HP 41/HP 48
Transitions

```
{ HOME }                          PRG
≪ IF HP41
  THEN STO 99
  ELSE 99 STO
  END ≫ 'RPL' STO◀
 IF  CASE START  FOR   DO  WHILE
```

William C. Wickes

HP 41/HP 48 Transitions Program Disk

As a convenience for readers of *HP 41/HP 48 Transitions* who use or have access to an IBM-compatible or MacIntosh personal computer, Larken Publications is offering a disk containing all of the HP 48 programs described in the book. By downloading the programs individually or collectively from your computer to your HP 48, you avoid the effort and errors of entering the programs manually from the calculator keyboard.

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

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

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

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Quantity		Unit Price	Price
	HP 41/HP 48 Transitions Program Disk	$10.00	
	(Optional) Airmail postage outside of USA, Canada, Mexico	1.00	

TOTAL $ _____

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

HP 41/HP 48 Transitions

William C. Wickes

William C Wickes

8/30/98

Dedicated to Carroll Alley, who got me my first HP 41, and has been a staunch supporter ever since.

CONTENTS

1. Introduction

The HP 41C, HP 41CV, and HP 41CX are generally recognized as the most successful family of programmable technical calculators ever produced. Their combination of calculator convenience with flexibility, customizability, and communications capability has been unmatched by any other product at a comparable price since the HP 41C introduction in 1979.

Part of the HP 41's long-lasting popularity has derived from the absence of any new products, from Hewlett-Packard or any other manufacturer, that add new hardware or software features to the all-around capabilities of the HP 41. The HP 42S, for example, is designed to execute HP 41 programs, much more quickly than the HP 41 itself, but the HP 42S lacks the plug-in ports of the HP 41, so that all programs must be typed in by hand. The HP 28S offers computation technology significantly advanced over that of the HP 41, but like the HP 42S, the HP 28S is not extensible and has no input mechanism other than its keyboard.

With the introduction of the HP 48SX, we see in many respects a rebirth of the HP 41. With its plug-in memory card ports, extensive customization capability, and built-in calculator-to-calculator and calculator-to-serial-device communications, the HP 48SX offers all of the advantages of the HP 41, implemented on greatly superior hardware. Furthermore, the HP 48SX incorporates the advanced software technology of the HP 28S. With the advent of the HP 48SX, the HP 41 has finally been retired from Hewlett-Packard's active product line. (Henceforth, for simplicity, we will refer generally to the *HP 41* and *HP 48*, dropping the alphabetic product designators C, CV, SX, etc., unless they are required in specific cases.)

Many HP 41 users may be a little apprehensive about adopting the HP 48 as the long-awaited replacement for their HP 41's, because of the differences in calculation style and programming language between the two calculators. This book is intended to help you make the transition from the HP 41 to the HP 48, by reviewing their similarities and differences. The general approach is to show how the HP 48 principles of operation are refinements or extensions of familiar HP 41 ideas. The book is not intended as a complete treatise on HP 48 operation, but rather as a guide to translating your HP 41 problem-solving techniques onto the HP 48, so that you will have a comfortable basis from which to explore the vast array of new computing resources that the HP 48 offers to you. We will not study basic HP 48 operation or keystrokes, except as they relate to the HP 41; it is recommended that you review the HP 48 owner's manual at least to obtain a working knowledge of the HP 48 before or during your reading of this book.

The following summarizes the contents of each chapter:

Chapter		Topics
1.	Introduction	Introductory material, notation conventions, HP 41 to HP 48 evolution.
2.	The HP 48 Stack	Understanding and using the unlimited stack.
3.	Variables	Storing and naming objects on the HP 48.
4.	HP 48 Programming Principles	HP 48 programming principles, conditional structures, and local variables.
5.	Program Development	Constructing HP 48 programs with input and output, halts, and prompts; key assignments.
6.	Program Conversion	Converting HP 41 programs to HP 48 programs.

1.1 Notation

Most calculator manuals use the term *function* to refer to any operation a calculator can perform, programmable or not. Here we will use the more precise HP 48 terminology, where *function* refers to specific mathematical operations, and the term *operation* is used for general calculator actions. A programmable operation, i.e., one that has a name that can be entered in a program, is called a *command*. Thus SIN is a command, but ▵ is an operation. We will use this terminology for HP 41 operations as well.

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

- All calculator commands and displayed results that appear in the text are printed in helvetica characters, e.g. DUP 1 2 SWAP. When you see characters like these, you are to understand that they represent specific HP 41 or HP 48 operations rather than any ordinary English-language meanings.

- Italics used within calculator operations sequences indicate varying inputs or results. For example, 123 'REG' STO means that 123 is stored in the specific variable REG, whereas 123 *'name'* STO indicates that the 123 is stored in a variable for which you

may choose any name you want. Similarly, $\ll$ *program* $\gg$ indicates an unspecified HP 48 program object; { *numbers* } might represent a list object containing numbers as its elements.

Italics are also used for emphasis in ordinary text.

- Calculator keys are displayed in helvetica characters surrounded by rectangular boxes, e.g. ENTER , EVAL , or EEX . The back-arrow key looks like this: ⇐ , and the HP 48 cursor keys like these: ◁ , ▷ , △ , and ▽ .

- A shifted key is shown with the key name in a box preceded by a shift key picture, ■ for the HP 41 shift key, and ⇦ or ⇨ for the HP 48 left- and right-shift keys.

- HP 48 menu keys for commands available through the various menus are printed with the key labels surrounded by boxes drawn to suggest the reverse characters you see in the display, like these: ▤SIGN▤ or ▤→LIST▤ .

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

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

HP 41 examples are presented in a similar manner, except that the four stack levels are labeled X:, Y:, Z: and T:.

For better legibility, we don't show individual letters and digits in key boxes--just 123 rather than 1 2 3 . We do show key boxes for the multi-letter keys on the keyboard and in menus.

In some HP 48 examples, a simple listing of the stack contents isn't adequate, so we use an actual calculator-generated picture of the calculator display, such as this picture from Chapter 5:

```
                                                      PRG
{ HOME }
Enter X:



100◀
PARTS PROB  HYP  MATR VECTR BASE
```
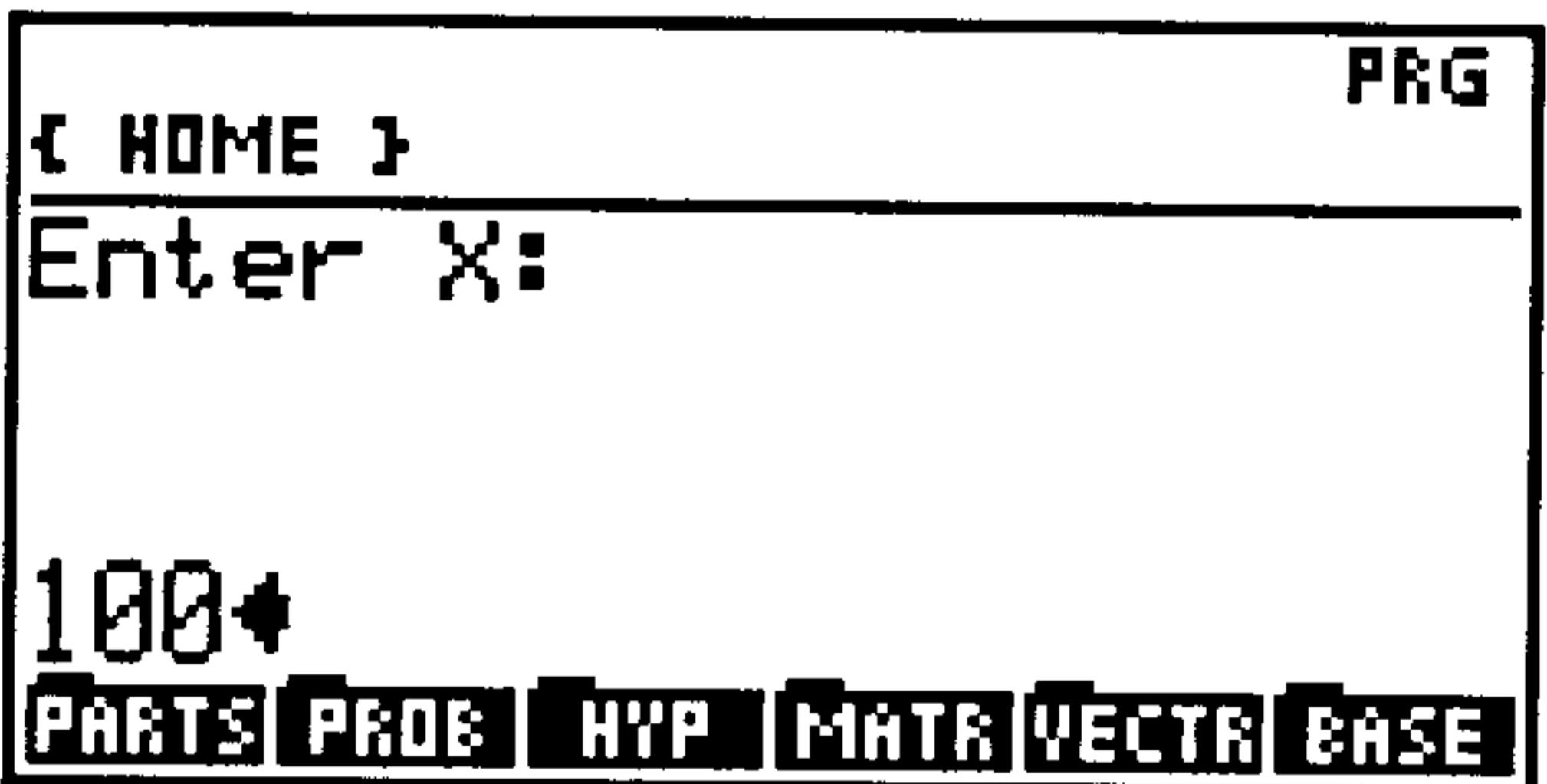
Some examples, however, are given in a more compact format than the keystroke exam-ple shown above. These examples consist of a *sequence* of calculator commands and data that you are to execute, together with results. The "right hand" symbol ☞ is used as a shorthand for "the calculator returns..." In the compact format, the above addition example is written as

<div align="center">123   456   +   ☞   579</div>

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

<div align="center">A   B   C   ROT   SWAP   ☞   B   A   C</div>

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

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

The ☞ symbol is also used in the *stack diagrams* that are part of HP 48 program list-ings. The stack diagrams show how to set up stack objects for execution of the pro-gram, where the objects to the left of the ☞ are the "input" objects, and the objects fol-lowing the ☞ are the program "outputs".

The most elaborate examples in this book are *programs*. Each program is listed in a box, showing the actual steps that make up the program, and comments to help you understand the steps. In addition, many program listings include a heading like this one

from Chapter 6:

SI	*Sine Integral*	AF4C
	level 1 *level 1*	
	x ☞ SI(x)	

The first line of the header shows a suggested program variable name, a brief description of the program, and the hexadecimal checksum for the program object. The rest of the header is a stack diagram that shows the program's inputs and outputs.

1.2 HP 41-to-HP 48 Evolution

Many HP 41 owners who pick up an HP 28 or an HP 48 are taken aback to find that the newer calculators depart in many important ways from the "standard" RPN design that had remained essentially unchanged from the HP 35 through the HP 41CX. The consistency of the standard design always made it easy for anyone to upgrade to a new HP calculator, since the basic operating methods remained the same. Without this comforting familiarity, the 28/48-class of calculator may seem rather alien and forbidding to prospective users.

Actually, the HP 28 and HP 48 were specifically designed with the HP 41 as a model, attempting to preserve and enhance the strengths of the HP 41 while remedying its weaknesses. In particular, the most basic calculator operations--keyboard arithmetic-- are maintained keystroke-for-keystroke from the old generation to the new. To add 1 plus 2, for example, you press ☐1☐ ☐ENTER☐ ☐2☐ ☐+☐ , regardless of whether you are using an HP 41, an HP 28, or an HP 48. For more complicated arithmetic, you can generally use the same keystroke RPN approach on any of the three calculators. This basic compatibility serves as the anchor point from which you can explore some of the areas in which the HP 48 differs from the HP 41.

The basic design philosophy of the HP 48 is to extend the calculator convenience and flexibility of the HP 41 to a wider class of mathematical and logical quantities than the HP 41 can handle. You know that the HP 41 is an excellent calculator for operations on floating-point real numbers. It provides an RPN stack, which is demonstrably the best interface for exploratory calculations. Its programming language is a straightforward encoding of keyboard steps, so you don't have to rethink a problem solution to capture it as a program after you have performed it once manually. However, when you stray out of the realm of ordinary numbers, the HP 41 begins to show its limitations. For example, the HP 41 has no simple provision for complex numbers. Various HP 41 programs that deal with complex numbers usually use pairs of registers to contain a single complex number. The four-level stack thus becomes a two-level complex number stack, but this obviously has major limitations. Not only does the limited stack depth

make calculations difficult, but even simple commands like + must be replaced with complex number versions--ordinary + would add the real and imaginary parts of one of the two stack numbers, rather than performing a proper complex addition.

The HP 48 deals with complex numbers in a much more straightforward way, by generalizing the design of the RPN stack so that each stack "register" (called a *level*) can hold an entire complex number as readily as a real number. Commands like + examine the stack to see what type of addition to perform--if the stack contains real numbers, a real number sum is computed; if either of the summands is complex, a complex addition is performed. To add the complex numbers $1+2i$ and $3+4i$, the sequence is:

$$(1,2) \quad \boxed{\text{ENTER}} \quad (3,4) \quad \boxed{+}$$

where complex numbers $x+iy$ are entered and displayed as ordered pairs (x,y) in the HP 48. Note that the logical sequence is just the same as the real number example given previously:

$$\textit{first-number} \quad \boxed{\text{ENTER}} \quad \textit{second-number} \quad \boxed{+} .$$

In the HP 48, every real number operation that could sensibly be extended to complex numbers is made to work uniformly with either type of number. This is what we meant at the beginning of the section by "extend the calculator convenience and flexibility of the HP 41 to a wider class of mathematical and logical quantities."

[The first calculator with built-in provision for complex numbers was the HP 15C, which uses dual stacks to hold the real and imaginary parts of complex numbers for RPN operations. A mode setting determines whether commands are applied to one stack or both. The HP 15C also allows you to manipulate real and complex matrices on its stacks. The even more general approach of the HP 28/HP 48 is a descendent of this pioneering design of the HP 15C.]

The desire to intermix complex numbers and real numbers on the same stack requires a more flexible stack design than that of the HP 41. On the HP 41, the stack is a permanent set of four *registers*--memory locations exactly big enough (7 bytes each) to contain one floating-point number. Since a complex number is made up of two numbers, it can't be stored in a single stack register, so complex number programs must resort to clumsy substitutes such as using the stack registers in pairs. The HP 48 solves this problem by eliminating the fixed stack registers and using instead a dynamically configured stack in which the entries on the stack can be of any size whatever. A internal system of memory pointers keeps track of the order and sizes of the entries, and the memory used by the stack grows and shrinks automatically as the stack contents change.

The generalized stack of the HP 48 is not limited even to numbers or arrays. The calculator can deal with a host of different kinds of *objects*: real and complex floating-point

numbers, integers, arrays, text strings, pictures, physically dimensioned numbers--even algebraic expressions and programs. *Object* is the HP 48 term for any "thing" that you can put on the stack--when you see that term, just think of a *number* as the prototype of an object. All of the different HP 48 objects can be manipulated on the stack, stored, recalled, etc., using the same logic and keystrokes as real numbers. This makes learning to use the HP 48 simple, since the methods used for ordinary numerical calculations can also be used for much more complicated problems--problems that on the HP 41 either require elaborate strategies, or can't be computed at all.

The HP 41 itself takes a small step towards a generalized stack with its ability to place *alpha data* in stack registers. The fixed size of the registers limits the individual alpha data objects to a maximum of six characters, but the basic idea is the same as that on the HP 48--you can move the alpha data around on the stack or store and recall the data using the the same commands as you use for numbers.

The HP 48 carries this idea to greater extremes by allowing various operations to work with many different object types. For example, consider the common programming task of combining two text strings into a single string (concatenation). On the HP 41, with the two strings initially represented as alpha data in the X- and Y-registers, you can concatenate them by using the sequence

$$\text{CLA} \quad \text{ARCL} \quad \text{X} \quad \text{ARCL} \quad \text{Y}$$

This combines the two strings in the alpha register, but you have to activate alpha mode to see the result, since the result is too long in general to return to a single stack register. Also, you can't store the combined string without taking it apart again. On the HP 48, the entire operation is condensed into a single execution of +. + works for strings the same way that it works for numbers: the two inputs are taken from the first two stack levels, and the sum is returned to level 1:

$$\text{"ABC"} \quad \text{"DEF"} \quad + \quad \text{☞} \quad \text{"ABCDEFG"}$$

In this case the "sum" is the concatenation of the two strings. The flexibility of the stack allows the inputs and the result to be strings of any size, so there is no need on the HP 48 for a separate alpha register.

1.2.1 Other HP 41 Extensions
We have shown how the goal of "generalizing the HP 41" leads to the HP 48 idea of a flexible stack that can hold a variety of objects as well as real numbers. In the following we will show how similar considerations of HP 41 limitations lead to other facets of the HP 48 design. In subsequent chapters we will study these topics in more detail.

1.2.1.1 The Unlimited Stack

All HP RPN calculators prior to the HP 28 had a data stack limited to four registers. The choice of four registers was actually derived from the limited memory of the HP 35, rather from any profound analysis that said that four was somehow a "correct" number for a stack. Later calculators like the HP 41 used RAM separate from the CPU for their stacks, but kept the four-level stack for the sake of familiarity with their predecessors.

The depth of the RPN stack determines the number of intermediate results that can be retained during an extended calculation without recourse to any explicit store and recall to non-stack memory registers. In computing the value of an algebraic expression, the stack depth corresponds to the maximum number of open parentheses that can be maintained at one time. So-called "algebraic" calculators in the HP 35 to HP 41 era generally advertised some number of parentheses levels that they could handle; this was their equivalent of a fixed RPN stack depth. (Internally, all calculators use RPN, so the algebraic calculators' parentheses levels really were a stack depth limitation.)

As a practical matter, the majority of common expressions likely to be evaluated on a calculator can readily be managed with a four-level stack, as long as you are willing to work through an expression from the inside out so as to avoid too many intermediate results. However, given the abundance of RAM on current calculators, there doesn't seem to be any reason to continue to impose an artificial limitation of four stack levels. The purpose of a calculator is to automate as much of the calculation process as possible. Requiring you to inspect each expression before calculation in order to find a path that doesn't overflow the stack is hardly consistent with that purpose. Moreover, if a calculator is to work with *symbolic* expressions, it is quite impractical to limit the number of nested parentheses. The HP 48 does not attempt to answer the question of the "right" number of stack levels--it simply does not limit the stack size at all.

The desirability of extending the stack does not come only from purely mathematical considerations. Consider the HP 41 program loop command ISG. ISG requires an argument that is a number of the form *aaa.bbbcc*, where the digits *aaa* are the loop index, *bbb* is the stopping value of the index, and *cc* is the index increment. This structure has two obvious disadvantages: 1) the index and its increment must be integers, limited in their numbers of digits, and 2) the argument must be assembled from the individual numbers by dividing and adding; similarly, it must be taken apart to obtain current values. Both of these problems would disappear if ISG could use three separate stack arguments. But that is impractical in a four-level stack calculator, since whenever ISG was executed, anything currently on the stack would have to be stored away to make room for ISG's arguments.

A fixed-depth stack also makes program development difficult, since you can't generally

leave things on the stack when calling a subroutine--the subroutine will likely need the stack for its own purposes. On the HP 48, you can write a subroutine without any regard for the stack use of the programs that call it. A subroutine can be just like a built-in command, in that it is entirely characterized by the number of arguments it takes from the stack, and the number of results that it returns to the stack. (As a matter of fact, most HP 48 commands do their computations on the same stack that you use, sometimes pushing dozens or even hundreds of objects on and off the stack. This doesn't matter, because any objects you have on the stack will still be there after you execute a command, except for the command's arguments).

1.2.1.2 Algebraic Expressions

There is generally little doubt among RPN calculator users that the RPN approach using a stack and postfix operations is easier and more efficient for exploratory calculation than the so-called "algebraic" method used by other calculators. The algebraic style attempts to reproduce more nearly the left-to-right calculation order of the common written form of algebraic expressions, but provides this familiarity at the cost of flexibility. However, the advantage of RPN is hard to perceive for calculations that are predetermined literal executions of written formulae, especially on a calculator with sufficient memory and display to show an entire formula as it is entered instead of a single number or function at a time.

The HP 48 philosophy is that the RPN vs. algebraic debate is made irrelevant by the availability of sufficient electronic resources--display and memory. The HP 48 is capable of interpreting and executing algebraic expressions complete with prefix, infix, and postfix functions, and parentheses. But it does these calculations by means of an RPN stack, and makes that same stack available to you so that you can save and reuse intermediate results. You have the best of both worlds:

- If you know in advance the complete mathematical form of a calculation, you can enter and evaluate it as an algebraic object.

- If you are working out the solution to a problem, and don't know in advance all of the steps, you can work through the problem with an RPN approach, applying functions to the results as they appear.

- In both cases, the results are held on the stack ready for use in further calculations.

The desirability of an algebraic capability becomes particularly obvious when applications like HP Solve or function plotting are considered. If you want to plot, for example, $x^3 - 5x^2 + 4x - 6$, you wouldn't want to have to translate it into the form of an HP 41 program:

```
01   RCL X
02   3
03   Y↑X
04   RCL Y
05   2
06   Y↑X
07   5
08   *
09   −
10   RCL Y
11   4
12   *
13   +
14   6
15   −
```

Not only does it take considerable mental effort to write this program sequence, but the final form is almost unrecognizable as a representation of the original expression. Compare it with the HP 48 algebraic object

$$'X^3 - 5X^2 + 4*X - 6',$$

and you can immediately see the advantages of the HP 48 approach. Even better, the HP 48 EquationWriter lets you enter or display the expression in a form even more closely resembling the written form:
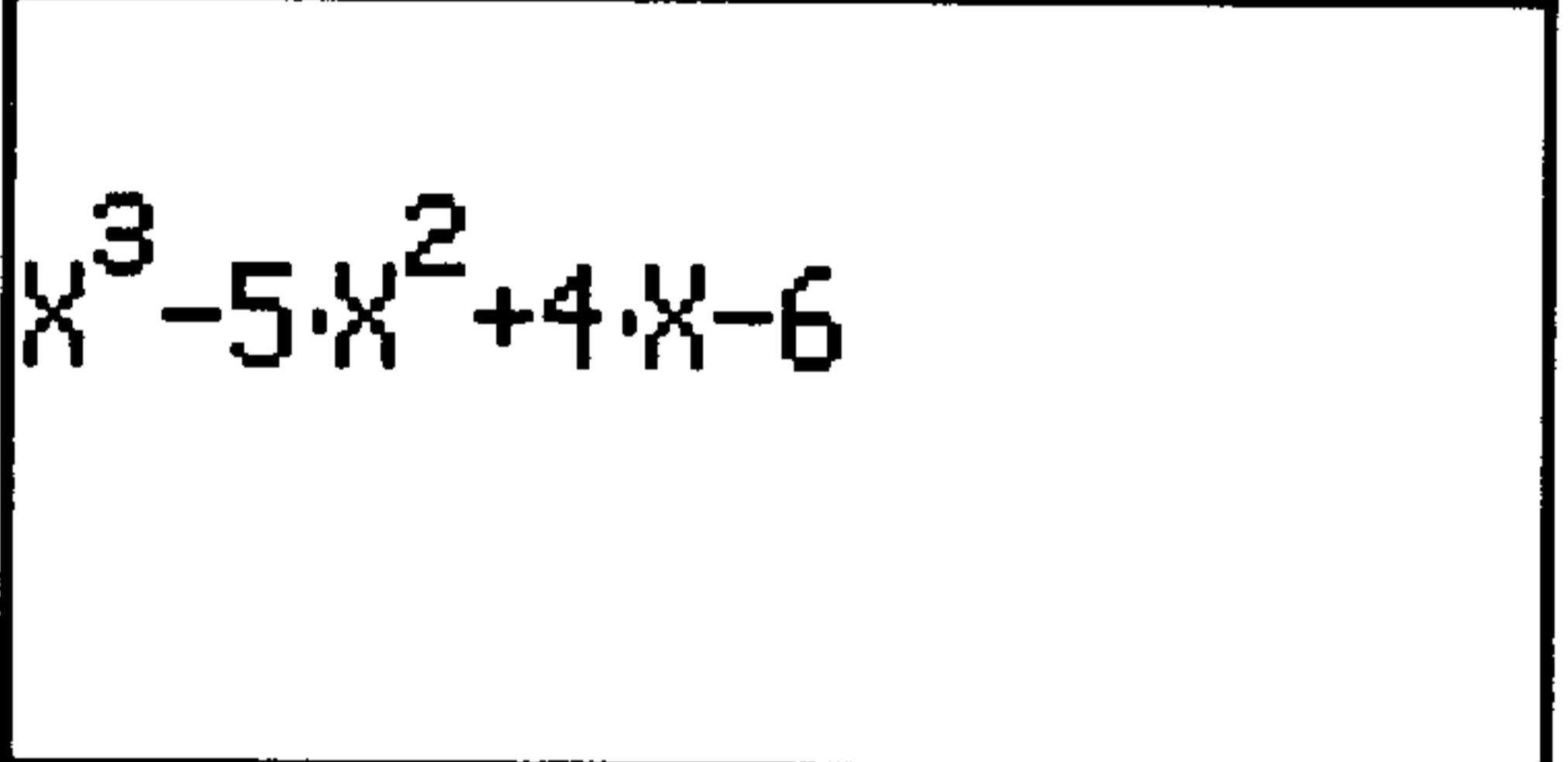
$$X^3 - 5 \cdot X^2 + 4 \cdot X - 6$$

1.2.1.3 The Command Line

In the HP 41, numbers are keyed directly into the X-register. Upon entry of a new number, the previous contents of each stack register are automatically pushed up to the next higher register. This design is again an economy from the early days of calculators when memory was very limited. The effect on the stack while you are entering a number is obviously a disadvantage--once you start a new entry, you lose the contents of

the T-register. With this in mind, the HP 48 replaces the X-register entry system with a much more flexible and powerful *command line* entry system. The command line lets you enter objects as text, with many of the resources of a text editor: multiple lines, a cursor that can be moved to any position, insert or replace modes, typing aids, etc. The use of *delimiters*--special characters that identify the various HP 48 object types--reduces the need for special editors for each type, and allows you to combine the entry of different types in a single command line. In effect, each command line that you enter is a program intended to be executed immediately. The command line program can consist of anything from a single number to be entered on the stack to an elaborate sequence of commands and data.

1.2.1.4 Variables
To support extended calculations and data storage, the HP 41 provides *storage registers* that can each contain a single number or alpha string. By using SIZE, you can select the number of registers that are available, the remainder of memory then being reserved for program storage.

Like the HP 41 stack registers, storage registers are of fixed size and therefore are awkward for storing anything other than individual numbers or six-character text strings. Furthermore, a register's designation by its register number does not provide any mnemonic clues about the register's contents. To address these problems, the HP 48 replaces numbered registers with named *variables*. Each variable contains one object of any type or size, and is identified by a name up to 127 characters long. A variable's name can therefore help to identify its contents--names such as LENGTH or FREQUENCY are typical--making keyboard use easier and programs more legible than is possible with numbered registers.

Named variables are also desirable for algebra operations. If you are trying to represent the expression $A + B + C$ in the calculator, you wouldn't want to have to write it as RCL 01 + RCL 02 + RCL 03. In the HP 48, the expression is represented by the algebraic object 'A+B+C', where A, B, and C are the names of variables. The correspondence between the HP 48 representation and the original expression is obvious and natural.

1.2.1.5 Programming
The HP 41 programming language is appealing because it is a natural extension of keyboard operations. A program consists of a sequence of numbered steps; in the simplest cases, the steps represent the same keystrokes that you would use when making the same calculation once by pressing keys. Recording the keystrokes as a program allows you to replay them automatically whenever you wish.

HP 48 programming starts with the same basic model as the HP 41: a program is a record of commands that you want to replay automatically. A HP 48 program *looks* considerably different from an HP 41 program, primarily because the HP 48 doesn't bother with line numbers. However, the sense of the programs is the same on the two calculators. Consider a program sequence that computes $[\sin(3\cos 30)]/5$. To perform this calculation by hand, you would press the same keys on either calculator:

$$30 \quad \boxed{\text{COS}} \quad 3 \quad \boxed{\times} \quad \boxed{\text{SIN}} \quad 5 \quad \boxed{/}$$

On the HP 41, a program to perform this calculation translates each keystroke into a numbered program line:

```
01   30
02   COS
03   3
04   *
05   SIN
06   5
07   /
```

On the HP 48, the equivalent program sequence matches the keystrokes even more closely, because it's not disrupted by the line numbers:

$$30 \quad \text{COS} \quad 3 \quad * \quad \text{SIN} \quad 5 \quad /$$

The HP 48 dispenses with line numbers in order to display as much of a program as possible on the screen while you are editing the program. Since the program steps aren't numbered, you are free to move the cursor around the screen at will and insert or delete objects.

The HP 41 and HP 48 start with the same general philosophy of translating keyboard operations one-for-one into program steps, but diverge in their approach to program branching (which is usually done only in programs). The HP 41 relies upon its GTO command for all program jumps and loops, along with primitive conditional skips and a very limited error trapping mechanism. This approach encourages the development of convoluted, illegible programs (called "spaghetti code" in computer jargon). In contrast, the HP 48 enforces a more organized program style by providing *program structures* such as IF...THEN...ELSE and FOR...NEXT for conditional branches and loops--it does not provide a GTO at all. These structures, together with an unlimited subroutine return stack, make it possible to develop programs as series of self-contained modules, each of which can perform a single understandable calculation and be labeled with a mnemonic name. The resulting programs can then be read easily and modified or extended in a straightforward manner.

2. The HP 48 Stack

In an RPN calculator, the *stack* is the focus of most operations. It is the place where the great majority of commands find their arguments and return their results. It's also the primary and most efficient means for commands and programs to transfer data and instructions so that small calculations can be linked together. In this chapter, we'll describe the fundamental HP 48 stack operations and compare them with their HP 41 counterparts. For the most part we will use real numbers as example objects, but keep in mind that all of the stack operations described here apply uniformly to any of the various HP 48 object types.

The HP 48 stack consists of series of numbered *levels*, each of which contains one object of any type. The stack is always filled from the lowest level up, so that there are never any empty levels between full ones. ENTER always moves new objects from the command line into level 1, pushing previous stack objects up to higher levels. Most commands remove their argument objects from the lowest levels, whereupon the objects in higher levels drop down. The only exceptions are some of the stack manipulation commands, which can move objects to or from arbitrary stack levels. There is no limit on the number of objects or levels of the stack; you can enter as many objects as available memory will permit.

The HP 48 provides an extensive set of stack manipulation commands, some permanently assigned to keys, and the rest contained in the stack menu (PRG ≣STK≣). All of the stack menu operations are programmable commands, which means that you can execute them by pressing the appropriate keys or by spelling their names into the command line. Most stack operations can also be executed by using the *interactive stack menu*, described in section 2.5.

Table 2.1 on the next page lists the stack operations found on the keyboard and in the stack menu. The individual operations are explained in subsequent sections.

2.1 Clearing the Stack

Perhaps the most common stack operation is "clearing" one or more objects, either to discard unnecessary objects so that others are moved to lower levels, or just to clear the decks for a new calculation. The latter is accomplished by CLEAR, which removes the entire contents of the stack in a single operation. CLEAR is usually executed from the keyboard (▷ CLR); a well-designed HP 48 program does not execute CLEAR because that might destroy stack objects needed by a second program that called it.

There are three commands for removing a specific number of objects from the lowest-

Table 2.1. HP 48 Stack Manipulations

	Operation	Action
Stack Clearing	DROP	Discard the level 1 object
	DROP2	Discard the objects in levels 1 and 2
	DROPN	Discard the first n objects
	CLEAR	Discard all stack objects
Reordering Arguments	SWAP	Exchange the objects in levels 1 and 2
	ROT	Rotate the level 3 object to level 1
	ROLL	Rotate the level n object to level 1
	ROLLD	Rotate the level 1 object to level n
Copying Objects	DUP	Copy the level 1 object
	OVER	Copy the level 2 object
	PICK	Copy the level n object
	DUPN	Copy the first n objects
Counting Objects	DEPTH	Count the number of objects on the stack
Object Recovery	LASTARG	Return the arguments used by the last command
	LAST STK	Restore the stack to its state before ENTER

numbered stack levels: DROP, DROP2, and DROPN. The basic command is DROP, which removes the object in level 1, and "drops" the remaining stack objects one level to fill in the empty level. Each DROP discards another object, and the stack drops one level.

DROP and CLEAR correspond to the HP 41 commands CLX and CLST, respectively. However, as we will discuss further in section 2.6.2, the HP 41 and the HP 48 differ considerably in their concepts of a "clear" stack. CLX clears the X-register, and CLST the entire stack, by replacing the contents of those registers with zeros. CLX is primarily intended for replacing the contents of X with a new value. By disabling stack lift, the zero CLX enters can be overwritten by a following entry. The HP 48 takes a simpler approach: DROP discards the level 1 object and doesn't replace it at all. Since the HP 48 has no stack-lift disable, the next entry always replaces the dropped object.

CLST is of very limited value in the HP 41, serving only to provide a "supply" of zeros. HP 48 CLEAR, on the other hand, removes all objects from the stack, and recovers the

associated memory.

DROP2 and DROPN are equivalent to repeated execution of DROP. DROP2 does just what its name implies: it removes two objects, from level 2 and level 1, then drops the remaining objects down two levels to fill in. The closest HP 41 equivalent to DROP2 is the sequence CLX RDN CLX RDN.

DROPN drops *n* objects in addition to a number *n* in level 1 (so actually *n + 1* objects are dropped--see section 2.2.4 for a discussion of stack depth parameters). Notice that although DROPN appears abbreviated as ≣DRPN≣ in menus, its correct name in a program is DROPN.

2.2 Rearranging the Stack

Dropping objects from the stack is not always the appropriate action when you need access to objects in higher-numbered stack levels--you may also need to preserve the low-numbered objects. In such cases, you need to employ stack rearrangement commands to change the order of the objects.

2.2.1 Exchanging Two Arguments

The simplest form of stack rearrangement is the exchange of the positions of the objects in levels 1 and 2. On the HP 41, this is accomplished by X<>Y, which is renamed to SWAP on the HP 48. SWAP is used for switching the arguments for a two-argument command, or more generally for changing the order in which the level 1 and 2 objects may be used. SWAP is easy to illustrate:

<div align="center">A B SWAP ☞ B A.</div>

2.2.2 Rolling the Stack

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

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

Register	Register contents		
	Before	*After* R↑	*After* R↓
T:	t	z	x
Z:	z	y	t
Y:	y	x	z
X:	x	t	y

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

The HP 48 provides a more general roll up/down capability with ROLL (roll up) and ROLLD (roll down). These work analogously to R↑ and R↓, respectively, but you must specify the number of stack levels you want to roll by placing the number in level 1. Each command drops the number from the stack, then rolls that number of the remaining stack objects. So 4 ROLL is equivalent to HP 41 R↑, and 4 ROLLD is the same as R↓:

Level	Stack Contents		
	Before	*After* 4 ROLL	*After* 4 ROLLD
4:	t	z	x
3:	z	y	t
2:	y	x	z
1:	x	t	y

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

- n ROLL means "bring the nth level object to level 1." That is, ROLL retrieves a previously entered or computed object that has been pushed up the stack by subsequent entries.

- n ROLLD means "move the level 1 object to level n." ROLLD moves the level 1 object "behind" other objects that you want to use first.

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

2.2.3 Copying Stack Objects

One of the strengths of RPN calculators is their ability to make copies of an object on the stack, so that you can reuse it without having to store the object in a data register or

variable. The simplest example of this facility is the HP 48 command DUP, which makes a second copy of the object in level 1, pushing the original copy to level 2, and all other stack objects up one level. The HP 41 counterparts of DUP are RCL X and ENTER↑, although the use of the latter command is complicated by its extra feature of disabling stack lift.

The HP 48 also lets you copy a block of stack objects with DUPN. The sequence n DUPN, where n is a real integer, makes copies of the first n objects on the stack. The order of the objects is preserved; for example

$$X \quad Y \quad Z \quad 3 \quad DUPN \quad \text{☞} \quad X \quad Y \quad Z \quad X \quad Y \quad Z.$$

DUP2 is a one-command version of 2 DUPN:

$$X \quad Y \quad DUP2 \quad \text{☞} \quad X \quad Y \quad X \quad Y.$$

In some cases it is desirable to copy an object that is not in level 1, by bringing a copy to level 1 while leaving the object in its original position relative to other objects. In the HP 48, this combination of ROLL, DUP, and ROLLD is represented by PICK, the general purpose stack copy command. PICK works like ROLL, returning the nth level object to level 1, but it leaves the original copy behind. The original therefore ends up in level $n+1$:

$$W \quad X \quad Y \quad Z \quad 4 \quad PICK \quad \text{☞} \quad W \quad X \quad Y \quad Z \quad W.$$

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

$$X \quad Y \quad OVER \quad \text{☞} \quad X \quad Y \quad X.$$

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

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

2.2.4 How Many Stack Objects?

Several HP 48 stack commands require you to supply an argument that specifies how many stack levels the command will affect. Because this argument is always taken from level 1, you might be uncertain about what the argument should be--should you count level 1, which contains the argument? The answer is no--always count the stack levels you need before the count is entered into level 1.

For example, suppose the stack looks like this:

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

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

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

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

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

2.3 Recovering Arguments

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

1. To allow you to re-use the same argument for a new command.

2. To help you reverse the effect of an incorrect command, by applying the inverse of the command to the same argument.

These two purposes are split into separate operations on the HP 48:

1. The capability of recovering an argument for reuse is provided by the LASTARG command. It is important to note, however, that whereas HP 41 LASTX returns only the X-register argument, LASTARG returns *all* of the arguments used by most recent command--up to five arguments. (No built-in HP 48 command uses more than four arguments, but the LASTARG system provides for up to five for the sake of library commands. Commands like DUPN or →ARRY, which appear to use an indefinite number of arguments, are considered for this purpose to use only one argument, which is the number or list in level 1 that specifies the number of stack levels that are involved.)

The arguments saved by HP 48 commands are kept in a special area of memory that is only accessible via LASTARG, rather than in an L-register that is almost an extension of the stack as in the HP 41. Also, a wider variety of HP 48 commands use stack arguments than is the case in the HP 41, so that the objects returned by LASTARG change more frequently in the HP 48. For example, DROP and SWAP both affect LASTARG, whereas HP 41 CLX and X<>Y do not affect the L-register. Only HP 48 commands like STD or HEX, that use no arguments at all, leave the LASTARG objects unchanged.

2. The use of HP 41 LASTX in recovering from incorrect commands is replaced in the HP 48 by the [LAST STK] (*last stack*--not programmable) operation. At the start of ENTER, a copy of the entire stack is saved. When all of the objects processed by ENTER have completed execution, you can cancel the stack effects of the objects by pressing [←] [LAST STK] . This discards the new stack and replaces it with the stack contents saved by ENTER.

3. The HP 48 [LAST CMD] operation offers a third method for object re-use that does not have a direct analog on the HP 41. The HP 48 saves each command line as it is used, keeping up to four at any time. The saved command lines can be retrieved by pressing [LAST CMD] one or more times. Once an old command line is reactivated, you can edit it or re-use it unchanged.

The objects saved for recovery by LASTARG, [LAST STK] , and [LAST CMD] can consume a substantial amount of memory if the objects are numerous or large. In some cases, this use of memory can actually prevent you from carrying out various operations. For this reason, the HP 48 gives you the option of disabling any or all of these features (and also the command stack), by means of the appropriate keys in the [←] [MODES] menu. You can also enable or disable LASTARG by respectively setting or clearing flag −55.

Two notes:

- Disabling LASTARG prevents commands that error from returning their arguments to the stack. This makes it harder to recover from an error, and also affects the design of error traps (section 4.6).

- If there is insufficient memory available to save the current stack for LAST STACK, the HP 48 shows the error message No Room for LAST STACK, and *automatically* disables LAST STK . This last step is necessary, since you would otherwise be unable to do anything--including trying to free some memory. All commands would fail as the HP 48 tries to save the stack first.

2.4 An Example of HP 48 Stack Manipulations

The following illustrates the use of several of the HP 48 stack commands. By following the steps, you can observe how to copy, move, and combine stack objects.

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

$$P + A + B$$
$$P + B \cdot F + A/F$$
$$P + B/F + A \cdot F,$$

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

■ *Solution*:

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

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

	Steps:		Results:
(Start)	4:		'P'
	3:		'A'
	2:		'B'
	1:		'F'

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

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

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

/ + +	6:	'P + (A + B)'
	5:	'F'
	4:	'P'
	3:	'A'
	2:	'B'
	1:	'P + (B*F + A/F)'

5 ROLLD	6:	'P + (A + B)'
	5:	'P + (B*F + A/F)'
	4:	'F'
	3:	'P'
	2:	'A'
	1:	'B'

4 PICK	7:	'P + (A + B)'
	6:	'P + (B*F + A/F)'
	5:	'F'
	4:	'P'
	3:	'A'
	2:	'B'
	1:	'F'

/ SWAP	6:	'P+(A+B)'
	5:	'P+(B*F+A/F)'
	4:	'F'
	3:	'P'
	2:	'B/F'
	1:	'A'

4 ROLL	6:	'P+(A+B)'
	5:	'P+(B*F+A/F)'
	4:	'P'
	3:	'B/F'
	2:	'A'
	1:	'F'

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

2.4.1 The Easy Way: Local Variables

The preceding example illustrates the use of 48 stack manipulation commands, but it does not necessarily represent the best way to solve the problem. Keeping track of numerous objects on the stack takes considerable care when you are writing or editing a program. In general, manipulating objects on the stack in a purely RPN manner yields the most efficient programs. However, there are other programming techniques that are easier and produce more legible programs.

If you were writing the above program on the HP 41, you could not carry out the example using only the stack, since as many as ten stack levels are needed. You would have to store the initial values in registers, then recall each to the stack as it is needed in the calculations. You can do the same thing in the HP 48, using variables instead of registers (Chapter 3), but this has the disadvantage of cluttering up user memory with a lot of variables, which you may or may not need after the program is finished. The cleanest method is to use *local* variables.

Using local variables, the solution to the problem in the preceding section is represented by the program object listed below. In the program, the sequence → p a b f takes the four initial values off the stack and assigns them to local variables p, a, b, and f (here we are using the convention of lower-case characters for local names). The rest of the program computes the three results, then discards the local variables. The obvious

advantage of this method is that you can write the program "instantly," since the program so closely resembles the written form of the expressions you are trying to compute. The use of local variables is explored in detail in section 4.7.

```
« → p a b f
  «  'p+a+b'  EVAL
     'p+b*f+a/f'  EVAL
     'p+b/f+a*f'  EVAL
  »
»
```

2.5 The Interactive Stack

In the HP 41, the easiest way to see the objects in the Y-, Z-, and T-registers is to roll the stack up or down. However, that has the disadvantage of changing the positions of the stack contents, so that you have to remember to restore the original state if you want to continue with a calculation. A less disruptive method is to use VIEW, which replaces the normal X-register display with a temporary display of the specified register. On the HP 48, rather than rolling the stack around, you can copy any object to level 1 without disrupting the rest of the stack, and then drop it to restore the original stack. Furthermore, the HP 48 also provides the *interactive stack*, which lets you view any stack object by scrolling the display up the stack. The interactive stack also lets you rearrange the stack by applying stack commands to objects in various levels selected by a pointer rather than a stack level argument.

The interactive stack is activated by pressing ▲ when there is no command line or at most a single-line command line active, or by pressing ▤ ↑STK ▤ (in the ← EDIT menu) when there is a multi-line command line. The interactive stack menu appears, and the colon in the level 1 indicator 1: changes to a triangle pointer, to show that the level 1 object is currently selected:

```
{ HOME }
4:                    "Example"
3:                       (0,1)
2:                         'π'
1▶            3.14159265359
 ECHO  VIEW  PICK  ROLL  ROLLD →LIST
```
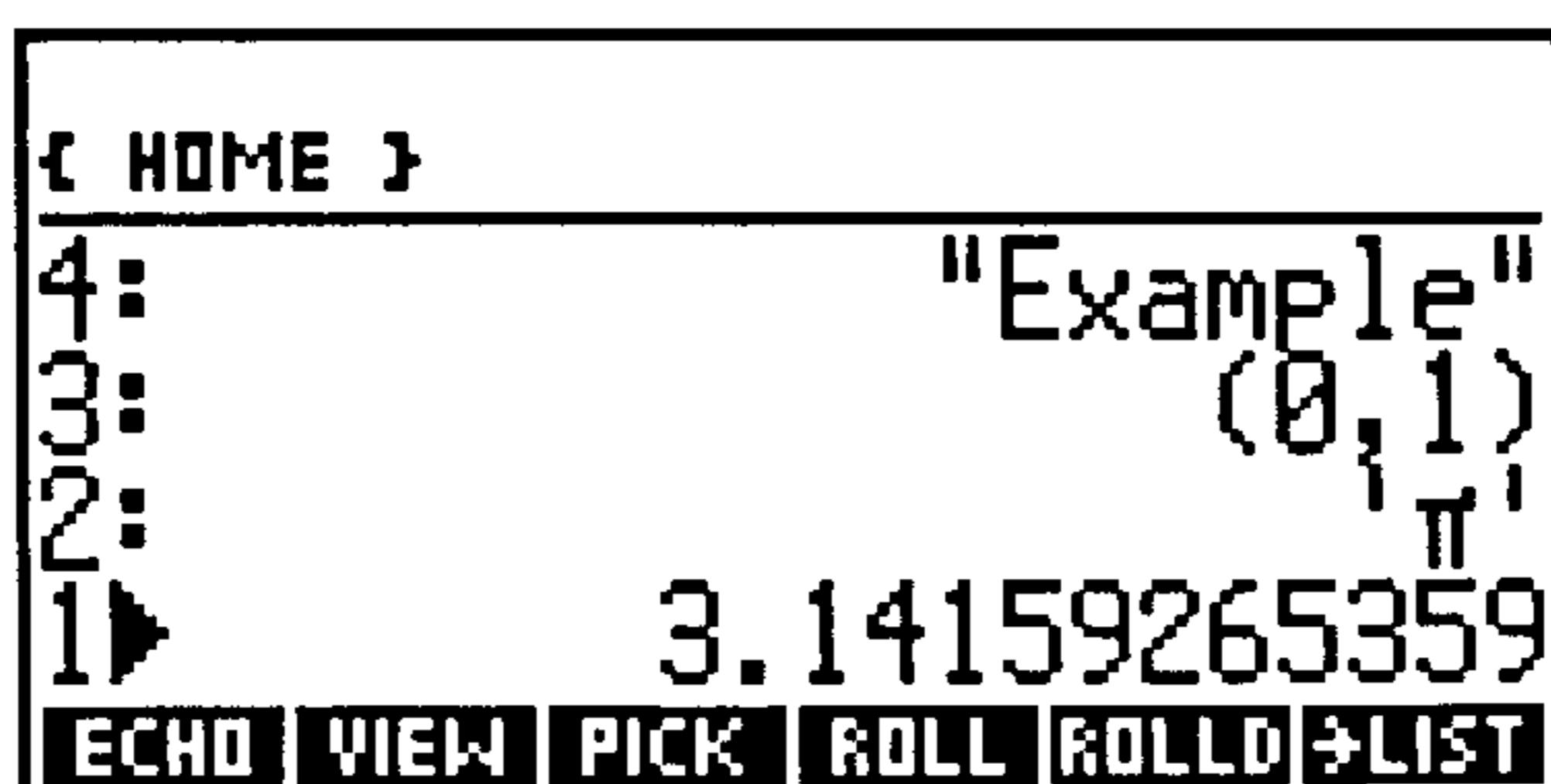
Note that the stack is redisplayed in single-line format, so that four stack levels can appear in the display. Pressing △ moves the selector to level 2; pressing the key repeatedly moves the arrow to the top of the stack display and then begins scrolling objects from higher levels into the window. ⑤△ moves up four levels; ⑥△ moves the arrow to the highest stack level. You can also move the arrow down using ▽ , ⑤ ▽ , and ⑥▽ .

"Selecting" an object consists of moving the arrow to point at it; the stack level number of the selected object is then an implicit argument for the stack operations that appear in the menu. For example, to move the object in level 5 to level 1, you press △ five times (or △ and ⑤△ , then press ≣ROLL≣ . This is equivalent to executing 5 ROLL, but is easier because the very act of moving the pointer up to level 5 to see where the object is not only automatically activates a menu containing ROLL, but also saves you from having to enter the 5.

The other operations in the interactive stack menu are reasonably self-explanatory, with a few observations:

- ≣ECHO≣ is for copying an object to the command line when you want a new copy of the object, either to modify it to make a new object, or to embed it in a command line sequence. It differs from EDIT or VISIT in that the new command line object does not replace the original stack object.

- ⬅ removes the selected object from the stack. It is equivalent to *n* ROLL DROP.

- ≣KEEP≣ discards all stack objects in levels *above* the selected object. It is intended for manual stack cleanup, and has no programmable equivalent since generally it is not a good idea for a program to discard objects that might have been on the stack before it began execution. It is, however, easy to write a program to replicate ≣KEEP≣ :

KEEPN	*Keep N Objects*		4DB5
...	*level 1*	| ...	
objects	*n*	☞ *n* objects	

`≪ →LIST → temp`	Combine *n* objects in a list, save as temp.
`  ≪ CLEAR  temp ≫`	Clear the stack, return the list.
`OBJ→  DROP`	Put the saved objects back on the stack.
`≫`	

Executing *n* KEEPN discards all stack objects in levels above level *n*.

2.6 Managing the Unlimited Stack

In our review of HP 48 stack manipulation commands, we have described the differences between the HP 48 commands and the similar commands provided by the HP 41. In addition to these individual command differences, there are also several general aspects of the use of the HP 48 stack that will require some adjustment if you are used to the HP 41-style stack. In the following sections, we will outline some of the significant differences in stack management between a four-level stack and an unlimited one.

2.6.1 Stack Housekeeping

The obvious key difference between the HP 48 and HP 41 stacks is that the HP 48 stack can contain an unlimited number of objects, whereas the HP 41 and all other previous RPN calculator stacks have been limited to four registers. An important advantage of the unlimited stack is that objects are never lost by being pushed off the end of the stack when a new object is entered. This is also a mild disadvantage--if you don't clear objects from the stack when you're through with them, more and more objects will pile up. This not only wastes memory, but can even slow down execution, since the calculator has to keep track of all of the stack objects. It can also be distracting to see old objects appear in the display when you've long since forgotten their purpose.

A general recommendation for HP 48 stack management is to clean up the stack after you have finished a calculation. By all means pile up as much as you want on the stack while you are working through a problem--that is the stack's purpose. But when you're finished, empty the stack. You can do this either at the beginning or the end of each calculation. We recommend the latter, since at that point you will best remember what each object is, and whether it's all right to throw it away.

"Clean up the stack" doesn't always mean to drop every object, or to use CLEAR. You may very well want to keep certain objects, by storing them in variables, or by using the interactive stack to delete individual objects. Notice that the STO command removes the object being stored from the stack, reducing the number of objects on the stack.

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

$$\boxed{\triangle} \quad \boxed{\triangleright}\boxed{\triangle} \quad \boxed{\equiv\text{→LIST}\equiv} \quad \boxed{\text{ENTER}}$$
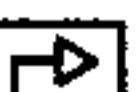
Now you can store the list into a variable named OLDST (for example) by typing

'OLDST' ⌊STO⌋ . This leaves the stack clear for new calculations. After completing any number of subsequent operations, you can restore the old stack by pressing

⌊VAR⌋   ≡OLDST≡   ⌊PRG⌋   ≡OBJ≡   ≡OBJ→≡   ⌊⇐⌋ .

(The DROP performed by ⌊⇐⌋ removes the object count returned by OBJ→.)

In a program, you can duplicate the above keystrokes with:

DEPTH →LIST 'OLDST' STO

to save the stack, and

OLDST LIST→ DROP

to restore it.

Another way of temporarily preserving the stack contents is to execute HALT (in the ⌊PRG⌋ ≡BRCH≡ menu). When a program (including the command line considered as a program--see section 2.8) is suspended by HALT, a new copy of the stack is saved which is independent of any previous ones. This means that you can do anything you want to the stack, including clearing it entirely or using ⌊⇦⌋ ⌊LAST STACK⌋ , without affecting the copy of the stack saved before the HALT. Then when you want to return to the original saved stack, just press ⌊⇦⌋⌊CONT⌋ followed by ⌊⇦⌋⌊LAST STACK⌋ .

2.6.2 A Really Empty Stack

An important property of the HP 48 stack not shared by other RPN calculators' stacks is its ability to be *empty*. That is, when you clear the stack with DROP or CLEAR, there's *nothing* left. If you try to execute a command that requires arguments, you'll get an outright error--Too Few Arguments. The HP 48 makes no attempt to supply default arguments.

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

Zero is not such an obvious choice for a default value in the HP 48, since the calculator

---

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

$$\text{"ABCDEFG"} \quad 3 \quad 4 \quad \text{SUB} \quad ☞ \quad \text{"CD"}.$$

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

$$\text{"ABCDEFG"} \quad \text{DUP} \quad 3 \quad 4 \quad \text{SUB} \quad ☞ \quad \text{"ABCDEFG"} \quad \text{"CD"}.$$

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

4:	"ABCDEFG"
3:	3
2:	4
1:	"CD"

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

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

2.6.4 Postfix Commands

The HP 48 feature that perhaps takes the most adjustment by HP 41 users is the extension of *postfix* syntax to commands that use a *prefix* form in the HP 41. A postfix command is executed after its arguments are entered; a prefix command, like HP 41 STO, SF (set flag), FIX, etc., is specified *before* its arguments.

For prefix commands, the "arguments" don't go on the stack at all. When you're working with a fixed-depth stack, you can't afford to lose objects from the end of the stack whenever you store an object or set the number of display digits. Instead, these HP 41 commands use a syntax that combines the argument with the command into a single unit. For example, when you press STO , the display shows STO _ _, indicating that STO expects an argument in the form of a two-digit number (which identifies the data

register). You must then press two digit keys, following which the store is actually executed. When you include STO in a program, the STO *nn* is treated as a single program line.

One important reason for the HP 48 to abandon the prefix syntax is the inflexibility of that form. In the HP 41, STO always requires a two-digit prefix, which is a nuisance on a calculator with up to 319 data registers, since it limits direct storage to registers 00 through 99. It's not so bad for FIX, SCI, and ENG, since a single-digit argument allows you to specify all possible display formats. But since the HP 48 has 8-byte floating-point numbers, you can display up to 12 mantissa digits, so that a 1-digit prefix syntax for these commands is inadequate.

The HP 48 uses a postfix syntax for all of its commands, including those that correspond to HP 41 prefix functions. This provides:

- Consistency--all commands work the same way. You don't have to remember which commands are prefix and which are postfix.

- Flexibility--no restrictions on the number of digits or characters in the arguments. FIX, SCI, ENG, SF, CF, etc., use floating-point numbers as arguments, so they are not limited to one or two digits. STO, RCL, etc., work with variable *names* that have any number of letters.

- Computed arguments--arguments can be computed using any other HP 48 operations, as well as entered manually.

The last point means that there is no necessity in the HP 48 for "indirect" addressing as it is defined in the HP 41. Indirect addressing is the case for which a command argument is not contained as part of a prefix command syntax, but is stored in a register. The number of the register is specified as the argument for the indirect form of the command. For example, in the HP 41, the indirect command STO IND 01 means "store the number in the X-register into the register specified in register 01." You use indirect addressing when you don't know a command argument in advance, but must compute it. This often occurs when you are working with a set of sequentially numbered data registers, as in matrix operations.

To reproduce the effect of indirect addressing in the HP 48, you can define a variable named INDEX (for example) to play the role of an index register. You store the name of the indirectly referenced variable in INDEX. Then the sequence 'INDEX' RCL STO is equivalent to STO IND 01 (where we have arbitrarily chosen register 01 to be the index register) on the HP 41. Assuming that the object you want to store is already in level 1, executing 'INDEX' RCL puts the name stored in INDEX onto the stack, pushing the first object to level 2. Then STO is ready to go, with its arguments correctly

positioned on the stack.

You will probably find that there is no real need for this HP 41-style indirect addressing on the HP 48, because it provides automatically indexed structures--arrays and lists. For example, suppose you want to create a sequence consisting of the reciprocals of the integers 1 through 10. On the HP 41, you might write a program as follows:

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

This program stores the 10 reciprocals into registers 01 through 10. An HP 48 equivalent is the following program RECIP. This program returns the reciprocals together as the elements of a list object.

RECIP	Compute 10 Reciprocals
	level 1
☞	{ reciprocals }

≪ 1 10	Put loop parameters on the stack.
FOR x	Start a loop with x as the index.
x INV	Compute the reciprocal of x.
NEXT	Loop.
10 →LIST	Pack up the 10 reciprocals into a list.
≫	

Notice that in the HP 48 program, you can keep each successive reciprocal on the stack, whereas on the HP 41 you have to find a block of unused registers big enough to hold the separate results.

2.6.5 Stack-lift Disable

"Stack lift" is the RPN calculator process in which entering a new object "lifts" the previous contents of the stack into higher levels. In the HP 41, certain operations temporarily disable stack lift, so that a subsequent a no-argument operation which returns a

result to the stack overwrites the contents of the X-register with the result rather than lifting the stack. This feature, which originally derived from the memory limitations of the first RPN calculators and has carried over into later products, is not present in the HP 48.

The purpose of stack-lift disable centers around the behavior of CLX in a one-line display calculator. The first RPN calculators were characterized by very limited memory. They could not afford the luxury of having a separate register for digit entry, so numbers were keyed directly into the X-register. Usually you don't want to overwrite the number already in X, so the calculators provided that entry of a new number "lifts" the existing stack contents into higher registers. Now suppose that you start to enter an incorrect number, and start pressing ⌫ to remove it. Consider what should happen if you press the back-arrow too many times, at least once after the last remaining digit of the newly entered number. You wouldn't want that to erase the number that was previously in the X-register, so ⌫ was designed to stop erasing in this case and just enter a zero into the X-register. But since that zero has nothing to do with your calculation, and will most likely just be in the way, the zero entry is coupled with a stack-lift disable. The next number you enter overwrites the phantom zero. For similar reasons, CLX works the same way (and, in fact, ⌫ on the HP 41 just performs CLX when digit entry is not active).

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

The HP 48 eliminates the need for stack-lift disable by providing a command line, where you can enter an object without disturbing the stack during entry. Since an object in the command line is not on the stack, you can abandon it (ON , or ⌫ repeatedly to empty the line) without having to leave a zero as a placeholder on the stack. Furthermore, if you want to get rid of an object that's already on the stack, you can use DROP (⌫), which discards the object and drops the remaining objects on the stack one level.

The HP 41 commands that disable stack-lift are ENTER↑, CLX, Σ+, and Σ−. The first two commands each serve dual purposes. CLX can be used either to eliminate an unwanted number from the stack, or just to enter a zero. ENTER↑ acts either to terminate digit entry, or to copy the X-register into Y. On the HP 48, the two roles of each of these operations are separated, as shown in Table 2.2 in section 2.8. The HP 41

summation commands Σ+ and Σ− disable stack-lift in order to return a running count of the number of data points that have been accumulated to the X-register, while allowing you to overwrite the count with the next data entry if you wish. The HP 48 uses its display's status area to show both the last data point and the count when Σ+ is executed from the STAT menu, so that it does not need to disturb the stack. Furthermore, the last-point display is a property of the menu rather than any individual command like Σ+, so that the prompting information is available any time the menu is active.

2.7 The Meaning of ENTER

The unmistakable hallmark of RPN calculators has always been the double-wide key ENTER↑ that you see instead of an algebraic calculator's = key. That familiar key (minus the arrow) is still the focal point of the HP 48, but its action has been extended along with other HP 28/48 generalizations of traditional calculator RPN.

In all RPN calculators including the HP 48, the fundamental purpose of ENTER is to terminate object entry. In pre-HP 28 calculators, the only objects that can be entered directly onto the stack are real numbers, so that terminating entry means only turning off digit entry mode and leaving the completed number in the X-register. In these calculators, ENTER↑ copies the contents of the X-register into Y, and disables stack-lift. However, this second role of ENTER↑ really has no necessary connection with the first; it just happens to be the way it was first designed on the HP 35, and has remained ever since on HP RPN calculators until the arrival of the HP 28C.

In the HP 48, ENTER retains the basic action of terminating entry and entering new objects. However, because the HP 48 replaces ordinary calculator digit entry with a command line that can contain any number of objects and commands, ENTER can invoke almost any of the calculator's capabilities as well as just entering numbers onto the stack. (The secondary role of duplicating the object in level 1 is provided separately by the command DUP. To preserve more keystroke consistency with HP 41-style calculators, the HP 48 ENTER key executes DUP if you press it when no command line is present. You should recognize this as only a keyboard convenience, not a property of ENTER itself.)

The basic definition of the HP 48 operation ENTER is:

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

This is a much-elaborated version of the old "terminate-digit-entry and enter a number onto the stack," but in simple cases, it amounts to the same thing. If you press a series of digit keys, then ENTER , you will end up with a number in level 1. The same key

sequence on an HP 41 yields the same result, except that the number is also copied into the Y-register.

On the HP 41, you can also terminate digit entry by pressing any key other than a digit key. In effect, the non-digit key first terminates digit entry then executes its own key definition. But it is not correct to say that the key performs ENTER↑, because no second copy of the number is created, nor, in general, is stack-lift disabled. In the HP 48, however, many keys do literally execute ENTER, then their own definitions. This feature, called *implicit* ENTER, is provided for keystroke similarity with previous RPN calculators, and for keystroke efficiency. Pressing the ENTER key itself is called *explicit* ENTER.

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

Note, however, that the sequence ⊡ ENTER ⊡ ENTER ⊞ does not give the same results on an HP 41 and on the HP 48. The second ENTER on the HP 41 duplicates the 2, so that the ⊞ adds 2+2 and leaves the 1 in the Y-register. In general, when you key in a number, then press ENTER↑ *n* times, you get *n+1* copies of the number (up to four). On the HP 48, the same sequence produces only *n* copies--the first ENTER moves the number from the command line into level 1; each subsequent press executes DUP once and makes one copy of the number.

In general you can use ENTER on the HP 48 pretty much the same as you would on the HP 41, that is, after every data object that you key in--explicitly by pressing ENTER , or implicitly by pressing a command key. To make use of the many-objects-at-a-time capabilities of the command line, just think of the command line as a "instant" program. You write this program, execute it, and purge it all in one operation.

2.8 HP 48 Translations of HP 41 Stack Commands

Table 2.2 lists the nearest HP 48 equivalents of the HP 41 stack manipulation commands.

Table 2.2. HP 41 and HP 48 Stack Commands		
HP 41 Command	**Purpose**	**Nearest HP 48 Equivalent**
CLX	Remove last entry Enter a 0	DROP 0
CLST	Clear the stack Enter four 0's	CLEAR 0 DUP DUP2
X<>Y	Exchange X and Y	SWAP
R↑	Roll up four levels Roll up the entire stack	4 ROLL DEPTH ROLL
RDN	Roll four levels down Roll down the entire stack	4 ROLLD DEPTH ROLLD
ENTER↑	Terminate digit entry Duplicate X into Y	ENTER or any delimiter or separator character DUP
LASTX	Recover last argument Correct an error	LASTARG ←| LAST STACK
RCL X	Copy X	DUP
RCL Y	Copy Y	OVER
RCL Z	Copy Z	3 PICK
RCL T	Copy T	4 PICK
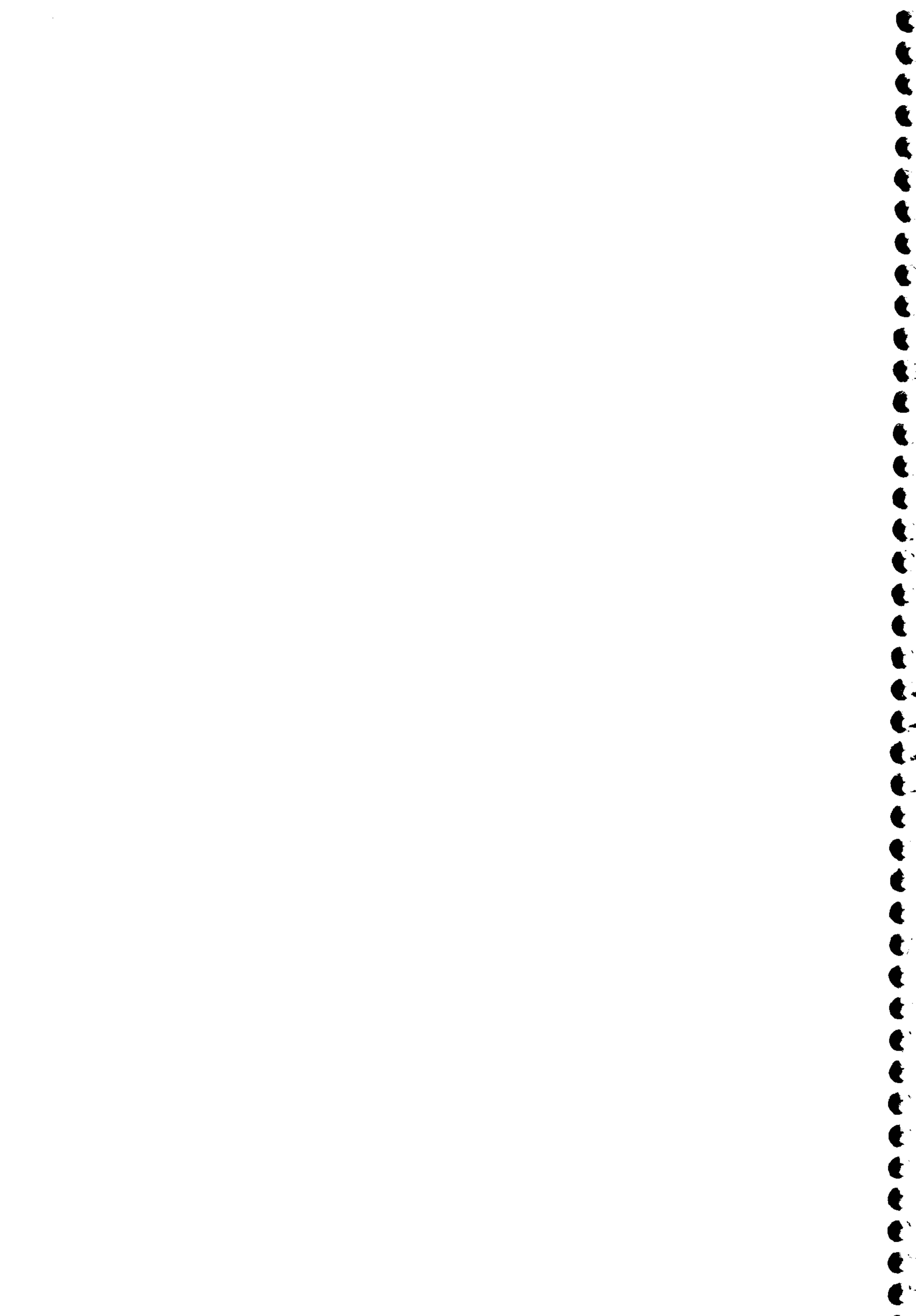
3. Variables

The HP 48 stack evolved from the fixed-size number registers of the HP 41 and its predecessors into a more flexible mechanism for working with objects of varying types and sizes. For similar reasons, the HP 48 replaces the HP 41's data registers with a general storage system in which objects are stored in named *variables*. Variables are similar to stack levels in that each holds one object of any type. But whereas the stack levels are numbered, and the order of stack objects is important, HP 48 variables are named with words, and their order is unimportant. A name is the means by which you access the object stored in a variable, and it also serves to label the variable, acting as a mnemonic form of register "number." With numbered registers, you must keep a mental or written record of what is stored in each register. By choosing appropriate variable names, you can substantially reduce the need for such records on the HP 48.

There are three kinds of HP 48 variables: *global*, *local*, and *port* variables. The discussion in this chapter focuses on global variables, which are most closely related to HP 41 storage registers. Local variables are "local" to specific programs; they are discussed in section 4.7. Port variables (*backup objects*) play a role similar to that of HP 41 extended memory registers, with the added advantage that the memory cards that contain the port variables can be used for data exchange between calculators and for archival storage.

In HP 48 terminology, a variable is "a combination of a name object and any other object that are stored together in memory." Looking at each part of this definition:

- *A combination...* A variable has two parts, the name and the other object. You can visualize a variable as a labeled box--the name is the label, the other object is the content of the box. The label is permanent, but you can change what is stored in the box.

- *of a name...* You always refer to a variable by its name. The name fills the role of the storage register number in the HP 41.

- *and any other object...* The contents of a variable can be any single object, of any size or complexity. You can only store one object in a variable, but if you want to store more, you can combine any number of objects into a list, then store the list in a variable.

- *that are stored together...* The name and the object are in fact contiguous in memory, but it is the logical combination that is significant, not the details of the storage.

- *in memory.* Global variables are all stored in a portion of memory called *user memory*, or VAR memory in connection with the VAR key. User memory can be organized into a directory system similar to that used on personal computers, by storing objects called *directories*. A *directory* is a collection of variables that is itself

an object, which you can put on the stack or store in a variable. The *home directory* is a built-in, permanent directory that contains all global variables including other directories.

HP 48 variables are an electronic realization of mathematical variables. In an expression like $x + 2$, you understand that x stands for some specific quantity (the value of x). The implication is that at any time you can substitute the value for the symbol, and actually carry out the calculation prescribed by the expression. In the HP 48, name objects are the symbols for variables. Executing a name replaces the symbol with the variable's value, which is the object stored in the variable.

You can only use an object stored in a variable by means of the variable's name. For this reason, it is often convenient to blur the distinction between *names* and *variables* by using the terms interchangeably, particularly in a mathematical context. We speak, for example, of the variables in an equation--actually, only the names are literally present in the equation. A *formal* variable in the HP 48 is a name for which no variable exists. The name potentially represents a value of some kind; the value becomes definite when you store an object in the variable.

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

3.1 Creating Global Variables

HP 41 data registers are created by SIZE, which reserves memory for a specified number of registers. Newly created registers initially contain the real number zero as a default value. You can not create or delete individual registers; commands that "clear" a register actually store a zero there. You also can not store in a register that has not been previously created by SIZE.

The HP 48 has no analog to HP 41 SIZE. A new variable is created automatically by the act of storing an object, when the name specified does not correspond to any existing variable. Global variables are created one at a time by the command STO, which takes a name and one additional object as its arguments. STO moves the object from the stack to the home directory (or the currently active subdirectory), where it and the

name are added to the current set of variables that appear in the VAR menu. HP 48 variables have no particular object type associated with them; you might create a variable initially with one type of object, but later you can store any other type of object into the same variable.

STO expects to find the name of the target variable in level 1, and the object to store in level 2. This means that the order of entering the variable name and executing STO is reversed from HP 41 style in which you press ⟨STO⟩ *before* entering the register number. Furthermore, you must *quote* the variable name in single quotes ("ticks") ' ' when you enter it; this causes the name to go onto the stack rather than recalling or executing the contents of an already existing variable with that name. To store 25 in a variable X, for example, you execute 25 'X' STO. If you forget the quotes and execute 25 X STO, you will most likely see an error message, since X will be executed rather than entered as a name, before the STO is executed. (If you know that there is no current variable X, then you can omit the quotes, since executing the name of a non-existent variable just returns the name back to the stack. This can save you a keystroke when you're creating a *new* variable.)

3.1.1 DEFINE

An alternate method of creating variables is provided by the command DEFINE (⟨⟵⟩ ⟨DEF⟩), which you can use to make simple numerical assignments. For example, to store 10 in a variable X, execute 'X=10' DEFINE. In general, DEFINE expects an equation of the form *name=value* in level 1; it stores the object *value* in a variable *name*. If you have selected numerical evaluation mode (set flag −3, or use the ⟨NUM⟩ key in the ⟨⟵⟩ ⟨MODES⟩ menu), *value* can be an expression--DEFINE evaluates the expression to a single object before storing it.

3.1.2 Renaming Variables

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

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

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

3.2 Recalling Values

There are two ordinary ways to "recall" the value of a variable:

- *Execute the variable's name.* You can do this by pressing the VAR menu key labeled with the name, by typing the name (without quotes) into the command line, or, if the name is already in level 1, by executing EVAL. Executing a global name executes the object stored in the named variable. Unless the stored object is itself a name, or a program, this just recalls the object to the stack. For example, if you have stored the number 25 in a variable named X, pressing ⬚X⬚ ⬚ENTER⬚ returns the number 25 to level 1.

- *Use RCL.* `'name' ` RCL returns the object stored in the variable *name* to the stack, without executing the object. RCL is primarily used for variables that contain program, names, and directory objects, in cases where you just want to put a copy of the stored object on the stack. For other types of objects, RCL has the same effect as simple execution of a variable's name, which requires fewer steps.

Recalling a variable's value by executing its name rather than using RCL follows directly from the mathematical meaning of a variable. For example, when you evaluate the expression A+B, you translate it into RPN order and execute the sequence A B +. In the HP 48, you can do literally that; with the values 10 and 20 stored in variables A and B, executing A B + returns the sum 30. You don't write expressions like "RCL A + RCL B", so there's no compelling reason to include the RCL's in the RPN form.

The diminished role of RCL in the HP 48 explains why it is relegated to a shifted key position. Even when you do want to use RCL, it is often more convenient to use the VAR menu keys; pressing ⬚↱⬚ followed by the menu key for a variable executes RCL. If you have trouble remembering that RCL is the right-shift and STO is the left-shift, notice that RCL is in the right-shift position above the ⬚STO⬚ key on the keyboard. The left-shift position is DEFINE, which is an alternate form of STO.

3.2.1 Recall From Stored Lists and Arrays

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

$$object \quad index \quad \text{GET} \quad ☞ \quad element,$$

where *index* specifies the element to retrieve:

- For a list or a vector, the index is either a real number or a list containing one real number.

- For an array, the index is either a real number representing the element number, counting in "row order" (left to right, top to bottom), or list of two real numbers representing the row and column numbers of the element.

The *object* in the above sequence can either be a list or an array, or the name of a variable in which a list or an array is stored. Thus,

$$\{A \quad B \quad C\} \quad 2 \quad GET \quad ☞ \quad 'B',$$

or

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

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

$$object \quad index \quad GETI \quad ☞ \quad object \quad index+ \quad element,$$

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

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

If *index* points to the last element, GETI returns either 1, { 1 }, or { 1 1 } for *index+*, as appropriate to cycle back to the first element. GETI also sets flag −64 when this occurs, or clears the flag otherwise, so that a program can easily determine when it has come to the end of a list or an array.

3.3 Altering the Contents of Variables

The most straightforward means of changing the contents of a variable is to store a new object into the variable using STO. However, there are a number of commands that let you compute a new value for a variable from its current value, without having to recall the stored object to the stack. These commands, found in the ⏩ MEMORY menu, are modeled on the HP 41 storage arithmetic commands ST+, ST−, ST∗, and ST/.

In addition to storage arithmetic, the HP 48 array commands CON, IDN, RDM, and TRN can be applied to arrays stored in variables. PUT and PUTI are the storing counterparts of GET and GETI, allowing you to alter individual elements in a stored list or array.

3.3.1 HP 48 Storage Arithmetic

Storage arithmetic commands let you combine a number in level 1 with a number stored in a variable, without having to recall the latter to the stack. For example, 25 'X' STO+ adds 25 to a number stored in X. More generally, STO+, STO−, STO∗, and

STO/ use a syntax similar to that of STO:

$$object \quad 'name' \quad STO\bullet,$$

where the $\bullet$ stands for any of the symbols +, −, *, or /. However,

$$'name' \quad object \quad STO\bullet$$

is also allowed. Either sequence combines the *object* in level 2 with the object stored in the variable *name*, leaving the result stored in the same variable. The object and the name are dropped from the stack. The two objects that are combined do not have to be numerical; any object types that are suitable for the corresponding stack arithmetic will work with the storage arithmetic commands. Note that the order of the *object* and the *'name'* on the stack is important:

- *object*   *'name'*   STO$\bullet$ computes

$$(new\ value)\ =\ (stack\ object) \begin{Bmatrix} + \\ - \\ * \\ / \end{Bmatrix} (old\ value).$$

In this case, STO$\bullet$ is equivalent to

$$DUP \quad RCL \quad ROT \quad SWAP \quad \bullet \quad SWAP \quad STO.$$

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

- *'name'*   *object*   STO$\bullet$ computes

$$(new\ value)\ =\ (old\ value) \begin{Bmatrix} + \\ - \\ * \\ / \end{Bmatrix} (stack\ object).$$

Here STO$\bullet$ is equivalent to

$$OVER \quad RCL \quad SWAP \quad \bullet \quad SWAP \quad STO.$$

With 1 stored in X, 'X' 3 STO− stores −2 in X.

You can understand these two choices by realizing that the storage arithmetic

commands combine a named object with a stack object in the same manner as if you replaced the name on the stack with the named object, then executed the stack command.

There is an ambiguity in this design when *both* stack arguments are name objects. In this case, the HP 48 interprets the level 1 name as the variable name; this arbitrary choice to match the sense of the arguments for STO was made as an easy-to-remember rule. Thus with the list { C D } stored in variable B, 'A' 'B' STO+ returns the list { A C D } to B (rather than adding or concatenating the name B to the contents of A). The rule does mean that you can not use STO+ to concatenate a name to the *end* of a list stored in a variable.

3.3.1.1 Counter Variables
INCR and DECR are specialized forms of STO+ and STO− that make it easy to use a global or local variable as a simple counter. INCR adds 1 to a real number stored in the variable specified by a name argument; DECR subtracts 1. Both commands return the result value to the stack. Thus *'name'* INCR is equivalent to the sequence *'name'* DUP 1 STO+ RCL, but executes about twice as fast.

INCR and DECR can be used in conjunction with program branch structures to replace many HP 41 uses of ISG and DSE.

3.3.2 Additional Storage Commands
In addition to the four storage arithmetic commands, the HP 48 has three storage commands that alter a stored number or array without requiring any stack argument other than the variable name. In each case, the result replaces the original object:

SNEG   negates the stored object.

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

SCONJ   computes the complex conjugate of the stored object.

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

you only need enough room for the original array.

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

CON   converts an arbitrary array into a constant array (all elements are the same), where the constant number is specified on the stack.

IDN   converts a square matrix into the identity matrix.

TRN   transposes and conjugates an array.

RDM   redimensions an array according to the dimensions specified by a list of one or two real numbers. Note that RDM can change the total size of an array if the new dimensions correspond to more or fewer elements than are in the original array.

PUT and PUTI allow you to store individual elements into an existing array or list, using a syntax similar to that of GET and GETI (section 3.2.1).

For example,

$$\{A \quad B \quad C\} \quad 2 \quad 'D' \quad PUT \quad \text{☞} \quad \{A \quad D \quad C\}.$$

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

$$'MAT' \quad \{ \quad 3 \quad 3 \quad \} \quad 25 \quad PUTI \quad \text{☞} \quad 'MAT' \quad \{ \quad 3 \quad 4 \quad \}$$

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

3.4 Purging Variables

In section 2.6.2, we discussed the differences between an "empty" stack in the HP 48, and a "clear" stack in the HP 41. Similar considerations apply to HP 48 variables compared with HP 41 storage registers.

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

An HP 48 variable, on the other hand, doesn't exist until you create it with STO.

Furthermore, when you remove a variable with PURGE, it is entirely deleted from memory. If you try to execute *'name'* RCL, when no variable with that name exists, you get a Undefined Name error message, the analog of the HP 41 NONEXISTENT message. Because variables can hold any objects, not just numbers, there is no such thing as a "clear" variable--no empty version of a purged variable left in user memory. You can certainly store zero in a variable if you want to use it as a counter or accumulator, but in general, the real number zero is no better choice as a default value than any number of other objects--an empty list, a null matrix, etc.

The ordinary use of PURGE follows the syntax *'name'* PURGE. This sequence removes the variable called *name* from user memory. If the VAR menu is active, with the *name* label showing in the display, you will see that label disappear, and the other labels move over to fill in the vacant position. As an added convenience, you can execute PURGE with a list of names: { $name_1$ $name_2$ ... $name_n$ } PURGE simultaneously removes the variables $name_1$, $name_2$, ..., $name_n$. An easy way to purge several variables is to press VAR [ ] , then press the menu key for each variable you wish to purge, followed by ENTER ⇦ PURGE . The ENTER turns off program-entry mode so that ⇦ PURGE executes PURGE rather than just adding the command name to the command line.

3.5 Grouping Variables

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

1. To save the contents of a block of registers in a second block, so that a new program can use the original registers without destroying the first program's data.

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

There are no direct equivalents for these commands in the HP 48 (see section 6.1.3). The flexibility of HP 48 variables allows you to achieve the above two purposes by using different approaches:

1. By making variable names unique to each program, there is never any reason for conflict between programs trying to use the same variables (unless the programs are deliberately exchanging data via variables). Better yet, you can use local variables (section 4.7) for temporary storage that is guaranteed to be unique to a program, without worrying at all about two programs using the same variable names.

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

3.6 The VAR Menu

The VAR menu is a visible and operational listing of the contents of user memory--the current collection of global variables. When you create a variable, its name automatically appears in the VAR menu; when you purge a variable, its name disappears from the menu. Pressing a VAR menu key automatically executes the name that appears in the menu label above the key. Also, pressing a left-shifted menu key stores an object from the stack into the specified variable; the right-shifted menu key recalls the variable contents.

New variable names are always added to the beginning of the menu, the left end of the first menu level that appears when you press VAR . In this sense, the VAR menu is a "last in, first out" arrangement similar to the stack (except that executing a name doesn't remove it from the menu). "First out" also means "found first." When a name is executed, the HP 48 searches user memory for the corresponding variable, starting with the newest variable and continuing in the reverse order of creation.

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

3.7 Port Variables

A *port variable* is a named object stored in a portion of memory configured to be separate from main global variable memory. One such region, called *port 0*, always exists in main RAM. If you have a plug-in RAM card, its memory can be merged as part of main RAM, or can be designated as separate, in which case it is called *port 1*

RAM or *port 2 RAM* according to the card slot in which it is inserted.

Port RAM plays a role much like that of extended memory in the HP 41. Extended memory lets you store data and programs for which there is not room in main RAM; however, the store/recall process is more complicated. HP 48 port RAM has important improvements over HP 41 extended memory:

- Objects are stored in port variables, which have properties very much like main memory global variables.

- Objects can be executed while they are stored in port variables. This means in particular that you can execute a program without having to copy it into main memory as you must on the HP 41.

- Plug-in RAM cards have batteries, so that you can remove the cards without loss of their contents. This makes the cards useful for memory archiving, and for information transfer between calculators.

A port variable is, like a global variable, a combination of name and an object stored together. Actually, the name and object are stored inside yet another object, called a *backup object* (similar to the way variables are stored inside of directory objects). In addition to the name and object, a backup object contains a checksum and a (memory) length, which the calculator uses at various times to verify that the backup object's contents are valid. In some circumstances (for example, when you restore an archived copy of main RAM from a personal computer), you can see a backup object on the stack, but usually backup objects remain in port memory where you store and recall their contents. We use the term *port variable* to refer to those contents, in order to stress the similarity of access to that of global variables. (The HP 48 Owner's Manual does not use this term, preferring to speak only of backup objects. This, however, can lead to confusion between the backup object itself and the object that is stored in it.)

Access to the contents of port variables is provided by means of *port-names*, which are ordinary global names that are *tagged* with a port number 0, 1, or 2. If you execute 123 LENGTH STO, the number 123 is stored in the global variable LENGTH. To store the number instead into a port variable in port 0, you use

<div align="center">123 :0:LENGTH STO.</div>

RCL, EVAL, and PURGE also work with port-names. Furthermore, for these last three commands, you can also use a "wild-card" port designator, the character "&" (α ← ENTER). Executing :&:LENGTH RCL searches port 2, port 1, port 0, and finally global variable memory, until it finds a variable LENGTH, then returns its contents.

Because port variables are primarily intended for archiving objects, the HP 48 deliberately makes it a little more difficult to change their contents than is the case for global variables:

- You can't use STO to store a new object in an already existing port variable. You must first use PURGE to remove the existing variable.

- You cannot purge a port variable while its contents are in use. "In use" means that the object has been copied to the stack, is stored in a local variable, or is part of a running or halted program. Thus if you want to purge a port variable, while keeping a copy of the stored object, you must not only recall the object before purging, but you must also store the copy somewhere other than on the stack before the purge will succeed. (You can also copy the object to the stack and execute NEWOB, which creates a new copy that is independent of the original stored version.)

Attempting to break either of these two rules results in the Object in Use error message.

4. HP 48 Programming Principles

The simplest kind of calculator programming consists of "keystroke capture," in which sequences of keystrokes are recorded in calculator memory instead of being executed immediately. The keystrokes are the same as those you would use if making a calculation once. Each set of recorded keystrokes is called a *program*, and can be replayed automatically any number of times. The first programmable hand-held calculator, the HP 65, was programmable in this style, and subsequent HP calculators have preserved the same general approach even as their programming capabilities increased.

The appeal of keystroke-capture, of course, is that once you learn the manual operation of a calculator, you don't need to learn anything new in order to program it. (Here we use the term *manual operation* to indicate the interactive use of the calculator where you press keys to select each individual operation.) However, a calculator's ability to execute a sequence of steps rapidly makes it practical to carry out calculations that would be too extensive or complicated to perform with simple manual operations. For this reason, even the HP 65 augmented its keystroke-capture programming with additional programmable operations that have little purpose in manual operation, such as logical tests, branches, and labels. The HP 41 provides the same set of program control commands as the HP 65, with the addition of a few more conditional tests such as ISG, DSE, and flag operations.

The HP 48 takes a different approach to the idea of keystroke capture programming. As we noted earlier (section 2.8), the HP 48 command line can contain any sequence of commands and objects, which are executed together when you press ENTER . You can perform a calculation either by pressing a series of immediate-execute command keys, or by pressing ENTRY first, accumulating the commands in the command line, then pressing ENTER --obtaining the same results either way. Entering a command into the command line delays its execution so that you can execute it as part of a sequence of other commands and objects; but this is just what programming is. In effect, the command line is a program that you write and execute immediately. All that is required is a means of saving the command line/program so that it can be re-executed repeatedly. This is provided by means of *program objects*.

Creating an HP 48 program object consists of entering a sequence of objects and commands enclosed within << >> delimiters. These program delimiters instruct the calculator to combine the objects and commands into a program object that is entered unexecuted onto the stack. You can then name the program by storing it in a variable. By doing so, you effectively extend the calculator's command set, since you can subsequently use the variable name just as you would a built-in command. Imagine, for example, that you have created two program objects named DOTHIS and DOTHAT.

Then if you want to create a program that performs both of the tasks done by DOTHIS
and DOTHAT, you just enter << DOTHIS DOTHAT >>, perhaps naming it DOBOTH.
This process is unlimited--you can use DOBOTH as an element of another program.
DOTHIS and DOTHAT themselves may be combinations of other programs' names. As
a matter of fact, the HP 48 commands that you use in your programs are themselves
programs written the same way, stored in the calculator's permanent memory (ROM).

We are using the term *sequence* to mean a series of objects and commands that are exe-
cuted in order. However, there are also command line and program entries that are not
objects, so we need to extend our definition of *sequence* for subsequent discussions to be
any series of objects *and* other program entries that can "stand alone," and can consti-
tute a program by itself if surrounded by << >> delimiters. A sequence can be an
entire program, or part of a program.

The non-object "entries" are *program structure words*, such as FOR, DO, →, END, etc.
These are not objects, because you can't put them on the stack or execute them indivi-
dually. You can only use them in certain specific combinations, like FOR...NEXT, or
IF...THEN...END. A complete combination, including the objects between the program
structure words, is called a *program structure*.

For example, in

$$<< \ 1 \ 2 \ \text{IF} \ A \ \text{THEN} \ B \ C \ \text{END} \ D \ >>$$

1 2 is a sequence, B C is a sequence, and 1 2 IF A THEN B C END D is a
sequence. IF, IF A, and IF A THEN are not sequences, because they contain incom-
plete program structures--you can not enter these by themselves without obtaining a
Syntax Error message.

4.1 Program Basics

The basic structure of an HP 48 program is very simple:

$$<< program \ body >>.$$

The << and >> are the program object delimiters that serve to identify this object as a
program. *Program body* is the sequence of objects and program structures that make up
the logical and computational definition of the program.

4.1.1 The ≪ ≫ Delimiters

The ≪ and ≫ that surround HP 48 programs serve a dual purpose. First, they are the delimiters that identify an object as a program. When you enter a program into the command line, the ≪ tells the HP 48 to create a program object from all of the objects, commands, names, etc., that follow, up to the next matching ≫. For every ≪, there is always a ≫; the ≫ ends the definition of the program started by the preceding ≪. When the HP 48 displays a program object after it has been created, the ≪ and ≫ identify the object to you as a program.

The second role of these delimiters is to postpone execution of a program sequence. When ≪ is encountered in program or command line execution, it is interpreted by the HP 48 to mean "put the following program object on the stack." This behavior of ≪ allows you to include programs within other programs:

$$\ll \quad object \quad \gg \quad \text{EVAL}$$

executes *object*, but

$$\ll \quad \ll \quad object \quad \gg \quad \gg \quad \text{EVAL}$$

leaves the program ≪ *object* ≫ on the stack.

The ⌊≪⌋ and ⌊≫⌋ keys are the closest analog the HP 48 has to the more traditional program mode keys you find on other calculators (PRGM on the HP 41). On the HP 48, instead of pressing a program-mode key to start program entry, you press ⌊◁⌋ ⌊≪ ≫⌋ , which enters a pair of program delimiters into the command line. The key also activates program-entry mode, indicated by the PRG annunciator, in which pressing a command key types the command's name in the command line instead of executing the command. Pressing ENTER after entering the program's objects terminates program entry, and enters the program object on the stack. You can even think of ⌊◁⌋ ⌊≪ ≫⌋ as a *programmable* PRGM key, since you can include those delimiters in programs, allowing programs themselves to enter new programs.

4.1.2 The Program Body

The *body* of an HP 48 program, that is, everything between the ≪ and the ≫, can consist of any combination of objects and program structures:

• Data objects;

• Quoted names, which go on the stack like data;

• Unquoted names--which act like user-defined commands;

- Commands;

- Program structures--loops, conditionals, and local variable structures.

In general, when a program is executed, all of the items from the above list that constitute the program body are executed sequentially. The nominal order of execution is start-to-finish, or "left-to-right, top to bottom" in the command line order in which the program was entered originally. Within a program structure, there may be repetitive loops or conditional jumps. Of course, there's nothing remarkable about this program flow--any programming language exhibits similar orderly execution.

The simplest programs are those which contain no program structures. Such programs contain only objects to be executed one after the other, starting with the first object after the $\ll$, and ending with the last object just before the $\gg$. Such programs are very easy to create: all you do is

1. Press the $\boxed{\Leftarrow}$ $\boxed{\ll \gg}$ key;

2. Press the keys for, or spell out, the objects you want the program to execute, in the same order used when you perform the calculation manually; then

3. End the program entry by pressing $\boxed{\text{ENTER}}$.

4. To name the program, enter a name (quoted) and press $\boxed{\text{STO}}$. You can consider the resulting variable as a named program.

To "run" a named program, you execute the program's name, either by pressing the appropriate VAR, CST, or LIBRARY menu key, or by typing the name into the command line and pressing $\boxed{\text{ENTER}}$. If the program itself is in level 1, you can run it by executing $\boxed{\text{EVAL}}$.

Examples:

1. $\ll$ 1 2 3 $\gg$ 'P123' STO   creates a program named P123 that enters the numbers 1, 2, and 3 onto the stack. The equivalent HP 41 program is

```
01  LBL "P123"
02  1
03  ENTER↑
04  2
05  ENTER↑
06  3
07  END
```

2. << 2 / SIN >> 'HSIN' STO   creates a program named HSIN, that returns the sine of 1/2 times the number in level 1. On the HP 41:

```
01  LBL "HSIN"
02  2
03  /
04  SIN
05  END
```

3. << + + SQ >> 'SUMSQ' STO   creates SUMSQ, which adds three numbers from the stack and squares the result. The HP 41 version:

```
01  LBL "SUMSQ"
02  +
03  +
04  X↑2
05  END
```

You can alter the basic start-to-finish execution flow of programs by adding program structures that define branches and loops. *Branches* are forward jumps in a program, that cause program sequences to be skipped. *Loops* contain backward jumps, which cause program sequences to be repeated one or more times. These structures are described later in this chapter.

Notice that HP 48 ENTER never appears in a program, unlike HP 41 ENTER↑. As we discussed in section 2.8, HP 48 ENTER means "execute the command line," which has no meaning in a program. When you enter consecutive numbers into a program, just use a space to separate them. For other types of objects, their delimiter characters serve to separate the objects--no space is necessary before, after, or between delimiters.

4.1.3 Structured Programming

A property of HP 48 programs that is common among many computer languages, but may be unfamiliar to HP 41 programmers, is their well-determined "entrance" and "exit." That is, in any HP 48 program there is only one point--the first object after the <<--where execution can begin. Similarly, there is only one exit, or point at which a program completes execution. A diagram to represent the execution flow in and out of an HP 48 program is very simple:

IN ——→ | << | *Body* | >> | ——→ OUT
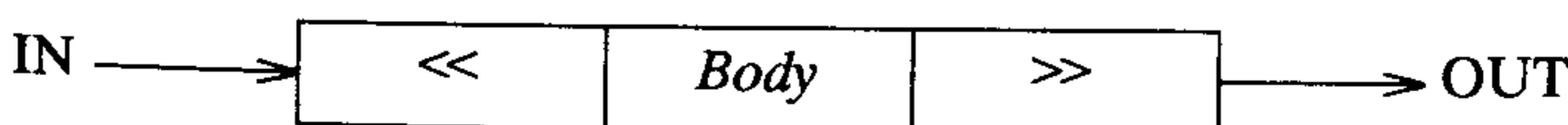
Contrast this diagram with one that illustrates a possible program flow in an HP 41 program:

```
                    |
                    v
        +-------------------+
        |     ...steps...   |
        +-------------------+
  ---->  |      LBL 01       |
        +-------------------+
        |     ...steps...   |
        +-------------------+
        |      GTO 03       |  ----->
        +-------------------+
        |     ...steps...   |
        +-------------------+
  ---->  |      LBL 02       |
        +-------------------+
        |     ...steps...   |
        +-------------------+
        |       END         |
        +-------------------+
                    |
                    v
```
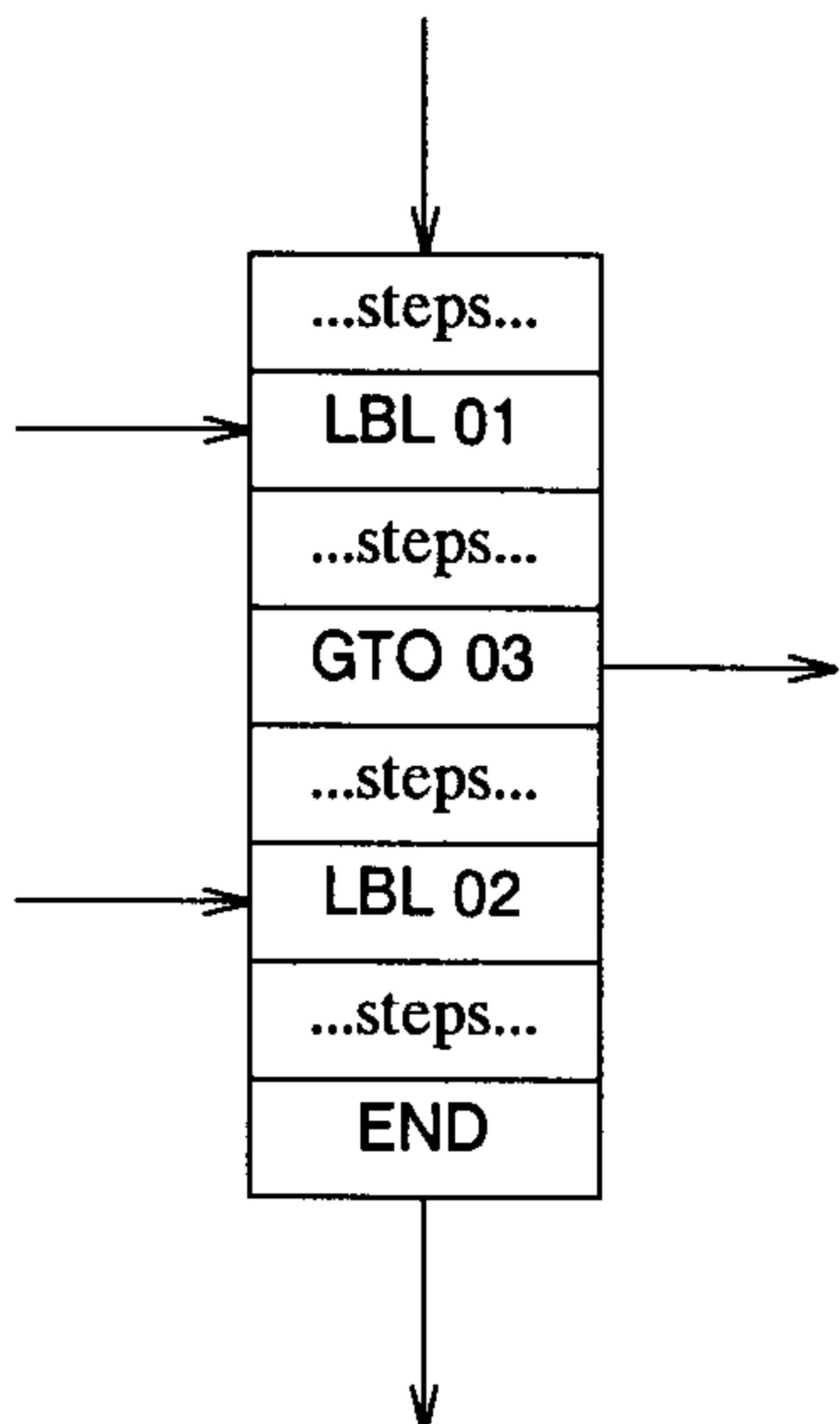
There is no restriction on the number of entrances and exits in an HP 41 program. The principal program entries that make this possible are *labels* and GTO commands. A GTO (go to) is an unconditional jump to a label with no return. With labels and GTO's, program execution can jump around from program to program, in and out of portions of programs, or round and round within a single program. At first glance (and more, if you're used to programming this way), this capability seems like an advantage. You may wonder why the HP 48 does not provide the same capability.

The answer is that the HP 48 is designed for *structured programming*. Structured programming consists of writing small programs as building blocks, or modules, from which bigger programs are assembled as series of subroutine executions. A *subroutine* is a program that is executed, or *called*, from within another program, and which returns to the original calling program when it is finished. Bigger programs themselves may become subroutines for even bigger programs, and so on. Each program, at every level, has a single entrance and exit; there is no jumping in and out of programs at intermediate points. Structured programming has the following advantages:

• Programs are easy to write. Each program can be designed to fulfill a single task, and can thus consist of relatively few steps. If a program gets too long, you just

divide it into smaller programs.

- Programs are easy to decipher. By choosing meaningful names for subprograms, you can read a program almost as text. For example, a program might look like this:

```
<<  GETINPUT  DOMATH
    IF  TOOBIG
    THEN  DISCARD
    ELSE  SAVE
    END
>>.
```

It is easy to understand what this program does. It gets input (GETINPUT), then does some calculations (DOMATH) on that input. Next, it checks a result to see if it's too large (IF TOOBIG); if so, it discards the result (THEN DISCARD), otherwise saves it (ELSE SAVE). At this level, you can see the overall structure of the program. To see more detail, you can examine the individual subroutines. For example, TOOBIG must be a program that tests one or more of the results returned by DOMATH, and returns a *true* flag (see section 4.3) if the results are too big according to some criterion. TOOBIG might be something like this:

```
<<  DUP2  +  LIMIT  >  >>.
```

This program makes copies of two numbers in levels 1 and 2, then adds them and tests to see if the sum is greater than the value of LIMIT (which might be a number, or another calculation to perform, etc.).

- Programs are easy to alter. In the above example, you can completely change the internal definition of TOOBIG, without worrying about the main program. All you have to do is ensure that TOOBIG works the same from an external point of view--it must take the right number of objects from the stack, and return the right number, etc. Similarly, you can change the value of LIMIT from a specific number to a program that computes a result, without any change in the design of TOOBIG.

In a programming language that permits GTO's into the middle of a program, any modification of a program must ensure that the correct entry conditions are met at any point at which execution can start. This is especially difficult to manage in languages like BASIC, where a GTO can jump to any line in a program, with no label or other indication to remind the programmer that execution may start at that line.

- Programs can be written without any regard to the internal behavior of programs that call them, or programs that they may call. All that matters about a program is its input and output, not the steps that it uses in its execution.

The last point is a key concept in HP 48 structured programming. A program is defined externally only in terms of its input and output:

1. The number and type of objects it takes from the stack;

2. The number and type of objects it returns to the stack;

3. The variables that it uses;

4. Flags that are tested or changed.

From the point of view of one program calling another as a subroutine, the first program doesn't have to care *at all* about how many stack levels or additional subroutine returns are needed by the subroutine. It just has to be sure to provide the correct inputs for the subroutine, and know where to find the results returned by the subroutine (usually on the stack). A program that calls a subroutine can also depend on having program execution return to it after the subroutine is finished, no matter how many other sub-subroutines are called by the subroutine.

4.1.4 Comparing HP 48 and HP 41 Programs
HP 41 programs have this general form:

```
01  LBL A
02  program line
03  program line
...
nn  RTN (or END)
```

HP 41 programs aren't required to start with a label, but it is most common to include a label as the first step, and to start execution at the beginning of a program, by using that label. Programs always finish with a RTN or an END. A single program may contain multiple RTN's--different programs are separated by END's.

If you match up the parts of HP 41 and HP 48 programs, you may observe that:

- The HP 41 line 01, and the label that marks the start of a program, are replaced in the HP 48 by the ≪ delimiter.

- The program steps or lines that make up an HP 41 program body are replaced by the objects that define the HP 48 program.

- The END, or last line of an HP 41 program, which acts as a subroutine return when the program is called as a subroutine, is replaced by the HP 48 >> delimiter.

HP 41 program lines or steps are always numbered. The line numbers help show the program flow unambiguously, and are useful in moving a program counter to specific points in a program for editing or single-stepping. The line numbers are artificial in the sense that they are not stored as part of a program (they are created as needed as part of the program mode display), and have nothing to do with a program's execution while it is running. All program branching is accomplished by GTO's or XEQ's to labels that are explicit program lines.

HP 48 programs have no line numbers. Whether this is an advantage or a disadvantage is a matter of taste. Since line numbers have no purpose during execution, showing them as part of a program can be considered as a superfluous complication. On the other hand, in a large program, line numbers can help you keep track of where you are looking in a program. The HP 48 does not show line numbers to permit more convenient editing of a program, and to permit the display of as many objects as possible on the screen at any time during editing.

Up to a point, you can write HP 41 programs in structured form, by avoiding the use of GTO's. Of course, you can't eliminate GTO's entirely, because the HP 41 doesn't provide any program structures in the HP 48 sense. But you can preserve a structured form by avoiding intertwined branches and loops. For example, to treat the program sequence

$$\begin{array}{ll} 01 & \text{LBL } 01 \\ 02 & \\ \dots & \\ 98 & \text{ISG } 00 \\ 99 & \text{GTO } 01 \end{array}$$

as a "structure," you must make sure that there are no labels between the LBL 01 and the GTO 01, so that program execution can never jump into the middle of the sequence.

The fixed size stacks available on the HP 41 makes truly structured programming in the style of the HP 48 rather difficult. The HP 41 has only a 6-level subroutine return stack, so that any time you write one program that calls another, you must verify that the subroutine does not itself call other routines (and so on), to such an extent that the return stack overflows and execution never returns to the original program. The four-register data stack produces a similar limitation: a program can't arbitrarily leave data

on the stack when calling a subroutine, in case the subroutine uses so many stack levels that the calling program's data gets pushed off the stack. After writing a program, if you later decide to modify one of its subroutines, you may also have to change the calling program if the new version of the subroutine uses an additional stack register.

Structured programming is not just a matter of programmer style in the HP 48--you have no option. The HP 48 won't let you write an unstructured program. There is no GTO command, and all branching and looping is accomplished by means of well-defined structures.

Table 4.1 matches various programming concepts in the HP 41 with their analogs in HP 48 programs. All of these topics are discussed in subsequent sections.

Table 4.1. HP 41 and HP 48 Programming Analogs

HP 41	HP 48
Program mode	Alpha-entry mode, command line
Program file	Program object
Global label	Program name
Local label	None
Line number	None
GTO *label*	None
Subroutines	Named programs
XEQ *label*	Execute by name or EVAL
RTN	>>
END	>>
ISG, DSE	DO...UNTIL...END
	WHILE...REPEAT...END
	FOR...NEXT
	FOR...STEP
	START...NEXT
	START...STEP
Flag 25 error handling	IFERR...THEN...ELSE...END
Test and Skip	IF...THEN...ELSE...END
	CASE...THEN...END...END
PROMPT	PROMPT
STOP	HALT
	INPUT
R/S	← CONT

4.2 Program Structures

A simple HP 48 program consisting of a sequence of objects can be broken into two or more programs at any point in the sequence. For example, the program

$$\ll \ 5 \ * \ 6 \ + \ 10 \ - \ \gg$$

is equivalent to the two programs

$$\ll \ 5 \ * \ \gg \ \ll \ 6 \ + \ 10 \ - \ \gg$$

executed consecutively.

A *program structure* is a program sequence that can not be broken into stand-alone sections. For example, the program

$$\ll \ \to \ x \ \ '2*x+3' \ \gg$$

can *not* be divided like this:

$$\ll \ \to \ x \ \gg \ \ll \ '2*x+3' \ \gg.$$

The first part would give a Syntax Error message if you entered it. Similarly, you can't break

$$\ll \ 1 \ 5 \ FOR \ n \ n \ SQ \ NEXT \ \gg$$

into

$$\ll \ 1 \ 5 \ FOR \ \gg \ \ll \ n \ n \ SQ \ NEXT \ \gg.$$

The FOR and the NEXT must be in the same program.

Program structures are defined by *program structure words*. These words are special command line words that do *not* represent objects, but cause other objects to be combined into structures. The structure words always appear in specific combinations that define complete structures. Table 4.2 lists all of the HP 48 program structures and their uses.

Before studying the various program structures further, we need to describe HP 48 test commands and flags, which are key concepts in understanding the execution of program structures.

Table 4.2. HP 48 Program Structures

Structure	Type	Typical Use
IF...THEN...ELSE...END	Conditional	Program decisions
CASE...THEN...END...END	Conditional	Selecting among multiple choices.
START...NEXT/STEP	Definite Loop	Execute a sequence a specified number of times.
FOR *index* ... NEXT/STEP	Indexed Definite Loop	Execute a sequence once for each value of an index.
DO...UNTIL...END	Indefinite Loop	Repeat a sequence until a condition is satisfied.
WHILE...REPEAT...END	Indefinite Loop	While a condition is satisfied, repeat a sequence.
→ *...names... procedure*	Local Variable Structure	User-defined functions. Creating local variables.
IFERR...THEN...ELSE...END	Error trap	Handling expected and unexpected command errors.

4.3 Tests and Flags

A calculator program "asks a question" by executing a *test* command. A test command is any command that in effect returns "true" or "false" as a result, which then can be used to choose a particular program branch to execute. The HP 48 differs from the HP 41 in that HP 48 tests return stack results called *flags*, whereas HP 41 test commands include immediate branching based on the test result.

In HP 48 terminology, the word *flag* has a dual meaning. One meaning is the traditional one inherited from the HP 41, where a flag is one of a group of numbered

memory locations that are used to store logical *true* or *false* values. A "memory location" in this context is just a binary bit; if the bit is 1, the flag is *true*; if it is 0, the flag is *false*. A *user flag* is one that can be set (made *true*), cleared (made *false*), or tested by by means of commands. A *system flag* is one reserved for use by the calculator operating system, and which can only be tested by the user, not set or cleared. The HP 41 has 56 flags, of which those numbered above 29 are system flags that can only be tested in programs. The HP 48 has 128 flags, numbered from -64 to $+63$; all can be set or cleared as well as tested. In both the HP 41 and the HP 48, some of the flags represent modes, such as the angle mode and the beeper enable/disable. Changing the state of one of these flags changes the corresponding calculator mode, and vice-versa.

The HP 48 introduces a second meaning to *flag* that has no equivalent in the HP 41. An HP 48 flag can be a true/false value independent of any number memory location. Specifically, a flag is represented on the stack by a real number, so that flags can be used as arguments or returned as results by commands. The HP 48 conventions for real number values used as flags are:

- As arguments to commands, 0 means *false*; any non-zero real number means *true*.

- When a command returns a flag result, 0 again means *false*; the value 1 means *true*.

With these ideas in mind, we can make the following definitions:

Test:	A command that returns a flag to the stack. Examples: SAME, ==, FS?
Logical operator:	A function that makes a logical combination of two flags (AND, OR, XOR), or inverts a flag (NOT), and returns a new flag.
Conditional:	A program structure that includes a structure word that uses a flag as an argument, and causes a program branch according to the flag's value. HP 48 conditionals are CASE...END, IF...THEN...(ELSE...)...END, DO...UNTIL...END, and WHILE... REPEAT...END.

In the HP 41, all test commands combine a test and a branching operation. If the test is *true*, one choice of branch is made; if *false*, another choice is made. For example, when a test such as FS? (flag set?) or X=Y? is *true*, the program line immediately following the test is executed. If the test is *false*, that next line is skipped.

In the HP 48, a test and the corresponding conditional branch are separate operations. To permit this separation, a test command returns its result as a (real-number) flag on the stack, which can then be manipulated like any other stack object. Consider a typical test command, >. > compares real numbers in levels 1 and 2: if the number in level 2

is greater than that in level 1, $>$ returns 1 (*true*); it returns 0 (*false*) if the level 2 number is equal or smaller. For example, to compare the values of X and Y in a program, you can use the sequence

$$X \quad Y \quad >.$$

This returns 1 (*true*) if X is greater than Y, or 0 (*false*) otherwise.

In a conditional structure, one particular structure word actually makes the branch decision, taking a flag from the stack for this purpose:

- the THEN in IF...THEN...(ELSE...) END (section 4.4.1).

- the THEN in CASE...THEN...END...END (section 4.4.2)

- the END in DO...UNTIL...END (section 4.5.2.1).

- the REPEAT in WHILE...REPEAT...END (section 4.5.2.2).

But note that you can include any number of intervening objects and commands between the point at which the flag is put on the stack, and the structure word that uses the flag for a branch decision. This separation of tests and decisions makes possible the use of logical operators to combine flags. For example, the logical operator AND takes two flags from the stack and returns a true flag if both of the original flags are *true*, and a false flag otherwise. The sequence

$$X \quad Y \quad > \quad Y \quad Z \quad > \quad AND$$

returns *true* only if X is greater than Y, *and* Y is greater than Z. Furthermore, since the logical operators and most tests (except SAME) are functions, you can rewrite the above sequence in a more legible manner:

$$'X>Y \quad AND \quad Y>Z' \quad \rightarrow NUM.$$

The $\rightarrow$NUM converts the algebraic expression into a real number suitable for use as a flag. It is not necessary if the symbolic expression is used as an argument for a conditional--the conditional automatically performs the numerical evaluation of the expression.

Suppose you want to write a program that returns the sum of two numbers if they are both greater than or equal to 1, and otherwise returns the difference. In HP 41 language, the program might look like this:

01	LBL"SUMDIFF"	
02	1	
03	X>Y?	Is the second number < 1?
04	GTO 01	If so, go to LBL 01.
05	RDN	If not, check the other number.
06	X<>Y	
07	R↑	
08	X>Y?	Is the first number < 1?
09	GTO 02	If so, go to LBL 01.
10	RDN	Drop the 1.
11	+	Add the two numbers.
12	RTN	
13	LBL 02	The first number is < 1.
14	RDN	
15	X<>Y	Restore the original order.
16	R↑	
17	LBL 01	One or both numbers are ≤ 1.
18	RDN	Drop the 1.
19	−	Compute the difference.
20	RTN	

In this program, there are separate tests and separate branches for each of the two numbers. The two branches have to be to different destinations (LBL 01 and LBL 02) because the stack is in a different configuration at the time of the tests. In the equivalent HP 48 program, the tests are logically combined before any branching is done:

```
<<
  DUP2                          Makes copies of the two numbers.
  IF
    1  ≥                        Test the first number.
    SWAP  1  ≥                  Test the second number.
    AND                         Are both tests true?
  THEN  +                       ...then add.
  ELSE  −                       ...otherwise subtract.
  END
>>
```

Notice how easy it is to read the HP 48 program compared to the HP 41 version. The two tests are right next to each other between the IF and the THEN. The two possible branches then follow immediately after the AND that combines the tests. You can also write this program using local variables and an algebraic expression, using the IFTE function (section 4.4.1.1):

$$\ll \quad \rightarrow \text{ x }\text{ y }\text{ 'IFTE (x>1 AND y>1, x+y, x−y)' } \quad \gg$$

You can think of HP 48 user flags as a kind of variable: the flag number is the variable name, the number 1 or 0 is the value. FS? plays the role of RCL for a user flag--it transfers the flag value to the stack. You use SF and CF to store the values 1 and 0, respectively, into a user flag. There's no single command to store a stack flag directly into a user flag, but the sequence

IF   SWAP   THEN   SF   ELSE   CF   END

will accomplish that, where the flag number is initially in level 1 and the new flag value is in level 2.

One consequence of using real numbers as flags for conditionals is that it is easy to test a real number to see if it is zero. In the sequence

IF   X   0   ≠   THEN   A   ELSE   B   END,

the 0 ≠ is superfluous; you can rewrite the sequence as

IF   X   THEN   A   ELSE   B   END.

4.3.1 HP 48 Test Commands

The HP 48 test command set is comparable to that found in the HP 41 and most other languages: it consists of numerical and text comparisons for equality, inequality, and order, plus user flag tests. But it is important to emphasize the difference in the argument order for the HP 48 tests compared with their HP 41 counterparts.

The order of test arguments in the HP 48 is chosen to be consistent with the argument order for other HP 48 functions: the arguments are entered onto the stack in the same order as they appear in algebraic expressions. For example, consider the "greater-than" operator >. In an expression, "is A greater than B?" is written as "A>B". A is the first argument, reading left-to-right; B is the second. The comparison is *true* if the first argument is greater than the second. If you rewrite the infix operator > in RPN, the expression becomes A B >, which indicates that A is entered into the stack before B. When > executes, A should be in level 2, and B in level 1.

This is the reverse of the order of HP 41 tests. X>Y? in the HP 41 means "is X (level 1) greater than Y (level 2)?" Therefore, when translating HP 41 programs to the HP 48, you must be careful to use the opposite tests in cases where the order of arguments is important.

Table 4.3. HP 48 and HP 41 Test Commands		
HP 48 Test	**Meaning**	**HP 41 Equivalent**
<	Less than?	X>Y?
≤	Less than or equal to?	X> =Y?
>	Greater than?	X<Y?
≥	Greater than or equal to?	X< =Y?
= =	Equal to?	X=Y?
≠	Not equal to?	X≠Y?
SAME	Object same?	X=Y?
FS?	User flag set?	FS?
FC?	User flag clear?	FC?
FS?C	User flag set?--clear	FS?C
FC?C	User flag clear?--clear	FC?C

4.3.2 SAME, = =, and =

It is important to distinguish carefully between the three HP 48 commands SAME, = =, and =, which may appear to have similar meanings. The first point to note is that = is *not* a test command, so it is fundamentally different from the other two commands, which are tests. = is a *function* that creates an equation from two expressions. Its execution does not return a flag; in symbolic evaluation mode, it does nothing other than evaluate its arguments. In numeric evaluation mode (including using →NUM) it is equivalent to subtractions (-), returning the numerical difference of the two sides of the equation.

= =, on the other hand, is a *test*, and always returns a flag when executed. = = is primarily intended for ordinary numerical equality comparisons. You can use = = in algebraic expressions as an infix operator, just like <, >, etc. = = and = must have different names to distinguish their quite different meanings, and to prevent ambiguity within algebraic expressions. You can think of A=B as an "assertion," whereas A= =B is a "question."

SAME is very similar to = =; in many cases you can use them interchangeably. Other than the fact that SAME is an RPN command that is not allowed in algebraic objects, the two commands differ only in the manner in which they deal with algebraic and binary integer objects:

- = = operates on algebraic objects like any other function, returning a symbolic result when appropriate. SAME compares the original objects themselves, always returning a flag. Thus, '1+2' 3 = = returns the *expression* '1+2= =3' (which evaluates to a

-65-

true flag), whereas '1+2' 3 SAME returns a *false* flag.

- When comparing binary integers, == ignores leading zeros and compares only the numerical values, so that the relative wordsize of the two integers does not matter. For SAME to return a true flag, the two integers must have the same wordsize as well as the same value.

4.4 Conditional Branches

4.4.1 The IF Structure

Many programming problems require a program to make simple decisions: "If this is true, do that--otherwise do something else." On the HP 41, this is most commonly handled by the combination of a test command followed by a GTO: if the test is true, the GTO is executed, so that execution jumps to a label; otherwise, execution continues immediately after the GTO. The HP 48 approach to this kind of branching is embodied in the *IF structure*, a program structure that has the general form:

IF *test-sequence* THEN *then-sequence* ELSE *else-sequence* END

You can read this structure as "if *test-sequence* is *true* (returns a true flag), then execute *then-sequence* and jump past the END. If *false*, skip the *then-sequence* and execute *else-sequence*."

The ELSE *else-sequence* portion of the structure is optional; for cases where "do something else" is just "do nothing," you can use:

IF *test-sequence* THEN *then-sequence* END,

which translates to "If *test-sequence* is *true*, execute *then-sequence*; otherwise, skip past the END."

■ *Example*. Test a user flag specified by its number in level 1, and display YES if the flag is set, or NO if it is clear:

```
IF   FS?
THEN   "YES"
ELSE   "NO"
END
1   DISP
```

This sequence is very easy to read, more so than the equivalent HP 41 sequence, in

which GTO's and LBL's break up the visual flow:

```
01   FS? IND X        IF the flag is set...
02   GTO 00
03   "NO"             ELSE "NO"
04   GTO 01
05   LBL 00
06   "YES"            THEN "YES"
07   LBL 01
08   AVIEW
```

■ *Example.* Order two numbers so that the smaller one is returned in level 1, the greater in level 2.

Here the HP 41 has the advantage of simplicity, since no GTO is required, and the test does not remove the stack arguments:

```
01  X>Y?
02  X<>Y
```

The HP 48 version:

DUP2	Copy the two numbers.
IF <	Test if the first is less than the second.
THEN SWAP	If so, switch the numbers.
END	

Because it is THEN that actually removes a flag from the stack and makes the branch decision, the position of the IF in the sequence that precedes THEN is unimportant:

```
1  2  IF  >  THEN  ...,  and
1  2  >  IF  THEN  ...,  and
IF  1  2  >  THEN  ...,
```

all produce the same result. You can choose to position the IF wherever you want to make a program the most readable. (The most memory-efficient form has a single object between the IF and the THEN. Thus of the three forms above, the first uses the least memory.)

4.4.1.1 Command Forms of IF

An alternate means of achieving IF structure branching is provided by the IFTE and IFT commands. For these commands, the various sequences included in an IF structure are entered as stack arguments, either as single objects or as programs. That is,

$$\textit{test-sequence} \ll \textit{then-sequence} \gg \ll \textit{else-sequence} \gg \text{IFTE}$$

is equivalent to

$$\text{IF } \textit{test-sequence} \text{ THEN } \textit{then-sequence} \text{ ELSE } \textit{else-sequence} \text{ END.}$$

Similarly,

$$\textit{test-sequence} \ll \textit{then-sequence} \gg \text{IFT}$$

is equivalent to

$$\text{IF } \textit{test-sequence} \text{ THEN } \textit{then-sequence} \text{ END.}$$

To use IFTE, you put a flag in level 3, an object (usually a program) representing the *then-sequence* in level 2, and an object representing the *else-sequence* in level 1. IFTE tests the flag; if the flag is *true* (non-zero), the *else-sequence* is dropped, and the *then-sequence* is executed. If the flag is *false* (zero), the *then-sequence* is dropped, and the *else-sequence* is executed. IFT works much the same way: the flag must be in level 2, and a *then-sequence* in level 1. If the flag is *true*, the *then-sequence* is executed, otherwise it is dropped.

- *Example.* Split a real or a complex number into its real and imaginary parts.

RC→R		Real/Complex-to-Real		8A8F
level 1		level 2	level 1	
x	☞	x	0	
(x,y)	☞	x	y	

« DUP TYPE	Get the input type.
« C→R »	Complex case (type ≠ 0).
0	Real case (type 0)--just push zero on the stack.
IFTE	Execute appropriate choice.
»	

(RTOP takes advantage of the fact that a real number is object type 0, which is the same as a false flag--a complex number is object type 1, which is the same as a true flag.)

There is no particular advantage within a single program to using IFT or IFTE rather than the corresponding IF structure, so which form you use is mostly a matter of taste.

However, the RPN command forms have an advantage for more sophisticated programming: their use allows you to place the *test-sequence*, the *then-sequence*, and the *else-sequence* in separate programs or program structures. If you use an IF structure, all must be contained in the same program.

IFTE is actually a function, which means you can use it in algebraic objects as well as in programs. It is a prefix function of three arguments:

$$IFTE(\textit{test-expression, then-expression, else-expression})$$

Notice that the arguments are in the same order as the stack arguments when IFTE is executed as an RPN command. All three arguments are ordinary expressions. *Test-expression* is evaluated, and its value is interpreted as a flag. If the flag is *true*, *then-expression* is evaluated; if the flag is *false*, *else-expression* is evaluated. Typically, the *test-expression* contains a comparison operator, so that evaluation automatically returns a flag.

■ *Example.* 'IFTE(X>0,X,1−X)' returns X if X>0, and 1−X otherwise.

IFT has no algebraic form. This is because algebraic objects must return a result when evaluated--an algebraic conditional can't "do nothing" if the test flag is false.

4.4.2 The CASE Structure

The IF structures described in the previous section are convenient for branching that is based on a single test to select between two choices. While it is possible to handle any more elaborate combinations of tests and choices with "nested" IF structures, the overall structure can get rather convoluted (although certainly less so than equivalent HP 41 programs). For more straightforward handling of multiple tests and choices, the HP 48 provides the *CASE structure*, which has the following general form:

```
CASE
    test-sequence₁ THEN then-sequence₁ END
    test-sequence₂ THEN then-sequence₂ END
        .
        .
        .
    test-sequenceₙ THEN then-sequenceₙ END
    else-sequence
END
```

You can read the CASE structure as "execute *test-sequence*$_1$, *test-sequence*$_2$, etc., until one *test-sequence* returns *true*. Then execute the corresponding *then-sequence,* and skip to past the final END. If no *test-sequence* returns true, then execute *else-sequence*.

■ *Example*. The program COUNT4 is a simple four "bin" counting routine.

COUNT4	*Count in 4 Ranges*	1A23
level 1		*level 1*
x	☞	

≪ CASE	
DUP 0 < THEN 1 END	Range 1 if $x<0$.
DUP 0 == THEN 2 END	Range 2 if $x = 0$.
1 ≤ THEN 3 END	Range 3 if $0<x\le1$.
4	Other tests failed, so x must be greater than 1 (range 4).
END	
'COUNTS' SWAP DUP2	Make two copies of the vector name and the index.
GET 1 + PUT	Get the element, add 1, put it back.
≫	

COUNT4 tests an argument x to see in which of four ranges its value lies. The total in each range is stored in the four-element vector COUNTS. The elements of the vector represent these ranges:

Element	Range
1	$x<0$
2	$x=0$
3	$0<x\le1$
4	$x>1$

Another way to make a multi-case choice is to create a list of programs, then select one of the programs from the list according to an index. For example, this sequence takes a real number from the stack, and executes a name corresponding to the number:

{ ONE TWO THREE FOUR FIVE }	List of name choices.
SWAP	Put the index in level 1.
GET	Get the indexed choice.
EVAL	Execute the selected name.

4.5 Loops

A *loop* is a program structure containing a sequence that is executed more than once. In a *definite loop*, the number of repeats is known in advance. In an *indefinite loop*, execution of the loop repeats until some specified condition is met.

4.5.1 Definite Loops

The most common form of definite loop is one in which a calculation is performed once for each value of an index which is incremented by steps between start and stop values. On the HP 41, this is commonly achieved by using ISG ("increment and skip if greater") or DSE ("decrement and skip if equal"). These commands use *start*, *stop*, *index* and *step* values combined into a single decimal "control number" of the form *iiii.fffcc*:

- *iiii* is the *index*, which is an integer of up to five digits. Its initial value is the *start* value.

- *fff* is the three-digit integer *stop* value.

- *cc* is the two-digit integer *step* size. If it is omitted, the default step size is one.

The control number is stored in a stack or data register. This scheme has an advantage over the HP 48 approach described below in that all three of the loop parameters are available as long as the register containing the control number is undisturbed, although it takes some calculation to extract the various parts of the control number (and to create it in the first place).

The HP 48 uses the *FOR...NEXT* and *FOR...STEP* structures for definite looping, in which the *start* and *stop* values are supplied as stack arguments, and the index is stored in a named local variable. The following examples show equivalent HP 41 and HP 48 sequences:

Task	HP 41 Sequence	HP 48 Sequence
Execute a sequence ten times.	1.010 STO 01 LBL 01 *sequence* ISG 01 GTO 01	1 10 START *sequence* NEXT

Task	HP 41 Sequence	HP 48 Sequence
Sum the integers between the values stored in variables (registers) R05 and R06.	RCL 06 1000 / RCL 05 + STO 01 0 LBL 01 RCL 01 INT + ISG 01 GTO 01	0 R05 R06 FOR n n + NEXT

The HP 48 FOR...NEXT loop is designed to repeat execution of a program sequence several times, making use of an *index* that is incremented by 1 at each iteration of the sequence. The general form of a FOR...NEXT loop is:

$$start \quad stop \quad \text{FOR} \quad name \quad sequence \quad \text{NEXT},$$

where

- *start* is the initial value of the index.

- *stop* is the final value of the index.

- FOR identifies the start of the structure; it removes the start and stop values from the stack.

- *name* is the name of the (local) variable that contains the index.

- *sequence* is any program sequence, which can contain any number of uses of *name*.

- NEXT is the structure word that identifies the end of the sequence. It increments the index by one, then tests its value against the stop value to determine whether to repeat the sequence.

You can read a FOR...NEXT loop as "For each value from *start* through *stop* of an index named *name*, execute the sequence that ends with NEXT."

■ *Example.* Compute the sum of the squares of the integers from 1 through 100.

■ *Solution.* The easiest method on the HP 48 is to use the summation function Σ:

$$'\Sigma(I=1,100,I^2)' \quad \text{EVAL}$$

An equivalent program to illustrate the FOR...NEXT loop is:

0	Initialize the sum.
1 100	Start and stop values.
FOR n	Begin a loop using index *n*.
n SQ +	Square the current index and add to sum.
NEXT	Increment n by 1. If n≤100, loop again.

Executing this sequence returns the answer 338350.

An HP 41 program to make the same calculation looks like this:

01	1.100	Control number.
02	0	Initial value of sum.
03	LBL 00	Start of loop.
04	RCL Y	Recall the control number.
05	INT	Current integer.
06	X↑2	Square it.
07	+	Add to the sum.
08	ISG Y	Increment the control number.
09	GTO 00	Repeat if < 100.

Some observations:

• *Start* and *stop* are *not* part of the FOR...NEXT program structure. FOR expects to take two numbers from the stack, but those numbers can be entered or computed at any time in advance of the FOR, as long as they are in levels 1 and 2 when the FOR executes. *Start* and *stop* can either be real numbers or algebraic objects that FOR can evaluate to real numbers.

• The *start* and *stop* values are removed from the stack by FOR. They are not accessible afterwards; if a program needs their values for other purposes, it should copy them or store them in variables before executing the FOR.

• The *index* is kept in a local variable identified by the name that immediately follows FOR. You can return the current value of the index by executing its name. You can also change the value of the index after the loop has started, by storing a real number into the local variable. The naming and use of the index variable are subject to the same restrictions as local variables created by → (section 4.7). After the loop is finished, the index variable is automatically purged.

- The name following a FOR is *not* part of the sequence that is repeated. For example,

$$1 \quad 10 \quad \text{FOR} \quad \text{n} \quad \text{n} \quad \text{NEXT}$$

puts integers 1 through 10 on the stack, but

$$1 \quad 10 \quad \text{FOR} \quad \text{n} \quad \text{NEXT}$$

does nothing.

- The sequence between FOR *name* and NEXT always executes at least once, even if the specified *stop* value is less than the *start* value.

- The *start* and *stop* values don't have to be integers. NEXT always increments the index by 1; the loop will repeat as long as the index is less than or equal to the stop value.

$$.5 \quad .6 \quad \text{FOR} \quad \text{n} \quad sequence \quad \text{NEXT}$$

executes *sequence* once, with n = .5.

- The combination FOR *name* acts like a single operation when you single-step the FOR.

4.5.1.1 Varying the Step Size
The FOR...STEP program structure is a variation of FOR...NEXT, that allows you to increment the loop index by amounts other than one, including negative values. A FOR...STEP structure looks like this:

$$start \quad stop \quad \text{FOR} \quad name \quad sequence \quad \text{STEP}.$$

Start, *stop*, *name*, and *sequence* play the same roles as in FOR...NEXT loops. The structure word STEP plays a similar role to NEXT, but allows you to control the amount by which the index is incremented (or decremented). STEP takes a real number step value from level 1, and adds it to the current value of the index. Then:

- If the *step* value is *positive*, the loop repeats if the index is less (more negative) than or equal to the stop value.

- If the *step* value is *negative*, the loop repeats if the index is greater (more positive) than or equal to the stop value.

Note that since STEP takes a number from the stack, *sequence* must end with the step value on the stack (the step value doesn't have to be the same each time).

■ *Example.* The program DFACT computes the double factorial $n!! \equiv n\,(n-2)(n-4)...1$, where n is an integer.

DFACT	*Double Factorial*	605F
	level 1 | *level 1*	
	n ☞ $n!!$	

≪ 1	Initialize the product.
SWAP 2	Loop from n down to 2.
FOR m	m is the index.
m *	Multiply the product by m.
−2 STEP	Decrement m by 2. Repeat if $m \geq 2$.
≫	

An HP 41 version:

01	LBL "DFACT"	
02	.00102	
03	+	Control word: start at x, stop at 1, decrement by 2.
04	1	Initial product.
05	LBL 00	Start of loop.
06	RCL Y	Recall control word.
07	INT	Current integer
08	*	Multiply times current product.
09	DSE Y	Decrement the integer.
10	GTO 00	Repeat if integer > 1.
11	END	

4.5.1.2 Looping with No Index

In some circumstances, there is no need for an index when a program sequence is to be repeated a fixed number of times. In such cases, you can use START in place of FOR. START...NEXT and START...STEP are the same as FOR...NEXT and FOR...STEP, respectively, except that the loop index is not accessible. The index name that must follow FOR is not used with START (if a name does follow START, it is just treated as part of the loop sequence, and has nothing to do with the loop index).

■ *Example.* The program VSUM sums the n elements of a vector.

VSUM	*Sum Vector Elements*		ACD8
	level 1		*level 1*
	[*vector*]	☞	*sum*

≪ OBJ→ OBJ→	Put the elements on the stack, with the number of elements in level 2, and a 1 in level 1.
SWAP OVER –	Loop start and stop values for $n-1$ additions.
START + NEXT	Execute + $n-1$ times.
≫	

4.5.1.3 Exiting from a Definite Loop

Definite loop structures are designed to repeat a predetermined number of times. Since the HP 48 has no GTO command that can cause program execution to jump out of a loop before it has completed the specified number of iterations, you should ordinarily use an indefinite loop (section 4.5.2) for calculations where you don't know in advance how many iterations are needed. However, the indefinite loops don't provide an automatic index like that in FOR...NEXT/STEP loops, so for some problems you may find it more convenient to use a definite loop with a contrived exit rather than an indefinite loop where you have to provide your own index.

All you have to do to cause a loop to exit before the prescribed number of iterations is to store a number greater than or equal to the stop index value into the index variable. In loops with a positive step size, an obvious choice for an exit value is MAXR, the largest number that the HP 48 can represent, although you have to be sure to convert the symbolic constant into a real number. For loops with a negative step, you can use –MAXR.

Typically, the exit from a definite loop is taken as the result of a test. The general form of such a loop is as follows:

```
     start   stop
     FOR  n  sequence
        IF  test
        THEN  MAXR  →NUM  'n'  STO
        END
     NEXT
```

This structure executes *sequence* for every value of *n* starting with *start*, and ends when

either *n* is greater than *stop*, or *test* returns a true flag.

■ *Example.* Determine the value of N for which $\sum_{n=1}^{N} n^2 \geq 1000$.

0 1 10000	Initial value of sum; *start* and *stop* values.
FOR n	Loop index is n.
n SQ +	Increment the sum.
IF DUP 1000 >	Is the sum ≥ 1000?
THEN n	The current value of the index is *N*.
MAXR →NUM 'n' STO	Set the index past the stop value.
END	
NEXT	

Executing this sequence returns the sum 1015, and the value 14 for *N*.

4.5.2 Indefinite Loops

An *indefinite loop* is a loop where the number of iterations is not determined in advance. Instead, the loop repeats indefinitely until some exit condition is satisfied. The HP 48 provides two program structures for indefinite looping, the *DO loop* and the *WHILE loop*. The primary difference between the two structures is the relative order of the test and the loop sequence. In a DO loop, the sequence is performed first, then the test; in a WHILE loop, the test is performed first.

4.5.2.1 DO Loops

The basic form of a DO loop structure is:

DO *loop-sequence* UNTIL *test-sequence* END.

Loop-sequence is any program sequence. *Test-sequence* is a second program sequence, which must leave a flag on the stack. END removes the flag; if the flag is *false* (zero), execution jumps back to the start of *loop-sequence*. If the flag is *true* (non-zero), execution proceeds with the remainder of the program after the END. You can read a DO loop as:

"Do *loop-sequence* repeatedly, until *test-sequence* is *true*."

The equivalent HP 41 form is as follows:

LBL 00   Start of loop-sequence.

.

.

.

Test   Reverse "UNTIL"
GTO 00

where *Test* is any of the HP 41 test commands.

In HP 48 programs the position of the UNTIL between DO and END is unimportant. That is, the division of the program steps into *loop-sequence* and *test-sequence* is only a matter of program legibility. Both *loop-sequence* and *test-sequence* are executed at each iteration of the loop, so it doesn't matter where you put the UNTIL. We recommend that you use the UNTIL to isolate that portion of the program that constitutes the logical test--the program steps which produce the flag that signals whether or not to repeat. The portion that precedes the UNTIL should be the part of the loop that computes the results used by the remainder of the program (after the END).

To reverse the sense of the test, that is, to make a loop that repeats until a test is *false*, you can either substitute an opposite test command (> for <, FC? for FS?, etc.), or insert a NOT immediately before the END:

DO *loop-sequence* UNTIL *test-sequence* NOT END.

■ *Example*. Compute $\sum_{n=1}^{\infty} \frac{1}{n^5}$.

■ *Solution*: The sequence below sums terms of the form n^{-5}, until two consecutive sums are equal. Executing the sequence on the HP 48 returns 1.03692775496, after 183 iterations.

0 'N' STO	Initialize a variable N as a counter.
0	Initialize the sum.
DO	Start of loop.
DUP	Copy the old sum.
'N' INCR	Get the (incremented) counter.
-5 ^ +	Add n^{-5}.
SWAP	New sum in level 2, old in level 1.
UNTIL	Start test-sequence.
OVER = =	True if old sum = new sum (leaves only new sum in level 1)
END	Repeat if test was *true*, otherwise done.

HP 41 version:

01	1	
02	STO 00	Initialize a counter.
03	0	Initialize the sum.
04	LBL 00	Start of loop.
05	ENTER↑	Copy the old sum.
06	ENTER↑	
07	1	
08	X<> 00	Recall the counter.
09	STO+ 00	Increment the counter.
10	−5	
11	Y↑X	
12	+	Add n^{-5} to the sum.
13	X≠Y?	True if old sum ≠ new sum.
14	GTO 00	Iterate.
15	END	

4.5.2.2 WHILE Loops

In a WHILE loop, a test sequence is defined in the first part of the structure:

WHILE *test-sequence* REPEAT *loop-sequence* END.

Here again *loop-sequence* is any program sequence, and *test-sequence* is any sequence that returns a flag. REPEAT removes the flag; if the flag is *true*, the program executes *loop-sequence*, then loops back to *test*. If the flag is *false*, *loop-sequence* is skipped, and execution proceeds with the remainder of the program after the END. You can read a WHILE loop like this:

"As long as *test-sequence* is *true*, keep repeating *loop-sequence*."

Note that the position of the REPEAT is important (unlike the case for UNTIL), since it marks the division between the *test-sequence*, which is executed at every iteration, and the *loop-sequence*, which is not executed on the final iteration.

To reverse the sense of the WHILE test, that is, to make a loop that repeats while a test is *false*, you can either substitute an opposite test (> for <, FC? for FS?, etc.), or insert a NOT immediately before the REPEAT:

WHILE *test-sequence* NOT REPEAT *loop-sequence* END.

■ *Example.* The program GCD finds the greatest common divisor (GCD) of two integers *m* and *n*. GCD repeatedly computes $r = m \bmod n$; if each successive r is non-

zero, it replaces n with r, m with n, and repeats. When r is finally zero, the value of n is the GCD.

On the HP 48:

GCD	Greatest Common Divisor		
level 2	level 1		level 1
n	m	☞	GCD(n,m)

≪ WHILE	Beginning of test-sequence.
DUP2	Make 2 copies of m and n.
MOD	Compute $r = m \bmod n$
DUP 0 ≠	Test $r \neq 0$.
REPEAT	If true, do the following:
ROT DROP	Replace m and n by new values.
END	Loop back and repeat the test-sequence.
ROT DROP2	Leave n in level 1.
≫	

HP 41 version:

01	LBL "GCD"	
02	LBL 00	
03	RCL Y	
04	RCL Y	Make copies of m and n.
05	MOD	Compute $r = m \bmod n$
06	X≠0?	If $r \neq 0$, try again
07	GTO 00	
08	RDN	
09	END	

The HP 41 version is shorter than the HP 48 program, because it does not need to remove the successive values of m from the stack--the four-level stack ensures that they don't pile up as they would on the HP 48.

4.6 Error Handling

The action of the HP 41 when an error occurs during program execution is controlled by flag 25. If that flag is *clear*, any error causes program execution to to halt. However, if the flag is *set*, program execution continues after an error, but:

- the erroring command is aborted, and the state of the HP 41 is preserved as it was before the command was executed, including the stack contents;

• flag 25 is cleared.

An *error trap* in an HP 41 program generally consists of setting flag 25 in advance of the execution of a suspect command or command sequence, then having the program branch afterwards according to the state of flag 25. If the flag remains set, then no error occurred. If the flag is clear, an error did occur and the program must react accordingly.

The HP 48 provides considerably more sophisticated error handling tools than the HP 41. As on the HP 41, an action on the HP 48 that produces an error beep and an error message display also normally causes any current program evaluation to stop. However, by means of the *IFERR structure*, programs can intercept any errors (except Out of Memory) and continue execution. The IFERR structure has the following general form:

IFERR *error-sequence* THEN *then-sequence* ELSE *normal-sequence* END,

where the three sequences are arbitrary program sequences. You can read an IFERR structure as:

"If any error occurs during the execution of *error sequence*, then execute *then-sequence* and continue execution after the END. If no error occurs, skip *then-sequence* and execute *normal-sequence*, and continue on after the END."

There does not have to be a *normal-sequence*--the ELSE *normal-sequence* is optional:

IFERR *error-sequence* THEN *then-sequence* END

executes *then-sequence* if an error occurs during *error-sequence*, but does nothing special otherwise.

The HP 41 analog to an HP 48 IFERR structure is a program sequence like this:

SF 25	
command	error *"sequence"*.
FC?C 25	"IFERR"
GTO 01	
normal-sequence	"ELSE..."
GTO 02	
LBL 01	"THEN..."
then-sequence	
LBL 02	"END"
. . .	

Advantages of the HP 48 system are flexibility, in that the *error-sequence* can contain one or more objects; and legibility--when you read a program it is easy to identify the various sequences because they are set off by the IFERR/THEN/ELSE/END structure words. Furthermore, you can easily nest HP 48 error traps, which is difficult on the HP 41 because there is only one error-ignore flag.

■ *Example.* Compute $\sin x/x$, where x is a stack argument, using an IFERR structure to handle the undefined result error condition at $x = 0$.

<div align="center">DUP SIN SWAP IFERR / THEN DROP2 1 END</div>

This sequence returns 1 for an argument of zero. Here the "protected" *error-sequence* is just the single command /. The analogous HP 41 program is:

01	ENTER↑	
02	SIN	
03	X<>Y	
04	SF 25	
05	/	
06	FS?C 25	No error?
07	GTO 02	
08	RDN	
09	RDN	Discard the zeros.
10	1	Return 1.
11	LBL 02	

It's important in an HP 41 program to clear flag 25 as soon as an error-sequence has finished, to prevent inadvertently masking the effects of some later unanticipated error. On the HP 48, this is not a concern, since the IFERR structure is very specific about the extent of the error-sequence.

The position of the IF structure word in the sequence preceding THEN in an IF structure is unimportant because it is THEN that actually makes the branch decision. However, the position of IFERR in an IFERR structure is significant; the IFERR and the succeeding THEN define the extent of the sequence for which errors are trapped. IFERR A B THEN intercepts errors in A and B, whereas A IFERR B THEN traps errors occurring only in B. The jump to the *then-sequence* happens immediately upon the error; any remaining steps preceding the THEN are skipped. Thus if an error occurs in A in the structure IFERR A B C THEN D END, B and C are not executed--execution jumps from the point in A where the error occurred directly to D.

Because the reaction to an error is usually specific to a particular error, it is generally a

good idea to keep the *error-sequence* short, containing as few as one object if possible. Then there is no ambiguity about which object caused the error, and no part of the sequence that will be skipped. Of course, even a single object may cause different types of errors. To sort out such possibilities, you can use the ERRN command to return the error number of the most recent error, and ERRM to return the text of the error message. For example, suppose that a program adds two arguments. The addition can fail either because the stack is empty, or because the arguments are of the wrong type. The following IFERR structure can deal with either problem:

IFERR +	
THEN ERRN	Get the error number.
IF #513d ==	Is it error 513 (Too Few Arguments)?
THEN GETMORE	Use GETMORE to get more arguments.
ELSE ERRM ABORT	If the arguments are the wrong type, return the error message as a string.
END	
END	

4.6.1 The Effect of LASTARG

The design of an IFERR structure must take into account whether last arguments recovery is active at the time an error occurs. If LASTARG is enabled, the arguments of the command that errors are restored to the stack. If LASTARG is disabled, the arguments are discarded. The $\sin x/x$ example in the preceding section assumes that LASTARG is enabled. The DROP2 in the *then-sequence* is intended to discard the two zeros that cause the division error, and which are restored by the error system. If LASTARG is disabled, the DROP2 is inappropriate because the two zeros are not returned after the error.

Since in most cases it is preferable for programs to work correctly regardless of the state of LASTARG, IFERR structures should include steps to determine whether LASTARG is active or not and to act accordingly. Flag -55 provides programmable control of LASTARG. A program can test the flag to determine if LASTARG is enabled or disabled, or it can set or clear the flag itself. These options allow two general approaches for designing error traps:

1. Set or clear flag -55 in the program before the error trap, then write the IFERR structure accordingly. Returning to the $\sin x/x$ example, either

    -55 CF DUP SIN SWAP IFERR / THEN DROP2 1 END

    or

    -55 SF DUP SIN SWAP IFERR / THEN 1 END

will work. This method has the disadvantage that it may alter the state of flag
−55 and thus affect other programs that may depend on the flag. As a rule, any
program that does depend on flag −55 or any other flag should itself set the flag
the way it wants, so this should not be a major limitation. (You can preserve the
state of the LASTARG flag, and all other flags, by executing RCLF, keeping the
resulting list on the stack or in a local variable, then using it with STOF at the end
of a sequence that alters the flags.)

2. Include a conditional in the *then-sequence* that can react to the current state of
flag −55 without altering it. For example,

```
DUP  SIN  SWAP
IFERR  /
THEN
   IF  −55  FC?
   THEN  DROP2
   END
   1
END
```

4.6.2 Exceptions

A mathematical *exception* is a condition encountered in the execution of certain func-
tions for which you are given a choice of how subsequent execution should proceed.
You can treat an exception as an execution-halting error, or have the calculator supply a
default result and continue normally. In both the HP 41 and the HP 48, you make your
choice by means of the various *exception action flags*, which are flag 24 on the HP 41
and flags −20, −21, and −22 on the HP 48.

The HP 41 recognizes only one type of exception, the *Out of Range* exception, which
occurs when a mathematical result is larger than the maximum HP 41 number
$9.999999999 \times 10^{99}$. The corresponding exception action flag is flag 24; if this flag is
clear, such an overflow produces an error. If the flag is set, the maximum value
$\pm 9.999999999 \times 10^{99}$ is returned.

HP 48 exception action flags act analogously to HP 41 flag 24. The state of each of
these flags determines whether the corresponding exception results in an error, or
returns a default result with no error.

A typical HP 48 exception is division by zero. The behavior of / when the divisor is
zero is controlled by flag −22, the *infinite result action* flag. If flag −22 is clear (the
default setting), division by zero is treated as an error, causing the Infinite Result error.
However, if flag −22 is set, no error is reported, and one of the values

±9.99999999999E499 (±MAXR) is returned, which are the HP 48's best representations of ±∞. The sign of the result is determined by the sign of the dividend.

The choice to error or to supply a default generally depends on whether the exceptional condition is expected. For example, if you don't anticipate that a program might cause a division by zero, it is better to clear flag −22 so that the program will halt and report the error. On the other hand, if you know that the division-by-zero situation can happen, and that ±MAXR is a good approximate result that lets a calculation proceed to meaningful results, then setting flag −22 is a good choice.

An HP 48 program can detect when an exception occurs even when the action flag is clear and execution does not halt. Flags −23 through −26 act as *signal* flags--when an exception occurs, the corresponding signal flag is set automatically. For example, flag −26 is set by an infinite result exception. Therefore, a program can clear flag −26, carry out a calculation with flag −22 set, and still determine if a division by zero occurred by testing flag −26.

In addition to the infinite result exception, the HP 48 also recognizes two other exceptions:

- *Overflow* (action flag −21, signal flag −25). Overflow occurs when a function returns a result that is finite, but larger than the HP 48 can represent, such as FACT(2000). With flag −21 clear (the default setting), overflowing functions return ±9.99999999999E499. An overflow is *not* the same as an infinite result, for which the correct value is ±∞ rather than a too-large finite number.

- *Underflow* (action flag −20, signal flags −24 and −23). Underflow occurs when a function returns a result that is not zero but smaller in absolute value than 1E−499 (MINR), the smallest non-zero number that the HP 48 can represent. If flag −20 is clear (the default setting), any underflowing function returns zero as its default result. Since zero has no sign, two signal flags are used: flag −24 is set to indicate that the function underflowed from the negative side of zero; flag −23 set indicates underflow to a small positive number.

Note that 0÷0 is *not* an exception. That result is mathematically undefined--it is neither an overflow nor an infinite result. There is no appropriate default result to supply, so the HP 48 always reports the Undefined Result error and halts execution. You can, of course, create your own exception handing by using an IFERR structure to trap this error.

HP 41 flag 24 is closest in effect to HP 48 flag −21. However, the HP 41 does not distinguish between infinite and finite-but-too-large results. For example, TAN(90°) and EXP(2000) are both treated as the same out-of-range exception. Moreover, division by

zero on the HP 41 always returns DATA ERROR--this error can only be trapped by a flag 25 error branch. In converting HP 41 programs to the HP 48, therefore, you can't just use HP 48 flag −21 or −22 to act as HP 41 flag 24. If, for example, you set HP 48 flag −22 to match HP 41 flag 24 in preventing TAN(90°) from producing an error, the HP 48 then does *not* report an error for division by zero, which the HP 41 always does.

4.7 Local Variables

The HP 48 variables and their associated names that you see in the VAR menu are referred to as *global* variables and names. The term *global* implies that these variables can be accessed by any program, or from the command line. The HP 48 also provides variables that are associated only with individual procedures. The use of these *local* variables and the corresponding *local name objects* is a very useful and powerful programming technique.

It is possible, with the "unlimited" stack provided by the HP 48, to carry out an arbitrarily complicated calculation on the stack without any use of variables to store inputs, intermediate results, or final outputs. The fastest and most efficient computation is usually achieved in this manner. However, it is not always program execution efficiency that is paramount, but rather the overall "throughput" of the problem solving process. If a calculator is easy to program, you can usually get a result in less total time even if the program itself may execute more slowly than if you developed a solution in an efficient but arcane style. Thus while you *can* write a HP 48 program that is a marvel of structure and efficiency by using only stack objects, the time and skill required for you to keep track of everything on the stack during program development may be too high a price for the result. In short, there is often a compelling advantage to assigning names to objects to simplify the programming process.

At first glance this seems to imply the use of global variables that are created by STO, are available at any time, and appear automatically in the VAR menu. However, while global variables are fine for "permanent" data and procedures, they are not as attractive for storing intermediate results. They stay around indefinitely, so that you have to remember to purge them to avoid cluttering up the VAR menu and to conserve memory. Furthermore, you have to be careful when you create a variable in one program to avoid using the same name as that used by another program, unless you deliberately intend the two programs to share a common variable.

HP 48 *local* variables are a means for saving intermediate data and results that is intermediate between using the stack exclusively and using global variables. Local variables exist only in a context defined by the program structure that creates them; therefore there is no question of name conflicts with global variables or other procedures' local variables. Also, when the defining structure has completed evaluation, all of its local

variables are automatically purged.

There are two methods by which you can create local variables. The primary method is by means of *local variable structures*, which use the program structure word → to create local variables. In addition, the FOR...NEXT/STEP loops described in section 4.5.1 use a local variable to store the current value of the loop index. Although the index variable is used for a special purpose, it is otherwise the same as a local variable created by →, with the same applicable commands and restrictions. In the remainder of this section, we will concentrate on local variable structures.

A *local variable structure* starts with the structure word → (called "arrow," or just "to") followed by one or more local names, and then by a program or an algebraic object referred to as the *defining procedure*. The closing delimiter (′ or ≫) that ends the defining procedure also marks the end of the structure:

$$\rightarrow \quad name_1 \quad name_2 \quad \cdots \quad name_n \quad \ll program \gg, \quad \text{or}$$

$$\rightarrow \quad name_1 \quad name_2 \quad \cdots \quad name_n \quad 'algebraic'.$$

The primary purpose of local variables is to provide a means of manipulating by name the stack arguments used by a procedure. You can think of the → as meaning "take objects from the stack and give them the following names; then evaluate a procedure defined using the names." Note that the procedure *is* evaluated, even though it is entered between quote delimiters ′ ′ or ≪ ≫.

→ takes objects from the stack and matches them each with one of the names that follows the →. The number of objects taken is determined by the number of names that are specified. The end of the series of names is marked by the delimiter ′ or ≪ that starts the defining procedure. The objects are matched in the order in which they appear in the stack; the object in the highest stack level goes with the first name; the object in level 1 is matched with the last name. A local variable is created for each of the names, with the local name as its variable name, and the matching object as its value. For example,

$$1 \quad 2 \quad 3 \quad 4 \quad \rightarrow \quad a \quad b \quad c \quad d$$

creates the local variables a with the value 1, b with value 2, c with value 3, and d with value 4.

■ *Example.* Compute the five integer powers x through x^5 of a number x in level 1. This first method does not use any variables except a loop index:

`« 2 5`	Powers 2 through 5.
`  FOR n`	Loop with index *n*.
`    n 1 - PICK`	Get a copy of the number.
`    n ^`	Raise to the *n*th power.
`  NEXT`	
`»`	

This is not a very complicated program. It is fast and efficient, because it uses only stack operations to obtain copies of the input number. The sequence n 1 - PICK is needed to return a new copy each time around because when the index is n, the original number has been pushed to level $n - 1$ by the growing stack of computed powers.

The program looks easy to write, but you do need a little thought to figure out where the input number will be on the stack at each iteration, and what stack operations are required to return a copy of the number. You can avoid the mental gymnastics by writing the program to remove the number from the stack at the outset, and name it with a local name:

`« → x`	Store the number as *x*.
`  « x 2 5`	Powers 1 through 5.
`    FOR n`	Loop with index n.
`      x n ^`	Compute x^n.
`    NEXT`	Repeat.
`  »`	
`»`	

The latter program is slightly longer than the previous version, but the time it takes you to write it should be less because there is no effort required to keep track of the input number on the stack. Any time the program needs the number, it just executes the local name. The lesson of this simple example becomes more important as the complexity of the programmed calculation increases, to the point where using local variables can make the difference between success and failure in the development of a program.

You can use local variable structures at any point in a program, not just at the beginning as in the example. The program CINT illustrates the use of a local variable to name an *intermediate* result. CINT computes the radius of a circle inscribed in a triangle, where the lengths of the sides of the triangle are specified on the stack. The formula is:

$$r = \frac{[s\,(s-a)(s-b)(s-c)]^{\frac{1}{2}}}{s}$$

where *a*, *b*, and *c* are the lengths of the sides, and $s = \frac{1}{2}(a + b + c)$.

CINT		*Circle in a Triangle*		3EBE
level 3	*level 2*	*level 1*		*level 1*
a	*b*	*c*	☞	*r*

≪ → a b c	Name the lengths of the sides.
≪ '(a+b+c)/2' EVAL → s	Compute and save *s*.
'√(s*(s-a)*(s-b)*(s-c))/s'	Compute *r*.
≫	End of local variable structure.
≫	

4.7.1 Comparison of Local and Global Variables and Names

Local names and variables are very similar to ordinary names and variables, but there are some important differences:

- *Global* variables are stored in a permanently established portion of memory we call VAR memory because of its relation to the [VAR] key, or *user memory* because it is the primary storage place for user-created objects. *Local* variables are stored in dynamically created "local memories," each of which is a segment of memory that acts like an independent VAR memory assigned to a particular procedure. When the procedure has finished evaluation, the associated local memory is deleted, including all of its local variables.

- *Local* names are a different object type (7) from *global* names (6). This is how the HP 48 system knows whether to find the variable corresponding to the name in VAR memory (global variables) or in a temporary local memory. When the HP 48 attempts to find a local variable, it searches the most recently created local memory first, then previous local memories in reverse chronological order, until it finds a variable matching the specified name.

- Executing a local name recalls to level 1 the object stored in the corresponding local variable, *without* executing the object. This means that if you store a program in a local variable, to execute that program you must execute the variable name followed by EVAL (or →NUM). The EVAL is not necessary for programs stored in *global* variables, since execution of a *global* name does automatically execute the stored object.

- Most commands that can work with *formal* global variables (names with no associated variables) do *not* accept local names as arguments: ∂, ∫, TAYLR, DRAW, ISOL, QUAD, ROOT.

- You can not delete a local variable with PURGE.

- *Local* names can be the same as HP 48 command names (except for single-character algebraic operator names like +, −, *, etc.). You can use local names i and e, but you should be careful not to use these names when you also want to use the symbolic constants i and e.

Occasionally you may encounter a local name for which the associated local variable no longer exists. For example, a defining procedure may leave the name of a local variable on the stack after it completes evaluation.

$$\ll \ 1 \ \rightarrow \ x \ \ll \ 'x' \ \gg \ \gg$$

leaves the local name 'x' on the stack after evaluation, but the corresponding local variable x that was given the value 1 is gone. You can not successfully execute this "formal local variable"--EVAL returns the Undefined Local Name error. You should try to avoid leaving left-over local names on the stack or in algebraic objects that result from symbolic calculations, to avoid confusion later.

5. Program Development

In the preceding chapter we described the nature of HP 48 programs, including the program structures that provide for branching and looping. In this chapter, we will study certain topics in program development, including program editing and debugging, and HP 48 mechanisms for input and output to and from a program's users.

5.1 Program Editing

Unlike the HP 41, the HP 48 does not have a specific "program mode" for entry or editing of programs. A program is created as a stack object using the command line, then named by storing the new object in a variable. Similarly, to make any alteration to an existing program in order to correct an error, optimize execution, or add features, you must edit a program the same way you edit any other object--in the command line. That is, you use EDIT or VISIT to create a text version of the program in the command line, use the facilities of the command line to make the changes you desire, then press ENTER to replace the old copy of the program with the new one. Re-entering the entire program this way ensures that objects and program structures are entered correctly.

The advantages of the HP 48 program editing approach are:

- The same editing methods apply to all HP 48 object types, so that you don't have to learn special techniques for each object type.

- No changes you make during an edit are "final" until you press ENTER . If you change your mind while you are editing a program, you can just press ATTN to cancel the edit and leave the program intact.

On the other hand, there are two important disadvantages:

- For a large program, it can take a substantial amount of time for the HP 48 to translate the entire program object into its text form, and, when you're done editing, to build the new program from the command line text.

- During the execution of ENTER, there must be memory available for as many as three versions of the program (the original, the command line text, and the new version) simultaneously. This restricts the size of the program that you can edit.

The latter disadvantage is the most serious, because it can happen that there isn't enough memory to permit any changes to an existing program, even if the changes don't increase the final size of the program. Both disadvantages dictate that you keep programs small, typically less than a few dozen objects each. If a program starts to get too big as you develop it, break it up into smaller subprograms that are executed by a short

main program. Even though this costs a little more memory for the subprogram names and variables, the smaller programs will be editable when a big single program is not.

5.2 Starting and Stopping

As we have discussed in previous sections, HP 48 programs are highly structured, and each has only a single entrance and exit. This fact makes starting and stopping an HP 48 program a different proposition from the simple run/stop capability of other calculators. On the HP 41, for example, you can stop a running program at any time by pressing [R/S] . When that key is pressed, the program halts after the currently executing step, and returns control to you. You can use [GTO] to move the program counter to another line or label, or run another program, etc. When you press [R/S] again, program execution resumes from wherever the program counter happens to be.

In the HP 48, if a program is to stop and be able to be restarted, it must include a HALT or PROMPT command in its definition. You can stop any program by pressing [ATTN] , but as you will see below, that terminates execution of the currently executing program and cancels pending returns to any other programs that may have called that program. (In more precise terms, the return stack is cleared, and the normal stack display and keyboard are reactivated.) HALT and PROMPT are the HP 48 versions of HP 41 STOP and PROMPT, respectively. PROMPT is the same as HALT (or STOP) with the addition that it displays a specified message in the display (in the status area, on the HP 48). HP 48 CONT (*continue*) is a programmable command which produces an effect similar to pressing [R/S] on the 41--except that CONT has no effect unless there is a program suspended. In the HP 41, [R/S] always causes program execution to resume from wherever the program counter happens to be.

When either HALT or PROMPT is executed, the program containing the command is *suspended*. The HALT annunciator turns on in the status display area on to remind you that there is a program awaiting completion. The keyboard is activated, and all calculator operations work normally. The HP 48 can maintain this state indefinitely--it behaves as if you had started up another calculator "inside" the halted program. This suspended calculator *environment* even has its own stack save (for [◄] |LAST STACK|), which is separate from the stack copy made before the suspended program was started. The calculator operates in the suspended environment until you execute CONT, whereupon the suspended program resumes execution at the object following the HALT or PROMPT.

You can "nest" suspended program environments one within another without limit (other than available memory). While one program is halted, you can run another program that itself halts and sets up another calculator environment with its own saved stack, and so on. When you execute CONT, the latest suspended environment is deleted, including the stack copy saved in that environment. When a program completes

execution, pressing ⬅ [LAST STACK] immediately afterwards restores the stack that was saved by the ENTER that started the program. To illustrate, enter the following program and name it A:

$$\ll \quad \text{CLEAR} \quad 1 \quad 2 \quad \text{HALT} \quad 3 \quad 4 \quad \gg \quad 'A' \quad \text{STO}$$

Then:

Keystrokes:			Results:		
➡ [CLR] 'X' 'Y'					
[ENTER]			2:	'X'	
			1:	'Y'	
A [ENTER]			2:	1	
			1:	2	HALT annunciator is on. The program has put 1 and 2 on the stack, and halted.
➡ [CLR]			2:		
			1:		
⬅ [LAST STACK]			2:	1	
			1:	2	[LAST STACK] restores the stack cleared by the last CLEAR.
⬅ [CONT]			4:	1	
			3:	2	
			2:	3	
			1:	4	The program A resumes, pushes 3 and 4 onto the stack, and is finished.

⬅ LAST STACK	2:	'X'

1:      'Y'   Back to the original environment; the last ENTER in this environment was the one that started the program A. Thus LAST STACK restores the stack as it was before that ENTER.

5.2.1 The ATTN key, DOERR and KILL

As the ATTN (*Attention!*) letters below the ⌐ON⌐ key suggest, this key is your means for getting the "attention" of the calculator. When you press ATTN, you tell the calculator to stop what it is doing: stop all operations, procedures, etc., clear any special displays, reactivate the keyboard, and show the standard stack display. This is a "gentle" interruption--stored variables are not affected, the stack is preserved, and the saved stack, command line stack, and last arguments are left intact. The procedure return stack is cleared, however, which means that programs interrupted by ATTN can't be continued.

When a program is suspended (by HALT), ATTN acts as above, but preserves the suspended environment. That is, any suspended programs (and all of the associated saved stacks) are unaffected by ATTN . This is a reassertion of the statement that all ordinary calculator operations can be carried out while a program is suspended without affecting the suspended program-- ATTN is considered "ordinary" in this sense.

ATTN is treated by the calculator as an error, error number zero, except that it is "silent." There is no error beep or error message, but otherwise it halts execution like any other error, and you can intercept this error with an error trap (section 4.6). A running program can even abort itself by executing 0 DOERR, which acts exactly as if the ATTN key were pressed. 0 DOERR is a particular case of the general form *n* DOERR, which executes the system error *n*. Such deliberate errors, unless trapped, act the same as inadvertent errors--the HP 48 beeps and displays the appropriate error message. A program might execute DOERR when it has intercepted an error but decides to abort anyway (using, for instance, ERRN DOERR to display the error message), or when it deliberately wants to error and use a built-in error message. If no built-in error message is suitable, you can supply your own error message with "*message*" DOERR, where "*message*" is a string object that specifies the error message. Usually this is done in combination with an IF structure, such as

IF *all-is-lost* THEN "Give Up" DOERR END.

Like [ATTN], DOERR works in the current suspended program environment--if there are any suspended programs, they are unaffected by DOERR. The only command that does affect suspended programs is KILL. KILL is an extended form of DOERR that not only terminates the current program, but also clears *all* suspended programs and turns off the HALT annunciator. All of the temporary saved stacks associated with the suspended program environments are deleted. You can use KILL in a program, but that is a rather drastic thing to do, since in general a program can not "know" what other programs are suspended when it is executed. It is better to use DOERR in a program, then execute KILL manually if needed. Most likely, your most frequent use of KILL will be to clear some half-finished program that you have been single-stepping, once you have found the problem you have been seeking.

5.2.2 Single-Stepping

The ≡SST≡ (single-step) key found in the [PRG] ≡CTRL≡ (program control) menu is a combination of CONT and HALT that lets you step through a program one object at a time. HP 48 single-stepping differs from that on the HP 41 in several important respects:

- You can't single-step an HP 48 program unless it is *suspended*--it must be executed from its beginning and halted prior to single-stepping. On the HP 41, using the [GTO] key, you can begin single-stepping anywhere in a program.

- Every HP 41 program step is self-contained, so every single-step operation is independent of what precedes or follows it. On the HP 48, certain program structures, such as IFERR sequences or local variable assignments, execute together as a single step.

- The HP 48 allows you either to execute a named subroutine as one step, or to single-step through the subroutine. The HP 41 only allows the latter.

- There is no back-step (BST) on the HP 48.

To understand the mechanics of HP 48 SST, you can picture it as the equivalent of pressing [◁] [CONT] with a HALT temporarily inserted immediately after the next object in the program. From this model, you can see that in order to single-step a program, it must be suspended. There are two ways to suspend a program for this purpose. The easiest way is to put the program or its name on the stack, and press ≡DBUG≡ (also in the program control menu). This begins execution of a program but halts it immediately, before executing the first object in the program. If you want instead to begin executing a program and halt it at a point other than the beginning, you must first edit the program and insert a HALT at the desired point, then execute the program and wait for it to stop. (You can also use [PROMPT], as long as you include a prompt string argument.) In either case, once the program is halted, you can use ≡SST≡ to execute each successive object in the program. At each ≡SST≡ press, the HP 48 executes the next

object in the suspended program, then halts and suspends the program again. To help you keep track of where you are in the program, each object is displayed in the top display line as it is executed (that display may be overwritten if the executed object affects the display). You can also preview the next object without executing it by pressing � NEXT (which displays the next one or two objects in the program).

In some cases, it is desirable to step through subroutines of a program--other programs that are executed by name within the main program. This is accomplished by using ⊟SST⊦⊟ (single-step "down") when you encounter a subroutine's name instead of ⊟SST⊟. While single stepping a subroutine, you can press ⟨◁⟨ CONT⟩ to complete execution of the subroutine, and resume single-stepping in the main program. Pressing ⟨◁⟨ CONT⟩ in the main program, or single-stepping its final >>, completes execution of the program and clears the suspended program environment.

Another consequence of the behavior of SST as a one-step CONT is that each SST clears the current suspended program environment, then creates a new one after the step. This means that you can't cancel any stack effects of the object that was single-stepped by pressing ⟨◁⟩ LAST STACK --the saved stack present before the SST is deleted by the SST.

Some additional notes about HP 48 SST:

- An IFERR structure is treated as a single object by SST. That is, when you press ⊟SST⊟ at an IFERR, the entire IFERR...THEN...ELSE...END structure is executed. If an error occurs between IFERR and THEN, the *then-sequence* between THEN and ELSE is executed; otherwise the *else-sequence* (if it is present) between ELSE and END is executed. The next ⊟SST⊟ will single step whatever object follows the END. If you want to step through individual parts of the IFERR structure, you must insert HALT(s) within the structure.

- If a single-stepped object causes an error, the error is reported normally, but the single step execution does not advance. If you press ⊟SST⊟ again, the HP 48 will attempt to execute the same object again. This gives you a chance to fix whatever it is that causes the error, such as a missing stack argument, then proceed with single-stepping.

- At any time while you are single-stepping a program, you can return to normal execution of the remainder of the program by pressing ⟨◁⟩ CONT .

5.3 Input and Output

In any RPN calculator, the stack is the basic input/output mechanism in programs as well as in manual operation. You can enter a program's input data as stack objects,

execute the program, then read its results from the stack. It is common, however, for programmers to simplify matters for program users by providing visual or audible prompting for input, and labels for output. In the HP 41, prompting is generally accomplished using PROMPT, which halts program execution and displays the alpha register to show the prompt message. A program can also use PROMPT to label its output, with sequences like this:

> 01 *"value ="*
> 02 ARCL X
> 03 PROMPT

This leaves the string *value =x* in the display, where *x* is the output value, here assumed to be in the X-register.

This same simple prompting and labeling can be done on the HP 48, but that calculator also provides several more elaborate methods for input and output.

5.3.1 Input Prompting

5.3.1.1 Stack Entry

The HP 48 PROMPT command (≣PROM≣ in the [PRG] ≣CTRL≣ menu) is a straightforward analog of the corresponding HP 41 command. It is designed to display a one- or two-line message in the status area, which serves to prompt a program user to enter data or perform other calculator operations before resuming program execution. The message is displayed in the medium font in the status area, where it will persist until the next ENTER. For example, the following sequence instructs the program user to enter a value for X, then to resume program execution by pressing ◁┐ [CONT] :

> "Enter X, then CONT" PROMPT.

Executing this sequence yields a display like this:

```
Enter X, then CONT

4:
3:
2:
1:
PARTS PROB  HYP  MATR VECTR BASE
```

PROMPT expects to find a string object in level 1 representing the the prompt message. If the message contains no newline characters, it is displayed in display line 1, and line 2 is automatically blanked. To display a message that spans lines 1 and 2, you must include a newline character at the line break:

<div align="center">
"Enter X,
then press CONT"   PROMPT
</div>

yields

```
┌──────────────────────────────┐
│Enter X,                      │
│then press CONT               │
│4:                            │
│3:                            │
│2:                            │
│1:                            │
│PARTS PROB  HYP  MATR VECTR BASE│
└──────────────────────────────┘
```
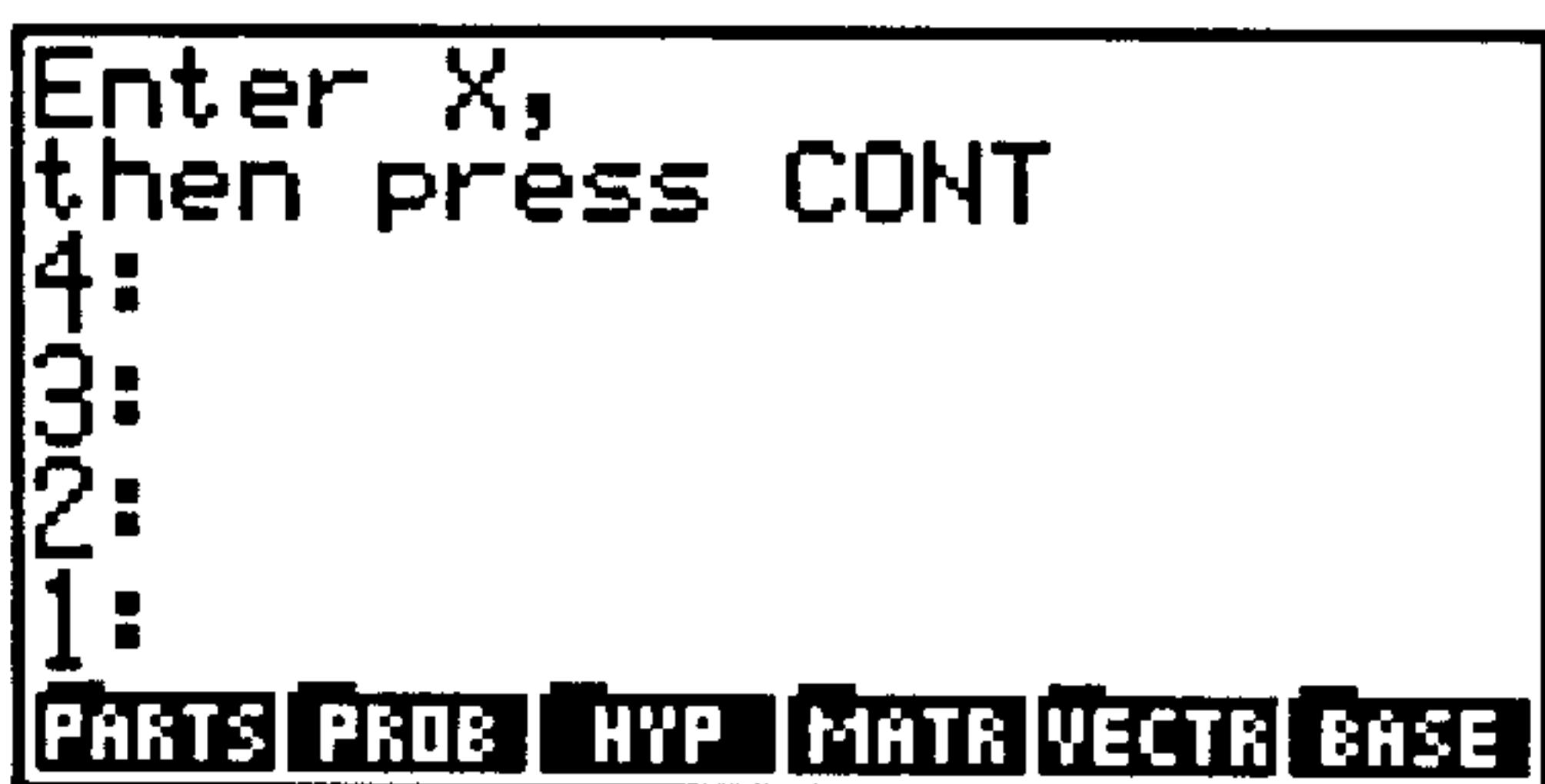
Each line of a message string should normally be 22 or fewer characters; if either is longer, the first 21 characters of that line are displayed with an ellipsis "..." in the right-most character position.

For multi-line text or graphical prompts, you can use DISP, →LCD, or PVIEW followed by FREEZE and HALT. The general steps are as follows:

1. Create the prompt. For text displays, this consists of one or more string objects, each of which is to be displayed in one medium-font display line. For graphical displays, the prompt can be a graphics object (grob) on the stack, or you can use the plotting screen (PICT).

2. Display the prompt.

   • DISP displays a single line of text, where the text is specified by a string object in level 2, and the display line by a real integer 1 - 7 in level 1. Normally, you precede one or more DISP's with CLLCD (*CLear LCD*), which blanks the status and stack areas of the display (lines 1 through 7).

   • →LCD blanks the stack and status display areas, and displays a grob (level 1) in the upper left corner of the display. The largest grob that can be displayed this way is 131×56; if the argument grob is larger in either direction, it is

cropped to 131×56.

- PVIEW displays the plotting screen. You must specify the coordinates of the plot that are to be placed at the upper left corner pixel of the display, by entering a complex number (x,y) for the scaled coordinates, or a list { #x #y } containing the pixel numbers. In the common case where the PICT grob is the same size as the LCD (131×64), the appropriate argument for PVIEW is { #0 #0 }.

3. "Freeze" the display. Normally, when a program halts, the calculator automatically redisplays the status area, stack, and menu labels, wiping out any special displays a program might have made. You can prevent the automatic display update by executing FREEZE, with a real integer argument that specifies which display areas are to remain unchanged. The status area, stack area, and menu labels are individually frozen by 1 FREEZE, 2 FREEZE, and 4 FREEZE, respectively; other values up to 7 freeze the two or three areas that "sum" to those values. Thus 3 FREEZE freezes the status and stack areas, and 7 FREEZE freezes all three areas.

4. Halt the program, using HALT.

■ *Example.* The following program sequence creates an input prompt for a quadratic equation solution.

QUADP →LCD	Display the quadratic equation.
"Enter a, b, and c;" 4 DISP	Display first text line.
"Press CONT" 5 DISP	Display second text line.
3 FREEZE	Freeze status and stack areas.
HALT	Suspend program execution.

Here QUADP is a global variable containing a graphics object representing the quadratic equation. The graphics object is created with the following sequence:

'a*x^2+b*x+c=0' 0 →GROB { #0d #15d } { #65d #32d } SUB

0 →GROB turns the algebraic object into a grob showing the equation in Equation-Writer form; SUB trims the grob down to a minimum size (66×18) to show the equation.

Executing the prompting sequence yields the following display:

```
┌─────────────────────────────────────┐
│          2                          │
│ a·x +b·x+c=0                         │
│                                     │
│ Enter a, b, and c;                  │
│ Press CONT                          │
│                                     │
│                                     │
│ PARTS PROB  HYP  MATR VECTR BASE    │
└─────────────────────────────────────┘
```
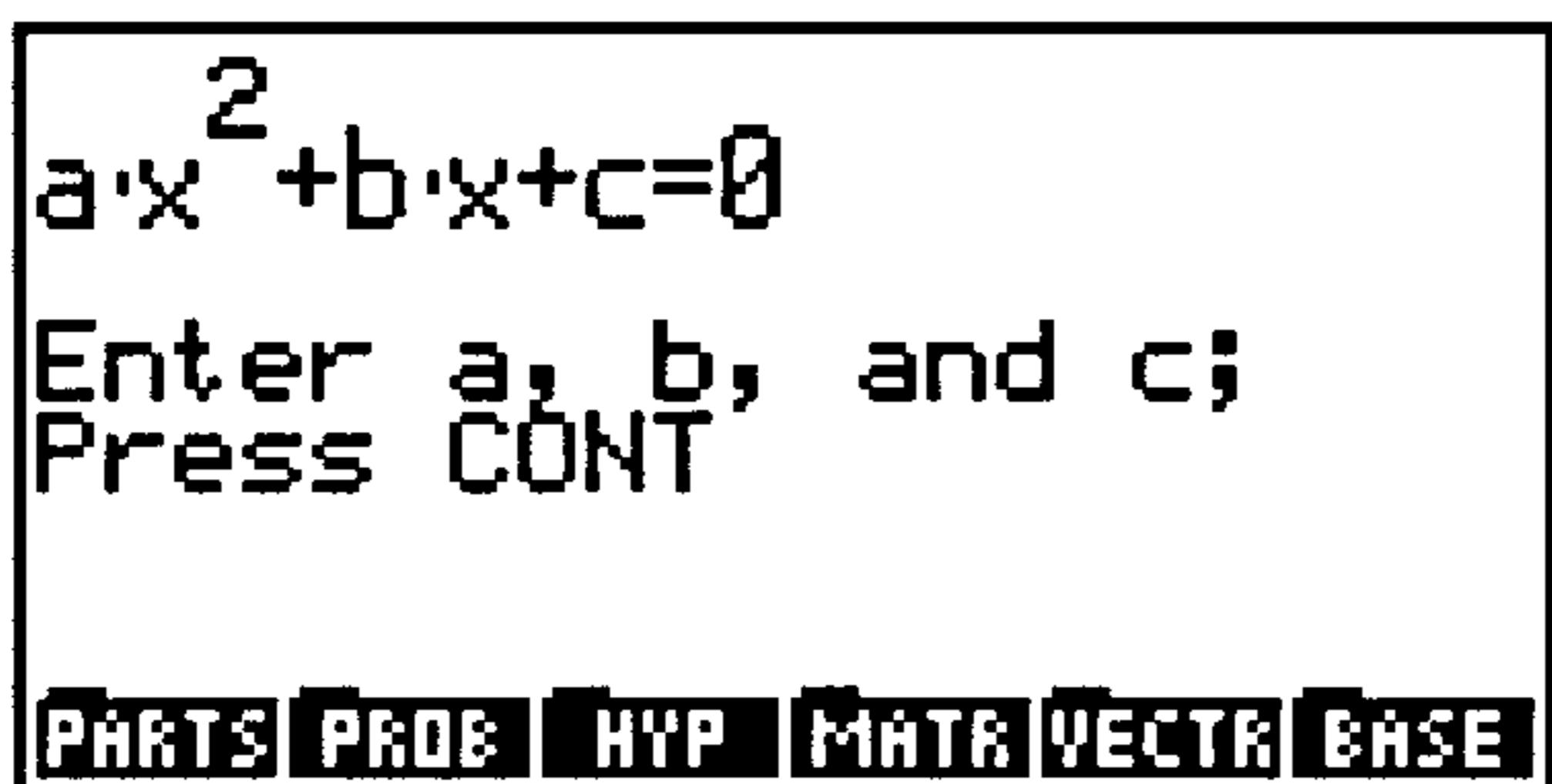
Prompting displays created with DISP, PVIEW, and →LCD may be more elaborate than those produced by PROMPT, but they only persist until the next key press.

5.3.1.2 Command Line Entry

The advantage of the stack entry methods described in the preceding section is that a program user has access to the HP 48 stack and any of the calculator's facilities when entering the input needed by the program. However, this access can also be a disadvantage, in circumstances where it is not desirable for the user to be able to alter stack contents, stored objects, etc. For these situations, the INPUT command provides for command line entry, for which the program can specify display prompting, initial command line contents and cursor position. INPUT automatically activates program entry mode, so that the user is restricted to object entry only. When INPUT executes, program execution is suspended, prompt text replaces the stack display, and the command line is activated. After entry, ENTER resumes program execution with the object following INPUT in the current program.

INPUT requires two arguments. A string in level two specifies a prompt that appears in the medium font in the stack display area; this prompt persists during keystroke entry, until ENTER terminates the INPUT operation. You can create a multi-line prompt by including newlines in the level 2 string. The level 1 argument can also be a string, which is used as the initial contents of the command line. For example, the following sequence prompts for a new value for a variable X:

<div align="center">"Enter X:" X STD →STR INPUT OBJ→ 'X' STO</div>

Here we have used the current value of X as the initial contents of the command line. When the sequence is executed with 100 stored in X, the following display appears:

```
                                              PRG
 { HOME }
 Enter X:



 100◀
 PARTS PROB  HYP  MATH VECTR BASE
```
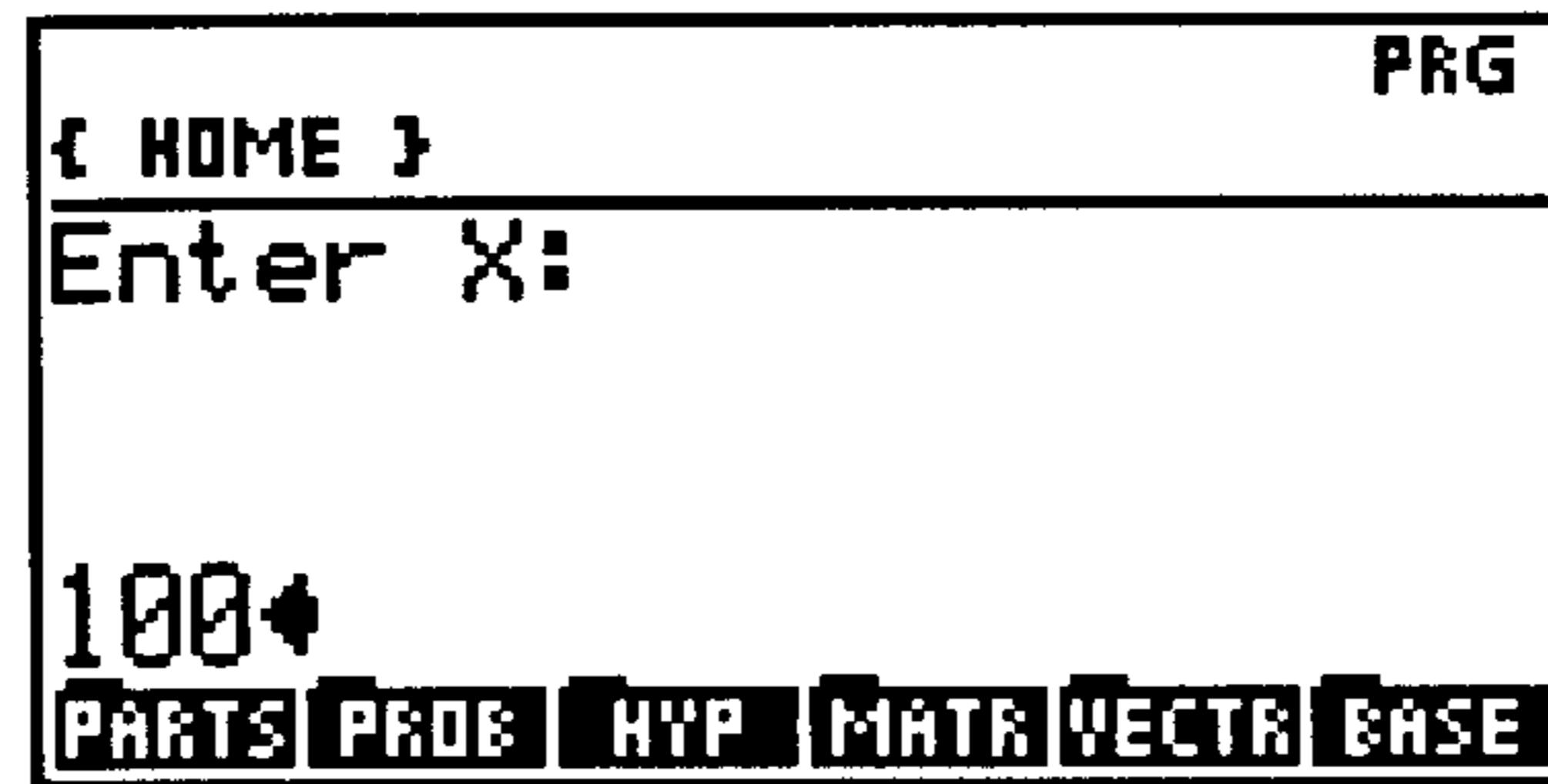
At this point, you can edit the current value, or press ATTN to clear the command line and type a new value for X. (If you press ATTN again, or any time the command line is empty, the program is aborted.) Pressing ENTER returns the contents of the command line to level 1 as a string object, and the program resumes execution with OBJ→.

In the example, the command line initially contains the level 1 string argument, with the insert cursor ◊ at the end of the string; upon ENTER, the command line string is pushed as is onto the stack. For additional control over the INPUT command line, you can use a list as the level 1 argument. The list can contain one or more elements (the order does not matter):

- To specify the initial command line text, include a string object (if this is the only element, then you can use the string object by itself as in the preceding example). The string may contain newlines, to produce a multi-line entry. If no string is specified, the command line will initially be empty.

- To place the cursor at a particular position in the command line, include a real integer to specify the character position, counting from the start of the command line (and including newlines in the count). Character number 0 specifies that the cursor is to be placed at the end of the command line (to the right of the last character). Alternatively, you can use a list {row column} that specifies the row (counting from the top down) and column (counting from the left) position for the cursor. Column number 0 indicates that the cursor is to be placed at the end of the specified row; row 0 specifies the last row of the command line. If no cursor position is specified, the cursor will be placed at the end of the command line.

  You can also use the cursor position object to select replace entry mode, in which typed characters overwrite the characters at the cursor. This is done by entering a *negative* character or row number. Positive numbers specify the default insert mode.

- To activate the command line in algebraic-program entry mode (ALG PRG), include the name ALG.

- To activate alpha-lock, include the name α.

- Since the command line contents are returned to the stack as a string, the INPUT normally does no syntax checking on the string following ENTER . However, if you include the name V (for verify), the string is checked for valid object syntax. If there is a syntax error, the HP 48 beeps and reactivates the command line with the highlighted error position, just as with ordinary command line entry. This is useful when you are using INPUT to enter objects in their standard form, i.e. you follow INPUT with OBJ→ to convert the result string to objects. If you don't use the V option and an entry has invalid object syntax, OBJ→ will error and abort the program. The V option allows the HP 48 to catch such errors before the program resumes.

Note that the symbols α, ALG, and V are entered into the INPUT strings as name objects--without any delimiters. However, these names are not executed, so it doesn't matter if you have variables with those names.

■ *Example*. In the following sequence, spaces within strings are marked by "^" characters for clarity.

"Enter temperature ^and pressure"	Two-line prompt string.
{ ":Temp:^ :Press:^"	Initial command line text.
{ 1 0 }	Cursor at end of first line.
V	
}	
INPUT	Stop for input.
OBJ→	Convert entered text into objects.

Executing this sequence yields the following display:

```
                                    PRG
{ HOME }
_____
Enter temperature
 and pressure

:Temp: ◀
: Press:
PARTS PROB  HYP  MATR VECTR BASE
```
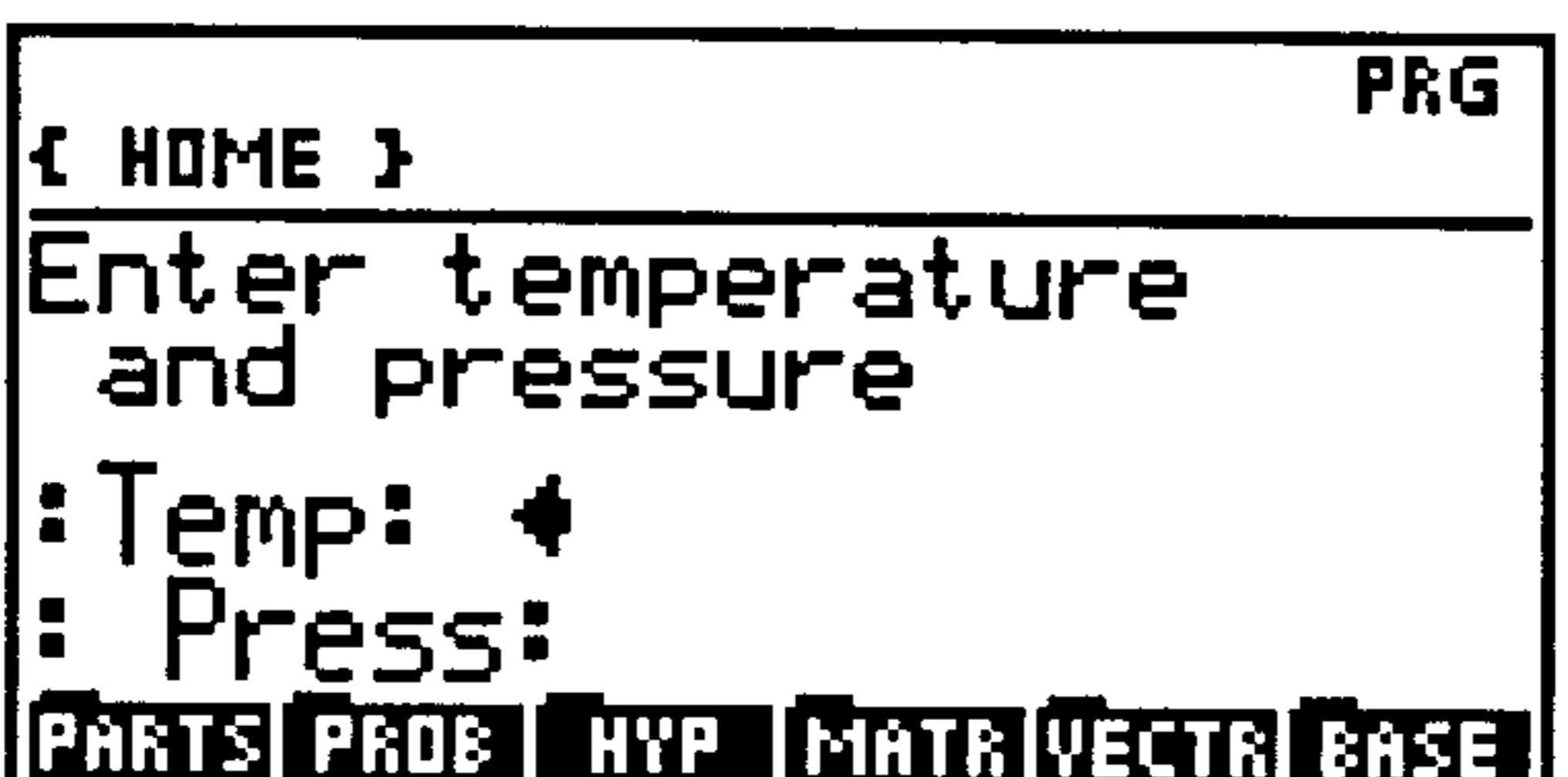
The cursor is at the end of the first row, following the tag :Temp: that indicates that a temperature should be entered. After entering the temperature, pressing ⊽ moves the cursor to the second row, following the :Press: tag. For example, these keystrokes

300_ ⟵|UNITS| |NXT| ≣TEMP≣ ≣ K ≣ ⊽
100000_ ⟵|UNITS| |NXT| ≣PRESS≣ ≣ PA ≣
|ENTER|

return the tagged object :Temp:300_K to level 2, and :Press:100000_Pa to level 1. Here the primary purpose of the tags is to indicate the command line order of the entries; the fact that the resulting stack objects are tagged will not interfere with any subsequent program calculations.

5.3.1.3 Custom Menus

In some cases, a program requires you to make a choice of action rather than to enter data. It is possible, of course, to use data to make a choice, as in "enter 1 to do this, or 2 to do that." An alternative method is to use *custom menus*, where a program halts and allows the user to press a menu key to indicate a choice and resume program execution. Consider this example:

"Choose left or right"	Prompt text.
{ { "LEFT" << 1 SF CONT >> }	First menu key definition.
{ "RIGHT" << 1 CF CONT >> }	Second menu key definition.
}	
TMENU	Create the temporary custom menu.
PROMPT	Stop for input.

Executing this sequence produces the following display:

```
Choose left or right

4:
3:
2:
1:
LEFT RIGHT
```
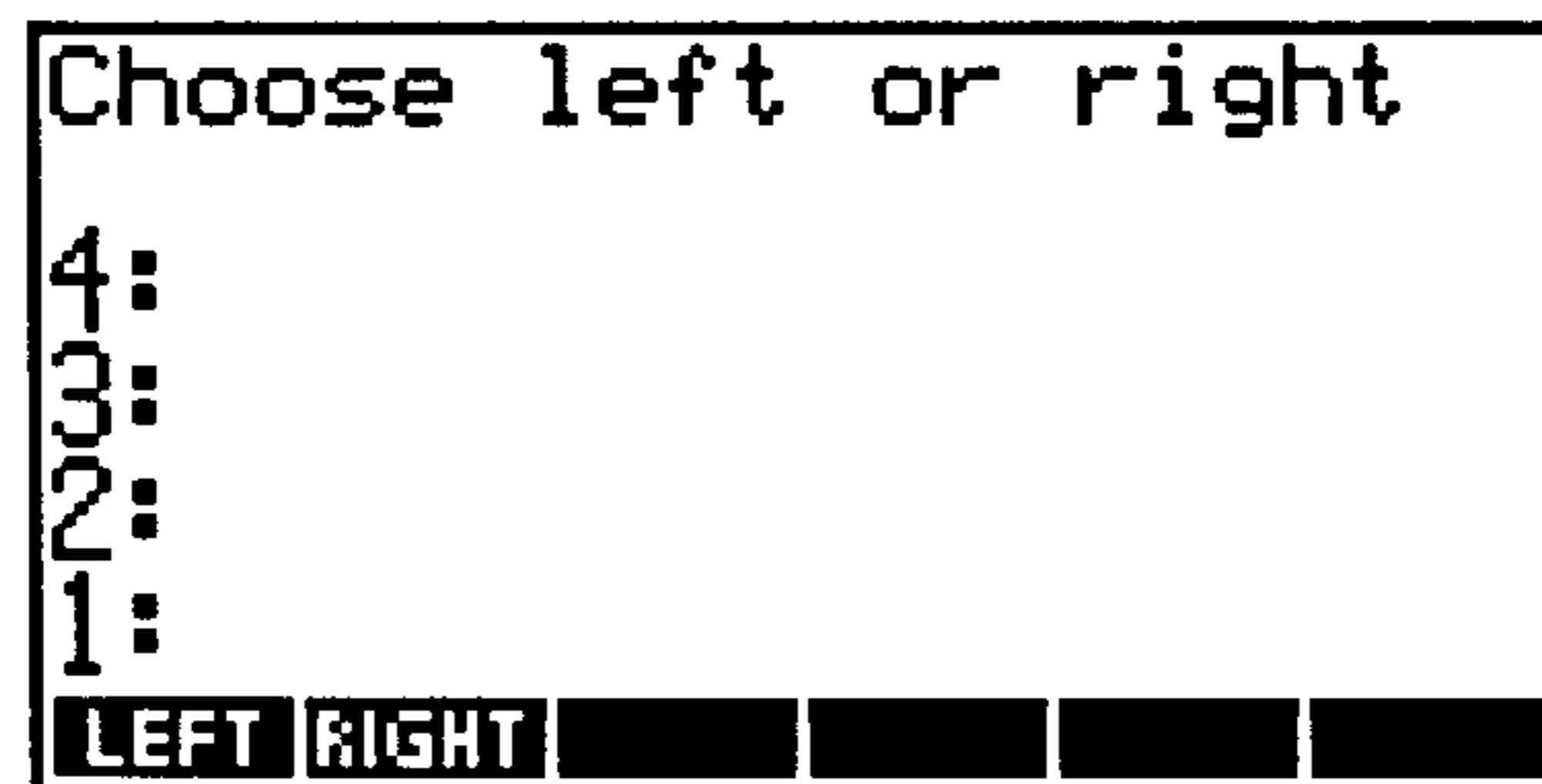
Now the user presses either ≣LEFT≣ or ≣RIGHT≣ , upon which the program resumes, with flag 1 set if "LEFT" was chosen, and clear if "RIGHT" was chosen. This program illustrates two HP 48 programming facilities:

- **TMENU** (*Temporary Menu*). This command is the same as MENU, except that its menu list does not replace the contents of the CST variable, so that the menu it creates is generally not recoverable once you change menus ([⏵] [LAST MENU] will recover it, when it was the menu immediately preceding the current one).

- *Programmable* **CONT.** When CONT is executed in a program, the program containing it is immediately terminated, and the most recently halted program resumes execution. In the above example, pressing ▤LEFT▤ executes the program ≪ 1 SF CONT ≫. This sets flag 1, then resumes the main program that was halted by PROMPT. Note that if there is anything following the CONT in the ▤LEFT▤ program, it does not get executed.

This kind of choice is commonly implemented on the HP 41 by means of local labels, which are automatically assigned to the top-row keys. You can even provide primitive key labels by displaying single characters in the display above the label keys.

5.3.2 Keystroke Input
The input methods outlined so far all suspend program execution indefinitely to wait for object entry; a special terminating key press (e.g. [ENTER] or [◀┐] [CONT]) is required to resume program execution. The HP 48 also provides two commands for entry of individual keystrokes while a program continues executing: KEY and WAIT.

5.3.2.1 KEY
When you press an HP 48 key, a code representing that key is entered into a special memory location called the *key buffer*. When the HP 48 is otherwise idle, it checks the key buffer to see if any key codes have been entered. If so, it removes the codes one at a time (in the same order in which they were pressed), then performs whatever operations are associated with the keys. This two-stage key processing is responsible for the HP 48's "type-ahead" capability, whereby up to 15 keystrokes can be stored in the buffer while the busy annunciator is on.

Programs can check and act on the contents of the key buffer by executing KEY. KEY attempts to remove the oldest keycode from the key buffer. If it succeeds, it returns a two-digit keycode rp to level 2 and a *true* flag (1) to level 1. The first digit r of the keycode is the keyboard row of the key; s is the column. If there are no entries in the key buffer, KEY returns only a false flag (0) to level 1, and no keycode. Note that the keycode does not include a key plane (shift) digit like that used by ASN (section 5.4; see also WAIT, below); a shift key acts like any other key in this case and returns a two-digit code.

By using KEY, programs can accept keyboard input, on a key-by-key basis, without

actually halting execution. Typically, this is done with a simple DO loop:

DO   UNTIL   KEY   END

executes indefinitely until a key is pressed, then returns the keycode. A more elaborate use of KEY is illustrated by the program PSE listed on the next page, which simulates the number entry method of the HP 41 command PSE. PSE waits for up to 1 second; if a key is pressed during that interval, the 1 second wait is started over. If the key is a numeric key (digits [0] through [9], [EEX], [.], or [–]), the key value is appended to a number string. [⇐] removes the last digit; all other keys (except [ON]) are ignored. When one second passes without any key press, PSE looks at the entered string; if it is not null, PSE converts it to a number in level 2 and returns a 1 (*true*) to level 1. If no keys were pressed at all, PSE returns 0 (*true*). You can easily modify PSE to accommodate more keys (including alpha keys if you want to emulate HP 41 PSE more completely) by adding key tests and actions to the CASE structure in the program.

5.3.2.2 WAIT

The WAIT command is designed to produce a simple pause in program execution. x WAIT produces a pause of x seconds, during which program execution does not proceed, but the display is not changed and no key entry is processed (the key buffer will still accumulate key codes). A common application of WAIT is to display messages or other pictures while a program is running. If your program shows a series of messages, you can put a WAIT after one or more of the display commands to ensure that the message remains visible long enough to be read conveniently.

It is also possible to make WAIT pause program execution indefinitely, by using 0 or –1 as its argument. For 0, the current display is not affected by WAIT; for –1, the menu labels are updated to reflect the current menu. In either case, execution resumes only when a key is pressed, then returning the corresponding key code to level 1. The keycode returned by WAIT is a three-digit code *rc.p* like that used by ASN (section 5.4), where r is the key row, c the column, and p the key plane. Note that 0 or –1 WAIT only terminates when a "complete" key is entered, either a non-shift key by itself or such a key preceded by one or more shift keys.

5.3.3 Output Labeling

As mentioned previously, the primary method of labeling output on the HP 41 is to append a computed result to an alpha register string, then display the alpha register. The HP 48 equivalent of this method is to append a computed object as text to a string object that contains the labeling text, then use DISP to display the complete string. The next program SLABEL illustrates the HP 48 method; it makes a labeled display of an object in level 2, using a string from level 1 plus an "=" that the program supplies. A

PSE	*HP 41-like PSE*	C45A

| | ☞ | *x* | 1 |
| | ☞ | 0 | |

```
<< CLLCD  ""                          Start with an empty string.
  WHILE                               Keep going as long as keys are pressed.
   0                                  False means no key press.
   TIME                               Note the time.
   WHILE                              Look for keys while
     TIME  OVER  -  .0001  <          less than one second has passed,
   IF  KEY                            and no key is pressed.
     THEN  4  ROLLD
       3  DROPN  1  DUP  0            A key was pressed, so process it.
   END
   REPEAT
   END  DROP
  REPEAT                              Repeat if a key was pressed.
   → key                              Save the key code.
   <<
     CASE
       { 92 82 83 84 72 73 74 62 63 64 }
       key  POS  DUP                  Digit key?
       THEN  47  +  CHR  +  END       Append the digit.
       DROP
       'key==53'  THEN  "E"  +  END   Append an "E" if the key is EEX.
       'key==85'  THEN  "-"  +  END   Append a "-".
       'key==93'  THEN  "."  +  END   Append a ".".
       'key==55'                      Backspace.
       THEN  DUP
         IF  SIZE  2  <               If < 2 characters...
         THEN  DROP  ""               ...then replace with null string;
         ELSE  0  OVER  SIZE          ...else remove the last character.
           1  -  SUB
         END
       END
       500  .2  BEEP                  Beep for any other key.
     END
     DUP  4  DISP                     Show the current string.
   >>
  END                                 Go get another key.
  IF  DUP  SIZE                       If there was an entry...
  THEN  STR→  1
  ELSE  DROP  0                       ...otherwise, return false.
  END
>>
```

copy of the object is left in level 1.

SLABEL	*Output Labeling Utility*		FD9B
	level 2 *level 1*	|	*level 1*
	object *"label"*	☞	*object*

≪ " = " +	Append " = " to the label string.
OVER	Copy the object.
+	Append the object text to the label string.
CLLCD 1 DISP	Clear the LCD and display the string.
3 FREEZE	Preserve the object display.
≫	

There is a myriad of possible variations of **SLABEL**; for example, the program could take the display line number as an additional argument rather than always using line 1. Or, the program could be extended to split its output over more than one line when the label plus result object are too long to fit on a single line.

5.3.3.1 Using Tagged Objects

Another approach to labeling output objects is to use *tagged objects*. A tagged object is a combination of any object with a *tag*, which is a text descriptor for the object. The tag serves to label the object while it is on the stack. The text does not interfere with any operations that might be applied to the object--the operations ignore the tag. For example,

<div align="center">

:First:1   :Second:2   +   ☞   3.

</div>

The tags :First: and :Second: are removed by **+**, which adds the 1 and the 2 to return 3.

Tagged objects are entered into the command line or programs with the tag text surrounded by :: delimiters, followed by any object. The command →TAG allows you to tag an object after it is created, for output labeling purposes. The tag is represented by a string or name in level 1; the object to be tagged is taken from level 2:

<div align="center">

300_K   "Temp"   →TAG   ☞   :Temp:300_K.

</div>

The tag remains attached to an object until you apply any operation to the object other than a simple stack operation such as **DUP** or **SWAP**. This also includes storing the object to a global variable, although storing to a local variable or a port variable does not remove the tag.

5.4 Key Assignments

An important strength of the HP 41 is its ability to redefine its keyboard so that one or more keys can execute user programs instead of their default operations. In the HP 48, this capability is refined and extended so that you can assign any object to any key, including any of the shifted key variations.

Like the HP 41, the HP 48 allows you to turn a "USER" mode on or off; when this mode is "on", any user key assignments are active and replace the default key definitions. ⏴ USR behaves as a shift key similar to α :

- Pressing ⏴ USR once modifies only the next shifted or unshifted key. The 1USR annunciator appears in the status area while that next keystroke is pending.

- Pressing ⏴ USR twice consecutively turns on USER mode, and the keyboard is redefined according to user key assignments. The 1USR annunciator changes to USER.

- Setting flag −61 eliminates 1USR mode, so that a single press of ⏴ USR turns USER mode on or off.

You can also turn USER mode on or off in a program by setting or clearing flag −62 (similar to HP 41 flag 27).

The HP 48 command ASN for making a single key assignment is modeled on the HP 41CX command PASN: ASN takes the object to be assigned to a key from level 2 (the alpha register for PASN), and a keycode number $rc.p$ from level 1. As usual, the digit r is the key row, c is the column, and p is the plane (shift). Thus

$$'ABC' \quad 34.3 \quad ASN$$

assigns the name ABC to ▷ EVAL (row 3, column 4, shift 3--▷).

On the HP 41, ASN is an interactive operation in which you specify the assignment command by spelling it out in alpha mode, and specify the key by pressing it. The following program ASN41 provides a similar interactive key assignment method for the HP 48. First it uses INPUT to prompt you to enter an object into the command line (you can press ATTN to cancel the new assignment). When you press ENTER , ASN41 displays (Press a key), and waits for a key press (using 0 WAIT). After the key press, the display shows the key code for one second, and the assignment is complete. If you press ENTER at the first prompt without entering any object, any current key assignment for the designated key is cleared.

ASN41	*ASN HP 41-style*	C8A7

`<< RCLF STD -55 CF`	Save current modes, activate STD and LASTARG.
`"Assign: " DUP { V }`	Prompts for definition object.
`IFERR INPUT`	Enter definition.
`THEN 3 DROPN`	If ATTN, then quit.
`ELSE`	Otherwise, proceed.
`  IF DUP "" SAME`	If no entry,
`  THEN "(Clear)" SWAP`	then show (Clear);
`  ELSE "{" OVER + STR→ 1 GET`	else convert entry to an object.
`  END`	
`  3 ROLLD + 3 DISP`	Show the object.
`  "To: " DUP 5 DISP`	Prompt for a key.
`  "(Press a key)`	Newline at end to clear line 7.
`  " 6 DISP`	
`  IFERR 0 WAIT`	Wait for a key.
`  THEN DROP 91`	If ATTN, then keycode 91.
`  END`	
`  SWAP OVER + 5 DISP`	Show the keycode.
`  "" 6 DISP`	
`  IF OVER "" SAME`	If definition is null,
`  THEN DELKEYS DROP`	clear the key definition;
`  ELSE ASN`	else make the assignment.
`  END 1 WAIT`	Pause to make the display visible.
`END STOF`	Restore old modes.
`>>`	

5.4.1 Clearing Key Assignments

To remove a single key assignment from the USER keyboard, you use the command DELKEYS with a keycode argument *rc.p*. In addition to this simple case, DELKEYS can also work with the following arguments:

- The (global or local) name S (for *System*). With this argument, DELKEYS deactivates the default key assignments; that is, all keys not specifically defined by ASN or STOKEYS are inactive in USER mode. Such keys merely beep when pressed. Clearing the default assignments is appropriate when you want to lock out key actions not specifically required by a program. The HP 48 itself does this at times-- for example, when the interactive stack is active, only the menu keys plus a few other keys have any effect.

- A list of key codes, in which case DELKEYS clears the assignments for the all of the specified keys. The list may also contain at its start the name S, in which case the system default assignments are deactivated along with the listed keys.

5.4.2 Multiple Key Assignments

You can also make several HP 48 key assignments at once by using STOKEYS. This command takes as its single argument a list of object pairs, where each pair consists of an object to be assigned followed by a keycode *rc.p*. For example,

$$\{ \ \ ABC \ \ 34.3 \ \ DEF \ \ 42.1 \ \ \} \ \ STOKEYS$$

assigns ABC to ▣ EVAL and DEF to COS . Furthermore, you can include the name S by itself at the start of the list to reactivate the system default assignments for unassigned keys, which you might have cleared with DELKEYS.

5.4.3 Recalling Current Assignments

RCLKEYS recalls all current key assignments, for editing or storage. The key data is returned as a list of object pairs like that used by STOKEYS. If the list includes the name S at its start, this indicates that the system default assignments are active for unassigned keys in USER mode.

6. Program Conversion

The last step in the process of adapting from the HP 41 to the HP 48 is converting programs that you have written for the HP 41 to the HP 48. Unfortunately, there is no magic prescription for this, short of adding an *HP 82210A HP 41CV Emulator Application Card*, which can automatically translate and run many HP 41 programs on the HP 48. This card, however, leaves programs in their HP 41 language form, so that you cannot easily modify the programs to take advantage of the HP 48's full capabilities. Here we will consider the problem of converting HP 41 programs into the HP 48 RPL language.

Because the HP 41 and the HP 48 both use RPN logic, you might expect that an HP 41 to HP 48 program conversion could be done easily line-by-line. However, although that approach is possible, it turns out not to be very easy:

- Many HP 41 programs make use of the four-level stack limit, and the automatic replication of the T-register contents. These uses are often hard to notice in HP 41 program listings unless they are carefully documented.

- There are significant differences in program control commands between the two calculators.

- A line-by-line translation will not take advantage of the HP 48's power and ease of programming.

To demonstrate the last point, consider a program that finds the roots of a quadratic equation $ax^2 + bx + c = 0$, where the coefficients a, b, and c are entered on the stack. The listing on the next page shows an HP 41 program to perform this calculation, together with an almost line-for-line translation into HP 48 language. This program returns the real part and imaginary part of the first root to the HP 41 T-register (HP 48 level 4) and Z-register (3), respectively, and the real and imaginary parts of the second root to Y (2) and X (1).

While the HP 48 version runs perfectly well, it does not make good use of the HP 48's resources. For example, the HP 48 handles complex numbers as easily as real numbers; thus there is no reason to test for the negative square root case, nor to return the real and imaginary parts of the roots separately. Also listed on the next page is a more efficient HP 48 program. The second version is only 1/3 as large as the first HP 48 program (57.5 bytes versus 170 bytes), and runs 50% faster (.09 seconds versus .14 seconds).

HP 41 and HP 48 Quadratic Equation Programs

	HP 41	Comments	HP 48
01	LBL"QUADR"		≪
02	RCL Z	X = a.	3 PICK
03	/	X = $c' = c/a$.	/
04	X<>Y	X = b.	SWAP
05	RCL Z	X = a.	3 PICK
06	/	X = b/a.	/
07	2		2
08	/	X = $b' = b/2a$.	/
09	CHS		NEG
10	STO 00	R00 = $-b'$.	DUP 'B' STO
11	X↑2		SQ
12	X<>Y		SWAP
13	–	X = $r = (b'^2 - c')^{\frac{1}{2}}$	–
14	CF 01		1 CF
15	X<0?		IF DUP 0 <
16	SF 01	Flag 01 set indicates complex roots.	THEN 1 SF END
17	ABS		ABS
18	SQRT		√
19	STO 01	R01 = $\sqrt{|r|}$.	DUP 'R' STO
20	RCL 00	b'.	B
21	X<>Y		SWAP
22	FS? 01	If complex case, then root is $b' + i\sqrt{|r|}$.	IF 1 FC?
23	GTO 00		
24	+	Real case: real part is $b' + \sqrt{r}$.	THEN +
25	0	Imaginary part is 0.	0
26	LBL 00		END
27	RCL 00	b'.	B
28	RCL 01	$\sqrt{|r|}$.	R
29	CHS		NEG
30	FS?C 01	If complex case, then root is $b' - i\sqrt{|r|}$.	IF 1 FC?C
31	RTN		
32	+	Real case: real part is $b' - \sqrt{r}$.	THEN –
33	0	Imaginary part is 0.	0 END
34	END		≫

Better HP 48 Quadratic Equation Program

≪	a b c
3 PICK /	a b c/a
SWAP ROT 2 * / NEG	c/a $-b/2a$
DUP SQ	c/a $-b/2a$ $b^2/4a^2$
ROT – √	$-b/2a$ $\sqrt{[(b/2a)^2 - c/a]}$
DUP2 +	$-b/2a$ $\sqrt{[(b/2a)^2 - c/a]}$ x_1
3 ROLLD –	x_1 x_2
≫	

The preceding examples illustrate that even when a HP 41 program is easily translatable line-by-line into HP 48 language, you generally obtain superior HP 48 programs if you start from a program's definition or algorithm rather than from the HP 41 program listing. This is particularly true when the program in question is primarily a representation of an algebraic expression, since the HP 48 has a wealth of capabilities for expressions, for which the HP 41 has no equivalent. In the case of the current example, the mathematical algorithm for the roots x is the formula

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

Using HP 48 local variables and algebraic objects, it is easy to translate the algorithm into a program:

	a b c
`«`	
`→  a  b  c`	Name the arguments.
`  «   '(-b+√(b^2-4*a*c))/(2*a)'   EVAL`	x_1
`      '(-b-√(b^2-4*a*c))/(2*a)'   EVAL`	x_2
`  »`	
`»`	

This version is longer (151 bytes) and slower (.244 seconds) than the optimized RPN version listed previously, primarily because it performs many of the arithmetic operations twice, once in each algebraic expression. However, because it corresponds so closely to the quadratic formula, it is very easy to create, and correspondingly easy to understand or modify later--virtues which often outweigh any advantages of speed or program size.

In some 41-to-48 translation cases, you may not have a clearly defined program algorithm and so must work from the HP 41 program listing. Or, you may wish for other reasons to keep a HP 48 program as similar as possible to its HP 41 equivalent. In the remainder of this chapter, we will show how to reproduce various HP 41 program constructs on the HP 48, and conclude by listing the closest HP 48 equivalents to HP 41 commands.

6.1 Storage Registers

As described in Chapter 3, the HP 48 replaces HP 41-style storage registers with named *variables*. In many translation cases, it is adequate to replace HP 41 register references with individual HP 48 variables named, for example, R00, R01, etc. However, you will generally find the translation task easier if you set up a single HP 48 variable to represent HP 41 register memory. The variable, which we'll call REG41 here, should contain a list, each element of which plays the role of a single register. To manage the list, you need an analog of the HP 41 SIZE or (PSIZE) command:

SIZE41	*Create or Resize Register List*	0C31
	level 1 | level 1	
	n *☞*	

≪ → n	Save the new size as n.
≪ 'REG41'	
IF DUP VTYPE 5 ≠	Check current REG41.
THEN { } OVER STO	Store initial null list.
END EVAL SIZE n −	Size change.
CASE DUP 0 >	Current REG41 too big?
THEN DROP REG41 1 n SUB END	Shrink to size *n*.
DUP 0 <	Current REG41 too small?
THEN NEG 1 →LIST 0 CON	Vector of zeros.
OBJ→ OBJ→ DROP →LIST	Change to a list.
REG41 SWAP +	Concatenate zeros to register list.
END	
DROP REG41	Current REG41 just right.
END 'REG41' STO	Replace previous REG41.
≫	
≫	

(Notice that if REG41 does not already exist in the current directory, but does exist in a parent directory, a new REG41 is created in the current directory, with the same contents as that in the parent directory for the register range requested for SIZE41.)

With REG41 defined, it is straightforward then to devise HP 48 translations of HP 41 STO and RCL, as shown by the programs RSTO and RRCL listed on the next page. Using these programs, you can convert HP 41 register store and recall operations to HP 48 equivalents, with the simple requirement that the order of operation and register number appears reversed:

STO 55	becomes	55 RSTO
RCL 27	becomes	27 RRCL
RCL X	becomes	'X' RRCL

'Y', 'Z', and 'T' are also valid arguments for RSTO and RRCL.

RSTO	Register STO		B6A2
	level 2	level 1		level 1
	object	n	☞	object

```
« IF DUP TYPE 0 ==                              Data register?
  THEN 'REG41' SWAP 1 + 3 PICK PUT
  ELSE                                          Must be a stack register.
    { « »                                       For STO X.
      « SWAP DROP DUP »                          STO Y.
      « ROT DROP DUP 3 ROLLD »                   STO Z.
      « 4 ROLL DROP DUP 4 ROLLD »                STO T.
    }                                           List of possible STO cases.
    { X Y Z T }                                 List of STO programs.
    ROT POS                                     Which argument?
    GET                                         Get the corresponding STO
                                                program.
    EVAL                                        Do the store.
  END
»
```

RRCL	Register RCL		9764
	level 1		level 1
	n	☞	object

```
« IF DUP TYPE 0 ==                              Data register?
  THEN 'REG41' SWAP 1 + GET                     Do register RCL.
  ELSE                                          Must be a stack register.
    { DUP                                       For RCL X.
      OVER                                       RCL Y.
      « 3 PICK »                                 RCL Z.
      « 4 PICK »                                 RCL T.
    }                                           List of possible RCL cases.
    { X Y Z T }                                 List of RCL programs.
    ROT POS                                     Which argument?
    GET                                         Get the corresponding RCL pro-
                                                gram.
    EVAL                                        Do the recall.
  END
»
```

The next two programs emulate HP 41 CLRG and CLRGX.

CLRG	*Clear Registers*	1D3B

`<< REG41 SIZE 0 CON`	Make a vector of zeros the same size as REG41.
`'REG41' STO`	Replace old registers.
`>>`	

CLRGX	*Clear Registers by X*	162E

	level 1		*level 1*
	bbb.eeecc	☞	*bbb.eeecc*

`<< DUP DUP`	
`IP 1 +`	*bbb* +1.
`SWAP FP 1000 *`	*eee.cc.*
`DUP IP 1 +`	*eee.*
`SWAP FP 100 *`	*cc.*
`3 ROLLD`	
`FOR n`	Loop from *bbb* +1 to *eee* +1.
`'REG41' n 0 PUT`	Zero *n*th register.
`DUP`	
`STEP`	
`DROP`	
`>>`	

6.1.1 Indirect Storage
The following programs reproduce HP 41 indirect storage operations:

RSTOIND	*Register Store Indirect*	09A1

	level 2	*level 1*		*level 1*
	object	*n*	☞	*object*

`<< RRCL RSTO`	
`>>`	

RRCLIND	Register RCL Indirect	F6B2

level 1		level 1
n	☞	object

```
≪  RRCL  RRCL
≫
```

6.1.2 Storage Arithmetic

The four HP 41 storage arithmetic commands STO+, STO−, STO*, and STO/ correspond to these HP 48 programs:

STOPLUS	HP 48 Register STO +	EE19

level 2	level 1		level 1
object	n	☞	object

```
≪  DUP2  RRCL  SWAP  +  SWAP  RSTO  DROP
≫
```

STOMINUS	HP 48 Register STO −	D121

level 2	level 1		level 1
object	n	☞	object

```
≪  DUP2  RRCL  SWAP  −  SWAP  RSTO  DROP
≫
```

STOTIMES	HP 48 Register STO *	45B0

level 2	level 1		level 1
object	n	☞	object

```
≪  DUP2  RRCL  SWAP  *  SWAP  RSTO  DROP
≫
```

STODIVIDE	*HP 48 Register STO/*	501F

level 2	level 1		level 1
object	n	☞	object

```
≪ DUP2 RRCL SWAP / SWAP RSTO DROP
≫
```

6.1.3 Block Moves

The next two programs are based on the HP 41CX REGSWAP and REGMOVE commands (see also section 3.5). These commands use a control number of the form *sss.dddnnn*, where *sss* is the number of the first register of a block of *nnn* registers called the *source*, and *ddd* is the number of the first register of a block of the same size called the *destination*. REGMOVE copies the contents of the source registers to the destination block; REGSWAP exchanges the contents of the two blocks.

The HP 48 program REGSWAP does not attempt to reproduce the behavior of the HP 41CX command for the case where the source and destination blocks overlap, i.e. $|sss - ddd| < nnn$. The HP 41CX owner's manual tells you to avoid this case, although the command does not error. On the HP 48, the higher-numbered block is copied to the lower-numbered position, then the original lower-numbered block is copied to the higher-numbered position.

REGMOVE	*Register Move*	A782

level 1		level 1
sss.dddnnn	☞	sss.dddnnn

≪ DUP DUP IP 1 +	First element of source.
SWAP FP 1000 * DUP IP 1 +	First element of destination.
SWAP FP 1000 *	Number of registers.
→ sss ddd nnn	Save list parameters.
≪ REG41 1 ddd 1 − SUB	List prior to source.
REG41 sss DUP nnn + 1 − SUB	Source list.
REG41 ddd nnn + OVER SIZE SUB	List after destination.
+ +	Assemble new list.
'REG41' STO	Replace old list.
≫	
≫	

REGSWAP	*Register Swap*	9ECB
	level 1 | *level 1*	
	sss.dddnnn ⊏⊐ *sss.dddnnn*	

≪ DUP DUP IP 1 +	First element of source.
SWAP FP 1000 * DUP IP 1 +	First element of destination.
SWAP FP 1000 *	Number of registers.
3 ROLLD DUP2	
IF > THEN SWAP END	Ensure source < destination.
DUP 4 PICK +	First element past destination.
3 PICK 5 ROLL +	First element past source.
→ sss ddd end mid	Save list parameters.
≪ REG41 1 sss 1 − SUB	List prior to source.
REG41 ddd end 1 − SUB +	Concatenate destination block.
REG41 mid OVER SIZE SUB	Intermediate list, with source block replaced.
DUP 1 ddd 1 − SUB	Int. list prior to destination.
REG41 sss mid 1 − SUB +	Concatenate original source block.
SWAP end OVER SIZE SUB	Intermediate list from end of destination block.
+	New list.
'REG41' STO	Replace old list.
≫	
≫	

It is possible to write more memory efficient versions of the above programs, where REG41 is copied to the stack, expanded into its elements using OBJ→, and the register contents are moved individually with stack manipulations. However, such versions would generally be significantly slower than those listed here.

6.1.4 The Alpha Register
The HP 48 has no need for an *alpha register* like that of the HP 41, since the HP 48 can handle text strings of any size as stack objects. For purposes of HP 41 program conversion, you can designate a HP 48 variable (here called ALPHA) to act as an alpha register. Executing the program CLA listed next initializes an empty alpha "register" for the HP 48; an HP 41 program line *"text"*, which replaces the alpha register contents with *text*, is then reproduced on the HP 48 with "text" ENTERα.

CLA	Clear ALPHA	F0F1

```
≪  ""  'ALPHA'  STO
≫
```

ENTERα	Enter Text	6F27
	level 1		
	"text"	☞	

≪ DUP SIZE DUP 24 – SWAP SUB 'ALPHA' STO ≫	Limit to 24 characters. Store in ALPHA.

The programs listed next mimic the common HP 41 alpha register operations. All assume that a variable containing a string object already exists; the programs could check this upon each execution, but this would generally slow execution.

APPEND	Append Text	4E98
	level 1		level 1
	"text"	☞	

```
≪  ALPHA  SWAP  +  ENTERα
≫
```

ARCL	Alpha RCL	A4D5
	level 1		level 1
	n	☞	

≪ RRCL APPEND ≫	

ASTO	*Alpha STO*	9D69

level 1		*level 1*
n	☞	

≪ ALPHA 1 6 SUB SWAP RSTO ≫	First 6 characters. Store in register *n*.

ASHF	*Alpha Shift*	9149

≪ ALPHA 7 OVER SIZE SUB ENTERα ≫	Remove first six characters. Store in ALPHA.

AROT	*Rotate Alpha Register*	2D14

≪ ALPHA DUP 2 OVER SIZE SUB
    SWAP 1 1 SUB
    + ENTERα
≫
	End of string.
	Get first character.
	Restore rotated string.

ATOX	*Alpha to X*	C8E5

		level 1
	☞	*n*

≪ ALPHA DUP NUM SWAP 2 OVER SIZE SUB ENTERα ≫	Get first character number. Remove first character from ALPHA.

XTOA	*X to Alpha*	45D0
	level 1	*level 1*
	n ☞ *n*	

```
« ALPHA OVER CHR APPEND DROP
»
```

6.2 Replacing GTO

One of the major tasks in converting an HP 41 program into the HP 48 language is replacing program jumps using GTO into HP 48 program structures. If a HP 41 program is sufficiently convoluted, this task may be difficult--again, it might be easier to return to the program's basic conceptual algorithm rather than attempting a line-by-line conversion.

The uses of GTO can generally classified as *conditional* and *unconditional*. In a conditional GTO, the GTO is executed or not according to the result of a program test:

- When the GTO is (logically) forward, to a new part of the program, we refer to this as a program *branch*--the program splits into two paths.

- A conditional GTO that jumps backwards, to repeat execution of a sequence one or more times, is called *iteration*.

- GTO's may be used to provide one or more *exits* from iteration loops.

Unconditional GTO's are used primarily to minimize program size through reuse of code common to more than one part of a program; and for exiting from an iteration loop.

We will consider the HP 48 representation of each of these uses of HP 41 GTO in the next sections.

6.2.1 Program Branches

Figure 6.1 illustrates the typical structure of an HP 41 program conditional branch. The *test* is any HP 41 test command, such as FS? or X>Y?. Test commands have the property that if the test is true, the next program step is executed, and if false, the next step is skipped. The most general branching is obtained when the test is followed by a GTO; for any other command, the test command effectively performs the GTO itself by conditionally skipping the next step.
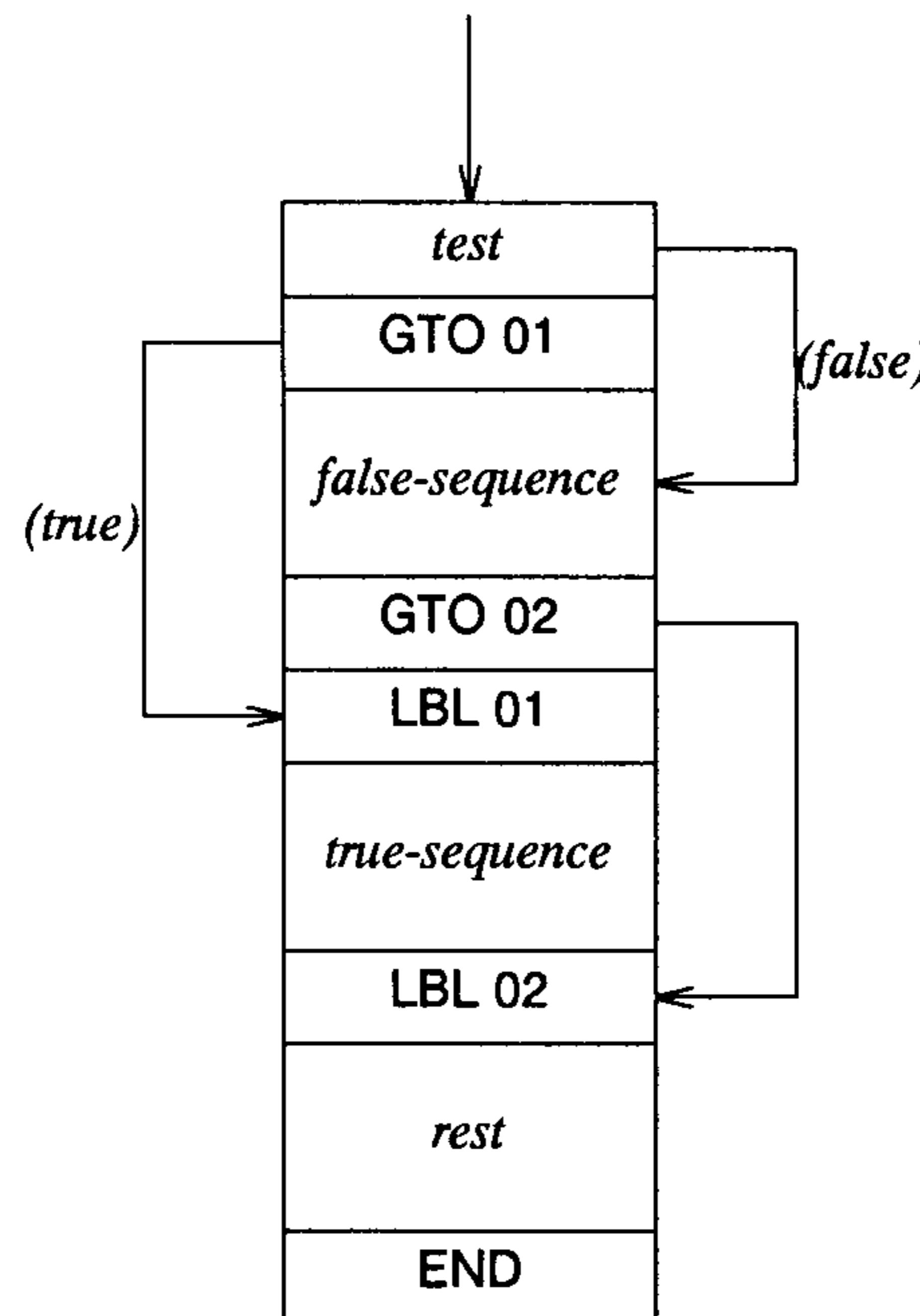
Figure 6.1. HP 41 Conditional Branching

On the HP 48, the equivalent of the HP 41 program in the Figure 6.1 is an IF structure (section 4.4.1):

IF *test* THEN *true-sequence* ELSE *false-sequence* END *rest*.

■ *Example.* This HP 41 routine tests a number in the X-register, taking its absolute value and setting flag 01 if it is negative, or clearing flag 01 otherwise:

```
01 X<0?
02 GTO 01
03 CF 01
04 GTO 02
05 LBL 01
06 ABS
07 SF 01
08 LBL 02
09 END
```

The HP 48 equivalent is

> IF   DUP   0   <   THEN   NEG   1   SF   ELSE   1   CF   END

6.2.2 Definite Iteration

Definite iteration is the repeated execution of a program sequence a predetermined number of times. In HP 41 programs, this is commonly achieved by the use of ISG or DSE witha register containing a number *iii.fffcc* that controls the iteration. *iii* is the initial (or current) value for a counter, *fff* is the final value, and *cc* is the counter increment. A typical program sequence is shown in Figure 6.2.
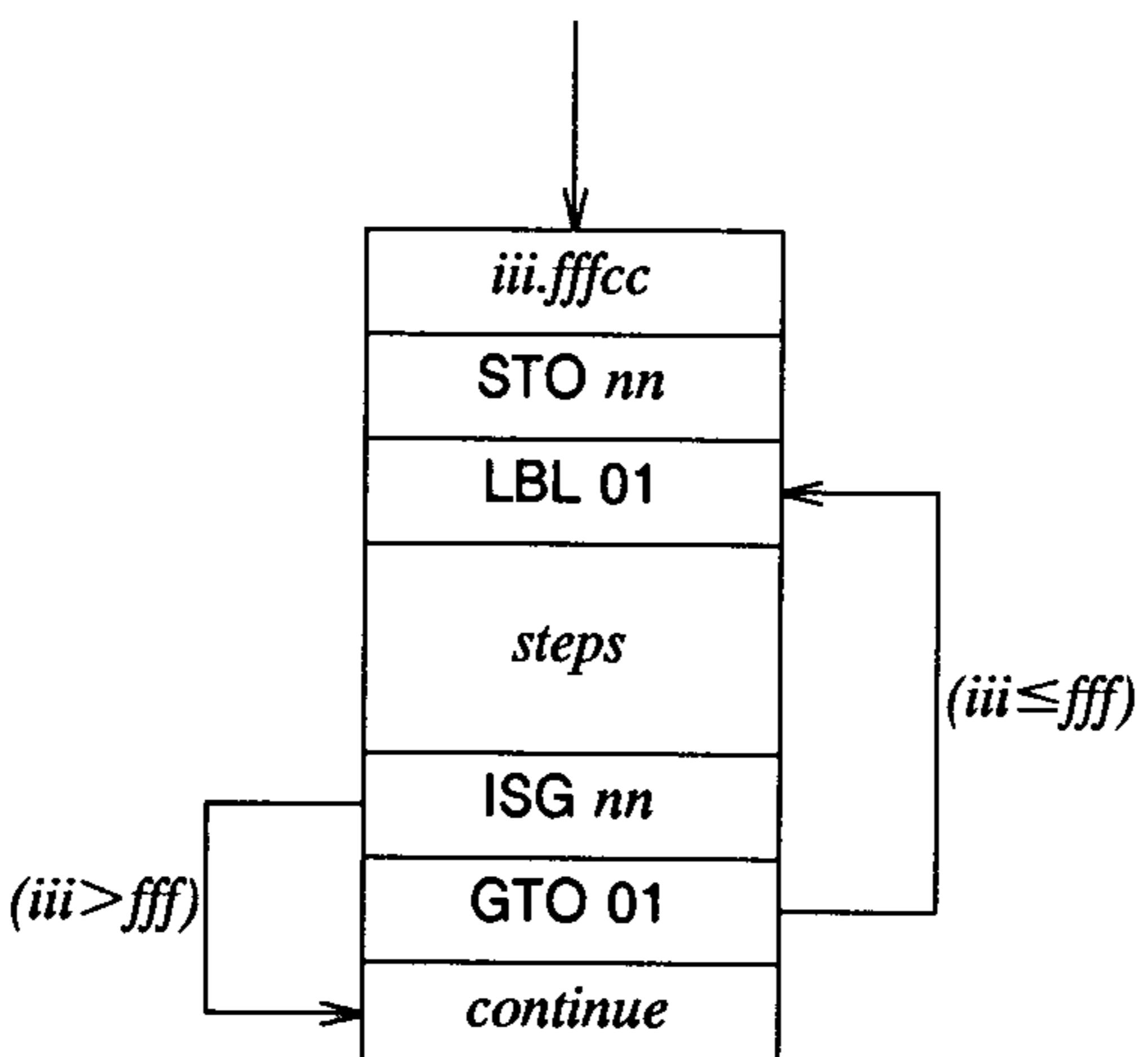
Figure 6.2. HP 41 Definite Iteration

The HP 48 provides the FOR...NEXT and FOR...STEP structures for definite iteration. The HP 48 translation of the above HP 41 program is

> *iii*   *fff*   FOR   *name*   ...*steps*...   *cc*   STEP

- For the special case of $cc = 1$, use NEXT in place of 1 STEP.

- To perform a DSE loop on the HP 48, use a negative *cc* with STEP, and add one to *fff*, since DSE does not iterate when *fff* = *iii*.

A local variable *name* contains the loop counter; its value can be recalled (by evaluating *name*) or changed (by storing into *name*) while the loop is executing.

■ *Example.* For an argument n, compute the sum $\sum_{i=1}^{n} n^2$.

HP 41 Version:

01	LBL"SUMSQ"	n is in X.
02	1000	
03	/	
04	1	
05	+	Control number $1.n$.
06	0	Initial value for sum.
07	LBL 01	Start of iteration loop.
08	RCL Y	Recall control number.
09	INT	Extract current counter.
10	X↑2	Square it.
11	+	Add to sum.
12	ISG Y	Increment counter.
13	GTO 01	Iterate.
14	END	Sum is in X.

HP 48 Version:

SUMSQ	*Sum of Squares*		751F
	level 1	*level 1*	
	n ☞	sum	

≪ 0	Initial value for sum.
1	Initial value of counter.
ROT	Final value of counter
FOR i	Start of iteration loop.
i SQ +	Add i^2 to sum.
NEXT	Iterate.
≫	

Notice that the HP 48 version also works with non-integer n.

6.2.3 Indefinite Iteration

Indefinite iteration is iteration where you don't know in advance how many repeats are necessary. Instead, a program sequence is repeated until a condition is met. On the HP 41, such a sequence is typically delimited at the start by a label, and at the end by a test command followed by a GTO to the label:
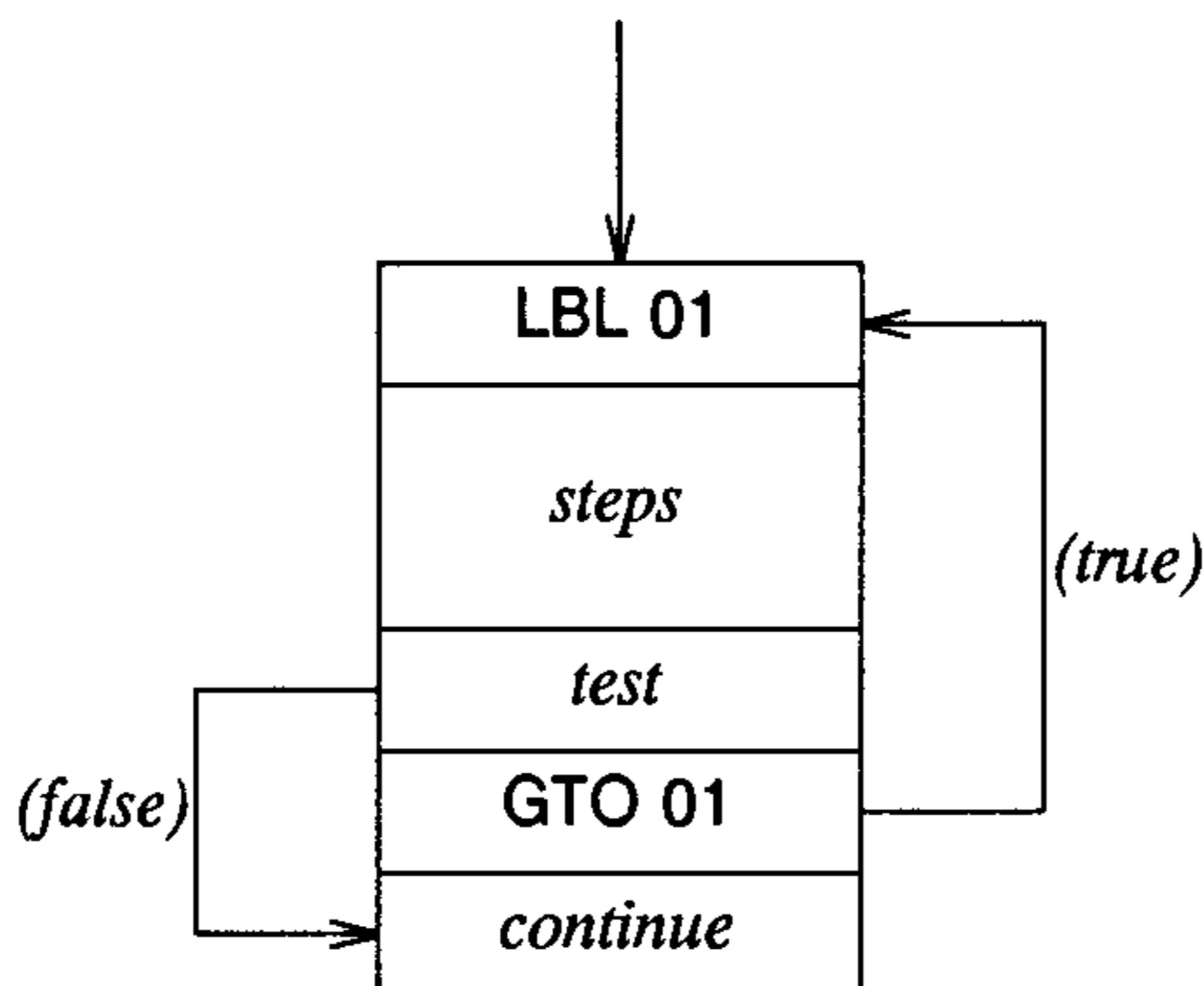
Figure 6.3. HP 41 Indefinite Iteration

Similar iteration is obtained on the HP 48 by using a DO loop (section 4.5.2.1):

$$\text{DO} \quad \textit{steps} \quad \text{UNTIL} \quad \textit{test} \quad \text{NOT} \quad \text{END} \quad \textit{continue}$$

The NOT is included in the HP 48 sequence so that the *test* has the same sense as in the HP 41 program. In the HP 41 case, the iteration continues while the test is *true*; the HP 48 DO loop normally repeats as long as the test is false, but the use of NOT reverses the test result.

■ *Example*: Compute the sum $\sum\limits_{i=1}^{\infty} \dfrac{1}{i^4}$.

In order to stop after a finite number of terms are accumulated, the programs below compute successive terms until the current sum no longer changes when a new term is added. This yields results that are approximately accurate to the full numerical precision of the calculators--10 places on the HP 41 and 12 places on the HP 48. This kind of stopping criterion does not always ensure full accuracy, depending on how rapidly the series converges. If the convergence is slow, the remaining terms in the sums that are not accumulated may in fact be large enough to affect at least the last decimal place of the programs' results.

HP 41 Version:

01	LBL"SUM4"	
02	0	Initial value for i.
03	0	Initial value for sum.
04	ENTER↑	Copy initial value.
05	LBL 01	Start of iteration loop.
06	RDN	Discard old sum.
07	X<>Y	
08	1	
09	+	Increment counter.
10	ENTER↑	
11	ENTER↑	
12	−4	
13	Y↑X	Next term.
14	RCL Z	Old sum.
15	+	New sum.
16	RCL Z	Old sum.
17	X≠Y	Did the sum change?
18	GTO 01	Then iterate.
19	END	Sum is in X.

HP 48 Version:

SUM4	*Sum Fourth Powers*	2589
	| *level 1*	
	☞ *sum*	

≪ 0	Starting value of n.
0	Initialize sum.
DO	Start of iteration sequence.
SWAP 1 +	Increment n.
DUP −4 ^	Next term.
3 PICK +	New sum.
ROT OVER	
UNTIL = =	Keep going until $sum(n+1) = sum(n)$.
END SWAP DROP	Drop n.
≫	

6.2.4 Reducing Program Size

The primary purpose of an unconditional GTO on the HP 41 is to save program memory by allowing two or more program sequences to jump to a common destination--the reverse of a program branch. Here is a prototype HP 41 program

where two parts of the program branch to a common point:

$$
\begin{array}{ll}
01 & \text{LBL "FIRST"} \\
02 & first_1 \\
... & \\
30 & first_n \\
31 & \text{GTO "COMMON"} \\
... & \\
55 & \text{LBL "SECOND"} \\
56 & second_1 \\
... & \\
80 & second_m \\
81 & \text{LBL "COMMON"} \\
82 & common_1 \\
... & \\
99 & common_p \\
\end{array}
$$

Program lines such as "$first_1$" are intended to represent arbitrary sequences of program steps. The line numbers are also unimportant.

On the HP 48, you must write FIRST, SECOND, and COMMON as separate program objects, where COMMON is used as a subroutine by the first two programs:

$$ \ll \quad first_1 \quad ... \quad first_n \quad \text{COMMON} \quad \gg \quad \text{'FIRST'} \quad \text{STO} $$

$$ \ll \quad second_1 \quad ... \quad second_m \quad \text{COMMON} \quad \gg \quad \text{'SECOND'} \quad \text{STO} $$

$$ \ll \quad common_1 \quad ... \quad common_p \quad \gg \quad \text{'COMMON'} \quad \text{STO} $$

The HP 48 "style" has the advantage that any program module, which may consist of a series of objects or a decision or loop structure, has only one entrance and one exit. This makes it simple to test modules independently, then combine them into larger modules, and so on. This style is also recommended for the HP 41, although its limited return stack requires you to use some care when you break up programs into subroutines.

6.2.5 Exits

HP 41 iteration loops that have more than one exit may use a GTO in the middle of the iteration sequence to jump out of the loop. These naturally are conditional jumps, since an unconditional jump out of an iteration loop would defeat the design of the iteration. A prototype program is shown in Figure 6.4.
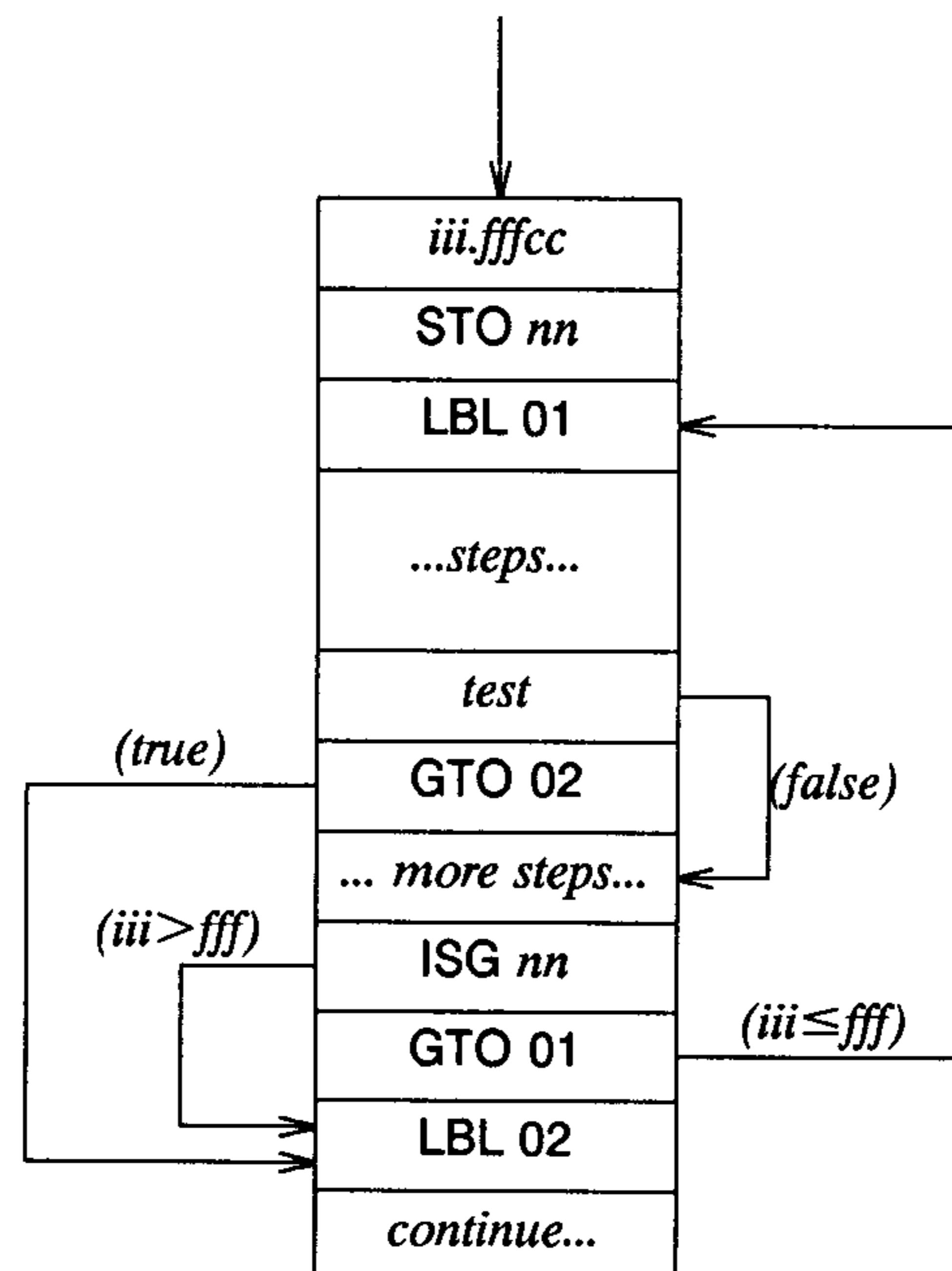
Figure 6.4. HP 41 Definite Iteration with an Exit

On the HP 48, you can't jump out of a FOR loop except at the STEP or NEXT. To reproduce the HP 41 structure in Figure 6.4, you must use a conditional branch *within* the loop, where the exit branch store the stop value in the loop counter so that STEP will not repeat the loop:

iii	Start value.
fff	Stop value.
FOR x	Start of loop.
steps	
IF *test*	Exit test.
THEN *fff* 'x' STO	Store *fff* in counter.
ELSE	
more steps	
END	End of conditional.
cc	step size
STEP	

■ *Example.* Compute $\sum_{n=0}^{100} \frac{1}{n^m}$, where m is a positive number. For this purpose, we will use a definite loop, but the loop will exit if addition of a term yields no change in the

sum.

HP 41 Version:

01	LBL "INVPOW"	
02	STO 01	Save *n*.
03	1.1	Loop control number.
04	STO 02	Save control number in R02.
05	0	Initial value of sum.
06	LBL 01	Start of loop.
07	RCL 01	*n*.
08	RCL 02	
09	INT	*m*
10	CHS	$-m$.
11	Y↑X	Next term
12	RCL Y	Current sum.
13	+	New sum.
14	X=Y?	No change?
15	GTO 02	...then exit.
16	ISG 02	...otherwise loop if $n < 100$
17	GTO 01	
18	LBL 02	
19	END	

HP 48 Version:

INVPOW	*Sum of Inverse Powers*	B5D4
	level 1	*level 1*
	m ☞	*sum*

≪ → m	Save the power argument.
≪ 0	Initial value for sum.
1 100	Loop limits.
FOR n	Loop with index n.
DUP	Copy the sum.
m n NEG ^	Compute the next term.
+	New sum.
SWAP OVER	
IF = =	Compare new and old sums.
THEN 100 'm' STO	If the same, then exit.
END	
NEXT	Next iteration.
≫	
≫	

6.3 An Example of Program Conversion

To demonstrate the process of converting HP 41 programs to the HP 48, we will translate the "Sine, Cosine, and Exponential Integrals" program from the the HP 41 *High Level Math* solution book. This program calculates the integrals defined as follows:

Sine Integral:

$$Si(x) = \int_0^x \frac{\sin t}{t} dt,$$

which may be found from the infinite series

$$Si(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)} \cdot (2n+1)!$$

$$Si(-x) = -Si(x)$$

Cosine Integral:

$$Ci(x) = \gamma + \ln x + \int_0^x \frac{\cos t - 1}{t} dt,$$

computed from

$$Ci(x) = \gamma + \ln x + \sum_{n=1}^{\infty} \frac{(-1)^n x^{2n}}{2n(2n)!}$$

$$Ci(-x) = Ci(x) - i\pi \text{ for } x > 0$$

where $\gamma = 0.5772156649$ (*Euler's constant*).

Exponential Integral:

$$Ei(x) = \int_{-\infty}^x \frac{e^t}{t} dt = \gamma + \ln x + \sum_{n=1}^{\infty} \frac{x^n}{n \cdot n!}$$

for x > 0.

The solution book notes that for Si(x) and Ci(x), the accuracy of the program output

decreases as x increases; for $x \approx 20$, results are accurate only to the second decimal place. Furthermore, for sufficiently large x, the terms of the sum for Ei(x) will overflow the floating-point number range of the calculator, causing a program-halting error. Our HP 48 translations of the programs do not attempt to improve upon these limitations; our focus is on the translation process rather than on the programs themselves.

Here is the solution book program listing:

01	LBL "SI"		40	CHS	
02	SF 27	Initialize.	41	STO 00	
03	STOP		42	1	
04	LBL A	Si(x)	43	STO 02	
05	STO 02		44	0	
06	STO 03		45	STO 01	
07	X↑2		46	LASTX	
08	CHS		47	XEQ 01	
09	STO 00		48	"C"	
10	1		49	GTO 00	
11	STO 01		50	LBL 01	
12	RCL 02		51	LN	
13	"S"		52	.5772156649	
14	LBL 00	Loop to add terms.	53	+	
15	RCL 00		54	RTN	
16	RCL 01		55	LBL C	Ei(x)
17	1		56	STO 03	
18	+		57	STO 00	
19	/		58	1	
20	LASTX		59	STO 02	
21	XEQ 02		60	0	
22	X≠Y		61	STO 01	
23	GTO 00		62	RCL 00	
24	GTO 04		63	XEQ 01	
25	LBL 02	Common subroutine	64	LBL 03	
26	1		65	RCL 00	
27	+		66	RCL 01	
28	STO 01		67	XEQ 02	
29	/		68	X≠Y	
30	RCL 02		69	GTO 03	
31	*		70	"E"	
32	STO 02		71	LBL 04	Display routine.
33	RCL 01		72	FIX 2	
34	/		73	"⊢I<"	
35	+		74	ARCL 03	
36	RTN		75	"⊢>="	
37	LBL B	Ci(x)	76	ARCL X	
38	STO 03		77	AVIEW	
39	X↑2		78	END	

The listing contains very few comments, so it provides a good test case for attempting a line-by-line translation.

The first step in the translation process is to identify the program's subroutines and represent them in the HP 48 as individual program variables. In the above program, the subroutines start at lines 14 (LBL 00), 25 (LBL 02), 47 (LBL 01), and 71 (LBL 04). The last of these is used as a common exit for all three integral programs, but on the HP 48 you must call the exit program as a proper subroutine without using GTO (section 6.2).

Here are the four subroutine translations:

LBL00	LBL 00 Subroutine	EB49
	level 1 | *level 1*	
	Initial Sum ☞ *Sum*	

≪ DO	Start of loop (LBL 00).
DUP	Extra copy of current sum.
0 RRCL	15 RCL 00
1 RRCL	16 RCL 01
1	17 1
+	18 +
/	19 /
LASTARG SWAP DROP	20 LASTX
LBL02	21 XEQ 02
UNTIL	
DUP ROT = =	22 X≠Y
END	23 GTO 00
≫	

LBL01	LBL 01 Subroutine	FDA8
	level 1 | *level 1*	
	x ☞ $LN(x) + \gamma$	

≪ LN	51 LN
.5772156649	52 .5772156649
+	53 +
≫	54 RTN

LBL02	*LBL 02 Subroutine*	1FDC

≪ 1	26 1
+	27 +
1 RSTO	28 STO 01
/	29 /
2 RRCL	30 RCL 02
*	31 *
2 RSTO	32 STO 02
1 RRCL	33 RCL 01
/	34 /
+	35 +
≫	36 RTN

LBL04	*LBL 04 Subroutine*	E308

≪ 2 FIX	72 FIX 2
"I(" APPEND	73 "⊢I<"
3 ARCL	74 ARCL 3
")=" APPEND	75 "⊢>="
'X' ARCL	76 ARCL X
AVIEW	77 AVIEW
≫	78 END

The remainder of the translation consists of converting the main programs. First note that the program starts with the sequence SF 27 STOP. This initializing routine only serves to turn on HP 41 USER mode, to take advantage of the automatic assignment of LBL's A through E to the HP 41's top row of keys. There is no corresponding automatic assignment of HP 48 keys; instead, you can use the following initializing routine:

SICOS	*Sine, Cosine, & Exponential Integrals*	87FC

≪ { SI CI EI } TMENU	Set up a temporary custom menu.
≫	

The translations of the three main programs are as follows:

SI	Sine Integral		BC56
	level 1	|	*level 1*
	x	☞	Si(x)

« 2 RSTO	05 STO 02
3 RSTO	06 STO 03
SQ	07 X↑2
NEG	08 CHS
0 RSTO	09 STO 00
1	10 1
1 RSTO	11 STO 01
DROP2	Discard unneeded stack objects.
2 RRCL	12 RCL 02
"S" ENTERα	13 "S"
LBL00	14 LBL 00
LBL04	24 GTO 04
»	

CI	Cosine Integral		AB46
	level 1	|	*level 1*
	x	☞	CI(x)

«	37 LBL B
3 RSTO	38 STO 03
DUP	Replaces LASTX in line 46.
SQ	39 X↑2
NEG	40 CHS
0 RSTO	41 STO 00
1	42 1
2 RSTO	43 STO 02
0	44 0
1 RSTO	45 STO 01
3 DROPN	Drop unneeded objects.
LBL01	47 XEQ 01
"C" ENTERα	48 "C"
LBL00	
LBL04	49 GTO 00
»	

EI	Exponential Integral	7624
	level 1 | level 1	
	x ☛ $EI(x)$	

≪	55 LBL C
3 RSTO	56 STO 03
0 RSTO	57 STO 00
1	58 1
2 RSTO	59 STO 02
0	60 0
1 RSTO	61 STO 01
3 DROPN	Drop unneeded objects.
0 RRCL	62 RCL 00
LBL01	63 XEQ 01
DO	64 LBL 03
DUP	Extra copy of current sum.
0 RRCL	65 RCL 00
1 RRCL	66 RCL 01
LBL02	67 XEQ 02
UNTIL	
DUP ROT = =	68 X≠Y
END	69 GTO 03
"E" ENTERα	70 "E"
LBL04	71 LBL 04 (XEQ 04)
≫	

The trickiest aspect of the translation of these programs from the HP 41 to the HP 48 is accounting for the effects of the four-level HP 41 stack, with its automatic duplication of the contents of the T-register when the stack drops, compared with the unlimited HP 48 stack. For example, in the HP 41 program, the loop at LBL 00 assumes that the current value of the sum is in the X-register at the beginning of each iteration. This is automatically provided because the initial copy is pushed up into the T-register during the course of the loop, and even though one copy is used in line 25, another is available to restart the loop thanks to the T-register duplication. This is not obvious from the program listing; you have to step through the execution to determine just what stack objects are expected at LBL 00. On the HP 48, the extra copy of the current sum is provided by inserting a DUP immediately following the DO in the subroutine LBL00.

6.3.1 Alternate Translations

The fact that the translated HP 48 programs listed above have execution times roughly the same as the corresponding HP 41 programs indicates that the strategy of making near-literal translations using the HP 48 simulations of HP 41 registers is not a particularly efficient approach. The HP 48 programs sacrifice the HP 48's strengths in the interest of HP 41 program similarity. In this section, we will show alternate approaches that do not attempt to match the HP 41 programs line-for-line, working instead directly from the program function definitions.

In the particular example of the sine, cosine, and exponential integrals, you can avoid programming altogether by making use of the HP 48's numerical integration capabilities combined with HP Solve. To evaluate the sine integral, for example, you need only to store the following expression as the current equation (EQ):

$$'\int(0,X,SIN(T)/T,T)'$$

When you press ⟦r▷⟧ ⟦SOLVE⟧ to activate the HP Solve variables menu, you will observe menu keys for X, T, and EXPR=. Since T is the variable of integration, its value has no meaning in this context and you can ignore it. Then to evaluate Si(9.8), for example, first select radians mode by pressing ⟦◁⟧ ⟦RAD⟧ if necessary, then

9.8   ⟦ X ⟧   ⟦EXPR=⟧   ⟦r▷⟧⟦→NUM⟧   ☞   1.66756961685

Here we have assumed that standard (STD) display mode is active, which specifies 12-digit accuracy for the integral. Also note that by setting flag −3 to activate numeric evaluation mode, you can omit the →NUM after each press of ⟦EXPR=⟧ .

Using the built-in integration facility of the HP 48 is the easiest way to compute the sine integral, but not necessarily the fastest. The version of SI listed on the next page is an implementation of the series approximation using appropriate HP 48 methods, which runs much faster than any of the versions listed previously.

SI	*Sine Integral*		AF4C
	level 1	|	*level 1*
	x	☞	$SI(x)$

`<<  DUP  SIGN  SWAP  ABS`	Remember the sign of x.
`  0  →  x  n`	
`  <<  x`	Initial value of sum.
`    DO`	
`      'n'  INCR`	Increment n.
`      -1  OVER  ^`	$(-1)^n$
`      SWAP  2  *  1  +`	$2n+1$
`      x  OVER  ^`	x^{2n+1}
`      SWAP  DUP  !  *`	$(2n+1)(2n+1)!$
`      /  *`	Next term.
`      OVER  +`	New sum.
`    UNTIL`	
`      DUP  ROT  = =`	New sum = old sum?
`    END`	
`    *`	Multiply by the sign of x.
`  >>`	
`>>`	

6.4 Command Equivalent Table

Table 6.1 starting on the next page lists each HP 41C/CV/CX command for which some corresponding HP 48 equivalent or replacement can be identified. For each HP 41 command the most closely related HP 48 command is listed, plus, where relevant, a "literal" HP 48 replacement that is either a short program sequence or one of the HP 48 programs listed in the preceding sections of this chapter.

Table 6.1. HP 48 Replacements for HP 41 Commands

HP 41 Command	HP 48 Command	HP 48 Literal	Remarks
⊢ (append)	+	APPEND	See section 6.1.4.
+	+		HP 48 + also accepts complex and array arguments.
−	−		HP 48 − also accepts complex and array arguments.
*	*		HP 48 * also accepts complex and array arguments.
/	/		HP 48 / also accepts complex and array arguments.
1/x	INV		HP 48 INV also accepts complex and array arguments.
10↑X	ALOG		HP 48 ALOG also accepts complex arguments.
ABS	ABS		HP 48 ABS also accepts complex and array arguments.
ACOS	ACOS		HP 48 ACOS also accepts complex and array arguments, and returns complex results for $|x| > 1$.
ADATE	DATE	0 TSTR 5 12 SUB APPEND	
ADV	CR		
ALEN	SIZE	ALPHA SIZE	
ANUM	STR→	ALPHA STR→	Assumes the entire alpha string represents a valid number.
AOFF			
AON			For alpha entry, use INPUT with the α parameter. See section 6.1.4.
ARCL n	'name' RCL	n ARCL	See section 6.1.4.
AROT		AROT	See section 6.1.4.
ASHF	SUB	ASHF	See section 6.1.4.
ASIN	ASIN		HP 48 ASIN also accepts complex arguments, and returns complex results for $|x| > 1$.
ASTO n	'name' STO	n ASTO	See section 6.1.4.
ATAN	ATAN		HP 48 ATAN also accepts complex arguments.
ATIME	TIME	DATE SWAP TSTR 14 22 SUB APPEND	
ATIME24	TIME	RCLF −41 SF SWAP ATIME STOF	ATIME as defined above.

HP 41 Command	HP 48 Command	HP 48 Literal	Remarks
ATOX	NUM	ATOX	
AVIEW	DISP	ALPHA 10 CHR + 1 DISP 1 FREEZE	See section 6.1.4.
BEEP	BEEP		Use combinations of BEEP's to produce audible signals like HP 41 BEEP.
CF n	n CF		
CHS	NEG		
CLA		CLA	See section 6.1.4.
CLD	CLLCD		
CLK12	−41 CF		
CLK24	−41 SF		
CLKEYS	0 DELKEYS		
CLKT			
CLKTD			
CLOCK	−40 SF		
CLRALMS	0 DELALARM		
CLRG	PURGE	CLRG	See section 6.1.
CLRGX	PURGE	CLRGX	See section 6.1.
CLΣ	CLΣ		HP 48 statistics data is kept in the variable ΣDAT.
CLST	CLEAR		See section 2.6.2.
CLX	DROP or 0		See section 2.6.2.
COS	COS		HP 48 COS accepts complex arguments.
D→R	D→R		
DATE	DATE		
DATE+	DATE+		
DDAYS	DDAYS		
DEC	DEC	"#" SWAP + "o" + B→R	
DEG	DEG		
DMY	−42 SF		
DOW	TSTR	0 TSTR 1 3 SUB "SUNMONTUEWEDTHUFRISAT" SWAP POS 1 − 3 /	
DSE	DECR		See section 6.2.2.
END	≫		See section 4.1.1.
ENG n	n ENG		HP 48 range is 0-11.
ENTER↑	DUP		See section 2.8.
E↑X	EXP		HP 48 EXP accepts complex arguments.

HP 41 Command	HP 48 Command	HP 48 Literal	Remarks
E↑X−1	EXPM1		
FACT	! or FACT		HP 48 $x!$ is $\Gamma(x+1)$
FC? n	n FC?		HP 48 flags are numbered $-64 \le n \le 63$
FC?C n	n FC?C		HP 48 flags are numbered $-64 \le n \le 63$
FS? n	n FS?		HP 48 flags are numbered $-64 \le n \le 63$
FS?C n	n FS?C		HP 48 flags are numbered $-64 \le n \le 63$
GETKEY	0 WAIT		
GETKEYX			
GRAD	GRAD		
GTO			See section 6.2.
HMS	→HMS		
HMS+	HMS+		
HMS−	HMS−		
HR	HMS→		
INT	IP		See also FLOOR, CEIL.
ISG	INCR		See section 6.2.2.
LASTX	LASTARG		See section 2.3.
LBL			See section 6.2.
LN	LN		HP 48 LN accepts complex arguments.
LN1+X	LNP1		
LOG	LOG		HP 48 LOG accepts complex arguments.
MDY	−42 SF		
MEAN	MEAN		
MOD	MOD	OVER SWAP MOD	
OCT	OCT	R→B →STR 1 OVER SIZE SUB "d" + OBJ→ B→R	
OFF	OFF		
ON			
P−R		−16 SF SWAP →V2 −16 CF V→ SWAP	
PASN	STOKEYS	ASN41	See section 5.4.
PCLPS	PURGE		
%	%	OVER SWAP %	
%CH	%CH	OVER SWAP %CH	
PI	π	π →NUM	

HP 41 Command	HP 48 Command	HP 48 Literal	Remarks
POSA	POS	ALPHA SWAP POS	
PROMPT	PROMPT		
PSE	WAIT	PSE	See section 5.3.2.1.
PSIZE		SIZE41	See section 6.1.
R↑	ROLL	4 ROLL	
R-D	R→D		
R-P		-16 CF SWAP →V2 -16 SF V→ SWAP	
RAD	RAD		
RCL	RCL	RRCL	See section 6.1.
RCLALM	RCLALARM		HP 48 alarm parameters returned as a list.
RCLFLAG	RCLFLAGS		HP 48 flags represented by a list of two binary integers.
RDN	ROLLD	4 ROLLD	
REGMOVE		REGMOVE	See section 6.1.3.
REGSWAP		REGSWAP	See section 6.1.3.
RND	RND	12 RND	
RTN	>>		
SCI n	n SCI		HP 48 argument range is 0-12.
SDEV	SDEV		
SETDATE	→DATE		
SETIME	→TIME		
SF n	n SF		HP 48 flags are numbered $-64 \le n \le 63$
Σ+	Σ+		
Σ-	Σ-		
ΣREG n	'name' STOΣ		
ΣREG?	RCLΣ		
SIN	SIN		HP 48 SIN accepts complex arguments.
SIGN	SIGN		HP 48 SIGN accepts complex arguments.
SIZE		SIZE41	See section 6.1.
SIZE?		REG41 SIZE 1 GET	See section 6.1.
SQRT	√		HP 48 √ also accepts complex arguments, and returns complex results for negative real arguments.
ST+	STO+	STOPLUS	See section 6.1.2.
ST-	STO-	STOMINUS	See section 6.1.2.
ST*	STO*	STOTIMES	See section 6.1.2.

HP 41 Command	HP 48 Command	HP 48 Literal	Remarks
ST/	STO/	STODIVIDE	See section 6.1.2.
STO *n*	*'name'* STO	*n* RSTO	See section 6.1.
STOFLAG	STOF		
STOP	HALT		See section 5.2.
T+X	→TIME	TIME HMS+ →TIME	
TAN	TAN		HP 48 TAN(90°) is an infinite result exception. TAN also works with complex arguments.
TIME	TIME		
TONE	BEEP		
VIEW *n*	DISP	*n* RRCL 10 CHR + 1 DISP 1 FREEZE	
X↑2	SQ		HP 48 SQ also accepts complex and array arguments.
X=0?	= =	DUP 0 = =	
X≠0	≠	DUP 0 ≠	
X<0?	<	DUP 0 <	
X< =0?	≤	DUP 0 ≤	
X>0?	>	DUP 0 ≥	
X=Y?	= =	DUP2 = =	
X≠Y	≠	DUP2 ≠	
X<Y?	>	DUP2 >	Sense of HP 41 and HP 48 tests is reversed.
X< =Y?	≥	DUP2 ≥	Sense of HP 41 and HP 48 tests is reversed.
X>Y?	<	DUP2 <	Sense of HP 41 and HP 48 tests is reversed.
X=NN? *n*	= =	*n* RRCL OVER = =	See section 4.3.1.
X≠NN *n*	≠	*n* RRCL OVER ≠	See section 4.3.1.
X<NN? *n*	<	*n* RRCL OVER SWAP <	See section 4.3.1.
X< =NN? *n*	≤	*n* RRCL OVER SWAP ≤	See section 4.3.1.
X> =NN? *n*	≥	*n* RRCL OVER SWAP ≥	See section 4.3.1.
X>NN? *n*	>	*n* RRCL OVER SWAP >	See section 4.3.1.
X<> *n*	RCL	*n* DUP RRCL 3 ROLLD RSTO DROP	
X<>F	RCLF	RCLF LIST→ DROP 64 STWS DUP SRB SLB 4 ROLL + SWAP 3 ROLLD 2 →LIST STOF	
X<>Y	SWAP		
XEQ *name*	*name*		

HP 41 Command	HP 48 Command	HP 48 Literal	Remarks
XTOA	CHR	XTOA	See section 6.1.4.
XYZALM	STOALARM		HP 48 alarm parameters entered in a list.
Y↑X	^		HP 48 ^ accepts complex arguments.

Program Index

Subject Index

HP 41/HP 48 Transitions

The HP 48SX Scientific Expandable Calculator is the heir-apparent of the HP 41, which dominated the field of programmable calculators for ten years. The new calculator combines the flexibility and customizability of the HP 41 with the revolutionary computational capabilities of the HP 28S, making an extraordinarily powerful personal tool.

The HP 48SX is fundamentally an RPN calculator, which provides a familiar basis from which you as an HP 41 user can learn the features of the HP 48SX. However, there are many differences in style and methodology between the two calculators, which together with the enormous range of HP 48SX facilities may present a barrier as you attempt to master the HP 48SX. In *HP 41/HP 48 Transitions*, Dr. William Wickes describes the logical evolution from the HP 41 to the HP 48SX, showing how many of the HP 48SX's actions are natural extensions and enhancements of familiar HP 41 operations. The book shows how your HP 41 procedures and methods can be adapted to the HP 48SX, including stack manipulations, storing and retrieving data, programming, and program conversions.

Dr. Wickes is the author of *HP-28 Insights*, the definitive book on HP 28 principles and methodology. *HP 41/HP 48 Transitions* is based on the HP 41 material in that book, and is the precursor to *HP 48 Insights*, a more complete treatise on the priniciples and applications of the HP 48. Dr. Wickes also wrote *Synthetic Programming on the HP-41C*, which has been a standard in HP 41 advanced programming since 1981.

Chapter Headings: