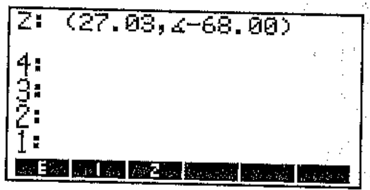
Key in the voltage value.

↰ [()] 10 ↱ [∡] 0

```
(10∡0◆
 E  I  Z
```

Store the voltage value. Then key in and store the current value. Solve for the impedance.

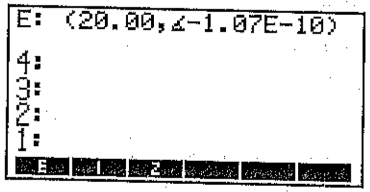
E
↰ [()] .37 ↱ [∡] 68 I
↰ Z



```
Z: (27.03,∡-68.00)
4:
3:
2:
1:
 E  I  Z
```

Recall the current and double it. Then find the voltage.

↱ I
2 [×]
I
↰ E



```
E: (20.00,∡-1.07E-10)
4:
3:
2:
1:
 E  I  Z
```

Press ↰ [MODES] FMT STD and [NXT] MODE ANGL RECT to restore Standard and Rectangular modes.

Turning Off the HP 48 from a Program

To turn off the calculator in a program:

- Execute the OFF command (PRG RUN menu).

The OFF command turns off the HP 48. If a program executes OFF, the program resumes when the calculator is next turned on.

2

Programming Examples

The programs in this chapter demonstrate basic programming concepts. These programs are intended to improve your programming skills, and to provide supplementary functions for your calculator.

At the end of each program, the program's *checksum* and size in bytes are listed to help make sure you typed the program in correctly. (The checksum is a binary integer that uniquely identifies the program based on its contents). To make sure you've keyed the program in correctly, store it in its name, put the name in level 1, then execute the BYTES command (↰ [MEMORY] BYTES). This returns the program's checksum to level 2, and its size in bytes to level 1. (If you execute BYTES with the program *object* in level 1, you'll get a different byte count.)

The programs in this chapter are also included in the online information of the Program Development Link software for developing HP 48 programs on computers. This software lets you load these programs from the online information into your HP 48 through its serial port.

The examples in this chapter assume the HP 48 is in its initial, default condition—they assume you haven't changed any of the HP 48 operating modes. (To reset the calculator to this condition, see "Memory Reset" in chapter 5 of the *HP 48 User's Guide*.)

Each program listing in this chapter gives the following information:

- A brief description of the program.
- A syntax diagram (where needed) showing the program's required inputs and resulting outputs.
- Discussion of special programming techniques in the program.
- Any other programs needed.
- The program listing.
- The program's checksum and byte size.

Fibonacci Numbers

This section includes three programs that calculate Fibonacci numbers:

- *FIB1* is a user-defined function that is defined *recursively* (that is, its defining procedure contains its own name). *FIB1* is short.
- *FIB2* is a user-defined function with a definite loop. It's longer and more complicated than *FIB1*, but faster.
- *FIBT* calls both *FIB1* and *FIB2* and calculates the execution time of each subprogram.

FIB1 and *FIB2* demonstrate an approach to calculating the *n*th Fibonacci number F_n, where:

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

FIB1 (Fibonacci Numbers, Recursive Version)

Level 1	→	Level 1
n	→	F_n

Techniques used in FIB1

- **IFTE (if-then-else function).** The defining procedure for *FIB1* contains the conditional *function* IFTE, which can take its argument either from the stack or in algebraic syntax.
- **Recursion.** The defining procedure for *FIB1* is written in terms of *FIB1*, just as F_n is defined in terms of F_{n-1} and F_{n-2}.

FIB1 program listing

Program:	**Comments:**
`«`	
`→ n`	Defines local variable *n*.

`'IFTE(n≤1,`
`n,`
`FIB1(n-1)+FIB1(n-2))'`
	The defining procedure, an algebraic expression. If $n \le 1$, $F_n = n$, else $F_n = F_{n-1} + F_{n-2}$.

`»`	
ENTER ' FIB1 STO	Stores the program in *FIB1*.

Checksum: # 41467d (press ' FIB1 ↰ MEMORY BYTES)
Bytes: 113.5

Example: Calculate F_6. Calculate F_{10} using algebraic syntax.

First calculate F_6.

VAR
6 FIB1

```
1:                8
```

Next, calculate F_{10} using algebraic syntax.

' FIB1 ↰ ( ) 10 EVAL

```
2:                8
1:               55
```

FIB2 (Fibonacci Numbers, Loop Version)

Level 1	→	Level 1
n	→	F_n

Techniques used in FIB2

- **IF ... THEN ... ELSE ... END.** *FIB2* uses the program-structure form of the conditional. (*FIB1* uses IFTE.)

- **START … NEXT (definite loop).** To calculate F_n, *FIB2* starts with F_0 and F_1 and repeats a loop to calculate successive values of F_i.

FIB2 program listing

```
Program:                  Comments:
«
 → n                      Creates a local variable structure.
 «
  IF n 1 ≤                If n ≤ 1,
  THEN n                  then Fn = n;
  ELSE                    otherwise ...
   0 1                    Puts F0 and F1 on the stack.
   2 n                    From 2 to n does the following
   START                  loop:
    DUP                   Copies the latest F (initially F1).
    ROT                   Gets the previous F (initially F0).
    +                     Calculates the next F (initially
                          F2).
   NEXT                   Repeats the loop.
   SWAP DROP              Drops Fn-1.
  END                     Ends the ELSE clause.
 »                        Ends the defining procedure.
»
ENTER ' FIB2 STO          Stores the program in FIB2.
```

Checksum: # 51820d (press ' FIB2 ↰ MEMORY BYTES)
Bytes: 89

Example: Calculate F_6 and F_{10}.

Calculate F_6.

VAR
6 FIB2

```
1:                    8
```

Calculate F_{10}.

10 FIB2

```
2:                    8
1:                   55
```

FIBT (Comparing Program-Execution Time)

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.

FIBT executes the TICKS command to record the execution time of *FIB1* and *FIB2* for a given value of n.

Level 1	→	Level 3	Level 2	Level 1
n	→	F_n	FIB1 TIME: *z*	FIB2 TIME: *z*

Techniques used in FIBT

- **Structured programming.** *FIBT* calls both *FIB1* and *FIB2*.
- **Programmatic use of calculator clock.** *FIBT* executes the TICKS command to record the start and finish of each subprogram.
- **Labeling output.** *FIBT* tags each execution time with a descriptive message.

Required Programs

- *FIB1* (page 2-2) calculates F_n using recursion.
- *FIB2* (page 2-3) calculates F_n using looping.

FIBT program listing

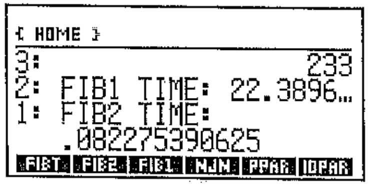
Program:	Comments:
`«`	
`DUP TICKS SWAP FIB1` `SWAP TICKS SWAP`	Copies *n*, then executes *FIB1*, recording the start and stop time.
`- B→R 8192 /`	Calculates the elapsed time, converts it to a real number, and converts that number to seconds. Leaves the answer returned by *FIB1* in level 2.
`"FIB1 TIME" →TAG`	Tags the execution time.
`ROT TICKS SWAP FIB2` `TICKS`	Executes *FIB2*, recording the start and stop time.
`SWAP DROP SWAP` `- B→R 8192 /`	Drops the answer returned by *FIB2* (*FIB1* returned the same answer). Calculates the elapsed time for *FIB2* and converts to seconds.
`"FIB2 TIME" →TAG`	Tags the execution time.
`»`	
ENTER ' FIBT STO	Stores the program in *FIBT*.

Checksum: # 22248d
Bytes: 135

Example: Calculate F_{13} and compare the execution time for the two methods.

Select the VAR menu and do the calculation.

VAR
13 FIBT



```
{ HOME }
3:                 233
2: FIB1 TIME: 22.3896…
1: FIB2 TIME:
     .082275390625
FIBT FIB2 FIB1 NJN PPAR IOPAR
```

F_{13} is 233. *FIB2* takes fewer seconds to execute than *FIB1* (far fewer if *n* is large). (The times required for the calculations depend on the contents of memory and other factors, so you may not get the exact times shown above.)

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)

PAD converts an object to a string, and if the string contains fewer than 22 characters, adds spaces to the beginning of the string till the string reaches 22 characters.

When a short string is displayed with DISP, it appears *left-justified*: its first character appears at the left end of the display. By adding spaces to the beginning of a short string, *PAD* moves the string to the right. When the string (including leading spaces) reaches 22 characters, it appears *right-justified*: its last character appears at the right end of the display. *PAD* has no effect on longer strings.

Level 1	→	Level 1
object	→	" *object*"

Techniques used in PAD

- **WHILE … REPEAT … END (indefinite loop).** The WHILE clause contains a test that executes the REPEAT clause and tests again (if true) or skips the REPEAT clause and exits (if false).

- **String operations.** *PAD* demonstrates how to convert an object to string form, count the number of characters, and combine two strings.

PAD program listing

```
Program:                    Comments:
«
 →STR                       Makes sure the object is in string
                            form. (Strings are unaffected by
                            this command.)
 WHILE                      Repeats if the string contains
  DUP SIZE 22 <             fewer than 22 characters.
 REPEAT                     Loop-clause adds a leading space.
  " " SWAP +
 END                        Ends loop.
»
ENTER ' PAD STO             Stores the program in PAD.
```

Checksum: # 38912d
Bytes: 61.5

PAD is demonstrated in the program *BDISP*.

PRESERVE (Save and Restore Previous Status)

PRESERVE stores the current calculator (flag) status, executes a program from the stack, and restores the previous status.

Level 1	→	Level 1
« *program* »	→	*result of program*
'*program name*'	→	*result of program*

Techniques used in PRESERVE

- **Preserving calculator flag status.** *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 structure to briefly remove the binary integer from the stack. Its defining procedure simply evaluates the program argument, then puts the binary integer back on the stack and executes STOF.
- **Error trapping.** *PRESERVE* uses IFERR to trap faulty program execution on the stack and to restore flags. DOERR shows the error if one occurs.

PRESERVE program listing

```
Program:                    Comments:
«
 RCLF                       Recalls the list of two 64-bit
                            binary integers representing the
                            status of the 64 system flags and
                            64 user flags.
 → f                        Stores the list in local variable f.
 «                          Begins the defining procedure.
  IFERR                     Starts the error trap.
   EVAL                     Executes the program placed on
                            the stack as the level 1 argument.
  THEN                      If the program caused an error,
   f STOF ERRN DOERR        restores flags, shows the error,
                            and aborts execution.
  END                       Ends the error routine.
  f STOF                    Puts the list back on the stack,
                            then restores the status of all
                            flags.
 »                          Ends the defining procedure.
»
ENTER ' PRESERVE STO        Stores the program in
                            PRESERVE.
```

Checksum: # 7284d
Bytes: 71

PRESERVE is demonstrated in the program *BDISP*

BDISP (Binary Display)

BDISP displays a real or binary number in HEX, DEC, OCT, and BIN bases.

Level 1	→	Level 1
# *n*	→	# *n*
n	→	*n*

Techniques used in BDISP

- **IFERR ... THEN ... END (error trap).** To accommodate real-number arguments, *BDISP* includes the command R→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 R→B command is placed inside an IFERR clause. No action is required when an error occurs (since a binary number is an acceptable argument), so the THEN clause contains no commands.
- **Enabling LASTARG.** In case an error occurs, the LASTARG recovery feature must be enabled to return the argument (the binary number) to the stack. *BDISP* clears flag −55 to enable this.
- **FOR ... NEXT loop (definite loop with counter).** *BDISP* executes a loop from 1 to 4, each time displaying *n* (the number) in a different base on a different line. The loop counter (named *j* in this program) is a local variable created by the FOR ... NEXT program structure (rather than by a → command), and automatically incremented by NEXT.
- **Unnamed programs as arguments.** A program defined only by its « and » delimiters (not stored in a variable) is not automatically evaluated, but is placed on the stack and can be used as an argument for a subroutine. *BDISP* demonstrates two uses for unnamed program arguments:
  - ☐ *BDISP* contains a main program argument and a call to *PRESERVE*. This program argument goes on the stack and is executed by *PRESERVE*.
  - ☐ *BDISP* also contains four program arguments that "customize" the action of the loop. Each of these contains a command to change the binary base, and each iteration of the loop evaluates one of these arguments.

  When *BDISP* creates a local variable for *n*, the defining procedure is an unnamed program. However, since this program is a defining procedure for a local variable structure, it *is* automatically executed.

Required Programs

- *PAD* (page 2-7) expands a string to 22 characters so that DISP shows it right-justified.
- *PRESERVE* (page 2-8) stores the current status, executes the main nested program, and restores the status.

BDISP program listing

Program: **Comments:**

```
«
 «                          Begins the main nested program.
  DUP                       Makes a copy of n.
  -55 CF                    Clears flag −55 to enable
                            LASTARG.
  IFERR                     Begins error trap.
   R→B                      Converts n to a binary integer.
  THEN                      If an error occurs, do nothing (no
  END                       commands in the THEN clause).
  → n                       Creates a local variable n and
  «                         begins the defining program.
   CLLCD                    Clears the display.
   « BIN »                  Nested program for BIN.
   « OCT »                  Nested program for OCT.
   « DEC »                  Nested program for DEC.
   « HEX »                  Nested program for HEX.
```

```
Program:               Comments:
  1 4                  Sets the counter limits.
  FOR j                Starts the loop with counter j.
    EVAL               Executes one of the nested base
                       programs (initially for HEX).
    n →STR             Makes a string showing n in the
                       current base.
    PAD                Pads the string to 22 characters.
    j DISP             Displays the string in the jth line.
  NEXT                 Increments j and repeats the
                       loop.
 »                     Ends the defining program.
 3 FREEZE              Freezes the status and stack
                       areas.
»                      Ends the main nested program.
PRESERVE               Stores the current flag status,
                       executes the main nested
                       program, and restores the status.
»
ENTER ' BDISP STO      Stores the program in BDISP
```

Checksum: # 18055d
Bytes: 191

Example: Switch to DEC base, display #100 in all bases, and check that *BDISP* restored the base to DEC.

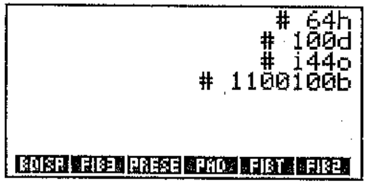
Clear the stack and select the MTH BASE menu. Make sure the current base is DEC and enter # 100.

⬅ CLEAR
MTH BASE
DEC
➡ # 100 ENTER

```
1:                 # 100d
```

Execute *BDISP*

VAR BDISP



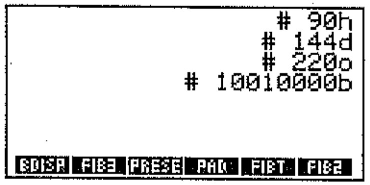
```
                   # 64h
                  # 100d
                  # 144o
              # 1100100b
```

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

CANCEL
MTH BASE

Although the main nested program left the calculator in BIN base, *PRESERVE* restored DEC base. To check that *BDISP* also works for real numbers, try 144.

VAR
144 BDISP



```
                   # 90h
                  # 144d
                  # 220o
              # 10010000b
```

Press CANCEL to return to the stack display.

Median of Statistics Data

This section contains two programs:

- *%TILE* returns the value of a specified percentile of a list.
- *MEDIAN* uses *%TILE* to calculate the median of the current statistics data.

(*%TILE* and *MEDIAN* are included in the TEACH function's EXAMPLES directory. See the entry for TEACH in chapter 3.)

%TILE (Percentile of a List)

%TILE sorts a list, then returns the value of a specified percentile of the list. For example, typing { *list* } 50 and pressing %TILE returns the median (50th percentile) of the list.

Level 2	Level 1	→	Level 1
{ list }	n	→	n^{th} *percentile of sorted list*

Techniques used in %TILE

- **FLOOR and CEIL.** For an integer, FLOOR and CEIL both return that integer; for a noninteger, FLOOR and CEIL return successive integers that bracket the noninteger.
- **SORT.** The SORT command sorts the list elements into ascending order.

%TILE program listing

Program: / **Comments:**

```
«
 SWAP SORT                 Brings the list to level 1 and sorts
                           it.
 DUP SIZE                  Copies the list, then finds its size.
 1 + ROT 100 / *           Calculates the position of the
                           specified percentile.
 → p                       Stores the center position in local
                           variable p.
 «                         Begins the defining procedure.
  DUP                      Makes a copy of the list.
  p FLOOR GET              Gets the number at or below the
                           center position.
  SWAP                     Moves the list to level 1.
  p CEIL GET               Gets the number at or above the
                           center position.
  + 2 /                    Calculates the average of the two
                           numbers.
 »                         Ends the defining procedure.
»
ENTER ' %TILE STO          Stores the program in %TILE.
```

Checksum: # 42718d
Bytes: 99

Example: Calculate the median of the list { 8 3 1 5 2 }.

⬅ {} 8 3 1 5 2 ENTER
VAR 50 %TILE

```
1:                    3
```

MEDIAN (Median of Statistics Data)

MEDIAN returns a vector containing the medians of the columns of the statistics data. Note that for a sorted list with an odd number of elements, the median is the value of the center element; for a list with an even number of elements, the median is the average value of the elements just above and below the center.

Level 1	→	Level 1
	→	[x_1 x_2 … x_m]

Techniques used in MEDIAN

- **Arrays, lists, and stack elements.** *MEDIAN* extracts a column of data from *ΣDAT* in vector form. To convert the vector to a list, *MEDIAN* puts the vector elements on the stack and combines them into a list. From this list the median is calculated using *%TILE*.

  The median for the *m*th column is calculated first, and the median for the first column is calculated last. As each median is calculated, ROLLD is used to move it to the top of the stack.

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

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

Required Program

- *%TILE* (page) sorts a list and returns the value of a specified percentile.

MEDIAN program listing

Program:	Comments:
`«`	
`RCLΣ`	Puts a copy of the current statistics matrix *ΣDAT* on the stack.
`DUP SIZE`	Puts the list { *n* *m* } on the stack, where *n* is the number of rows in *ΣDAT* and *m* is the number of columns.
`OBJ→ DROP`	Puts *n* and *m* on the stack, and drops the list size.
`→ Σ n m`	Creates local variables for *s*, *n*, and *m*.
`«`	Begins the defining procedure.
`'ΣDAT' TRN`	Recalls and transposes *ΣDAT*. Now *n* is the number of columns in *ΣDAT* and *m* is the number of rows. (To key in the Σ character, press ↱ Σ, then delete the parentheses.)
`1 m`	Specifies the first and last rows.
`FOR j`	For each row, does the following:
`Σ-`	Extracts the last row in *ΣDAT*. Initially this is the *m*th row, which corresponds to the *m*th column in the original *ΣDAT*. (To key in the Σ− command, press ↰ STAT DATA Σ-.)
`OBJ→ DROP`	Puts the row elements on the stack. Drops the index list { *n* }.
`n →LIST`	Makes an *n*-element list.
`50 %TILE`	Sorts the list and calculates its median.
`j ROLLD`	Moves the median to the proper stack level.
`NEXT`	Increments *j* and repeats the loop.

Program:	**Comments:**
`m →ARRY`	Combines all the medians into an *m*-element vector.
`s STOΣ`	Restores *ΣDAT* to its previous value.
`»`	
`»`	Ends the defining procedure.
ENTER ' MEDIAN STO	Stores the program in *MEDIAN*.

Checksum: # 57504d
Bytes: 140

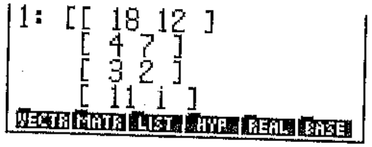
Example: Calculate the median of the following data.

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

There are two columns of data, so *MEDIAN* will return a two-element vector.

Enter the matrix.

⮫ MATRIX
18 ENTER 12 ENTER ▼
4 ENTER 7 ENTER
3 ENTER 2 ENTER
11 ENTER 1 ENTER
31 ENTER 48 ENTER
20 ENTER 17 ENTER
ENTER



```
1: [[ 18 12 ]
    [ 4 7 ]
    [ 3 2 ]
    [ 11 1 ]
```

Store the matrix in *ΣDAT*, and calculate the median.

⮨ STAT DATA
CLΣ Σ+
VAR MEDIA

```
1:           [ 14.5 9.5 ]
```

Expanding and Collecting Completely

This section contains two programs:

- *MULTI* repeats a program until the program has no effect on its argument.
- *EXCO* calls *MULTI* to completely expand and collect an algebraic.

MULTI (Multiple Execution)

Given an object and a program that acts on the object, *MULTI* applies the program to the object repeatedly until the program no longer changes the object.

Level 2	Level 1	→	Level 1
object	*« program »*	→	$\mathit{object}_{\text{result}}$

Techniques used in MULTI

- **DO … UNTIL … END (indefinite loop).** The DO clause contains the steps to be repeated. The UNTIL clause contains the test that repeats both clauses again (if false) or exits (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 convenient to store an object in a local variable when you don't know beforehand how many copies you'll need. An object stored in a local variable is simply put on the stack when the local variable is evaluated. *MULTI* uses the local variable name to put the program argument on the stack and then executes EVAL to execute the program.

MULTI program listing

```
Program:                  Comments:
«
 → p                      Creates a local variable p that
                          contains the program from level 1.
 «                        Begins the defining procedure.
  DO                      Begins the DO loop clause.
   DUP                    Makes a copy of the object, now
                          in level 1.
   p EVAL                 Applies the program to the
                          object, returning its new version.
   DUP                    Makes a copy of the new object.
   ROT                    Moves the old version to level 1.
  UNTIL                   Begins the DO test clause.
   SAME                   Tests whether the old version and
                          the new version are the same.
  END                     Ends the DO structure.
 »                        Ends the defining procedure.
»
ENTER ' MULTI STO         Stores the program in MULTI.
```

Checksum: # 34314d
Bytes: 56

MULTI is demonstrated in the next programming example.

EXCO (Expand and Collect Completely)

EXCO repeatedly executes EXPAN on an algebraic until the algebraic doesn't change, then repeatedly executes COLCT until the algebraic doesn't change. In some cases the result will be a number.

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*.

Level 1	→	Level 1
'algebraic'	→	*'algebraic'*
'algebraic'	→	*z*

Techniques used in EXCO

- **Subroutines.** *EXCO* calls the program *MULTI* twice. It is more efficient to create program *MULTI* and simply call its name twice than write each step in *MULTI* two times.

Required Programs

- *MULTI* (page 2-19) repeatedly executes the programs that *EXCO* provides as arguments.

EXCO program listing

```
Program:                  Comments:
«
 « EXPAN »                Puts a program on the stack as
                          the level 1 argument for MULTI.
                          The program executes the
                          EXPAN command.
 MULTI                    Executes EXPAN until the
                          algebraic object doesn't change.
 « COLCT »                Puts another program on the
                          stack for MULTI. The program
                          executes the COLCT command.
 MULTI                    Executes COLCT until the
                          algebraic object doesn't change.
»
ENTER ' EXCO STO          Stores the program in EXCO.
```

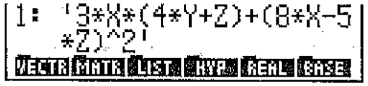
Checksum: # 48008d
Bytes: 65.5

Example: Expand and collect completely the expression:

$$3x(4y+z)+(8x-5z)^2$$

Enter the expression.

' 3 × X ×
⬑ ( ) 4 × Y + Z ▶ +
⬑ ( ) 8 × X − 5 × Z
▶ y^x 2
ENTER



```
1: '3*X*(4*Y+Z)+(8*X-5
   *Z)^2'
```

Select the VAR menu and start the program.

VAR EXCO

```
1: '64*X^2+12*X*Y-77*X
   *Z+25*Z^2'
```

Minimum and Maximum Array Elements

This section contains two programs that find the minimum or maximum element of an array:

- *MNX* uses a DO ... UNTIL ... END (indefinite) loop.
- *MNX2* uses a FOR ... NEXT (definite) loop.

MNX (Minimum or Maximum Element—Version 1)

MNX finds the minimum or maximum element of an array on the stack.

Level 1	→	Level 2	Level 1
[[*array*]]	→	[[*array*]]	z_{min} or z_{max}

Techniques used in MNX

- **DO ... UNTIL ... END (indefinite loop).** The DO clause contains the sort instructions. The UNTIL clause contains the system-flag test that determines whether to repeat the sort instructions.
- **User and system flags for logic control:**
  - ☐ *User* flag 10 defines the sort: When flag 10 is set, *MNX* finds the maximum element; when flag 10 is clear, it finds the minimum element. *You* determine the state of flag 10 at the beginning of the program.
  - ☐ *System* flag −64, the Index Wrap Indicator flag, determines when to end the sort. While flag −64 is clear, the sort loop continues. When the index invoked by GETI wraps back to the first array element, flag −64 is *automatically* set, and the sort loop ends.
- **Nested conditional.** An IF ... THEN ... END conditional is nested in the DO ... UNTIL ... END conditional, and determines the following:
  - ☐ Whether to maintain the current minimum or maximum element, or make the current element the new minimum or maximum.
  - ☐ The sense of the comparison of elements (either < or >) based on the status of flag 10.
- **Custom menu.** *MNX* builds a custom menu that lets you choose whether to sort for the minimum or maximum element. Key 1, labeled MAX, sets flag 10. Key 2, labeled MIN, clears flag 10.
- **Logical function.** *MNX* executes XOR (*exclusive OR*) to test the combined state of the relative value of the two elements and the status of flag 10.

MNX program listing

Program: / **Comments:**

```
«
 { { "MAX"
  « 10 SF CONT » }
  { "MIN"
  « 10 CF CONT » } }
```

Defines the option menu. MAX sets flag 10 and continues execution. MIN clears flag 10 and continues execution.

```
 TMENU
 "Sort for MAX or MIN?"
 PROMPT
```

Displays the temporary menu and a prompt message.

```
 1 GETI
```

Gets the first element of the array.

```
 DO
```

Begins the DO loop.

```
  ROT ROT GETI
```

Puts the index and the array in levels 1 and 2, then gets the new array element.

```
  4 ROLL DUP2
```

Moves the current minimum or maximum array element from level 4 to level 1, then copies both.

```
  IF
   < 10 FS? XOR
```

Tests the combined state of the relative value of the two elements and the status of flag 10.

```
  THEN
   SWAP
  END
```

If the new element is either less than the current maximum or greater than the current minimum, swaps the new element into level 1.

```
  DROP
```

Drops the other element off the stack.

```
 UNTIL
  -64 FS?
```

Begins the DO test-clause.

Tests if flag −64 is set—if the index reached the end of the array.

```
 END
 SWAP DROP 0 MENU
```

Ends the DO loop.

Swaps the index to level 1 and drops it. Restores the last menu.

```
»
```

ENTER ' MNX STO

Stores the program in *MNX*.

Checksum: # 57179d
Bytes: 210.5

Example: Find the maximum element of the following matrix:

$$\begin{bmatrix} 12 & 56 \\ 45 & 1 \\ 9 & 14 \end{bmatrix}$$

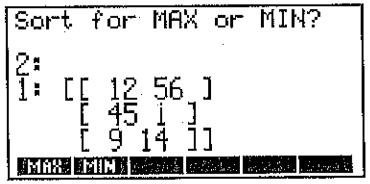
Enter the matrix.

↱ MATRIX
12 ENTER 56 ENTER ▼
45 ENTER 1 ENTER
9 ENTER 14 ENTER
ENTER

```
1: [[ 12 56 ]
    [ 45 1 ]
    [ 9 14 ]]
```

Select the VAR menu and execute *MNX*.

VAR MNX



```
Sort for MAX or MIN?
2:
1: [[ 12 56 ]
    [ 45 1 ]
    [ 9 14 ]]
```

Find the maximum element.

MAX

```
2: [[ 12 56 ] [ 45 1…
1:                 56
```

MNX2 (Minimum or Maximum Element—Version 2)

Given an array on the stack, *MNX2* finds the minimum or maximum element in the array. *MNX2* uses a different approach than *MNX*: it executes OBJ→ to break the array into individual elements on the stack for testing, rather than executing GETI to index through the array.

Level 1	→	Level 2	Level 1
[[*array*]]	→	[[*array*]]	z_{max} or z_{min}

Techniques used in MNX2

- **FOR . . . NEXT (definite loop).** The initial counter value is 1. The final counter value is $nm - 1$, where nm is the number of elements in the array. The loop-clause contains the sort instructions.
- **User flag for logic control.** *User* flag 10 defines the sort: When flag 10 is set, *MNX2* finds the maximum element; when flag 10 is clear, it finds the minimum element. *You* determine the status of flag 10 at the beginning of the program.
- **Nested conditional.** An IF . . . THEN . . . END conditional is nested in the FOR . . . NEXT loop, and determines the following:
  - ☐ Whether to maintain the current minimum or maximum element, or make the current element the new minimum or maximum.
  - ☐ The sense of the comparison of elements (either < or >) based on the status of flag 10.
- **Logical function.** *MNX2* executes XOR (*exclusive OR*) to test the combined state of the relative value of the two elements and the status of flag 10.
- **Custom menu.** *MNX2* builds a custom menu that lets you choose whether to sort for the minimum or maximum element. Key 1, labeled MAX, sets flag 10. Key 2, labeled MIN, clears flag 10.

MNX2 program listing

Program: / **Comments:**

```
«
 {{ "MAX"
  « 10 SF CONT » }
  { "MIN"
  « 10 CF CONT » }}
```

Defines the temporary option menu. MAX sets flag 10 and continues execution. MIN clears flag 10 and continues execution.

```
 TMENU
 "Sort for MAX or MIN?"
  PROMPT
```

Displays the temporary menu and a prompting message.

```
 DUP OBJ→
```

Copies the array. Returns the individual array elements to levels 2 through $nm+1$, and returns the list containing n and m to level 1.

```
 1
```

Sets the initial counter value.

```
 SWAP OBJ→
```

Converts the list to individual elements on the stack.

```
 DROP * 1 -
```

Drops the list size, then calculates the final counter value ($nm - 1$).

```
 FOR n
```

Starts the FOR . . . NEXT loop.

```
  DUP2
```

Saves the array elements to be tested (initially the last two elements). Uses the last array element as the current minimum or maximum.

```
  IF
   > 10 FS? XOR
```

Tests the combined state of the relative value of the two elements and the status of flag 10.

```
  THEN
   SWAP
  END
```

If the new element is either less than the current maximum or greater than the current minimum, swaps the new element into level 1.

Program:	Comments:
`DROP`	Drops the other element off the stack.
`NEXT`	Ends the FOR ... NEXT loop.
`0 MENU`	Restores the last menu.
`»`	
ENTER ' MNX2 STO	Stores the program in *MNX2*.

Checksum: # 12277d
Bytes: 200.5

Example: Use *MNX2* to find the minimum element of the matrix from the previous example:

$$\begin{bmatrix} 12 & 56 \\ 45 & 1 \\ 9 & 14 \end{bmatrix}$$

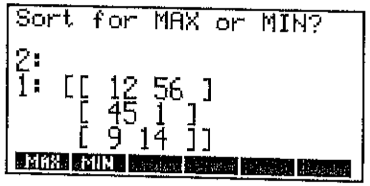
Enter the matrix (or retrieve it from the previous example).

↱ MATRIX
12 ENTER 56 ENTER ▼
45 ENTER 1 ENTER
9 ENTER 14 ENTER
ENTER

```
1: [[ 12 56 ]
    [ 45 1 ]
    [ 9 14 ]]
```

Select the VAR menu and execute *MNX2*.

VAR MNX2



```
Sort for MAX or MIN?
2:
1: [[ 12 56 ]
    [ 45 1 ]
    [ 9 14 ]]
```

Find the minimum element.

MIN

```
2: [[ 12 56 ]  [ 45 1…
1:                   1
```

Applying a Program to an Array

APLY makes use of list processing to transform each element of an array according to a desired procedure. The input array must be numeric, but the output "array" may be symbolic. Since arrays cannot actually contain symbolic objects, a convention for symbolic "pseudo-arrays" is used. Each row of elements is grouped into a single list and the set of rows is grouped into a list. For example, a 2 × 2 pseudo-array looks like this:

{ { $element_{11}$ $element_{12}$ }
{ $element_{21}$ $element_{22}$ } }

The procedure applied to each element must be a program that takes exactly one argument (i.e. the element) and returns exactly one result (i.e. the transformed element).

Level 2	Level 1	→	Level 1
[*array*]	« *program* »	→	[[*array*]] *or* {{ *array* }}

Techniques used in APLY

- **Manipulating Meta-Objects.** *Meta-objects* are composite objects like arrays and lists that have been disassembled on the stack. APLY illustrates several approaches to manipulating the elements and dimensions of such objects.
- **Application of List Processing.** APLY makes use of DOSUBS (although DOLIST might also have been used) to perform the actual transformation of array elements.
- **Using an IFERR ... THEN ... ELSE ... END Structure.** The entire symbolic pseudo-array case is handled within a error structure—triggered when the →ARRY command generates an error when symbolic elements are present.
- **Using Flags.** User flag 1 is used to track the case when the input array is a vector.

APLY program listing

Program: / **Comments:**

```
«
 → a p
```

Store the array and program in local variables.

```
  «
```

Begin the main local variable structure.

```
   1 CF
```

Make sure the flag 1 is clear to begin the procedure.

```
   a DUP SIZE
```

Retrieve the dimensions of the array.

```
   DUP SIZE
   IF 1 ==
```

Determine if the array is a vector.

```
   THEN 1 SF 1 +
```

If array is a vector,set flag 1 and add a second dimension by treating the vector as an $n \times 1$ matrix.

```
    SWAP OBJ→ OBJ→ DROP
```

Disassemble the original vector, leaving the element count, n, in level 1.

```
    1 + ROLL
```

Roll the elements up the stack and bring the "matrix" dimensions of the vector to level 1.

```
   ELSE DROP2 a OBJ→
```

If array is a matrix, clean up the stack and decompose the matrix into its elements, leaving its dimension list on level 1.

```
   END DUP OBJ→ DROP *
```

Duplicate the dimension list and compute the total number of elements.

```
   SWAP OVER 2 +
   ROLLD →LIST
```

Roll up the element count and combine all elements into a list. Note that the elements in the list are in row-major order.

```
   1 p DOSUBS
```

Recalls the program and uses it as an argument for DOSUBS (DOLIST works in this case as well). Result is a list of transformed elements.

Program: / **Comments:**

```
   OBJ→ 1 + ROLL
```

Disassembles the result list and brings the array dimensions to level 1.

```
   IFERR
```

Begins the error-trapping structure. Its purpose is to find and handle the cases when the result list contains symbolic elements.

```
    IF 1 FS?
```

Was original array a vector?

```
    THEN OBJ→ DROP →LIST
```

If the original array was a vector, then drop the second dimension (1) from the dimension list.

```
    END →ARRY
```

Convert the elements into a array with the given dimensions. If there are symbolic elements present, an error will be generated and the error-clause which follows will be executed.

```
   THEN
```

Begin the error clause.

```
   OBJ→
```

Put the array dimensions on levels 2 and 1. If the array is a vector, level 1 contains a 1.

```
    IF 1 FC?C
```

Is original array a matrix? Clear flag 1 after performing the test.

```
    THEN DROP
```

Drop the number of matrix elements.

```
    END → n m
```

Store the array dimensions in local variables.

```
     « 1 n
      FOR i
```

Begin local variable structure and initiate FOR..NEXT loop for each row.

```
        m →LIST
```

Collect a group of elements into a row (a list).

```
        'm*(n-i)+1' EVAL
        ROLLD
```

Computes the number of elements to roll so that the next row can be collected.

```
      NEXT
      n →LIST
```

Repeat loop for the next row.

Gather rows into a list, forming a list of lists (symbolic pseudo-array).

Program:	Comments:
`»`	
`END 1 CF`	Close the local variable structure and end the IFERR..THEN..END structure. Clear flag 1 before exiting the program.
`»`	
`»`	
(ENTER) (') APLY (STO)	Stores the program in *APLY*

Checksum: # 49768d
Bytes: 319

Example: Apply the function, $f(x) = Ax^3 - 7$ to each element x of the vector [3 -2 4 -8 1].

(⬅) ([ ]) 3 (SPC) 2 (+/−) 4 (SPC) 8 (+/−) 1 (ENTER)
(⬅) («») 3 (y^x) A (×) 7 (−) (ENTER)
(VAR) APLY

```
1: { { '27*A-7' } { '-
   (8*A)-7' } { '64*A-
   7' } { '-(512*A)-7'
   } { 'A-7' } }
```

Converting Between Number Bases

nBASE converts a positive decimal number (x) into a tagged string representation of the equivalent value in a different number base (b). Both x and b must be real numbers. *nBASE* automatically rounds both arguments to the nearest integer.

Level 2	Level 1	→	Level 1
x	*b*	→	*x* base*b*: "*string*"

Techniques used in nBASE

- **String Concatenation and Character Manipulation.** nBASE makes use of several string and character manipulation techniques to build up the result string.
- **Tagged Output.** nBASE labels ("tags") the output string with its original arguments so that the output is a complete record of the command.
- **Indefinite Loops.** nBASE accomplishes most of its work using indefinite loops—both DO..UNTIL..END and WHILE..REPEAT..END loops.

nBASE program listing

Program:	Comments:
`«`	
`1 CF 0 RND SWAP 0 RND`	Clear flag 1 and round both arguments to integers.
`→ b n`	Store the base and number in local variables.
`«`	Begin the outer local variable structure.
`n LOG b LOG /`	Computes the ratio of the common logarithms of number and base.
`10 RND`	Rounds result to remove imprecision in last decimal place.
`IP n 0`	Find the integer part of log ratio, recall the original number, and initialize the counter variable *k* for use in the DO..UNTIL loop.
`→ 1 m k`	Store the values in local variables.

Program:

```
   «
    ""
    DO
```

Comments:

Begin inner local variable structure, enter an empty string and begin the DO..UNTIL..END loop.

```
      'm' EVAL b i
      'k' EVAL - ^
```

Compute the decimal value of the $(i-k)$th position in the string.

```
     DUP2 MOD
```

Makes a copy of the arguments and computes the decimal value still remaining that must be accounted for by other positions.

```
     IF DUP 0 ==
      'm' EVAL b ≥
      AND
```

Is the remainder zero *and* $m \geq b$?

```
     THEN 1 SF
```

If the test is true, then set flag 1.

```
     END 'm' STO
```

Store the remainder in m.

```
     / IP
```

Compute the number of times the current position-value goes into the remaining decimal value. This is the "digit" that belongs in the current position.

```
     IF DUP 10 ≥
```

Is the "digit" ≥ 10?

```
     THEN 55 + CHR
```

Then convert the digit into a alphabetic digit (such as A, B, C, …).

```
     END +
     'k' 1 STO+
```

Append the digit to the current result string and increment the counter variable k.

Program:

```
    UNTIL 'm' EVAL 0 ==
```

Comments:

Repeat the DO..UNTIL loop until $m = 0$ (i.e. all decimal value has been accounted for).

```
    END
    IF i FS?C
```

Is flag 1 set? Clear the flag after the test.

```
    THEN 0 +
```

Then add a placeholder zero to the result string.

```
     WHILE i 'k' EVAL
      - 0 ≠
```

Begin WHILE..REPEAT loop to determine if additional placeholder zeros are needed. Loop repeats as long as $i \neq k$.

```
     REPEAT 0 +
      i 'k' STO
```

Add an additional placeholding zero and increment k before repeating the test-clause.

```
     END
    END
   »
```

End the WHILE..REPEAT..END loop, the IF..THEN..END structure, and the inner local variable structure.

```
   " base" b +
   n SWAP + →TAG
```

End the outermost IF..THEN..ELSE..END structure and create the label string and tag the result string using the original arguments.

```
  »
»
```

(ENTER) (') nBASE (STO) Stores the program in *nBASE*.

Checksum: # 19700d
Bytes: 417.5

Example: Convert 1000_{10} to base 23.

1000 (ENTER) 23 (VAR) NBASE

```
1:  1000 base23: "1KB"
```

Verifying Program Arguments

The two utility programs in this section verify that the argument to a program is the correct object type.

- *NAMES* verifies that a list argument contains exactly two names.
- *VFY* verifies that the argument is either a name or a list containing exactly two names. It calls *NAMES* if the argument is a list.

You can modify these utilities to verify other object types and object content.

NAMES (Check List for Exactly Two Names)

If the argument for a program is a list (as determined by *VFY*), *NAMES* verifies that the list contains exactly two names. If the list does not contain exactly two names, an error message appears in the status area and program execution is aborted.

Level 1	→	Level 1
{ valid list }	→	
{ invalid list }	→	*(error message in status area)*

Techniques used in NAMES

- **Nested conditionals.** The outer conditional verifies that there are two objects in the list. If so, the inner conditional verifies that both objects are names.
- **Logical functions.** *NAMES* uses the AND command in the inner conditional to determine if *both* objects are names, and the NOT command to display the error message if they are not both names.

NAMES program listing

```
Program:                       Comments:
«
 IF                            Starts the outer conditional
                               structure.
  OBJ→                         Returns the n objects in the list
                               to levels 2 through (n + 1), and
                               returns the list size n to level 1.
  DUP 2 SAME                   Copies the list size and tests if
                               it's 2.
 THEN                          If the size is 2, moves the
  DROP                         objects to levels 1 and 2, and
  IF                           starts the inner conditional
                               structure.
   TYPE 6 SAME                 Tests if the first object is a
                               name: returns 1 if so, otherwise
                               0.
   SWAP TYPE 6 SAME            Moves the second object to level
                               1, then tests if it is a name
                               (returns 1 or 0).
   AND                         Combines test results: Returns
                               1 if both tests were true,
                               otherwise returns 0.
   NOT                         Reverses the final test result.
  THEN                         If the objects are not both
  "List needs two names"       names, displays an error
  DOERR                        message and aborts execution.
  END                          Ends the inner conditional
                               structure.
 ELSE                          If the list size is not 2, drops the
  DROPN                        list size, displays an error
  "Illegal list size"          message, and aborts execution.
  DOERR
 END                           Ends the outer conditional.
»
ENTER ' NAMES STO              Stores the program in NAMES.
```

Checksum: # 40666d
Bytes: 141.5

NAMES is demonstrated in the program *VFY*.

VFY (Verify Program Argument)

VFY verifies that an argument on the stack is either a name or a list that contains exactly two names.

Level 1	→	Level 1
'name'	→	*'name'*
{ *valid list* }	→	{ *valid list* }
{ *invalid list* }	→	{ *invalid list* } *(and error message in status area)*
invalid object	→	*invalid object (and error message in status area)*

Techniques used in VFY

- **Utility programs.** *VFY* by itself has little use. However, it can be used with minor modifications by other programs to verify that specific object types are valid arguments.
- **CASE … END (case structure).** *VFY* uses a case structure to determine if the argument is a list or a name.
- **Structured programming.** If the argument is a list, *VFY* calls *NAMES* to verify that the list contains exactly two names.
- **Local variable structure.** *VFY* stores its argument in a local variable so that it can be passed to *NAMES* if necessary.
- **Logical function.** *VFY* uses NOT to display an error message.

Required Programs

- *NAMES* (page 2-36) verifies that a list argument contains exactly two names.

VFY program listing

```
Program:                        Comments:
«
 DUP                            Copies the original argument
                                to leave on the stack.
 DTAG                           Removes any tags from the
                                argument for subsequent
                                testing.
 → argm                         Stores the argument in local
                                variable argm.
 «                              Begins the defining procedure.
  CASE                          Begins the case structure.
    argm TYPE 5 SAME            Tests if the argument is a list.
   THEN                         If so, puts the argument back
    argm NAMES                  on the stack and calls NAMES
   END                          to verify that the list is valid,
                                then leaves the CASE
                                structure.
    argm TYPE 6 SAME NOT        Tests if the argument is not a
   THEN                         name. If so, displays an error
    "Not name or list"          message and aborts execution.
    DOERR
   END
  END                           Ends the CASE structure.
 »                              Ends the defining procedure.
»
ENTER ' VFY STO                 Enters the program, then
                                stores it in VFY.
```

Checksum: # 36796d
Bytes: 139.5

Example: Execute *VFY* to test the validity of the name argument *BEN*. (The argument is valid and is simply returned to the stack.)

' BEN ENTER
VAR VFY

```
1:                    'BEN'
```

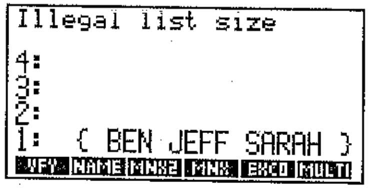
Example: Execute *VFY* to test the validity of the list argument { *BEN JEFF SARAH* }. Use the name from the previous example, then enter the names *JEFF* and *SARAH* and convert the three names to a list.

' JEFF ENTER
' SARAH ENTER
3 PRG LIST →LIST


```
1:  { BEN JEFF SARAH }
```

Execute *VFY*. Since the list contains too many names, the error message is displayed and execution is aborted.

VAR VFY


```
Illegal list size
4:
3:
2:
1:  { BEN JEFF SARAH }
```

Converting Procedures from Algebraic to RPN

This section contains a program, →*RPN*, that converts an algebraic expression into a series (list) of objects in equivalent RPN order. Note that →RPN is a program provided with the TEACH command. You can find it in the EXAMPLES directory by pressing EXAM PRGS →RPN.

Level 1	→	Level 1
'symb'	→	*{ objects }*

Techniques used in →RPN

- **Recursion.** The →RPN program calls itself as a subroutine. This powerful technique works just like calling another subroutine as long as the stack contains the proper arguments before the program calls itself. In this case the level 1 argument is tested first to be sure that it is an algebraic expression before →RPN is called again.
- **Object Type-Checking.** →RPN uses conditional branching that depends on the object type of the level 1 object.
- **Nested Program Structures.** →RPN nests IF ... THEN ... END structures inside FOR ... NEXT loops inside a IF ... THEN ... ELSE ... END structure.
- **List Concatenation.** The result list of objects in RPN order is built by using the ability of the + command to sequentially append additional elements to a list. This is a handy technique for gathering results from a looping procedure.

→RPN program listing

```
Program:                          Comments:
«
 OBJ→                             Take the expression apart.
 IF OVER                          If the argument count is
 THEN → n f                       nonzero, then store the count
                                  and the function.
  «                               Begins local variable defining
                                  procedure.
   1 n                            Begins FOR ... NEXT loop,
   FOR i                          which converts any algebraic
                                  arguments to lists.
    IF DUP TYPE 9 SAME            Tests whether argument is an
                                  algebraic.
    THEN →RPN                     If argument is an algebraic,
                                  convert it to a list first.
    END n ROLLD                   Roll down the stack to prepare
                                  for the next argument.
   NEXT                           Repeat the loop for the next
                                  argument.
   IF DUP TYPE 5 ≠                Tests to see if level 1 object is a
                                  list.
   THEN 1 →LIST                   If not a list, then convert it to
                                  one.
   END                            Ends the IF ... THEN ...
                                  END structure.
```

Program:

```
  IF n 1 >

  THEN 2 n
   START +
   NEXT
  END f +

 »

ELSE i →LIST SWAP DROP


END

»
```

Comments:

Tests to see if there is more than one argument.

Combine all of the arguments into a list.

Append the function to the end of the list.

End the local variable defining procedure.

For functions with no arguments, converts to a simple list.

End the IF ... THEN ... ELSE ... END structure.

Checksum: # 28598d
Bytes: 189.5

Example: Convert the following algebraic expression to a series of objects in RPN syntax: `'A*COS(B+√(C/D))-X^3'`.

['] A [×] [COS] B [+] [√] [↰] [()] C [÷] D [▶] [▶] [−] X [y^x] 3 [ENTER] →RPN

```
1: { A B C D / √ + COS
   * X 3 ^ - }
```

Bessel Functions

This section contains a program, *BER*, that calculates the real part $\mathrm{Ber}_n(x)$ of the Bessel function $J_n(xe^{3\pi i/4})$. When $n = 0$,

$$\mathrm{Ber}(x) = 1 - \frac{(x/2)^4}{2!^2} + \frac{(x/2)^8}{4!^2} - \cdots$$

Level 1	→	Level 1
z	→	*Ber(z)*

Techniques used in BER

- **Local variable structure.** At its outer level, *BER* consists solely of a local variable structure and so has two properties of a user-defined function: it can take numeric or symbolic arguments from the stack, or it can take arguments in algebraic syntax. However, because *BER* uses a DO ... UNTIL ... END loop, its defining procedure is a *program*. (Loop structures are not allowed in algebraic expressions.) Therefore, unlike user-defined functions, *BER* is not differentiable.
- **DO ... UNTIL ... END loop (indefinite loop with counter).** *BER* calculates successive terms in the series using a counter variable. When the new term does not differ from the previous term to within the 12-digit precision of the calculator, the loop ends.
- **Nested local variable structures.** The outer structure is consistent with the requirements of a user-defined function. The inner structure allows storing and recalling of key parameters.

BER program listing

```
Program:                          Comments:
«
 → x                              Creates local variable x.
 «                                Begins outer defining procedure.
  'x/2' →NUM 2 i                  Enters x/2, the first counter
  → xover2 j sum                  value, and the first term of the
                                  series, then creates local
                                  variables.
  «                               Begins inner defining procedure.
   DO                             Begins the loop.
    sum                           Recalls the old sum and
    'sum+(-i)^(j/2)*              calculates the new sum.
    xover2^(2*j)/SQ(j!)'
    EVAL
    2 'j' STO+                    Increments the counter.
    DUP 'sum' STO                 Stores the new sum.
   UNTIL                          Ends the loop clause.
    ==                            Tests the old and new sums.
   END                            Ends the loop.
   sum                            Recalls the sum.
  »                               Ends inner defining procedure.
 »                                Ends outer defining procedure.
»
```

(ENTER) (') BER (STO) — Stores the program in *BER*.

Checksum: # 36388d
Bytes: 200.5

Example: Calculate Ber(3).

(VAR)
3 BER

```
1:              -.2213802496
 BER  VFY  NAME MNX2 MNX  EXCO
```

Calculate Ber(2) in algebraic syntax.

(') BER (↰)(( )) 2
(EVAL)

```
1:              .751734182714
 BER  VFY  NAME MNX2 MNX  EXCO
```

Animation of Successive Taylor's Polynomials

This section contains three programs that manipulate graphics objects to display a sequence of Taylor's polynomials for the sine function.

- *SINTP* draws a sine curve, and saves the plot in a variable.
- *SETTS* superimposes plots of successive Taylor's polynomials on the sine curve plot from *SINTP*, and saves the resulting graphics objects in a list.
- *TSA* uses the ANIMATE command to display in succession each graphics object from the list built in *SETTS*.

SINTP (Converting a Plot to a Graphics Object)

SINTP draws a sine curve, returns the plot to the stack as a graphics object, and stores that graphics object in a variable. Make sure your calculator is in Radians mode.

Techniques used in SINTP

- **Programmatic use of PLOT commands.** *SINTP* uses PLOT commands to build and display a graphics object.

SINTP program listing

Program:	Comments:
`«`	
`'SIN(X)' STEQ`	Stores the expression for sin x in *EQ*.

`FUNCTION '-2*π' →NUM`
`DUP NEG XRNG`
`-2 2 YRNG`
	Sets the plot type and x- and y-axis display ranges.

`ERASE DRAW`	Erases *PICT*, then plots the expression.
`PICT RCL 'SINT' STO`	Recalls the resultant graphics object and stores it in *SINT*.
`»`	
ENTER ' SINTP STO	Stores the program in *SINTP*

Checksum: # 1971d
Bytes: 91.5

SINTP is demonstrated in the program *TSA*.

SETTS (Superimposing Taylor's Polynomials)

SETTS superimposes successive Taylor's polynomials on a sine curve and stores each graphics object in a list.

Techniques used in SETTS

- **Structured programming.** *SETTS* calls *SINTP* to build a sine curve and convert it to a graphics object.
- **FOR . . . STEP (definite loop).** *SETTS* calculates successive Taylor's polynomials for the sine function in a definite loop. The loop counter serves as the value of the order of each polynomial.
- **Programmatic use of PLOT commands.** *SETTS* draws a plot of each Taylor's polynomial.
- **Manipulation of graphics objects.** *SETTS* converts each Taylor's polynomial plot into a graphics object. Then it executes + to combine each graphics object with the sine curve stored in *SINT*, creating nine new graphics objects, each the superposition of a Taylor's polynomial on a sine curve. *SETTS* then puts the nine new graphics objects, and the sine curve graphics object itself, in a list.

SETTS program listing

Program:	Comments:
`«`	
`SINTP`	Plots a sine curve and stores the graphics object in *SINT*.
`1 17 FOR n`	Sets the range for the FOR loop using local variable n.
`'SIN(X)' 'X' n TAYLR` `STEQ ERASE DRAW`	Plots the Taylor's polynomial of order n.
`PICT RCL SINT +`	Returns the plot to the stack as a graphics object and executes + to superimpose the sine plot from *SINT*.
`2 STEP`	Increments the loop counter n by 2 and repeats the loop.

`SINT`
`10 →LIST`
`'TSL' STO`
	Puts the sine curve graphics object on the stack, then builds a list containing it and the nine graphics objects created in the loop. Stores the list in *TSL*.

`»`	
ENTER ' SETTS STO	Stores the program in *SETTS*.

Checksum: # 28102d
Bytes: 138.5

SETTS is demonstrated in the program *TSA*.

TSA (Animating Taylor's Polynomials)

TSA displays in succession each graphics object created in *SETTS*.

Techniques used in TSA

- **Passing a global variable.** Because *SETTS* takes several minutes to execute, *TSA* does not call *SETTS*. Instead, you must first execute *SETTS* to create the global variable *TSL* containing the list of graphics objects. *TSA* simply executes that global variable to put the list on the stack.

- **FOR ... NEXT (definite loop).** *TSA* executes a definite loop to display in succession each graphics object from the list.

TSA program listing

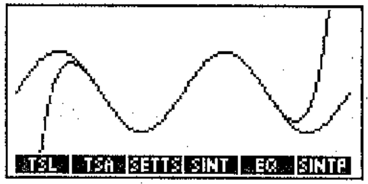
Program:	**Comments:**
`«`	
`TSL OBJ→`	Puts the list *TSL* on the stack and converts it to 10 graphics objects and the list count.
`{ { #0 #0 } .5 0 } +`	Set up the parameters for ANIMATE.
`ANIMATE`	Displays the graphics in succession.
`11 DROPN`	Removes the graphics objects and list count from the stack.
`»`	
ENTER ' TSA STO	Stores the program in *TSA*.

Checksum: # 59350d
Bytes: 96.5

Example: Execute *SETTS* and *TSA* to build and display in succession a series of Taylor's polynomial approximations of the sine function.

Set Radians mode and execute *SETTS* to build the list of graphics objects. (*SETTS* takes several minutes to execute.) Then execute *TSA* to display each plot in succession. The display shows *TSA* in progress.

↰ RAD (if necessary)
VAR SETTS
TSA



Press CANCEL to stop the animation. Press ↰ RAD to restore Degrees mode.

Programmatic Use of Statistics and Plotting

This section describes a program *PIE* you can use to draw pie charts. *PIE* prompts for single variable data, stores that data in the statistics matrix *ΣDAT*, then draws a labeled pie chart that shows each data point as a percentage of the total.

Techniques used in PIE

- **Programmatic use of PLOT commands.** *PIE* executes XRNG and YRNG to define x- and y-axis display ranges in user units, and executes ARC and LINE to draw the circle and individual slices.
- **Programmatic use of matrices and statistics commands.**
- **Manipulating graphics objects.** *PIE* recalls *PICT* to the stack and executes GOR to merge the label for each slice with the plot.
- **FOR ... NEXT (definite loop).** Each slice is calculated, drawn, and labeled in a definite loop.
- **CASE ... END structure.** To avoid overwriting the circle, each label is offset from the midpoint of the arc of the slice. The offset for each label depends on the position of the slice in the circle. The CASE ... END structure assigns an offset to the label based on the position of the slice.
- **Preserving calculator flag status.** Before specifying Radians mode, *PIE* saves the current flag status in a local variable, then restores that status at the end of the program.
- **Nested local variable structures.** At different parts of the process, intermediate results are saved in local variables for convenient recall as needed.
- **Temporary menu for data input.**

PIE program listing

Program:

Comments:

```
«
 RCLF → flags
```

Recalls the current flag status and stores it in variable *flags*.

```
 «
  RAD
```

Sets Radians mode.

```
  {{ "SLICE" Σ+ }
   { }
   { "CLEAR" CLΣ }
   { } { }
   { "DRAW" CONT }}
```

Defines the input menu: Key 1 executes Σ+ to store each data point in *ΣDAT*, key 3 clears *ΣDAT*, and key 6 continues program execution after data entry.

```
  TMENU
```

Displays the temporary menu.

```
  "Key values into
  SLICE,■DRAW
  restarts program."
  PROMPT
```

Prompts for inputs.

■ represents the newline character (⮫)(↵) after you enter the program on the stack.

```
  ERASE 1 131 XRNG
  1 64 YRNG CLLCD
```

Erases the current *PICT* and sets plot parameters.

```
  "Please wait...■
  Drawing Pie Chart"
  1 DISP
```

Displays "drawing" message.

```
  (66,32) 20 0 6.28
  ARC
```

Draws the circle.

```
  PICT RCL →LCD
```

Displays the empty circle.

```
  RCLΣ TOT /
```

Recalls the statistics data matrix, computes totals, and calculates the proportions.

```
  DUP 100 *
```

Converts the proportions to percentages.

```
  → prcnts
```

Stores the percentage matrix in *prcnts*.

```
  «
   2 π →NUM * *
   0
```

Multiplies the proportion matrix by 2π, and enters the initial angle (0).

Program:

Comments:

```
   → prop angle
```

Stores the angle matrix in *prop* and angle in *angle*.

```
   «
    prop SIZE OBJ→
    DROP SWAP
```

Sets up 1 to *m* as loop counter range.

```
    FOR n
```

Begins loop-clause.

```
     (66,32) prop n GET
     'angle' STO+
```

Puts the center of the circle on the stack, then gets the *n*th value from the proportion matrix and adds it to *angle*.

```
     angle COS angle SIN
     R→C 20 * OVER +
     LINE
```

Computes the endpoint and draws the line for the *n*th slice.

```
     PICT RCL
```

Recalls *PICT* to the stack.

```
     angle prop n GET
     2 / - DUP DUP
     COS SWAP SIN R→C
     26 * (66,32) +
```

For labeling the slice, computes the midpoint of the arc of the slice.

```
     SWAP
     CASE
```

Starts the CASE structure to test *angle* and determine the offset value for the label.

```
       DUP 1.5 ≤
      THEN
       DROP
      END
```

From 0 to 1.5 radians, doesn't offset the label.

```
       DUP 4.4 ≤
      THEN
       DROP 15 -
      END
```

From 1.5 to 4.4 radians, offsets the label 15 user units left.

```
       5 <
      THEN
       (3,2) +
      END
     END
```

From 4.4 to 5 radians, offsets the label 3 units right and 2 units up.

Ends the CASE structure.

Program:	Comments:
`prcnts n GET`	Gets the *n*th value from the percentage matrix, rounds it to one decimal place, and converts it to a string with "%" appended.
`i RND`	
`→STR "%" +`	
`i →GROB`	Converts the string to a graphics object.
`GOR DUP PICT STO`	Adds the label to the plot and stores the new plot.
`→LCD`	Displays the updated plot.
`NEXT`	Ends the loop structure.
`{ } PVIEW`	Displays the finished plot.
`»`	
`»`	
`flags STOF`	Restores the original flag status.
`» 0 MENU`	Restores the previous menu. (You must first press CANCEL to clear the plot.)
`»`	
ENTER ' PIE STO	Stores the program in *PIE*.

Checksum: # 1177d
Bytes: 765

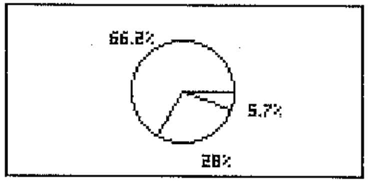
Example: The inventory at Fruit of the Vroom, a drive-in fruit stand, includes 983 oranges, 416 apples, and 85 bananas. Draw a pie chart to show each fruit's percentage of total inventory.

VAR PIE

```
Key values into SLICE,
DRAW restarts program.
4:
3:
2:
1:
```

Clear the current statistics data. (The prompt is removed from the display.) Key in the new data and draw the pie chart.

CLEAR
983 SLICE
416 SLICE
85 SLICE
DRAW



Press CANCEL to return to the stack display.

Trace Mode

This section contains two programs, *αENTER* and *βENTER*, which together provide "trace mode" for the HP 48 using an external printer. To turn on "trace mode," set flag −63 and activate User mode. To turn off "trace mode," clear flag −63 or turn off User mode.

Techniques used in αENTER and βENTER

- **Vectored ENTER.** Setting flag −63 and activating User mode turns on vectored ENTER. When vectored ENTER is turned on and variable *αENTER* exists, the command-line text is put on the stack as a string and *αENTER* is evaluated. Then, if variable *βENTER* exists, the command that triggered the command-line processing is put on the stack as a string and *βENTER* is evaluated.

αENTER program listing

Program:	Comments:
`«`	
`PR1`	Prints the command line text, then converts the string to an object and evaluates it.
`OBJ→`	
`»`	
ENTER ' αENTER STO	Stores the program in *αENTER*. (Press α ↱ A to type α. You *must* use this name.)

Checksum: # 51789d
Bytes: 25.5

βENTER program listing

Program:

```
«
 PR1 DROP
 PRSTC
»
```

Comments:

Prints the command that caused the processing, then drops it and prints the stack in compact form.

[ENTER] ['] βENTER [STO]

Stores the program in *βENTER*. (Press [α] [↱]B to type β. You *must* use this name.)

Checksum: # 37631d
Bytes: 28

Inverse-Function Solver

This section describes the program *ROOTR*, which finds the value of x at which $f(x) = y$. You supply the variable name for the program that calculates $f(x)$, the value of y, and a guess for x (in case there are multiple solutions).

Level 3	Level 2	Level 1	→	Level 1
'*function name*'	y	x_{guess}	→	x

Techniques used in ROOTR

- **Programmatic use of root-finder.** *ROOTR* executes ROOT to find the desired x-value.
- **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.

ROOTR program listing

Program:

```
«
 → fname yvalue xguess
 «
  xguess 'XTEMP' STO

   « XTEMP fname
   yvalue - »
   'XTEMP'
   xguess
   ROOT
 »
 'XTEMP' PURGE
»
```

Comments:

Creates local variables.
Begins the defining procedure.
Creates variable *XTEMP* (to be solved for).
Enters program that evaluates $f(x) - y$.
Enters name of unknown variable.
Enters guess for *XTEMP*.
Solves program for *XTEMP*.
Ends the defining procedure.
Purges the temporary variable.

[ENTER] ['] ROOTR [STO]

Stores the program in *ROOTR*.

Checksum: # 13007d
Bytes: 163

Example: Assume you often work with the expression $3.7x^3 + 4.5x^2 + 3.9x + 5$ and have created the program *X→FX* to calculate the value:

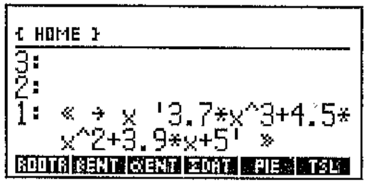
```
« → x '3.7*x^3+4.5*x^2+3.9*x+5' »
```

You can use *ROOTR* to calculate the *inverse* function.

Example: Find the value of x for which *X→FX* equals 599.5. Use a guess in the vicinity of 1.

Start by keying in *X→FX*:

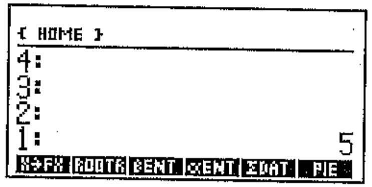
[↰] [« »] [↱] [→] x [SPC] ['] 3.7
[×] x [y^x] 3 [+] 4.5 [×] x [y^x] 2
[+] 3.9 [×] x [+] 5 [ENTER]



```
{ HOME }
3:
2:
1: « → x '3.7*x^3+4.5*
    x^2+3.9*x+5' »
```

Store the program in *X→FX*, then enter the program name, the *y*-value 599.5, and the guess 1, and execute *ROOTR*:

' X→FX STO
' VAR X→FX ENTER
599.5 ENTER 1 ROOTR



```
{ HOME }
4:
3:
2:
1:                    5
```

Animating a Graphical Image

Program *WALK* shows a small person walking across the display. It animates this custom graphical image by incrementing the image position in a loop structure.

Techniques used in WALK

- **Custom graphical image.** (Note that the programmer compiles the full information content of the graphical image before writing the program by building the image *interactively* in the Graphics environment and then returning it to the command line.)
- **FOR ... STEP (definite loop).** *WALK* uses this loop to animate the graphical image. The ending value for the loop is MAXR. Since the counter value cannot exceed MAXR, the loop executes indefinitely.

WALK program listing

Program:	**Comments:**
`«`	

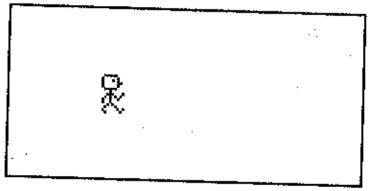
`GROB 9 15 E300`
`140015001C001400E300`
`8000C110AA0094009000`
`4100220014102800`
	Puts the graphical image of the walker in the command line. (Note that the hexadecimal portion of the graphics object is a continuous integer `E300` ... `2800`. The linebreaks do *not* represent spaces.)

`→ walk`	Creates local variable *walk* containing the graphics object.
`«`	
`ERASE { # 0d # 0d }` `PVIEW`	Clears *PICT*, then displays it.
`{ # 0d # 25d }` `PICT OVER walk GXOR`	Puts the first position on the stack and turns on the first image. This readies the stack and *PICT* for the loop.
`5 MAXR FOR i`	Starts the loop to generate horizontal coordinates indefinitely.
`i 131 MOD R→B`	Computes the horizontal coordinate for the next image.
`# 25d 2 →LIST`	Specifies a fixed vertical coordinate. Puts the two coordinates in a list.
`PICT OVER walk GXOR`	Displays the new image, leaving its coordinates on the stack.
`PICT ROT walk GXOR`	Turns off the old image, removing its coordinates from the stack.
`5 STEP`	Increments the horizontal coordinate by 5.
`»`	
`»`	
ENTER ' WALK STO	Stores the program in *WALK*.

Checksum: # 18146d
Bytes: 240.5

Example: Send the small person out for a walk.

(VAR) WALK



Press (CANCEL) when you think the walker's tired.

3

Command Reference

This chapter contains an alphabetical listing of the programmable commands and functions available on the HP 48. The listings include the following information:

- a brief definition of what the command or function does
- a stack diagram showing the arguments it requires (if any)
- the keys to press to gain access to it
- any flags that may affect how it works
- additional information about how it works and how to use it
- an example of its use
- related commands or functions

The next few pages explain how to read the stack diagrams in the command reference, how commands are alphabetized, and the meaning of the command classifications at the upper right corner of each stack diagram.

How to Read Stack Diagrams

Each entry in the command reference includes a *stack diagram*. This is a table showing the *arguments* that the command, function, or analytic function takes from the stack and the *results* that it returns to the stack. The "→" character in the table separates the arguments from the results. The stack diagram for a command may contain more than one "argument → result" line, reflecting all possible combinations of arguments and results for that command.