HEWLETT-PACKARD

HP-28C
Programming Examples
Welcome to the HP-28C

This booklet, *HP-28C Programming Examples*, contains 19 programs for your HP-28C. These programs are useful and, more importantly, they demonstrate a variety of programming techniques. You'll find a list of the techniques on page 6.

Before trying the examples in this booklet, please read “How To Use This Booklet” on page 7. It contains important information on the conventions observed in this booklet.

This booklet assumes you've read the *HP-28C Getting Started Manual*. At a minimum, you should know:

- How to enter numbers and expressions.
- How to enter programs and edit existing programs.
- How to use menus.

You can find detailed information about programming in the *HP-28C Reference Manual*, especially in the following sections.

- Programs
- PROGRAM BRANCH
- PROGRAM CONTROL
- PROGRAM TEST
Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

© 1987 by Hewlett-Packard Co.

This document contains proprietary information that is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

Portable Computer Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.

Printing History

Edition 1       June 1987       Mfg No. 00028-90099
Contents

7 How To Use This Booklet
9 Programming Examples
9 RENAME (Renaming a Variable)
11 Box Functions
11 BOXS (Surface of a Box)
13 BOXS Without Local Variables
14 BOXR (Ratio of Surface to Volume of a Box)
16 Fibonacci Numbers
16 FIB1 (Fibonacci Numbers, Recursive Version)
17 FIB2 (Fibonacci Numbers, Loop Version)
19 Comparison of FIB1 and FIB2
20 Single-Step Execution
23 Expand and Collect Completely
23 MULTI (Multiple Execution)
24 EXCO (Expand and Collect Completely)
26 Displaying a Binary Integer
26 PAD (Pad With Leading Spaces)
27 PRESERVE (Save and Restore Previous Status)
28 BDISP (Binary Display)
32 Summary Statistics
33 SUMS (Summary Statistics Matrix)
34 £GET (Get an Element of £COV)
35 £X2 (Sum of Squares of x)
35 £Y2 (Sum of Squares of y)
36 £XY (Sum of Products of x and y)
39 Median of Statistics Data
39 SORT (Sort a List)
41 LMED (Median of a List)
42 MEDIAN (Median of Statistics Data)
45 Index
List of Techniques

9  Basic stack manipulations
11  Local-variable structure
12  User function
14  Nested user functions
16  IFTE (If-Then-Else function)
16  Recursion
17  IF ... THEN ... ELSE ... END (conditional)
17  START ... NEXT (definite loop)
23  DO ... UNTIL ... END (indefinite loop)
23  Programs as arguments
23  Evaluation of local variables
25  Structured programming
26  WHILE ... REPEAT ... END (indefinite loop)
26  String operations
27  RCLF and STOF
27  Local-variable structure
28  IFERR ... THEN ... END (error trap)
29  Enabling LAST
29  FOR ... NEXT (definite loop with index)
29  Subprograms
32  Matrix operations
32  Programs usable in algebraic objects
32  ΣPAR convention
39  Bubble sort
39  Nested definite loops
39  Nested local-variable structures
39  FOR ... STEP and FOR ... NEXT (definite loops)
41  FLOOR and CEIL
42  Arrays, lists, and stack elements
42  FOR ... NEXT (definite loop with index)
How To Use This Booklet

For each program you'll find the following information.

- A description of its purpose.
- A diagram showing its effect on the stack.
- A list of techniques that it demonstrates.
- A list of other programs that it requires (if applicable).
- A program listing with comments.
- An example that shows how to use it.

Each type of information is described in more detail below.

**Stack Diagram.** A stack diagram is a two-column table showing “Arguments” and “Results”. “Arguments” shows what must be on the stack before the program is executed; “Results” shows what the program leaves on the stack.

Note that the stack diagram doesn’t show everything; a program that changes user memory or displays objects might have no effect on the stack.

**Techniques.** This is the most interesting part. When you understand how a technique is used in this booklet, you can use it in your own programs.

**Required Programs.** Some programs call others as subroutines. You can enter the required programs and the calling program in any order, but you must enter all of them before executing the calling program.

The HP-28C can’t hold all the programs in this booklet at one time. Before purging one program to make room for another, make sure the program you’re purging isn’t required by another program that interests you.
**Program and Comments.** This booklet formats the program listing to show a program’s structure and process. You don’t need to follow the format of the listing when you enter a program. However, be sure to key in spaces where they appear in the listing or between objects appearing on separate lines.

You can key in a program character by character, or you can use the menus to key it in command by command. It makes no difference as long as the result matches the listing.

When you key in the program you can omit all closing parentheses and delimiters that appear at the very end of the program; when you press **[ENTER]** the closing parentheses and delimiters are added for you.

**Example.** The examples observe the following conventions.

The illustrations assume STD display format. To select STD display format, press STD **[ENTER]** or use the MODE menu.

A box represents a key on the calculator keyboard.

```
USER
ON
ENTER
EVAL
STO
```

In some cases a box represents a shifted key on the HP-28C. The shift key is *not* shown explicitly.

```
CLEAR
CTRL
CONT
BINARY
VISIT
STAT
```

The “inverse” highlight represents a menu label.

```
Σ-CL , Σ+, NE , in the STAT menu.
SST and KILL in the CTRL menu.
DEC in the BINARY menu.
```

Variable names in the USER menu also appear as menu labels.

Menus typically include more than one menu level. Press **NEXT** and **PREV** to roll through the menu levels. In the examples, **NEXT** and **PREV** are *not* shown explicitly.
Programming Examples

The most important technique demonstrated in this booklet is structured programming: small programs used to build other programs. The following programs are used in other programs.

- BOXS is used in BOXR.
- MULTI is used in EXCO.
- PAD and PRESERVE are used in BDISP.
- \( \Sigma \)GET is used in \( \Sigma X_2 \), \( \Sigma Y_2 \), and \( \Sigma XY \).
- SORT and LMED are used in MEDIAN.

RENAME (Renaming a Variable)

Recall the contents of a variable, purge the variable, and store the contents in a new variable.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>2: 'name' (old)</td>
<td>2:</td>
</tr>
<tr>
<td>1: 'name' (new)</td>
<td>1:</td>
</tr>
</tbody>
</table>

Techniques:

- Basic stack manipulations.
**Program**

```
<
  OVER
  RCL
  ROT
  PURGE
  SWAP

  STO
>
```

**Comments**

Begin the program.
Copy the old name to level 1.
Recall the contents of the variable.
Move the old name to level 1.
Purge the old variable.
Put the contents and new name in the correct order.
Create the new variable.
End the program.

Put the program on the stack.
Store the program as RENAME.

**Example.** Create a variable A with contents 10, then rename A to B, then evaluate B to check that its value is 10.

Clear the stack and select the USER menu.

```
CLEAR
USER
```

Create a variable A with contents 10.

```
10 [ENTER]
'A [STO]
```

Rename variable A to B.

```
'A [ENTER]
'B [ENTER]
[RENAM]
```

Check the value of B.

```
'B
```

**RENAME (Renaming a Variable)**
Box Functions

This section contains two programs:

- BOXS calculates the total surface area of a box.
- BOXR uses BOXS to calculate the ratio of surface to volume for a box.

**BOXS (Surface of a Box)**

Given the height, width, and length of a box, calculate the total area of its six sides.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>3: height</td>
<td>3:</td>
</tr>
<tr>
<td>2: width</td>
<td>2:</td>
</tr>
<tr>
<td>1: length</td>
<td>1: area</td>
</tr>
</tbody>
</table>

**Techniques:**

- Local-variable structure. Local variables allow you to assign names to arguments without conflicting with global variables. Like global variables, local variables are convenient because you can use arguments any number of times without tracking their positions on the stack; unlike global variables, local variables disappear when the program structure that creates them is done.

A local-variable structure has three parts.

1. A command named "→". When you key in this command, remember to put spaces before and after it. (Like any command, → is spelled using normal characters and is recognized only when it's set off by spaces. Don't confuse this one-character command with delimiters like # or «.)

2. One or more names.

3. A procedure (expression, equation, or program) that includes the names. This procedure is called the defining procedure.
When a local-variable structure is evaluated, a local variable is created for each name. The values for the local variables are taken from the stack. The defining procedure is then evaluated, substituting the values of the local variables.

To appreciate the power of local variables, compare the version of BOXS given below with the version that appears on page 13.

- **User function.** This type of program works in either RPN or algebraic syntax. A user function is a program with two characteristics: (1) It consists solely of a local-variable structure. (2) The defining procedure is an expression.

**Program**

```
« Begin the program.
→ h w l

'2*(h*w+h*l+w*l)'

»
```

**Comments**

Begin the program.
Create local variables for height, width, and length. By convention, lower-case letters are used. The values are taken from the stack (in RPN) or from the arguments to the user function (in algebraic syntax).
The defining expression for the surface area. Evaluating the user function causes evaluation of this expression, returning the area to the stack.
End the program.

Put the program on the stack.
Store the program as BOXS.

**Example.** One of the advantages of user functions is that they work in either RPN or algebraic syntax. Calculate the surface of a box 12 inches high, 16 inches wide, and 24 inches long; make the calculation first in RPN and then in algebraic syntax.

For the RPN version, first enter the height and width.

```
USER
12 ENTER
16 ENTER
```

12 BOXS (Surface of a Box)
Then key in the length and execute BOXS.

\[
\begin{array}{c}
24 \quad \text{BOXS} \\
\end{array}
\]

The surface area is 1728 square inches.

Now try the algebraic version.

\[
\begin{array}{c}
'\text{BOXS} \ (12, 16, 24) \quad \text{EVAL} \\
\end{array}
\]

Again, the surface area is 1728.

**BOXS Without Local Variables**

The following program uses only stack operations to calculate the surface of a box. Compare this program with BOXS.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>3: height</td>
<td>3:</td>
</tr>
<tr>
<td>2: width</td>
<td>2:</td>
</tr>
<tr>
<td>1: length</td>
<td>1: area</td>
</tr>
</tbody>
</table>

**Program**

```
«
  DUP2 *  \Calculate \text{wl}.
  ROT     \Move \text{w} \text{to level 1}.
  4 PICK  \Copy \text{h} \text{to level 1}.
  *      \Calculate \text{wh}.
  +      \Calculate \text{wl} + \text{wh}.
  ROT ROT\Move \text{h} \text{and} \text{l} \text{to levels 2 and 1}.
  *      \Calculate \text{hl}.
  +      \Calculate \text{wl} + \text{wh} + \text{hl}.
  2 *    \Calculate 2(\text{wl} + \text{wh} + \text{hl}).
  »       \End the program.
```

Because this version of BOXS isn't a user function, it can't be used in algebraic syntax.
BOXR (Ratio of Surface to Volume of a Box)

Given the height, width, and length of a box, calculate the ratio of its surface to its volume.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>3: height</td>
<td>3:</td>
</tr>
<tr>
<td>2: width</td>
<td>2:</td>
</tr>
<tr>
<td>1: length</td>
<td>1: area/volume</td>
</tr>
</tbody>
</table>

Techniques:

- Nested user functions. BOXR is a user function whose defining expression uses BOXS in its calculation. In turn, BOXR could be used to define other user functions.

  Recall that BOXS was defined using \( h, w, \) and \( l \) as local variables, and note below that BOXS takes \( x, y, \) and \( z \) as arguments in the definition for BOXR. It makes no difference if the local variables in the two definitions match, or if they don't match, because each set of local variables is independent of the other. However, it's essential that local variables be consistent within a single definition.

Program

```
< « x y z
  'BOXS(x,y,z)
  /(x*y*z)' 
»
```

Comments

- Begin the program.
- Create local variables for height, width, and length. This program uses \( x, y, \) and \( z \), rather than \( h, w, \) and \( l \).
- Begin the defining expression with the user function BOXS.
- Divide by the volume of the box.
- End the program.

```
ENTER
'BOXR STO
```

- Put the program on the stack.
- Store the program as BOXR.
Example. Calculate the ratio of surface to volume for a box 9 inches high, 18 inches wide, and 21 inches long; make the calculation first in RPN and then in algebraic syntax.

For the RPN version, first enter the height and width.

\[
\begin{align*}
\text{USER} & \\
9 & \text{ ENTER} \\
18 & \text{ ENTER}
\end{align*}
\]

Then key in the length and execute BOXR.

\[
\begin{align*}
21 & \text{ BOXR} \\
\end{align*}
\]

The ratio is \(0.428571428571\).

Now try the algebraic version.

\[
\begin{align*}
\text{BOXR}(9, 18, 21 & \text{ EVAL} \\
3: & \\
2: & \text{ .428571428571} \\
1: & \text{ .428571428571}
\end{align*}
\]

Again, the ratio is \(0.428571428571\).
Fibonacci Numbers

Given an integer $n$, calculate the $n$th Fibonacci number $F_n$, where

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

This section includes two programs, each demonstrating an approach to this problem.

- FIB1 is a user function that is defined recursively – its defining expression contains its own name. FIB1 is short, easy to understand, and usable in algebraic objects.
- FIB2 is a program with a definite loop. It’s not usable in algebraic objects, it’s longer and more complicated than FIB1, but it’s faster.

FIB1 (Fibonacci Numbers, Recursive Version)

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1: \ n$</td>
<td>$1: \ F_n$</td>
</tr>
</tbody>
</table>

Techniques:

- User function. See the description on page 12.
- IFTE (If-Then-Else function). The defining expression for FIB1 contains the conditional function IFTE, which can be used in either RPN or algebraic syntax. (FIB2 uses the program structure IF . . . THEN . . . ELSE . . . END.)
- Recursion. The defining expression for FIB1 is written in terms of FIB1, just as $F_n$ is defined in terms of $F_{n-1}$ and $F_{n-2}$. 
Program Comments

«
  \rightarrow n
  IFTE(n\leq1,
    n,
    \text{FIB1}(n-1)+\text{FIB1}(n-2))
»

BEGIN the program.
Define a local variable.
Begin the defining expression.
If \( n \leq 1 \),
Then \( F_n = n \);
Else \( F_n = F_{n-1} + F_{n-2} \).
End the defining expression.
End the program.

ENTER
'FIB1 STO

Put the program on the stack.
Store the program as FIB1.

Example. Calculate \( F_6 \) using RPN syntax and \( F_{10} \) using algebraic syntax.

First calculate \( F_6 \) using RPN.

USER
6 FIB1

Next calculate \( F_{10} \) using algebraic syntax.

' FIB1 ( 10 EVAL

FIB2 (Fibonacci Numbers, Loop Version)

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( n )</td>
<td>1: ( F_n )</td>
</tr>
</tbody>
</table>

Techniques:

- Local-variable structure. See the description on page 11.
- IF \ldots THEN \ldots ELSE \ldots END. FIB2 uses the program-structure form of the conditional. (FIB1 uses IFTE.)
- START \ldots NEXT (definite loop). To calculate \( F_n \), FIB2 starts with \( F_0 \) and \( F_1 \) and repeats a loop to calculate successive \( F_i \)'s.
Program

```
«
  → n
  «
      IF n 1 ≤
      THEN n
      ELSE
          0 1
          2 n
          START
          DUP
            ROT
            +
          NEXT
          SWAP DROP
      END
  »
»
```

Comments

Begin the program.
Create a local variable.
Begin the defining program.
If \( n \leq 1 \),
Then \( F_n = n \);
Begin ELSE clause.
Put \( F_0 \) and \( F_1 \) on the stack.
From 2 to \( n \),
Do the following loop:
Make a copy of the latest \( F \) (initially \( F_1 \)).
Move the previous \( F \) (initially \( F_0 \)) to level 1.
Calculate the next \( F \) (initially \( F_2 \)).
Repeat the loop.
Drop \( F_{n-1} \).
End ELSE clause.
End the defining program.
End the program.

Put the program on the stack.
Store the program as FIB2.

Example. Calculate \( F_6 \) and \( F_{10} \). Note that FIB2 is faster than FIB1.

Calculate \( F_6 \).

```
USER
6 ← FIB2
```

Calculate \( F_{10} \).

```
10 ← FIB2
```

18 FIB2 (Fibonacci Numbers, Loop Version)
Comparison of FIB1 and FIB2

FIB1 calculates intermediate values $F_i$ more than once, while FIB2 calculates each intermediate $F_i$ only once. Consequently, FIB2 is faster.

The difference in speed increases with the size of $n$ because the time required for FIB1 grows exponentially with $n$, while the time required for FIB2 grows only linearly with $n$.

The diagram below shows the beginning steps of FIB1 calculating $F_{10}$. Note the number of intermediate calculations: 1 in the first row, 2 in the second row, 4 in the third row, and 8 in the fourth row.
Single-Step Execution

It's easier to understand how a program works if you execute it step by step, seeing the effect on the stack of each step. Doing this can help you "debug" your own programs or help you understand programs written by others.

This section shows you how to execute FIB2 step by step, but you can apply these rules to any program. The general rules are:

1. Use VISIT to insert the command HALT in the program. Place HALT where you want to begin single-step execution. (You'll see how the position of HALT within FIB2 affects execution.)

2. Execute the program. When the HALT command is executed, the program stops (indicated by the "stopsign" annunciator).

3. Select the PROGRAM CONTROL menu.

4. Press SST once to see the next program step displayed and then executed.

5. You can now:
   - Keep pressing SST to display and execute sequential steps.
   - Press CONT to continue normal execution.
   - Press KILL to abandon further program execution.

6. When you want the program to run normally again, use VISIT to remove HALT from the program.

For the first example, insert HALT as the first command in FIB2.

Clear the stack and select the USER menu.

```
| CLEAR |
| USER  |
```

Use VISIT to return FIB2 to the command line.

```
  FIB2 VISIT
```

* FIB2: IF n 1 \leq
  THEN n ELSE 0 1 2 n
  START DUP ROT + NEXT
  SWAP DROP END » »
Use the cursor menu keys to insert HALT as shown.

\[
\begin{align*}
& \text{HALT } + n \times \text{ IF } n 1 \leq \\
& \text{THEN n ELSE 0 1 2 n} \\
& \text{START DUP ROT + NEXT} \\
& \text{SWAP DROP END } \rightarrow \rightarrow \\
\end{align*}
\]

Store the edited version of FIB2.

Store the edited version of FIB2.

\[\text{ENTER}\]

Calculate \( F_1 \). At first, nothing happens except that the "stopsign" annunciator appears.

\[1 \leftarrow \text{FIB2}\]

Select the PROGRAM CONTROL menu and execute SST (single-step). (Watch the top line of the display to see the first step displayed before it's executed.)

\[\text{CTRL SST}\]

Note that \( \rightarrow n \) constitutes one step; "step" is a logical unit rather than simply the next object in the program.

Look at the general rules at the beginning of this section. You've performed the first four steps, and now you can choose one of the three alternatives for step 5. For this example, press \( \rightarrow \text{SST} \) repeatedly until the "stopsign" annunciator disappears, indicating that FIB2 is completed. (These single-steps not shown here.)

The calculation for \( F_1 \) executes only the THEN clause in FIB2. For the second example, execute \( 3 \leftarrow \text{FIB2} \) and single-step through the calculation for \( F_3 \). This executes the ELSE clause, including the START . . . NEXT loop. You'll see that, for \( n = 3 \), the START . . . NEXT loop is executed twice.

For the third example, suppose you want to single-step the START . . . NEXT loop as a whole—seeing the stack before each iteration of the loop, but not single-stepping all the steps in FIB2 or in the loop itself. To do so, move the HALT command inside the loop. Then FIB2 won't halt until it
reaches the loop, and you can use CONT (continue) to execute the loop one iteration at a time.

Use VISIT to return FIB2 to the command line.

Use the cursor menu keys to delete HALT. Then insert HALT as shown (on the third line, after START).

Store the edited version of FIB2.

Start the calculation for $F_3$. FIB2 will halt before performing the loop.

Continue execution of the loop. FIB2 will halt before performing the loop a second time.

Continue execution of the loop. Because this is the last iteration of the loop, FIB2 will execute to completion.

When you’re done experimenting with FIB2, don’t forget to use VISIT to remove the HALT command.
Expand and Collect Completely

This section contains two programs:

- MULTI repeats a program until the program has no effect.
- EXCO uses MULTI to expand and collect completely.

MULTI (Multiple Execution)

Given an object and a program that acts on the object, apply the program to the object repeatedly until the object is unchanged.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>2: object</td>
<td>2:</td>
</tr>
<tr>
<td>1: « program »</td>
<td>1: resulting object</td>
</tr>
</tbody>
</table>

Techniques:

- DO ... UNTIL ... END (indefinite loop). The DO clause contains the steps to be repeated; the UNTIL clause contains the test that determines whether to repeat both clauses again (if false) or to exit (if true).

- Programs as arguments. Although programs are commonly named and then executed by calling their names, programs can also be put on the stack and used as arguments to other programs.

- Evaluation of local variables. The program argument to be executed repeatedly is stored in a local variable. It’s handy to store an object in a local variable when you don’t know beforehand how many copies you’ll need.

MULTI demonstrates one of the differences between global and local variables: if a global variable contains a name or program, the contents of the variable are evaluated when the name is evaluated; but the contents of a local variable are always simply recalled. Consequently, MULTI uses the local name to put the program argument on the stack and then executes an explicit EVAL command to evaluate the program.
Program

«
→ p
«
DO
DUP
p EVAL
UNTIL
DUP
ROT
SAME
END
»

Comments

Begin the program.
Create a local variable \( p \) that contains the program argument.

Begin the defining program.
Begin the DO clause.

Make a copy of the object.

Apply the program to the object, returning a new version. (The EVAL command is necessary to execute the program because local variables always return their contents to the stack unevaluated.)

Begin the UNTIL clause.

Make a copy of the new version of the object.

Move the old version to level 1.

Test whether the old version and the new version are the same.

End the UNTIL clause.

End the defining program.

End the program.

Example. MULTI is demonstrated in the next program.

EXCO (Expand and Collect Completely)

Given an algebraic object, execute EXPAN repeatedly until the algebraic doesn’t change, then execute COLCT repeatedly until the algebraic doesn’t change. In some cases the result will be a number.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: 'algebraic'</td>
<td>1: 'algebraic'</td>
</tr>
<tr>
<td>1: 'algebraic'</td>
<td>1: ( z )</td>
</tr>
</tbody>
</table>
Techniques:

- Structured programming. EXCO calls the program MULTI twice. Even if you don’t use MULTI anywhere else, the efficiency of repeating all the commands in MULTI by simply including its name a second time justifies writing MULTI as a separate program.

Required Programs:

- MULTI (page 23) repeatedly executes the programs that EXCO provides as arguments.

Program

```
«
  « EXPAN »
  MULTI
  « COLCT »
  MULTI
»
```

Comments

Begin the program.
Put EXPAN on the stack.
Execute EXPAN until the algebraic object doesn’t change.
Put COLCT on the stack.
Execute COLCT until the algebraic object doesn’t change.
End the program.

Put the program on the stack.
Store the program as EXCO.

Example. Expand and collect completely the expression

\[3x(4y + z) + (8x - 5z)^2\]

Enter the expression.

```
\[3\times X \times (4\times Y + Z) + (8\times X - 5\times Z)^2\]
```

Expand and collect completely.

```
\[12\times X \times Y - 77\times X \times Z + 64\times X^2 + 25\times Z^2\]
```

Expressions with many products of sums or with powers can take many iterations of EXPAN to expand completely, resulting in a long execution time for EXCO.
Displaying a Binary Integer

This section contains three programs:

- PAD is a utility program that converts an object to a string for right-justified display.
- PRESERVE is a utility program for use in programs that change the calculator’s status (angle mode, binary base, and so on).
- BDISP displays a binary integer in HEX, DEC, OCT, and BIN bases. It calls PAD to show the displayed numbers right-justified, and it calls PRESERVE to preserve the binary base.

PAD (Pad With Leading Spaces)

Convert an object to a string and, if the string contains fewer than 23 characters, add spaces to the beginning.

When a short string is displayed by using DISP, it appears left-justified – its first character appears at the left end of the display. The position of the last character is determined by the length of the string.

By adding spaces to the beginning of a short string, PAD moves the position of the last character to the right. When the string is 23 characters long, it appears right-justified – its last character appears at the right end of the display.

PAD has no effect on strings that are longer than 22 characters.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: object</td>
<td>1: &quot; object&quot;</td>
</tr>
</tbody>
</table>

Techniques:

- WHILE . . . REPEAT . . . END (indefinite loop). The WHILE clause contains a test that determines whether to execute the REPEAT clause and test again (if true) or to skip the REPEAT clause and exit (if false).
- String operations. PAD demonstrates how to convert an object to string form, count the number of characters, and concatenate two strings.
Program

```
«
  →STR

  WHILE
  DUP SIZE 23 <
  REPEAT
    " " SWAP +
  END
»
```

Comments

Begin the program.
Make sure the object is in string form. (Strings are unaffected by this command.)
Begin WHILE clause.
Does the string contain fewer than 23 characters?
Begin REPEAT clause.
Add a leading space.
End REPEAT clause.
End the program.

Put the program on the stack.
Store the program as PAD.

Example.  PAD is demonstrated in the program BDISP.

PRESERVE (Save and Restore Previous Status)

Given a program on the stack, store the current status, execute the program, and then restore the previous status.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: « program »</td>
<td>1: (result of program)</td>
</tr>
</tbody>
</table>

Techniques:

- RCLF and STOF. PRESERVE uses RCLF (*recall flags*) to record the current status of the calculator in a binary integer and STOF (*store flags*) to restore the status from that binary integer.

- Local-variable structure. PRESERVE creates a local variable just to remove the object from the stack briefly; its defining program does little except evaluate the program argument on the stack.
Program

\[ \begin{align*}
\langle & \text{Begin the program.} \\
& \text{RCLF} \\
& \rightarrow f \\
\langle & \text{Begin the defining program.} \\
& \text{EVAL} \\
& f \text{ STOF} \\
\rangle & \text{End the defining program.} \\
\rangle & \text{End the program.} \\
\end{align*} \]

\[ \begin{align*}
\text{ENTER} \\
'\text{PRESERVE} & \text{ STO}
\end{align*} \]

Comments

Begin the program.
Recall a 64-bit binary integer representing the status of all 64 user flags.
Store the binary integer in a local variable \( f \).
Begin the defining program.
Execute the program argument.
Restore the status of all 64 user flags.
End the defining program.
End the program.

Example. PRESERVE is demonstrated in the program BDISP.

BDISP (Binary Display)

Display a number in HEX, DEC, OCT, and BIN bases.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: # ( n )</td>
<td>1: # ( n )</td>
</tr>
<tr>
<td>1: ( n )</td>
<td>1: ( n )</td>
</tr>
</tbody>
</table>

Techniques:

- IFERR \ldots THEN \ldots END (error trap). To accommodate real numbers, BDISP includes the command \( \text{R}\rightarrow\text{B} \) (real-to-binary). However, this command causes an error if the argument is already a binary integer.

To maintain execution if an error occurs, the \( \text{R}\rightarrow\text{B} \) command is placed inside an IFERR clause. Because no action is required when an error occurs, the THEN clause contains no commands.
Enabling LAST. In case an error occurs, LAST must be enabled to return the argument to the stack. BDISP sets flag 31 to programmatically enable the LAST recovery feature.

FOR . . . NEXT loop (definite loop with index). BDISP executes a loop from 1 to 4, each time displaying n in a different base on a different line.

The loop index (named j in this program) is a local variable. It's created by the FOR . . . NEXT program structure (rather than by a command) and it's automatically incremented by NEXT.

Subprograms. BDISP demonstrates three uses for subprograms.

1. BDISP contains a main subprogram and a call to PRESERVE. The main subprogram goes on the stack and is evaluated by PRESERVE.

2. When BDISP creates a local variable for n, the defining program is a subprogram.

3. There are four subprograms that “customize” the action of the loop. Each subprogram contains a command to change the binary base and a marker (h, d, o, or b) to indicate the base. Each iteration of the loop executes one of these subprograms.

**Required Programs:**

- PAD (page 26) expands a string to 23 characters so that DISP shows it right-justified.

- PRESERVE (page 27) stores the current status, executes the main subprogram, and restores the status.
Program

```
«
«
DUP
31 SF
IFERR
  R→B
THEN
END

→ n
«
  CLLCD
  « BIN "b" »
  « OCT "o" »
  « DEC "d" »
  « HEX "h" »
  1 4
  FOR j
    EVAL
    n →STR
    SWAP +
    PAD
    j DISP
  NEXT
»
»
PRESERVE
```

Comments

Begin the program.
Begin the main subprogram.
Make a copy of n.
Set flag 31 to enable LAST.
Begin error trap.
Convert n to a binary integer.
If an error occurred,
Do nothing (no commands in THEN clause).
Create a local variable n.
Begin the defining program.
Clear the display.
Subprogram for BIN.
Subprogram for OCT.
Subprogram for DEC.
Subprogram for HEX.
First and last index values.
Start loop with index j.
Evaluate one of the base subprograms (initially the one for HEX).
Make a string showing n in the current base.
Add the base marker.
Pad the string to 23 characters.
Display the string in the jth line.
Increment j and repeat the loop.
End the defining program.
End the main subprogram.
Store the current status, execute the main subprogram, and restore the status.
End the program.

Put the program on the stack.
Store the program as BDISP.
Example. Switch to DEC base, display # 100 in all bases, and check that BDISP restored the base to DEC.

Clear the stack and select the BINARY menu.

```
CLEAR
BINARY
```

Make sure the current base is DEC and key in # 100.

```
DEC
#100 ENTER
```

Execute BDISP. (Don’t switch menus, since you’ll want to see the BINARY menu in the next step.)

```
BDISP ENTER
```

Return to the normal stack display and check the current base.

```
ON
```

Although the main subprogram left the calculator in BIN base, PRESERVE restored DEC base.

To check that BDISP also works for real numbers, try 144.

```
USER
144 BDISP
```

BDISP (Binary Display)  31
Summary Statistics

For paired-sample statistics it's often useful to calculate the sum of the squares ($\Sigma x^2$ and $\Sigma y^2$) and the sum of the products ($\Sigma xy$) of the two variables. This section contains five programs:

- **SUMS** creates a variable $\Sigma COV$ that contains the covariance matrix for the current statistics matrix $\Sigma DAT$.
- **$\Sigma GET$** extracts a number from the specified position in $\Sigma COV$.
- **$\Sigma X2$** uses $\Sigma GET$ to extracts $\Sigma x^2$ from $\Sigma COV$.
- **$\Sigma Y2$** uses $\Sigma GET$ to extracts $\Sigma y^2$ from $\Sigma COV$.
- **$\Sigma XY$** uses $\Sigma GET$ to extracts $\Sigma xy$ from $\Sigma COV$.

If $\Sigma DAT$ contains $n$ columns, $\Sigma COV$ is an $n \times n$ matrix. The programs $\Sigma X2$, $\Sigma Y2$, and $\Sigma XY$ refer to $\Sigma PAR$ (*statistics parameters*) to determine which columns contain the $x$ data (called $C_1$) and the $y$ data (called $C_2$).

**Techniques:**

- **Matrix operations.** These programs demonstrate how to transpose a matrix, how to multiply two matrices, and how to extract one element from a matrix.

- **Programs usable in algebraic objects.** Because $\Sigma X2$, $\Sigma Y2$, and $\Sigma XY$ conform to algebraic syntax (no arguments from the stack, one result put on the stack), you can use their names like ordinary variables in an expression or equation.

- **$\Sigma PAR$ convention.** Several paired-sample statistics commands use a variable named $\Sigma PAR$ to specify a pair of columns in $\Sigma DAT$. $\Sigma PAR$ contains a list with four numbers, the first two specifying columns. (The other two numbers are the slope and intercept from linear regression.)

  SUMS ensures that $\Sigma PAR$ exists by executing 0 PREDV DROP; the command PREDV (*predicted value*) creates $\Sigma PAR$ with default values if $\Sigma PAR$ doesn't already exist, and DROP removes the predicted value computed for 0.

  $\Sigma X2$, $\Sigma Y2$, and $\Sigma XY$ use the values stored in $\Sigma PAR$ to determine which element to extract from $\Sigma COV$. 

32 Summary Statistics
SUMS (Summary Statistics Matrix)

Create a variable $\Sigma COV$ that contains the covariance matrix of the statistics matrix $\Sigma DAT$.

As an example, if $\Sigma DAT$ is the $n \times 2$ matrix

$$
\begin{bmatrix}
  x_1 & y_1 \\
  x_2 & y_2 \\
  \vdots & \vdots \\
  x_n & y_n
\end{bmatrix}
$$

then $\Sigma COV$ will contain the covariance matrix

$$
\begin{bmatrix}
  \Sigma x^2 & \Sigma xy \\
  \Sigma xy & \Sigma y^2
\end{bmatrix}
$$

### Arguments and Results

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>1:</td>
</tr>
</tbody>
</table>

### Program

```
«
RCL\Sigma
DUP
TRN
SWAP *
'\Sigma COV' STO
0 PREDV DROP
»
```

### Comments

Begin the program.
Recall the contents of the $n \times m$ statistics matrix $\Sigma DAT$.
Make a copy.
Transpose the matrix. The result is an $m \times n$ matrix.
Multiply the matrices to produce the $m \times m$ covariance matrix. (Without swapping the matrices, the product would be an $n \times n$ matrix.)
Store the covariance matrix in a variable $\Sigma COV$.
Make sure $\Sigma PAR$ exists.
End the program.

Put the program on the stack.
Store the program as SUMS.
**ΣGET (Get an Element of ΣCOV)**

Given \( p \) and \( q \), each indicating either the first or second position in ΣPAR, extract the \( rs \) element from ΣCOV, where \( r \) and \( s \) are the corresponding first or second elements in ΣPAR.

ΣGET is called by ΣX2, ΣY2, and ΣXY with the following arguments.

- For ΣX2, \( p = 1 \) and \( q = 1 \).
- For ΣY2, \( p = 2 \) and \( q = 2 \).
- For ΣXY, \( p = 1 \) and \( q = 2 \).

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>2: 1 or 2</td>
<td>2:</td>
</tr>
<tr>
<td>1: 1 or 2</td>
<td>1: ( rs ) element of ΣCOV</td>
</tr>
</tbody>
</table>

**Program**

```
«
ΣCOV
ΣPAR
DUP
5 ROLL
GET
SWAP
4 ROLL
GET
2 →LIST
GET
»
```

**Comments**

- Begin the program.
- Put the covariance matrix on the stack.
- Put the list of statistics parameters on the stack.
- Make a copy.
- Move \( p \) to level 1.
- Get \( r \), the \( p \)th element in ΣPAR.
- Move ΣPAR to level 1.
- Move \( q \) to level 1.
- Get \( s \), the \( q \)th element in ΣPAR.
- Put \( \{ r, s \} \) on the stack.
- Get the \( rs \) element from ΣCOV.
- End the program.

**Result**

```
LGET
```

Put the program on the stack.
Store the program as ΣGET.
**ΣX² (Sum of Squares of x)**

Calculate $\Sigma x^2$, where the $x$'s are the elements of $C_1$ (the column specified by the first parameter in `ΣPAR`).

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>1: $\Sigma x^2$</td>
</tr>
</tbody>
</table>

**Program**

```
<
1 1
ΣGET
>
```

**Comments**

- Begin the program.
- Specify $C_1$ twice.
- Extract $\Sigma x^2$.
- End the program.

```
ENTER
'ΣX² STO
```

**ΣY² (Sum of Squares of y)**

Calculate $\Sigma y^2$, where the $y$'s are the elements of $C_2$ (the column specified by the second parameter in `ΣPAR`).

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>1: $\Sigma y^2$</td>
</tr>
</tbody>
</table>

**Program**

```
<
2 2
ΣGET
>
```

**Comments**

- Begin the program.
- Specify $C_2$ twice.
- Extract $\Sigma y^2$.
- End the program.

```
ENTER
'ΣY² STO
```

**ΣY² (Sum of Squares of y)**
\( \sum xy \) (Sum of Products of \( x \) and \( y \))

Calculate \( \Sigma xy \), where the \( x \)'s and \( y \)'s are corresponding elements of \( C_1 \) and \( C_2 \) (the columns specified by the first and second parameters in \( \Sigma PAR \)).

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>( \Sigma xy )</td>
</tr>
</tbody>
</table>

**Program**

```
<
1 2
\Sigma GET
```

**Comments**

- **Begin the program.**
- **Specify \( C_1 \) and \( C_2 \).**
- **Extract \( \Sigma xy \).**
- **End the program.**

**Example.** Calculate \( \Sigma X_2 \), \( \Sigma Y_2 \), and \( \Sigma XY \) for the following statistics data:

\[
\begin{bmatrix}
18 & 12 \\
4 & 7 \\
3 & 2 \\
11 & 1 \\
31 & 48 \\
20 & 17
\end{bmatrix}
\]

The general steps are as follows.

1. Enter the statistical data.
2. Execute SUMS to create the covariance matrix \( \Sigma COV \).
3. Execute \( \Sigma X_2 \), \( \Sigma Y_2 \), and \( \Sigma XY \).
4. If \( \Sigma DAT \) contains more than two columns (that is, if each data point contains more than two variables):
   a. Execute COLS to specify new values for \( C_1 \) and \( C_2 \). The values are stored in \( \Sigma PAR \).
   b. Execute \( \Sigma X_2 \), \( \Sigma Y_2 \), and \( \Sigma XY \).
Now try the example given above.

Clear the stack, select the STAT menu, and clear ΣDAT.

Enter the data and then check that you entered all six data points.

\[ \begin{align*}
18,12 & \quad \Sigma + \\
4,7 & \quad \Sigma + \\
3,2 & \quad \Sigma + \\
11,1 & \quad \Sigma + \\
31,48 & \quad \Sigma + \\
20,17 & \quad \Sigma + \\
\end{align*} \]

Drop the number of data points.

Create the covariance matrix ΣCOV.

Calculate Σx².

Calculate Σy².

ΣXY (Sum of Products of x and y) 37
Calculate $\Sigma xy$.

If the statistics matrix had more than two columns, you could specify new values for $C_1$ and $C_2$. For practice, specify $C_1=1$ and $C_2=2$ (the current values).

The command COL$\Sigma$ is available in the STAT menu, but here it's easier to spell out the command name and stay in the USER menu.

You could now execute $\Sigma X^2$, $\Sigma Y^2$, and $\Sigma XY$ for the new pair of columns $C_1$ and $C_2$.

Don't forget the execute SUMS again whenever you add or delete data from the statistics matrix $\Sigma DAT$. 

38 $\Sigma XY$ (Sum of Products of $x$ and $y$)
Median of Statistics Data

This section contains three programs:

- **SORT** orders the elements of a list.
- **LMED** calculates the median of a sorted list.
- **MEDIAN** uses **SORT** and **MED** to calculate the median of the current statistics data.

**SORT (Sort a List)**

Sort a list into ascending order.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: { list }</td>
<td>1: { sorted list }</td>
</tr>
</tbody>
</table>

**Techniques:**

- **Bubble sort.** Starting with the first and second numbers in the list, **SORT** compares adjacent numbers and moves the larger number toward the end of the list. This process is done once to move the largest number to the last position in list, then again to move the next largest to the next-to-last position, and so on.

- **Nested definite loops.** The outer loop controls the stopping position each time the process is done; the inner loop runs from 1 to the stopping position each time the process is done.

- **Nested local-variable structures.** **SORT** contains two local-variable structures, the second inside the defining program of the first. This nesting is done for convenience; it's easier to create the first local variable as soon as its value is computed, thereby removing its value from the stack, rather than computing both values and creating both local variables at once.

- **FOR ... STEP and FOR ... NEXT (definite loops).** **SORT** uses two indexes: –1 STEP decrements the index for the outer loop each iteration; NEXT increments the index for the inner loop by 1 each iteration.
Program

«
DUP SIZE 1

FOR j
  1 j

FOR k
  k GETI \rightarrow n1

«
  GETI \rightarrow n2

«
  DROP
  IF n1 n2 >
  THEN
    k n2 PUTI
    n1 PUT
  END
»
»
NEXT

-1 STEP
»

ENTER
'SORT' STO

Comments

Begin the program.
From the last position to the first position,
Begin the outer loop with index j.
From the first position to the jth position,
Begin the inner loop with index k.
Get the kth number in the list and store it in a local variable n1.
Begin outer defining program.
Get the next number in the list and store it in a local variable n2.
Begin inner defining program.
Drop the index.
If the two numbers are in the wrong order,
Then do the following:
Put the second one back in the kth position.
Put the kth one back in the next position.
End of THEN clause.
End inner defining program.
End outer defining program.
Increment k and repeat the inner loop.
Decrement j and repeat the outer loop.
End the program.

Put the program on the stack.
Store the program as SORT.

Example. Sort the list \{ 8, 3, 1, 2, 5 \}.

USER
\{ 8, 3, 1, 2, 5 \} \rightarrow SORT

40  SORT (Sort a List)
LMED (Median of a List)

Given a sorted list, calculate the median. If the list contains an odd number of elements, the median is the value of the center element. If the list contains an even number of elements, the median is the average value of the elements just above and below the center.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: { sorted list }</td>
<td>1: median of sorted list</td>
</tr>
</tbody>
</table>

Techniques:

- FLOOR and CEIL. For an integer, FLOOR and CEIL both return that integer; for a non-integer, FLOOR and CEIL return successive integers that bracket the non-integer.

Program

```
«
  DUP SIZE
  1 + 2 /
  » p

«
  DUP
  p FLOOR GET
  SWAP
  p CEIL GET
  + 2 /
  »
»
```

Comments

- Begin the program.
- The size of the list.
- The center position in the list (fractional for even-sized lists).
- Store the center position in local variable p.
- Begin the defining program.
- Make a copy of the list.
- Get the number at or below the center position.
- Move the list to level 1.
- Get the number at or above the center position.
- The average of the two numbers at or near the center position.
- End the defining program.
- End the program.

ENTER
'LMED [STO]

Put the program on the stack.
Store the program as LMED.
**Example.** Calculate the median of the list you sorted using SORT.

LMED is called by MEDIAN.

**MEDIAN (Median of Statistics Data)**

Return a vector representing the medians of the columns of the statistics data.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>1: [ x_1 x_2 \cdots x_m ]</td>
</tr>
</tbody>
</table>

**Techniques:**

- Arrays, lists, and stack elements. MEDIAN extracts a column of data from \( \Sigma \text{DAT} \) in vector form. To convert the vector to a list, MEDIAN puts the vector elements on the stack and then combines them into a list. From this list the median is calculated using SORT and LMED.

The median for the \( m \)th column is calculated first, and the median for the first column is calculated last, so as each median is calculated, it is moved to the stack level above the previously calculated medians.

After all medians are calculated and positioned correctly on the stack, they're combined into a vector.

- FOR \ldots\ NEXT (definite loop with index). MEDIAN uses a loop to calculate the median of each column. Because the medians are calculated in reverse order (last column first), the index is used to reverse the order of the medians.

**Required Programs:**

- SORT (page 39) arranges a list in ascending order.
- LMED (page 41) calculates the median of a sorted list.
Program

```
RCLE

DUP SIZE

LIST→ DROP

→ n m

'ΣDAT' TRN

1 m

FOR j

Σ-

ARRY→ DROP

n →LIST

SORT

LMED

j ROLLD

NEXT

m 1 →LIST

→ARRY

»

SWAP

STOΣ

»
```

Comments

Begin the program.

Put a copy of the current statistics matrix ΣDAT on the stack for safekeeping.

Put the list \( \{ n \ m \} \) on the stack, where \( n \) is the number of rows in ΣDAT and \( m \) is the number of columns.

Put \( n \) and \( m \) on the stack. Drop the list size.

Create local variables for \( n \) and \( m \).

Begin the defining program.

Transpose ΣDAT. Now \( n \) is the number of columns in ΣDAT and \( m \) is the number of rows.

The first and last rows.

For each row, do the following:

Extract the last row in ΣDAT. Initially this is the \( m \)th row, which corresponds to the \( m \)th column in the original ΣDAT.

Put the row elements on the stack.

Drop the index list \( \{ n \} \), since \( n \) is already stored in a local variable.

Make an \( n \) -element list.

Sort the list.

Calculate the median of the list.

Move the median to the proper stack level.

Increment \( j \) and repeat the loop.

Make the list \( \{ m \} \).

Combine all the medians into an \( m \) -element vector.

End the defining program.

Move the orginal ΣDAT to level 1.

Restore ΣDAT to its previous value.

End the program.

Put the program on the stack.

Store the program as MEDIAN.
Example. Calculate the median of the data on page 36. (This example assumes you've keyed in the data.) There are two columns of data, so MEDIAN will return a two-element vector.

Calculate the median.

```
USER
MEDIAN
```

The medians are 14.5 for the first column and 9.5 for the second column.
Index

A
Algebraic syntax and user functions, 12
Array operations, 42

B
Basic stack manipulations, 9
BINARY menu, 8
Binary-integer display, 28
Box functions, 11
BOXS without local variables, 13
Bubble sort, 39

C
CEIL, 41
CLEAR, 8
Clear display, 30
CLLCD, 30
CLΣ, 8, 37
COLCT, 25
Conditional structure, 17
CONT, 8, 20
Covariance matrix, 33
Creating local variables, 12
CTRL menu, 8

D
Debugging, 20
DEC, 8
Decrementing an index, 39
Defining expression, 12
Defining procedure, 11
Defining program, 18
Definite loop, 17, 39
with index, 29, 39, 42
nested, 39
DISP, 30
Display format, 8
Displaying a binary integer, 26
DO, 23
DUP2, 13

E
ELSE, 17
Enabling LAST, 29
Error trap, 28
EVAL, 23, 28, 30
Evaluation of local variables, 23
Examples, conventions in, 8
EXPAN, 25
Expand and collect, 23, 24

F
Fibonacci numbers, 16
loop version, 17
recursive version, 16
two versions compared, 19
FLOOR, 41
FOR, 29, 39, 42
Format of displayed numbers, 8
Format of program listings, 8
G
GET, 34
Get an element of \(\Sigma\)COV, 34
Global variables, 11
Global variables, evaluation, 23

H
HALT, 20

I
IF, 17
IFERR, 28
If-Then-Else function, 16
Indefinite loop, 23, 26
Index, 29, 39, 42
  as local variable, 29
decrementing, 39

K
Keys, shifted, 8
KILL, 8, 20

L
Labels, in menu, 8
Lists, 42
Local variables
  evaluation, 23
  lower-case convention, 12
Local-variable structure, 11, 27
  nested, 39
Loop, single-stepping, 20
Loop index
  as local variable, 29
decrementing, 39

M
Matrix operations, 32
Median of a list, 41
Median of statistics data, 39, 42
Menu labels, 8
Menus
  next and previous levels, 8
  used in program entry, 8

Multiple execution, 23

N
Nested definite loops, 39
Nested local-variable structures, 39
Nested user functions, 14
NEXT, 8, 29, 39, 42
\(\Sigma\), 8, 37

O
Order of program entry, 7
OVER, 10

P
Pad with leading spaces, 26
PICK, 13
PREDV, 32
PREV, 8
Procedure, defining, 11
Program entry, format, 8
Programs
  as arguments, 23
  usable in algebraic objects, 32
PURGE, 10
Purging a variable, 9
Purging programs, 7

R
RCL, 10
RCLF, 27
Real-to-binary, 28
Recalling a variable, 9
Recursion, 16
Renaming a variable, 9
REPEAT, 26
Required programs, order of
  entry, 7
Restore status, 27
ROLL, 34
ROT, 10, 13
RPN, and user functions, 12
S
Σ+, 8, 37
Σ-, 43
ΣCOL, 38
Shifted keys, 8
Single-step execution, 20
Sort a list, 39
ΣPAR convention, 32
SST, 8, 20
Stack diagram, defined, 7
Stack elements, 42
Standard display format, 8
STAT menu, 8, 37
Status, restoring, 27
STD, 8
STEP, 39
STO, 10
STOF, 27
Storing a variable, 9
String operations, 26
Structured programming, 9, 25
Subprograms, 29
Sum of products of x and y, 36
Sum of squares of x, 35
Sum of squares of y, 35
Summary statistics, 32
Summary statistics matrix, 33
Surface of a box, 11
SWAP, 10

T
Techniques, list of, 6
THEN, 17, 28

U
UNTIL, 23
User function, 12
  in algebraic syntax, 12
  nested, 14
USER menu, 8

V
Variable names, in USER menu, 8
Variables, 9
  evaluation, 23
  local and global, 11
Vectors, 42
VISIT, 8, 20

W
WHILE, 26
Contents

Page 7 How To Use This Booklet
9 Programming Examples
9 RENAME (Renaming a Variable)
11 Box Functions
16 Fibonacci Numbers
20 Single-Step Execution
23 Expand and Collect Completely
26 Displaying a Binary Integer
32 Summary Statistics
39 Median of Statistics Data
45 Index