMMING THE HP-41



GRAPEVINE



This incredibly frie
the power and sophis
ing, conversational st
dation out of learning
former Hewlett-Pack
jargon-free approach
subject. And if a dash
rations are added, the
EASY and ENJOYAB

Packed with examp
zes, this self-paced bo
all the ins and outs o
RAMMING THE HP-4
calculator. Come alo
yourself using your c

This book
an

FROM THE
GRAPEVIN

92234T

This incredibly friendly book will help you discover and understand the power and sophistication within your HP-41 calculator. Its refreshing, conversational style and unique, readable format take the intimidation out of learning to program your calculator. The authors, both former Hewlett-Packard support engineers, realize that a relaxed, jargon-free approach is the best possible way to present a technical subject. And if a dash of humor along with some clever cartoon illustrations are added, then learning to program your HP-41 becomes both EASY and ENJOYABLE!

Packed with examples, review questions, explanations and fun quizzes, this self-paced book lets you work along at your own rate, learning all the ins and outs of programming. "AN EASY COURSE IN PROGRAMMING THE HP-41" is the easiest and fastest way to master your calculator. Come along on this Easy Course, and you'll soon find yourself using your calculator like an expert!

This book applies to the HP-41C, the HP-41CV,
and the ultimate: the HP-41CX.

FROM THE PRESS AT
GRAPEVINE PUBLICATIONS, INC.

92234T

ISBN 0-931011-09-4