Chapter 21
Programming in User RPL language

User RPL language is the programming language most commonly used to program the calculator. The program components can be put together in the line editor by including them between program containers « » in the appropriate order. Because there is more experience among calculator users in programming in the RPN mode, most of the examples in this Chapter will be presented in the RPN mode. Also, to facilitate entering programming commands, we suggest you set system flag 117 to SOFT menus. The programs work equally well in ALG mode once they have been debugged and tested in RPN mode. If you prefer to work in the ALG mode, simply learn how to do the programming in RPN and then reset the operating mode to ALG to run the programs. For a simple example of User RPL programming in ALG mode, refer to the last page in this chapter.

An example of programming

Throughout the previous Chapters in this guide we have presented a number of programs that can be used for a variety of applications (e.g., programs CRMC and CRMT, used to create a matrix out of a number of lists, were presented in Chapter 10). In this section we present a simple program to introduce concepts related to programming the calculator. The program we will write will be used to define the function *f(x) = sinh(x)/(1+x*^{2}*)*, which accepts lists as argument (i.e., x can be a list of numbers, as described in Chapter 8). In Chapter 8 we indicated that the plus sign, , acts as a concatenation operator for lists and not to produce a term-by-term sum. Instead, you need to use the ADD operator to achieve a term-by-term summation of lists. Thus, to define the function shown above we will use the following program:

```
        «'x' STO x SINH 1 x SQ ADD / 'x' PURGE »
```

To key in the program follow these instructions:

Keystroke sequence:	Produces:	Interpreted as:
↱ « »	`«`	Start an RPL program
['] ALPHA ↰ X ▶ STO▶	`'x' STO`	Store level 1 into variable x
ALPHA ↰ X	`x`	Place x in level 1
↰ MTH HYP SINH	`SINH`	Calculate sinh of level 1
1 SPC ALPHA ↰ X ↰ x^2	`1 x SQ`	Enter 1 and calculate x^2

↰ MTH LIST ADD	ADD	Calculate $(1+x^2)$,
÷	/	then divide
['] ALPHA ↰ X ▶	`'x'`	
↰ PRG MEM DIR PURGE	`PURGE`	Purge variable x
ENTER		Program in level 1

To save the program use: ['] ALPHA ↰ G STO▶

Press VAR to recover your variable menu, and evaluate g(3.5) by entering the value of the argument in level 1 (3 . 5 ENTER) and then pressing g. The result is 1.2485…, i.e., g(3.5) = 1.2485. Try also obtaining g({1 2 3}), by entering the list in level 1 of the display: ↰ {} 1 SPC 2 SPC 3 ENTER and pressing g. The result now is {SINH(1)/2 SINH(2)/5 SINH(3)/10}, if your CAS is set to `EXACT` mode. If your CAS is set to APPROXIMATE mode, the result will be {0.5876.. 0.7253… 1.0017…}.

Global and local variables and subprograms

The program g, defined above, can be displayed as

```
« 'x' STO x SINH 1 x SQ ADD / 'x' PURGE »
```

by using ↱ g.

Notice that the program uses the variable name x to store the value placed in level 1 of stack through the programming steps `'x' STO`. The variable x, while the program is executing, is stored in your variable menu as any other variable you had previously stored. After calculating the function, the program purges (erases) the variable x so it will not show in your variable menu after finishing evaluating the program. If we were not to purge the variable x within the program its value would be available to us after program execution. For that reason, the variable x, as used in this program, is referred to as *a global variable*. One implication of the use of x as a global variable is that, if we had a previously defined a variable with the name x, its value would be replaced by the value that the program uses and then completely removed from your variable menu after program execution.

From the point of view of programming, therefore, a *global variable* is a variable that is accessible to the user after program execution. It is possible to use a local variable within the program that is only defined for that program and will not be available for use after program execution. The previous program could be modified to read:

```
« → x « x SINH 1 x SQ ADD / » »
```

The arrow symbol (→) is obtained by combining the right-shift key ↱ with the 0 key, i.e., ↱ → . Also, notice that there is an additional set of programming symbols (« ») indicating the existence of a _sub-program_, namely « `x SINH 1 x SQ ADD /` », within the main program. The main program starts with the combination → `x`, which represents assigning the value in level 1 of stack to a _local variable_ x. Then, programming flow continues within the sub-program by placing *x* in the stack, evaluating *SINH(x),* placing *1* in the stack, placing *x* in the stack, squaring *x*, adding *1* to *x*, and dividing stack level 2 (*SINH(x)*) by stack level 1 (*1+x*^{2}). The program control is then passed back to the main program, but there are no more commands between the first set of closing programming symbols (») and the second one, therefore, the program terminates. The last value in the stack, i.e., *SINH(x)/* (*1+x*^{2}), is returned as the program output.

The variable x in the last version of the program never occupies a place among the variables in your variable menu. It is operated upon within the calculator memory without affecting any similarly named variable in your variable menu. For that reason, the variable x in this case is referred to as a variable local to the program, i.e., a _local variable_.

Note: To modify program [illegible], place the program name in the stack (' [illegible] ENTER), then use ↰ ▼ . Use the arrow keys (◀ ▶ ▲ ▼) to move about the program. Use the backspace/delete key, ◀, to delete any unwanted characters. To add program containers (i.e., « »), use ↱ « », since these symbols come in pairs you will have to enter them at the start and end of the sub-program and delete one of its components with the delete key ◀ to produce the required program, namely:

```
« → x « x SINH 1 x SQ ADD / »  ».
```
When done editing the program press ENTER . The modified program is stored back into variable [illegible].

Global Variable Scope

Any variable that you define in the HOME directory or any other directory or sub-directory will be considered a *global variable* from the point of view of program development. However, the *scope* of such variable, i.e., *the location in the directory tree where the variable is accessible*, will depend on the location of the variable within the tree (see Chapter 2).

The rule to determine a variable's scope is the following*: a global variable is accessible to the directory where it is defined and to any sub-directory attached to that directory, unless a variable with the same name exists in the sub-directory under consideration*. Consequences of this rule are the following:

- A global variable defined in the HOME directory will be accessible from any directory within HOME, unless redefined within a directory or sub-directory.
- If you re-define the variable within a directory or sub-directory this definition takes precedence over any other definition in directories above the current one.
- When running a program that references a given global variable, the program will use the value of the global variable in the directory from which the program is invoked. If no variable with that name exist in the invoking directory, the program will search the directories above the current one, up to the HOME directory, and use the value corresponding to the variable name under consideration in the closest directory above the current one.

A program defined in a given directory can be accessed from that directory or any of its sub-directories.

All these rule may sound confusing for a new calculator user. They all can be simplified to the following suggestion: *Create directories and sub-directories with meaningful names to organize your data, and make sure you have all the global variables you need within the proper sub-directory.*

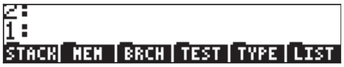
Local Variable Scope

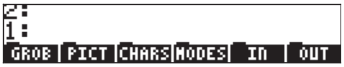
Local variables are active only within a program or sub-program. Therefore, their scope is limited to the program or sub-program where they're defined. An example of a local variable is the index in a FOR loop (described later in this chapter), for example `« → n x « 1 n FOR j x NEXT n →LIST » »`

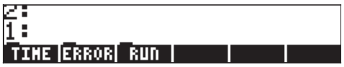
The PRG menu

In this section we present the contents of the PRG (programming) menu with the calculator's system flag 117 set to SOFT menus. With this flag setting sub-menus and commands in the PRG menu will be shown as soft menu labels. This facilitates entering the programming commands in the line editor when you are putting together a program.

To access the PRG menu use the keystroke combination ← PRG . Within the PRG menu we identify the following sub-menus (press NXT to move to the next collection of sub-menus in the PRG menu):







Here is a brief description of the contents of these sub-menus, and their sub-menus:

STACK:	Functions for manipulating elements of the RPN stack
MEM:	Functions related to memory manipulation
DIR:	Functions related to manipulating directories
ARITH:	Functions to manipulate indices stored in variables
BRCH:	Collection of sub-menus with program branching and loop functions
IF:	IF-THEN-ELSE-END construct for branching
CASE:	CASE-THEN-END construct for branching

START:	START-NEXT-STEP construct for branching
FOR:	FOR-NEXT-STEP construct for loops
DO:	DO-UNTIL-END construct for loops
WHILE:	WHILE-REPEAT-END construct for loops
TEST:	Comparison operators, logical operators, flag testing functions
TYPE:	Functions for converting object types, splitting objects, etc.
LIST:	Functions related to list manipulation
ELEM:	Functions for manipulating elements of a list
PROC:	Functions for applying procedures to lists
GROB:	Functions for the manipulation of graphic objects
PICT:	Functions for drawing pictures in the graphics screen
CHARS:	Functions for character string manipulation
MODES:	Functions for modifying calculator modes
FMT:	To change number formats, comma format
ANGLE:	To change angle measure and coordinate systems
FLAG:	To set and un-set flags and check their status
KEYS:	To define and activate user-defined keys (Chapter 20)
MENU:	To define and activate custom menus (Chapter 20)
MISC:	Miscellaneous mode changes (beep, clock, etc.)
IN:	Functions for program input
OUT:	Functions for program output
TIME:	Time-related functions
ALRM:	Alarm manipulation
ERROR:	Functions for error handling
IFERR:	IFERR-THEN-ELSE-END construct for error handling
RUN:	Functions for running and debugging programs

Navigating through RPN sub-menus

Start with the keystroke combination ← PRG , then press the appropriate soft-menu key (e.g., MEM). If you want to access a sub-menu within this sub-menu (e.g., DIR within the MEM sub-menu), press the corresponding key. To move up in a sub-menu, press the NXT key until you find either the reference to the upper sub-menu (e.g., MEM within the DIR sub-menu) or to the PRG menu (i.e., PRG).

Functions listed by sub-menu

The following is a listing of the functions within the PRG sub-menus listed by sub-menu.

STACK	**MEM/DIR**	**BRCH/IF**	**BRCH/WHILE**	**TYPE**
DUP	PURGE	IF	WHILE	OBJ→
SWAP	RCL	THEN	REPEAT	→ARRY
DROP	STO	ELSE	END	→LIST
OVER	PATH	END		→STR
ROT	CRDIR		**TEST**	→TAG
UNROT	PGDIR	**BRCH/CASE**	==	→UNIT
ROLL	VARS	CASE	≠	C→R
ROLLD	TVARS	THEN	<	R→C
PICK	ORDER	END	>	NUM
UNPICK			≤	CHR
PICK3	**MEM/ARITH**	**BRCH/START**	≥	DTAG
DEPTH	STO+	START	AND	EQ→
DUP2	STO-	NEXT	OR	TYPE
DUPN	STOx	STEP	XOR	VTYPE
DROP2	STO/		NOT	
DROPN	INCR	**BRCH/FOR**	SAME	**LIST**
DUPDU	DECR	FOR	TYPE	OBJ→
NIP	SINV	NEXT	SF	→LIST
NDUPN	SNEG	STEP	CF	SUB
	SCONJ		FS?	REPL
MEM		**BRCH/DO**	FC?	
PURGE	**BRCH**	DO	FS?C	
MEM	IFT	UNTIL	FC?C	
BYTES	IFTE	END	LININ	
NEWOB				
ARCHI				
RESTO				

LIST/ELEM	**GROB**	**CHARS**	**MODES/FLAG**	**MODES/MISC**
GET	→GROB	SUB	SF	BEEP
GETI	BLANK	REPL	CF	CLK
PUT	GOR	POS	FS?	SYM
PUTI	GXOR	SIZE	FC?	STK
SIZE	SUB	NUM	FS?C	ARG
POS	REPL	CHR	FS?C	CMD
HEAD	→LCD	OBJ→	FC?C	INFO
TAIL	LCD→	→STR	STOF	
	SIZE	HEAD	RCLF	**IN**
LIST/PROC	ANIMATE	TAIL	RESET	INFORM
DOLIST		SREPL		NOVAL
DOSUB	**PICT**		**MODES/KEYS**	CHOOSE
NSUB	PICT	**MODES/FMT**	ASN	INPUT
ENDSUB	PDIM	STD	STOKEYS	KEY
STREAM	LINE	FIX	RECLKEYS	WAIT
REVLIST	TLINE	SCI	DELKEYS	PROMPT
SORT	BOX	ENG		
SEQ	ARC	FM,	**MODES/MENU**	**OUT**
	PIXON	ML	MENU	PVIEW
	PIXOF		CST	TEXT
	PIX?	**MODES/ANGLE**	TMENU	CLLCD
	PVIEW	DEG	RCLMENU	DISP
	PX→C	RAD		FREEZE
	C→PX	GRAD		MSGBOX
		RECT		BEEP
		CYLIN		
		SPHERE		

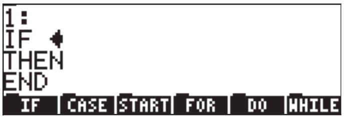
TIME	**ERROR**	**RUN**
DATE	DOERR	DBUG
→DATE	ERRN	SST
TIME	ERRM	SST↓
→TIME	ERR0	NEXT
TICKS	LASTARG	HALT
		KILL
TIME/ALRM	**ERROR/IFERR**	OFF
ACK	IFERR	
ACKALARM	THEN	
STOALARM	ELSE	
RCLALARM	END	
DELALARM		
FINDALARM		

Shortcuts in the PRG menu

Many of the functions listed above for the PRG menu are readily available through other means:

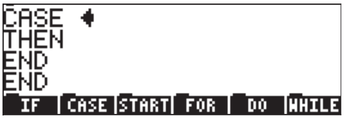
- Comparison operators (≠, ≤, <, ≥, >) are available in the keyboard.
- Many functions and settings in the MODES sub-menu can be activated by using the input functions provided by the MODE key.
- Functions from the TIME sub-menu can be accessed through the keystroke combination ↱ TIME .
- Functions STO and RCL (in MEM/DIR sub-menu) are available in the keyboard through the keys STO▸ and ↰ RCL .
- Functions RCL and PURGE (in MEM/DIR sub-menu) are available through the TOOL menu (TOOL).
- Within the BRCH sub-menu, pressing the left-shift key (↰) or the right-shift key (↱) before pressing any of the sub-menu keys, will create constructs related to the sub-menu key chosen. This only works with the calculator in RPN mode. Examples are shown below:

IF



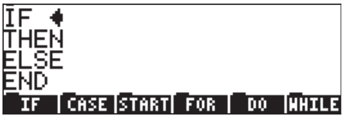
```
1:
IF ◀
THEN
END
```

CASE



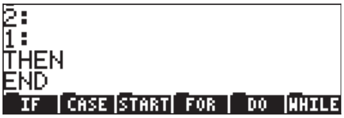
```
CASE ◀
THEN
END
END
```

IF



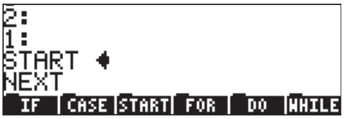
```
IF ◀
THEN
ELSE
END
```

CASE



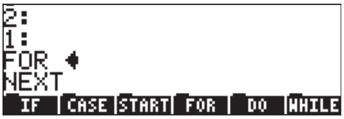
```
2:
1:
THEN
END
```

START



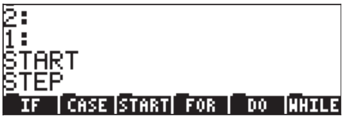
```
2:
1:
START ◀
NEXT
```

FOR



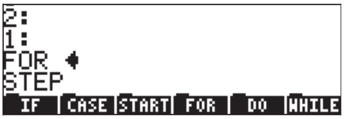
```
2:
1:
FOR ◀
NEXT
```

START



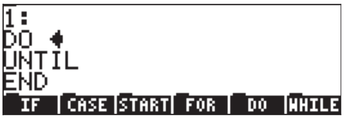
```
2:
1:
START
STEP
```

FOR



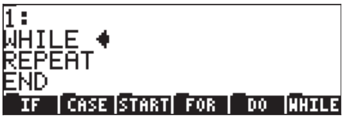
```
2:
1:
FOR ◀
STEP
```

DO



```
1:
DO ◀
UNTIL
END
```

WHILE



```
1:
WHILE ◀
REPEAT
END
```

Notice that the insert prompt (◀) is available after the key word for each construct so you can start typing at the right location.

Keystroke sequence for commonly used commands

The following are keystroke sequences to access commonly used commands for numerical programming within the PRG menu. The commands are first listed by menu:

STACK

DUP	← PRG STACK DUP
SWAP	← PRG STACK SWAP
DROP	← PRG STACK DROP

MEM DIR

PURGE	← PRG MEM DIR PURGE
ORDER	← PRG MEM DIR ORDER

BRCH IF

IF	← PRG BRCH IF IF
THEN	← PRG BRCH IF THEN
ELSE	← PRG BRCH IF ELSE
END	← PRG BRCH IF END

BRCH CASE

CASE	← PRG BRCH CASE CASE
THEN	← PRG BRCH CASE THEN
END	← PRG BRCH CASE END

BRCH START

START	← PRG BRCH START START
NEXT	← PRG BRCH START NEXT
STEP	← PRG BRCH START STEP

BRCH FOR

FOR	← PRG BRCH FOR FOR
NEXT	← PRG BRCH FOR NEXT
STEP	← PRG BRCH FOR STEP

BRCH DO

DO	← PRG BRCH DO DO
UNTIL	← PRG BRCH DO UNTIL
END	← PRG BRCH DO END

BRCH WHILE

WHILE	↰ PRG BRCH WHILE WHILE
REPEAT	↰ PRG BRCH WHILE REPEA
END	↰ PRG BRCH WHILE END

TEST

==	↰ PRG TEST ≠
AND	↰ PRG TEST NXT AND
OR	↰ PRG TEST NXT OR
XOR	↰ PRG TEST NXT XOR
NOT	↰ PRG TEST NXT NOT
SAME	↰ PRG TEST NXT SAME
SF	↰ PRG TEST NXT NXT SF
CF	↰ PRG TEST NXT NXT CF
FS?	↰ PRG TEST NXT NXT FS?
FC?	↰ PRG TEST NXT NXT FC?
FS?C	↰ PRG TEST NXT NXT FS?C
FC?C	↰ PRG TEST NXT NXT FC?C

TYPE

OBJ→	↰ PRG TYPE OBJ→
→ARRY	↰ PRG TYPE →ARRY
→LIST	↰ PRG TYPE →LIST
→STR	↰ PRG TYPE →STR
→TAG	↰ PRG TYPE →TAG
NUM	↰ PRG TYPE NXT NUM
CHR	↰ PRG TYPE NXT CHR
TYPE	↰ PRG TYPE NXT TYPE

LIST ELEM

GET	↰ PRG LIST ELEM GET
GETI	↰ PRG LIST ELEM GETI
PUT	↰ PRG LIST ELEM PUT
PUTI	↰ PRG LIST ELEM PUTI
SIZE	↰ PRG LIST ELEM SIZE
HEAD	↰ PRG LIST ELEM NXT HEAD
TAIL	↰ PRG LIST ELEM NXT TAIL

LIST PROC

REVLIST	← PRG LIST PROC REVLI
SORT	← PRG LIST PROC NXT SORT
SEQ	← PRG LIST PROC NXT SEQ

MODES ANGLE

DEG	← PRG NXT MODES ANGLE DEG
RAD	← PRG NXT MODES ANGLE RAD

MODES MENU

CST	← PRG NXT MODES MENU CST
MENU	← PRG NXT MODES MENU MENU
BEEP	← PRG NXT MODES MISC BEEP

IN

INFORM	← PRG NXT IN INFOR
INPUT	← PRG NXT IN INPUT
MSGBOX	← PRG NXT OUT MSGBO
PVIEW	← PRG NXT OUT PVIEW

RUN

DBUG	← PRG NXT NXT RUN DBG
SST	← PRG NXT NXT RUN SST
SST↓	← PRG NXT NXT RUN SST↓
HALT	← PRG NXT NXT RUN HALT
KILL	← PRG NXT NXT RUN KILL

Programs for generating lists of numbers

Please notice that the functions in the PRG menu are not the only functions that can be used in programming. As a matter of fact, almost all functions in the calculator can be included in a program. Thus, you can use, for example, functions from the MTH menu. Specifically, you can use functions for list operations such as SORT, ΣLIST, etc., available through the MTH/LIST menu.

As additional programming exercises, and to try the keystroke sequences listed above, we present herein three programs for creating or manipulating lists. The program names and listings are as follows:

LISC:

```
« → n x « 1 n FOR j x NEXT n →LIST » »
```

CRLST:

```
« → st en df « st en FOR  j j df STEP en st - df / FLOOR 1 +
→LIST » »
```

CLIST:

```
« REVLIST DUP DUP SIZE 'n' STO ΣLIST SWAP TAIL DUP SIZE 1 - 1
SWAP FOR j DUP ΣLIST SWAP TAIL NEXT 1 GET n →LIST REVLIST 'n'
PURGE »
```

The operation of these programs is as follows:

(1) *LISC*: creates a list of n elements all equals to a constant c.
Operation: enter n, enter c, press LISC
Example: 5 ENTER 6.5 ENTER LISC creates the list: {6.5 6.5 6.5 6.5 6.5}

(2) *CRLST*: creates a list of numbers from n_1 to n_2 with increment Δn, i.e., $\{n_1, n_1+\Delta n, n1+2\cdot\Delta n, \ldots n_1+N\cdot\Delta n\}$, where $N=\text{floor}((n_2-n_1)/\Delta n)+1$.
Operation: enter n_1, enter n_2, enter Δn, press CRLST
Example: .5 ENTER 3.5 ENTER .5 ENTER CRLST produces: {0.5 1 1.5 2 2.5 3 3.5}

(3) *CLIST*: creates a list with cumulative sums of the elements, i.e., if the original list is $\{x_1\ x_2\ x_3 \ldots x_N\}$, then CLIST creates the list:

$$\{x_1, x_1+x_2, x_1+x_2+x_3, \ldots, \sum_{i=1}^{N} x_i\}$$

Operation: place the original list in level 1, press CLIST.
Example: {1 2 3 4 5} ENTER CLIST produces {1 3 6 10 15}.

Examples of sequential programming

In general, a program is any sequence of calculator instructions enclosed between the program containers « and ». Subprograms can be included as part of a program. The examples presented previously in this guide (e.g., in Chapters 3 and 8) 6 can be classified basically into two types: (a) programs generated by defining a function; and, (b) programs that simulate a sequence of stack operations. These two types of programs are described next. The general form of these programs is input→process→output, therefore, we refer to them as <u>sequential programs</u>.

Programs generated by defining a function

These are programs generated by using function DEFINE (← DEF) with an argument of the form:

'function_name(x_1, x_2, …) = expression containing variables x_1, x_2, …'

The program is stored in a variable called `function_name`. When the program is recalled to the stack, by using → function_name. The program shows up as follows:

« → x_1, x_2, … 'expression containing variables x_1, x_2, …'».

To evaluate the function for a set of input variables x_1, x_2, …, in RPN mode, enter the variables into the stack in the appropriate order (i.e., x_1 first, followed by x_2, then x_3, etc.), and press the soft menu key labeled function_name. The calculator will return the value of the function function_name(x_1, x_2, …).

<u>Example</u>: *Manning's equation for wide rectangular channel.*
As an example, consider the following equation that calculates the unit discharge (discharge per unit width), q, in a wide rectangular open channel using Manning's equation:

$$q = \frac{C_u}{n} y_0^{5/3} \sqrt{S_0}$$

where C_u is a constant that depends on the system of units used [C_u = 1.0 for units of the International System (S.I.), and C_u = 1.486 for units of the English System (E.S.)], n is the Manning's resistance coefficient, which depends on the type of channel lining and other factors, y_0 is the flow depth, and S_0 is the channel bed slope given as a dimensionless fraction.

Note: Values of the Manning's coefficient, n, are available in tables as dimensionless numbers, typically between 0.001 to 0.5. The value of Cu is also used without dimensions. However, care should be taken to ensure that the value of y0 has the proper units, i.e., m in S.I. and ft in E.S. The result for q is returned in the proper units of the corresponding system in use, i.e., m^2/s in S.I. and ft^2/s in E.S. Manning's equation is, therefore, not *dimensionally consistent*.

Suppose that we want to create a function q(Cu, n, y0, S0) to calculate the unit discharge q for this case. Use the expression

'q(Cu,n,y0,S0)=Cu/n*y0^(5./3.)*√S0',

as the argument of function DEFINE. Notice that the exponent 5./3., in the equation, represents a ratio of real numbers due to the decimal points. Press VAR, if needed, to recover the variable list. At this point there will be a variable called q in your soft menu key labels. To see the contents of q, use ↱ q. The program generated by defining the function `q(Cu,n,y0,S0)` is shown as:

« → Cu n y0 S0 'Cu/n*y0^(5./3.)*√S0' ».

This is to be interpreted as "enter Cu, n, y0, S0, in that order, then calculate the expression between quotes." For example, to calculate q for Cu = 1.0, n = 0.012, y0 = 2 m, and S0 = 0.0001, use, in RPN mode:

1 ENTER 0.012 ENTER 2 ENTER 0.0001 ENTER q

The result is 2.6456684 (or, q = 2.6456684 m^2/s).

You can also separate the input data with spaces in a single stack line rather than using ENTER.

Programs that simulate a sequence of stack operations

In this case, the terms to be involved in the sequence of operations are assumed to be present in the stack. The program is typed in by first opening the program containers with ↱ « » . Next, the sequence of operations to be performed is entered. When all the operations have been typed in, press ENTER to complete the program. If this is to be a once-only program, you can at this point, press EVAL to execute the program using the input data available. If it is to be a permanent program, it needs to be stored in a variable name.

The best way to describe this type of programs is with an example:

<u>Example</u>: Velocity head for a rectangular channel.
Suppose that we want to calculate the velocity head, h_v, in a rectangular channel of width b, with a flow depth y, that carries a discharge Q. The specific energy is calculated as $h_v = Q^2/(2g(by)^2)$, where g is the acceleration of gravity ($g = 9.806\ m/s^2$ in S.I. units or $g = 32.2\ ft/s^2$ in E.S. units). If we were to calculate h_v for Q = 23 cfs (cubic feet per second = ft^3/s), b = 3 ft, and y = 2 ft, we would use: $h_v = 23^2/(2\cdot 32.2\cdot (3\cdot 2)^2)$. Using the RPN modethe calculator, interactively, we can calculate this quantity as:

2 ENTER 3 × ← x² 3 2 . 2 ×
2 × 2 3 ← x² ▶ ÷

Resulting in 0.228174, or h_v = 0.228174.

To put this calculation together as a program we need to have the input data (Q, g, b, y) in the stack in the order in which they will be used in the calculation. In terms of the variables Q, g, b, and y, the calculation just performed is written as (do not type the following):

y ENTER b × ← x² g × 2 × Q ← x² ▶ ÷

As you can see, y is used first, then we use b, g, and Q, in that order. Therefore, for the purpose of this calculation we need to enter the variables in the inverse order, i.e., (do not type the following):

Q ENTER g ENTER b ENTER y ENTER

For the specific values under consideration we use:

23 ENTER 32.2 ENTER 3 ENTER 2 ENTER

The program itself will contain only those keystrokes (or commands) that result from removing the input values from the interactive calculation shown earlier, i.e., removing Q, g, b, and y from (do not type the following):

y ENTER b × ← x^2 g × 2 × Q ← x^2 ▶ ÷

and keeping only the operations shown below (do not type the following):

ENTER × ← × 2 × ← x^2 ▶ ÷

Note: When entering the program do not use the keystroke ▶, instead use the keystroke sequence: ← PRG STACK SWAP.

Unlike the interactive use of the calculator performed earlier, we need to do some swapping of stack levels 1 and 2 within the program. To write the program, we use, therefore:

→ « »	Opens program symbols
×	Multiply y with b
← x^2	Square (b·y)
×	Multiply $(b·y)^2$ times g
2 ×	Enter a 2 and multiply it with g· $(b·y)^2$
← PRG STACK SWAP	Swap Q with 2·g· $(b·y)^2$
← x^2	Square Q
← PRG STACK SWAP	Swap 2·g· $(b·y)^2$ with Q^2
÷	Divide Q^2 by 2·g· $(b·y)^2$
ENTER	Enter the program

The resulting program looks like this:

```
« * SQ * 2 * SWAP SQ SWAP / »
```

Note: SQ is the function that results from the keystroke sequence ← x² .

Save the program into a variable called hv:

' ALPHA ← H ALPHA ← V STO▸

A new variable hv should be available in your soft key menu. (Press VAR to see your variable list.) The program left in the stack can be evaluated by using function EVAL. The result should be 0.228174…, as before. Also, the program is available for future use in variable hv. For example, for Q = 0.5 m^3/s, g = 9.806 m/s^2, b = 1.5 m, and y = 0.5 m, use:

0.5 SPC 9.806 SPC 1.5 SPC 0.5 hv

Note: SPC is used here as an alternative to ENTER for input data entry.

The result now is 2.26618623518E-2, i.e., hv = $2.26618623518\times10^{-2}$ m.

Note: Since the equation programmed in hv is dimensionally consistent, we can use units in the input.

As mentioned earlier, the two types of programs presented in this section are *sequential programs,* in the sense that the program flow follows a single path, i.e., INPUT→ OPERATION →OUTPUT. Branching of the program flow is possible by using the commands in the menu ← PRG BRCH . More detail on program branching is presented below.

Interactive input in programs

In the sequential program examples shown in the previous section it is not always clear to the user the order in which the variables must be placed in the stack before program execution. For the case of the program q, written as

```
« →Cu n y0 S0 'Cu/n*y0^(5/3)*√S0' »,
```

it is always possible to recall the program definition into the stack (→ q) to see the order in which the variables must be entered, namely, `→Cu n y0 S0`. However, for the case of the program hv, its definition

« * SQ * 2 * SWAP SQ SWAP / »

does not provide a clue of the order in which the data must be entered, unless, of course, you are extremely experienced with RPN and the User RPL language.

One way to check the result of the program as a formula is to enter symbolic variables, instead of numeric results, in the stack, and let the program operate on those variables. For this approach to be effective the calculator's CAS (Calculator Algebraic System) must be set to `symbolic` and `exact` modes. This is accomplished by using MODE CAS, and ensuring that the check marks in the options `_Numeric` and `_Approx` are removed. Press OK OK to return to normal calculator display. Press VAR to display your variables menu.

We will use this latter approach to check what formula results from using the program hv as follows: We know that there are four inputs to the program, thus, we use the symbolic variables S4, S3, S2, and S1 to indicate the stack levels at input:

ALPHA S 4 ENTER ALPHA S 3 ENTER ALPHA S 2 ENTER ALPHA S 1 ENTER

Next, press hv. The resulting formula may look like this

`'SQ(S4)/(S3*SQ(S2*S1)*2)',`

if your display is not set to textbook style, or like this,

$$\frac{SQ(S4)}{S3\cdot SQ(S2\cdot S1)\cdot 2}$$

if textbook style is selected. Since we know that the function SQ( ) stands for x^2, we interpret the latter result as

$$\frac{S4^2}{2\cdot S3\cdot(S2\cdot S1)^2},$$

which indicates the position of the different stack input levels in the formula. By comparing this result with the original formula that we programmed, i.e.,

$$h_v = \frac{Q^2}{2g(by)^2},$$

we find that we must enter y in stack level 1 (S1), b in stack level 2 (S2), g in stack level 3 (S3), and Q in stack level 4 (S4).

Prompt with an input string

These two approaches for identifying the order of the input data are not very efficient. You can, however, help the user identify the variables to be used by prompting him or her with the name of the variables. From the various methods provided by the User RPL language, the simplest is to use an input string and the function INPUT (← PRG NXT IN INPUT) to load your input data.

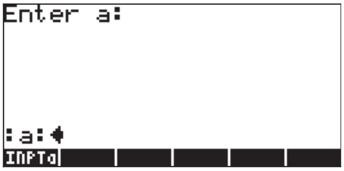
The following program prompts the user for the value of a variable a and places the input in stack level 1:

```
« "Enter a: " {"↵:a: " {2 0} V } INPUT OBJ→ »
```

This program includes the symbol :: (tag) and ↵(return), available through the keystroke combinations ← :: and → ↵, both associated with the . key. The tag symbol (::) is used to label strings for input and output. The return symbol (↵) is similar to a carriage return in a computer. The strings between quotes (" ") are typed directly from the alphanumeric keyboard.

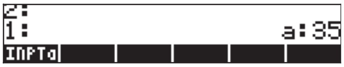
Save the program in a variable called INPTa (for INPuT a).

Try running the program by pressing the soft menu key labeled INPTa.


Enter a:

:a:

The result is a stack prompting the user for the value of a and placing the cursor right in front of the prompt :a: Enter a value for a, say 35, then press ENTER. The result is the input string `:a:35` in stack level 1.


```
2:
1:                    a:35
```

A function with an input string

If you were to use this piece of code to calculate the function, f(a) = 2*a^2+3, you could modify the program to read as follows:

```
« "Enter a:" {"↵:a:" {2 0} V }
INPUT OBJ→ → a « '2*a^2+3' » »
```

Save this new program under the name 'FUNCa' (FUNCtion of a):

Run the program by pressing FUNCa. When prompted to enter the value of a enter, for example, 2, and press ENTER. The result is simply the algebraic $2a^2+3$, which is an incorrect result. The calculator provides functions for debugging programs to identify logical errors during program execution as shown below.

Debugging the program

To figure out why it did not work we use the DBUG function in the calculator as follows:

' FUNCa ENTER	Copies program name to stack level 1
← PRG NXT NXT RUN DBUG	Starts debugger
SST↓	Step-by-step debugging, result: "Enter a:"
SST↓	Result: {" ↵ a:" {2 0} V}
SST↓	Result: user is prompted to enter value of a
2 ENTER	Enter a value of 2 for a. Result: "↵:a:2"
SST↓	Result: a:2

SST↓	Result: empty stack, executing →a
SST↓	Result: empty stack, entering subprogram «
SST↓	Result: ‘2*a^2+3’
SST↓	Result: ‘2*a^2+3’, leaving subprogram »
SST↓	Result: ‘2*a^2+3’, leaving main program»

Further pressing the SST↓ soft menu key produces no more output since we have gone through the entire program, step by step. This run through the debugger did not provide any information on why the program is not calculating the value of $2a^2+3$ for a = 2. To see what is the value of a in the sub-program, we need to run the debugger again and evaluate a within the sub-program. Try the following:

VAR	Recovers variables menu
' FUNCa ENTER	Copies program name to stack level 1
← PRG NXT NXT RUN DBG	Starts debugger
SST↓	Step-by-step debugging, result: “Enter a:”
SST↓	Result: {“ ↵a:” {2 0} V}
SST↓	Result: user is prompted to enter value of a
2 ENTER	Enter a value of 2 for a. Result: “↵a:2”
SST↓	Result: a:2
SST↓	Result: empty stack, executing →a
SST↓	Result: empty stack, entering subprogram «

At this point we are within the subprogram « ‘2*a^2+3’ » which uses the local variable a. To see the value of a use:

ALPHA ← A EVAL	This indeed shows that the local variable a = 2

Let’s kill the debugger at this point since we already know the result we will get. To kill the debugger press KILL. You receive an `<!> Interrupted` message acknowledging killing the debugger. Press ON to recover normal calculator display.

Note: In debugging mode, every time we press SST↓ the top left corner of the display shows the program step being executed. A soft key function called SST is also available under the RUN sub-menu within the PRG menu. This can be used to execute at once any sub-program called from within a main program. Examples of the application of SST will be shown later.

Fixing the program
The only possible explanation for the failure of the program to produce a numerical result seems to be the lack of the command →NUM after the algebraic expression '2*a^2+3'. Let's edit the program by adding the missing EVAL function. The program, after editing, should read as follows:

```
« "Enter a: " {"↵:a: " {2 0} V } INPUT
   OBJ→ → a « '2*a^2+3' →NUM » »
```

Store it again in variable FUNCa, and run the program again with a = 2. This time, the result is 11, i.e., $2*2^2+3 = 11$.

Input string for two or three input values

In this section we will create a sub-directory, within the directory HOME, to hold examples of input strings for one, two, and three input data values. These will be generic input strings that can be incorporated in any future program, taking care of changing the variable names according to the needs of each program.

Let's get started by creating a sub-directory called PTRICKS (Programming TRICKS) to hold programming tidbits that we can later borrow from to use in more complex programming exercises. To create the sub-directory, first make sure that you move to the HOME directory. Within the HOME directory, use the following keystrokes to create the sub-directory PTRICKS:

' ALPHA ALPHA P T R I C K S ENTER	Enter directory name 'PTRICKS'
← PRG MEM DIR CRDIR	Create directory
VAR	Recover variable listing

A program may have more than 3 input data values. When using input strings we want to limit the number of input data values to 5 at a time for the simple reason that, in general, we have visible only 7 stack levels. If we use stack level 7 to give a title to the input string, and leave stack level 6 empty to facilitate reading the display, we have only stack levels 1 through 5 to define input variables.

Input string program for two input values

The input string program for two input values, say a and b, looks as follows:

```
« "Enter a and b: " {"↵:a:↵:b: " {2 0} V } INPUT OBJ→ »
```

This program can be easily created by modifying the contents of INPTa. Store this program into variable INPT2.

Application: evaluating a function of two variables
Consider the ideal gas law, pV = nRT, where p = gas pressure (Pa), V = gas volume(m^3), n = number of moles (gmol), R = universal gas constant = 8.31451_J/(gmol*K), and T = absolute temperature (K).

We can define the pressure p as a function of two variables, V and T, as p(V,T) = nRT/V for a given mass of gas since n will also remain constant. Assume that n = 0.2 gmol, then the function to program is

$$p(V,T) = 8.31451 \cdot 0.2 \cdot \frac{T}{V} = (1.662902\_\frac{J}{K}) \cdot \frac{T}{V}$$

We can define the function by typing the following program

```
« →V T '(1.662902_J/K)*(T/V)' »
```

and storing it into variable p.

The next step is to add the input string that will prompt the user for the values of V and T. To create this input stream, modify the program in p to read:

```
« "Enter V and T: " {"↵ :V:↵ :T: " {2 0} V }
INPUT OBJ→ → V T '(1.662902_J/K)*(T/V)'  »
```

Store the new program back into variable p. Press p to run the program. Enter values of V = 0.01_m^3 and T = 300_K in the input string, then press

`ENTER`. The result is 49887.06_J/m^3. The units of J/m^3 are equivalent to Pascals (Pa), the preferred pressure unit in the S.I. system.

Note: because we deliberately included units in the function definition, the input values must have units attach to them in input to produce the proper result.

Input string program for three input values

The input string program for three input values, say a ,b, and c, looks as follows:

```
« “Enter a, b and c: “ {“↵ :a:↵ :b:↵ :c: “ {2 0} V } INPUT
                                    OBJ→ »
```

This program can be easily created by modifying the contents of INPT2 to make it look like shown immediately above. The resulting program can then be stored in a variable called INPT3. With this program we complete the collection of input string programs that will allow us to enter one, two, or three data values. Keep these programs as a reference and copy and modify them to fulfill the requirements of new programs you write.

Application: evaluating a function of three variables

Suppose that we want to program the ideal gas law including the number of moles, n, as an additional variable, i.e., we want to define the function

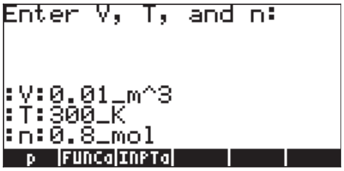
$$p(V,T,n) = (8.31451\_\frac{J}{K})\frac{n\cdot T}{V},$$

and modify it to include the three-variable input string. The procedure to put together this function is very similar to that used earlier in defining the function p(V,T). The resulting program will look like this:

```
« “Enter V, T, and n:“ {“↵ :V:↵ :T:↵ :n:“ {2 0} V } INPUT
OBJ→ →V T n  ‘(8.31451_J/(K*mol))*(n*T/V)‘»
```

Store this result back into the variable p.To run the program, press p.

Enter values of V = 0.01_m^3, T = 300_K, and n = 0.8_mol. Before pressing ENTER, the stack will look like this:



```
Enter V, T, and n:


:V:0.01_m^3
:T:300_K
:n:0.8_mol
```

Press ENTER to get the result 199548.24_J/m^3, or 199548.24_Pa = 199.55 kPa.

Input through input forms

Function INFORM (← PRG NXT IN INFOR.) can be used to create detailed input forms for a program. Function INFORM requires five arguments, in this order:

1. A title: a character string describing the input form
2. Field definitions: a list with one or more field definitions {s_1 s_2 … s_n}, where each field definition, s_i, can have one of two formats:
    a. A simple field label: a character string
    b. A list of specifications of the form {"label" "helpInfo" $type_0$ $type_1$ … $type_n$}. The "label" is a field label. The "helpInfo" is a character string describing the field label in detail, and the type specifications is a list of types of variables allowed for the field (see Chapter 24 for object types).
3. Field format information: a single number *col* or a list {*col tabs*}. In this specification, *col* is the number of columns in the input box, and *tabs* (optional) specifies the number of tab stops between the labels and the fields in the form. The list could be an empty list. Default values are *col* = 1 and *tabs* = 3.
4. List of reset values: a list containing the values to reset the different fields if the option RESET is selected while using the input form.
5. List of initial values: a list containing the initial values of the fields.

The lists in items 4 and 5 can be empty lists. Also, if no value is to be selected for these options you can use the NOVAL command (← PRG NXT IN NOVAL).

After function INFORM is activated you will get as a result either a zero, in case the CANCL option is entered, or a list with the values entered in the fields in the order specified and the number 1, i.e., in the RPN stack:

```
2:    {v₁ v₂ … vₙ}
1:              1
```

Thus, if the value in stack level 1 is zero, no input was performed, while it this value is 1, the input values are available in stack level 2.
<u>Example 1</u> - As an example, consider the following program, INFP1 (INput Form Program 1) to calculate the discharge Q in an open channel through Chezy's formula: $Q = C\cdot(R\cdot S)^{1/2}$, where C is the Chezy coefficient, a function of the channel surface's roughness (typical values 80-150), R is the hydraulic radius of the channel (a length), and S is the channel bed's slope (a dimensionless numbers, typically 0.01 to 0.000001). The following program defines an input form through function INFORM:

```
« " CHEZY'S EQN" { { "C:" "Chezy's coefficient" 0} { "R:" "Hydraulic radius" 0 } { "S:" "Channel bed slope" 0} } { } { 120 1 .0001} { 110 1.5 .00001 }  INFORM »
```

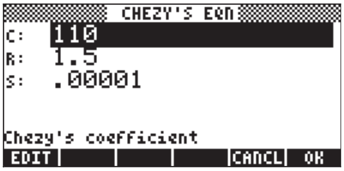
In the program we can identify the 5 components of the input as follows:

1. Title: `" CHEZY'S EQN"`
2. Field definitions: there are three of them, with labels "C:", "R:", "S:", info strings "Chezy coefficient", "Hydraulic radius", "Channel bed slope", and accepting only data type 0 (real numbers) for all of the three fields:

```
{ { "C:" "Chezy's coefficient" 0} { "R:" "Hydraulic radius" 0 } { "S:" "Channel bed slope" 0} }
```

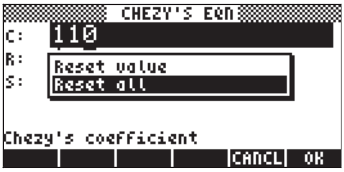
3. Field format information: { } (an empty list, thus, default values used)
4. List of reset values: `{ 120 1 .0001}`
5. List of initial values: `{ 110 1.5 .00001}`

Save the program into variable INFP1. Press INFP1 to run the program. The input form, with initial values loaded, is as follows:

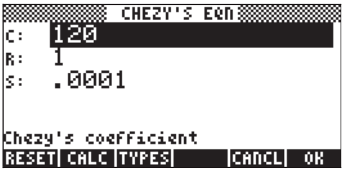


```
CHEZY'S EQN
C:  110
R:  1.5
S:  .00001

Chezy's coefficient
EDIT |    |    |    |CANCL| OK
```

To see the effect of resetting these values use NXT RESET (select *Reset all* to reset field values):

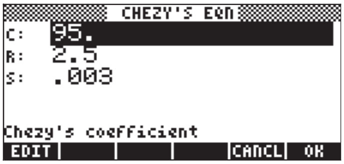


```
CHEZY'S EQN
C:  110
R:  Reset value
S:  Reset all

Chezy's coefficient
     |    |    |    |CANCL| OK
```



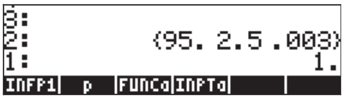
```
CHEZY'S EQN
C:  120
R:  1
S:  .0001

Chezy's coefficient
RESET| CALC |TYPES|    |CANCL| OK
```

Now, enter different values for the three fields, say, C = 95, R = 2.5, and S = 0.003, pressing OK after entering each of these new values. After these substitutions the input form will look like this:



```
CHEZY'S EQN
C:  95.
R:  2.5
S:  .003

Chezy's coefficient
EDIT |    |    |    |CANCL| OK
```

Now, to enter these values into the program press OK once more. This activates the function INFORM producing the following results in the stack:



```
3:
2:           {95. 2.5 .003}
1:                       1.
INFP1|  p  |FUNCa|INPTa|     |
```

Thus, we demonstrated the use of function INFORM. To see how to use these input values in a calculation modify the program as follows:

```
« " CHEZY'S EQN" { { "C:" "Chezy's coefficient" 0} { "R:"
"Hydraulic radius" 0 } { "S:" "Channel bed slope" 0} } { } { 120
1 .0001} { 110 1.5 .00001 }  INFORM IF THEN OBJ→ DROP → C R S
'C*(R*S)' →NUM "Q" →TAG ELSE "Operation cancelled" MSGBOX
END »
```

The program steps shown above after the INFORM command include a decision branching using the IF-THEN-ELSE-END construct (described in detail elsewhere in this Chapter). The program control can be sent to one of two possibilities depending on the value in stack level 1.   If this value is 1 the control is passed to the commands:

```
OBJ→ DROP → C R S 'C*√(R*S)' →NUM "Q" →TAG
```

These commands will calculate the value of Q and put a tag (or label) to it. On the other hand, if the value in stack level 1 is 0 (which happens when a CANCEL is entered while using the input box) , the program control is passed to the commands:

```
"Operation cancelled" MSGBOX
```

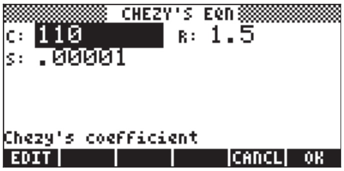
These commands will produce a message box indicating that the operation was cancelled.

Note: Function MSGBOX belongs to the collection of output functions under the PRG/OUT sub-menu. Commands IF, THEN, ELSE, END are available under the PRG/BRCH/IF sub-menu. Functions OBJ→, →TAG are available under the PRG/TYPE sub-menu. Function DROP is available under the PRG/STACK menu. Functions → and →NUM are available in the keyboard.

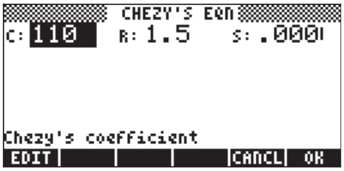
Example 2 – To illustrate the use of item 3 (Field format information) in the arguments of function INFORM, change the empty list used in program INFP1 to { 2 1 }, meaning 2, rather than the default 3, columns, and only one tab stop between labels and values. Store this new program in variable INFP2:

```
« " CHEZY'S EQN" { { "C:" "Chezy's coefficient" 0} { "R:"
"Hydraulic radius" 0 } { "S:" "Channel bed slope" 0} } {
2 1 } { 120 1 .0001} { 110 1.5 .00001 }  INFORM IF THEN
OBJ→  DROP  →  C  R  S  'C*(R*S)'  →NUM  "Q"  →TAG  ELSE
"Operation cancelled" MSGBOX END »
```

Running program INFP2 produces the following input form:


```
CHEZY'S EQN
C: 110          R: 1.5
S: .00001

Chezy's coefficient
EDIT |    |    |    |CANCL| OK
```

Example 3 – Change the field format information list to { 3 0 } and save the modified program into variable INFP3. Run this program to see the new input form:


```
CHEZY'S EQN
C: 110     R: 1.5     S: .0001

Chezy's coefficient
EDIT |    |    |    |CANCL| OK
```

Creating a choose box

Function CHOOSE (←PRG NXT IN CHOOS) allows the user to create a choose box in a program. This function requires three arguments:

1. A prompt (a character string describing the choose box)
2. A list of choice definitions {c_1 c_2 … c_n}. A choice definition c_i can have any of two formats:
    a. An object, e.g., a number, algebraic, etc., that will be displayed in the choose box and will also be the result of the choice.
    b. A list {object_displayed object_result} so that object_displayed is listed in the choose box, and object_result is selected as the result if this choice is selected.
3. A number indicating the position in the list of choice definitions of the default choice. If this number is 0, no default choice is highlighted.

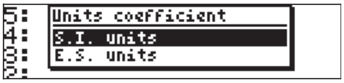
Activation of the CHOOSE function will return either a zero, if a CANCL action is used, or, if a choice is made, the choice selected (e.g., v) and the number 1, i.e., in the RPN stack:

```
2:            v
1:            1
```

Example 1 – Manning's equation for calculating the velocity in an open channel flow includes a coefficient, C_u, which depends on the system of units used. If using the S.I. (Systeme International), C_u = 1.0, while if using the E.S. (English System), C_u = 1.486. The following program uses a choose box to let the user select the value of C_u by selecting the system of units. Save it into variable CHP1 (CHoose Program 1):

```
« "Units coefficient" { { "S.I. units" 1}
   { "E.S. units" 1.486}  } 1 CHOOSE »
```

Running this program (press CHP1) shows the following choose box:



```
5: Units coefficient
4: S.I. units
3: E.S. units
2:
```

Depending on whether you select `S.I. units` or `E.S. units`, function CHOOSE places either a value of 1 or a value of 1.486 in stack level 2 and a 1 in level 1. If you cancel the choose box, CHOICE returns a zero (0).

The values returned by function CHOOSE can be operated upon by other program commands as shown in the modified program CHP2:

```
« "Units coefficient" { { "S.I. units" 1} { "E.S. units"
1.486}  } 1 CHOOSE IF THEN "Cu" →TAG ELSE "Operation
cancelled" MSGBOX END »
```

The commands after the CHOOSE function in this new program indicate a decision based on the value of stack level 1 through the IF-THEN-ELSE-END construct. If the value in stack level 1 is 1, the commands `"Cu" →TAG` will produced a tagged result in the screen. If the value in stack level 1 is zero, the commands `"Operation cancelled" MSGBOX` will show a message box indicating that the operation was cancelled.

Identifying output in programs

The simplest way to identify numerical program output is to "tag" the program results. A tag is simply a string attached to a number, or to any object. The string will be the name associated with the object. For example, earlier on, when debugging programs INPTa (or INPT1) and INPT2, we obtained as results tagged numerical output such as `:a:35.`

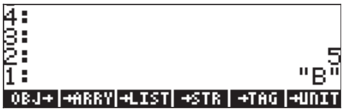
Tagging a numerical result

To tag a numerical result you need to place the number in stack level 2 and the tagging string in stack level 2, then use the →TAG function (← PRG TYPE →TAG) For example, to produce the tagged result `B:5.`, use:

5 ENTER → "" ALPHA B ← PRG TYPE →TAG

Decomposing a tagged numerical result into a number and a tag

To decompose a tagged result into its numerical value and its tag, simply use function OBJ→ (← PRG TYPE OBJ→). The result of decomposing a tagged number with →OBJ is to place the numerical value in stack level 2 and the tag in stack level 1. If you are interested in using the numerical value only, then you will drop the tag by using the backspace key ◄. For example, decomposing the tagged quantity `B:5` (see above), will produce:

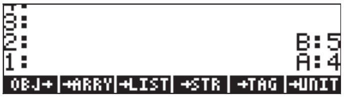
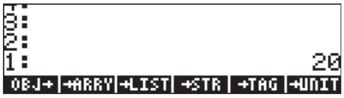


```
4:
3:
2:                 5
1:               "B"
```

"De-tagging" a tagged quantity

"De-tagging" means to extract the object out of a tagged quantity. This function is accessed through the keystroke combination: ← PRG TYPE NXT DTAG. For example, given the tagged quantity `a:2`, DTAG returns the numerical value 2.

Note: For mathematical operations with tagged quantities, the calculator will "detag" the quantity automatically before the operation. For example, the left-hand side figure below shows two tagged quantities before and after pressing the ⊠ key in RPN mode:


```
3:
2:                          B:5
1:                          A:4
OBJ→ →ARRY →LIST →STR →TAG →UNIT
```

```
3:
2:
1:                           20
OBJ→ →ARRY →LIST →STR →TAG →UNIT
```

Examples of tagged output

Example 1 – tagging output from function FUNCa
Let's modify the function FUNCa, defined earlier, to produce a tagged output. Use ⇨ FUNCa to recall the contents of FUNCa to the stack. The original function program reads

```
« "Enter a: " {"↵:a: " {2 0} V } INPUT OBJ→ → a « '2*a^2+3'
                    →NUM » »
```

Modify it to read:

```
« "Enter a: " {"↵:a: " {2 0} V } INPUT OBJ→ → a « '2*a^2+3'
                 →NUM "F" →TAG » »
```

Store the program back into FUNCa by using ⇦ FUNCa. Next, run the program by pressing FUNCa. Enter a value of 2 when prompted, and press ENTER. The result is now the tagged result `F:11`.

Example 2 – tagging input and output from function FUNCa
In this example we modify the program FUNCa so that the output includes not only the evaluated function, but also a copy of the input with a tag.
Use ⇨ FUNCa to recall the contents of FUNCa to the stack:

```
« "Enter a: " {"↵:a: " {2 0} V } INPUT OBJ→ → a « '2*a^2+3'
                 →NUM "F" →TAG » »
```

Modify it to read:

```
« "Enter a: " {"↵:a: " {2 0} V } INPUT OBJ→ → a « '2*a^2+3'
                    EVAL "F" →TAG a SWAP» »
```

(Recall that the function SWAP is available by using ← PRG STACK SWAP). Store the program back into FUNCa by using ← FUNCa. Next, run the program by pressing FUNCa . Enter a value of 2 when prompted, and press ENTER . The result is now two tagged numbers `a:2.` in stack level 2, and `F:11.` in stack level 1.

Note: Because we use an input string to get the input data value, the local variable `a` actually stores a tagged value (`:a:2`, in the example above). Therefore, we do not need to tag it in the output. All what we need to do is place an `a` before the SWAP function in the subprogram above, and the tagged input is placed in the stack. It should be pointed out that, in performing the calculation of the function, the tag of the tagged input a is dropped automatically, and only its numerical value is used in the calculation.

To see the operation of the function FUNCa, step by step, you could use the DBUG function as follows:

' FUNCa ENTER	Copies program name to stack level 1
← PRG NXT NXT RUN DBG	Starts debugger
SST↓	Step-by-step debugging, result: "Enter a:"
SST↓	Result: {" ↵a:" {2 0} V}
SST↓	Result: user is prompted to enter value of a
2 ENTER	Enter a value of 2 for a. Result: "↵a:2"
SST↓	Result: a:2
SST↓	Result: empty stack, executing →a
SST↓	Result: empty stack, entering subprogram «
SST↓	Result: '2*a^2+3'
SST↓	Result: empty stack, calculating
SST↓	Result: 11.,
SST↓	Result: "F"
SST↓	Result: F: 11.
SST↓	Result: a:2.
SST↓	Result: swap levels 1 and 2
SST↓	leaving subprogram »
SST↓	leaving main program »

Example 3 – tagging input and output from function p(V,T)
In this example we modify the program ▒▒ so that the output tagged input values and tagged result. Use ↱ ▒▒ to recall the contents of the program to the stack:

```
«“Enter V, T, and n:“ {“↵ :V:↵ :T:↵ :n:“ {2 0} V } INPUT
      OBJ→ →V T n  ‘(8.31451_J/(K*mol))*(n*T/V)‘ »
```

Modify it to read:

```
«“Enter V, T and n: “ {“↵ :V:↵ :T:↵ :n:“  {2 0} V } INPUT
OBJ→ →V T n «V T n‘(8.31451_J/(K*mol))*(n*T/V)‘ EVAL “p”
                      →TAG » »
```

Note: Notice that we have placed the calculation and tagging of the function p(V,T,n), preceded by a recall of the input variables V T n, into a sub-program [the sequence of instructions contained within the inner set of program symbols « »]. This is necessary because without the program symbol separating the two listings of input variables (`V T N « V T n`), the program will assume that the input command

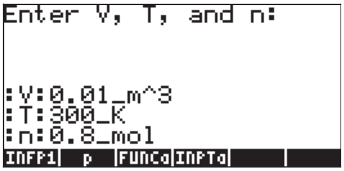
`→V T N V T n`

requires six input values, while only three are available. The result would have been the generation of an error message and the interruption of the program execution.

To include the subprogram mentioned above in the modified definition of program ▒▒, will require you to use ↱ «» at the beginning and end of the sub-program. Because the program symbols occur in pairs, whenever ↱ «» is invoked, you will need to erase the closing program symbol (») at the beginning, and the opening program symbol («) at the end, of the sub-program.

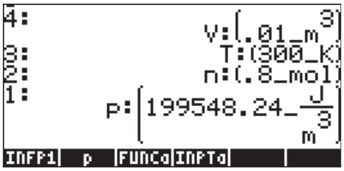
To erase any character while editing the program, place the cursor to the right of the character to be erased and use the backspace key ⬅.

Store the program back into variable p by using ← p. Next, run the program by pressing p. Enter values of V = 0.01_m^3, T = 300_K, and n = 0.8_mol, when prompted. Before pressing ENTER for input, the stack will look like this:



```
Enter V, T, and n:


:V:0.01_m^3
:T:300_K
:n:0.8_mol
INFP1  p  FUNCa INPTa
```

After execution of the program, the stack will look like this:



```
4:                  V:(.01_m^3)
3:                   T:(300_K)
2:                   n:(.8_mol)
1:         p:(199548.24_J/m^3)
INFP1  p  FUNCa INPTa
```

In summary: The common thread in the three examples shown here is the use of tags to identify input and output variables. If we use an input string to get our input values, those values are already pre-tagged and can be easily recall into the stack for output. Use of the →TAG command allows us to identify the output from a program.

Using a message box

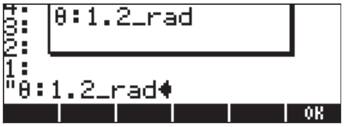
A message box is a fancier way to present output from a program. The message box command in the calculator is obtained by using ← PRG NXT OUT MSGBX. The message box command requires that the output string to be placed in the box be available in stack level 1. To see the operation of the MSGBOX command try the following exercise:

→ "" ALPHA → T ALPHA ← : : / . 2

→ _ ALPHA ← R ALPHA ← A ALPHA ← D

← PRG NXT OUT MSGBX

The result is the following message box:


```
4:  θ:1.2_rad
3:
2:
1:
"θ:1.2_rad
                              OK
```

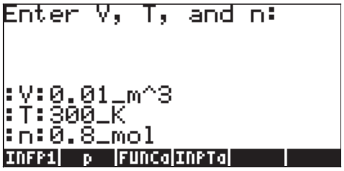
Press OK to cancel the message box.

You could use a message box for output from a program by using a tagged output, converted to a string, as the output string for MSGBOX. To convert any tagged result, or any algebraic or non-tagged value, to a string, use the function →STR available at ← PRG TYPE →STR.

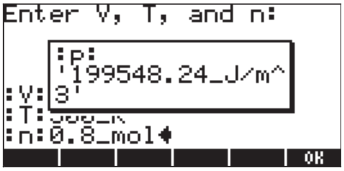
Using a message box for program output
The function p, from the last example, can be modified to read:

```
« "Enter V, T and n: " {"↵ :V:↵ :T:↵ :n: " {2 0} V } INPUT
OBJ→ →V T n « V T n'(8.31451_J/(K*mol))*(n*T/V)' EVAL "p"
→TAG →STR MSGBOX » »
```

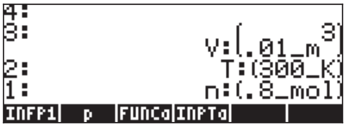
Store the program back into variable p by using ← p. Run the program by pressing p. Enter values of V = 0.01_m^3, T = 300_K, and n = 0.8_mol, when prompted.
As in the earlier version of p, before pressing ENTER for input, the stack will look like this:


```
Enter V, T, and n:


:V:0.01_m^3
:T:300_K
:n:0.8_mol
INFP1  p  FUNCa INPTa
```

The first program output is a message box containing the string:


```
Enter V, T, and n:
   :p:
   '199548.24_J/m^
:V:3'
:T:300_K
:n:0.8_mol
                              OK
```

Press OK to cancel message box output. The stack will now look like this:


```
4:
3:
               V:[.01_m^3]
2:              T:(300_K)
1:             n:(.8_mol)
INPP1  p  FUNCa INPTa
```

Including input and output in a message box
We could modify the program so that not only the output, but also the input, is included in a message box. For the case of program p, the modified program will look like:

```
« "Enter V, T and n: " {"↵ :V:↵ :T:↵ :n: " {2 0} V } INPUT
OBJ→ → V T n « V →STR "↵ " + T →STR "↵ " +  n →STR "↵ " +
'(8.31451_J/(K*mol))*(n*T/V)' EVAL "p" →TAG →STR + + +
MSGBOX » »
```

Notice that you need to add the following piece of code after each of the variable names V, T, and n, within the sub-program:

```
→STR "↵ " +
```

To get this piece of code typed in the first time use:

← PRG TYPE →STR → " → ↵ ▶ +

Because the functions for the TYPE menu remain available in the soft menu keys, for the second and third occurrences of the piece of code (→STR "↵ " +) within the sub-program (i.e., after variables T and n, respectively), all you need to use is:

→STR → " → ↵ ▶ +

You will notice that after typing the keystroke sequence ↱ ← a new line is generated in the stack.

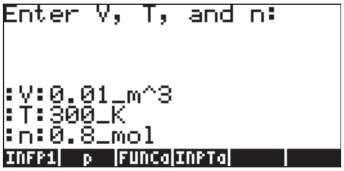
The last modification that needs to be included is to type in the plus sign three times after the call to the function at the very end of the sub-program.

Note: The plus sign (+) in this program is used to *concatenate* strings. *Concatenation* is simply the operation of joining individual character strings.

To see the program operating:

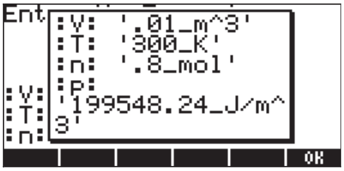
- Store the program back into variable p by using ↰ p.
- Run the program by pressing p.
- Enter values of V = 0.01_m^3, T = 300_K, and n = 0.8_mol, when prompted.

As in the earlier version of [p], before pressing [ENTER] for input, the stack will look like this:



```
Enter V, T, and n:


:V:0.01_m^3
:T:300_K
:n:0.8_mol
```

The first program output is a message box containing the string:



```
:V: '.01_m^3'
:T: '300_K'
:n: '.8_mol'
:p:
'199548.24_J/m^
3'
```

Press OK to cancel message box output.

Incorporating units within a program
As you have been able to observe from all the examples for the different versions of program presented in this chapter, attaching units to input values may be a tedious process. You could have the program itself attach those units to the input and output values. We will illustrate these options by modifying yet once more the program , as follows.

Recall the contents of program to the stack by using , and modify them to look like this:

Note: I've separated the program arbitrarily into several lines for easy reading. This is not necessarily the way that the program shows up in the calculator's stack. The sequence of commands is correct, however. Also, recall that the character ↵ does not show in the stack, instead it produces a new line.

```
« "Enter V,T,n [S.I.]: " {"↵ :V:↵ :T:↵ :n: " {2 0} V }
INPUT OBJ→ →V T n
« V '1_m^3' *  T '1_K' *  n '1_mol' *  →V T n
« V "V" →TAG →STR "↵ "+ T "T" →TAG →STR "↵ "+  n "n" →TAG
→STR "↵ "+
'(8.31451_J/(K*mol))*(n*T/V)' EVAL "p" →TAG →STR + + +
MSGBOX » » »
```

This new version of the program includes an additional level of sub-programming (i.e., a third level of program symbols « », and some steps using lists, i.e.,

`V '1_m^3' * { }+ T '1_K' * + n '1_mol' * + EVAL →V T n`

The *interpretation* of this piece *of code* is as follows. (We use input string values of `:V:0.01`, `:T:300`, and `:n:0.8`):

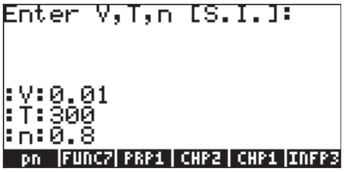
1. `V`	: The value of V, as a tagged input (e.g., V:0.01) is placed in the stack.

2. `‘1_m^3’`	: The S.I. units corresponding to V are then placed in stack level 1, the tagged input for V is moved to stack level 2.
3. `*`	: By multiplying the contents of stack levels 1 and 2, we generate a number with units (e.g., 0.01_m^3), but the tag is lost.
4. `T ‘1_K’ *`	:Calculating value of T including S.I. units
5. `n ‘1_mol’ *`	: Calculating value of n including units
6. `→V T n`	: The values of V, T, and n, located respectively in stack levels 3, 2, and 1, are passed on to the next level of sub-programming.

To see this version of the program in action do the following:

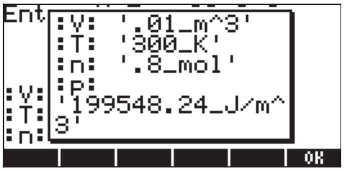
- Store the program back into variable p by using [↰][p].
- Run the program by pressing [p].
- Enter values of V = 0.01, T = 300, and n = 0.8, when prompted (no units required now).

Before pressing ENTER for input, the stack will look like this:



```
Enter V,T,n [S.I.]:


:V:0.01
:T:300
:n:0.8
```

Press ENTER to run the program. The output is a message box containing the string:



```
:V: '.01_m^3'
:T: '300_K'
:n: '.8_mol'
:p:
'199548.24_J/m^
3'
```

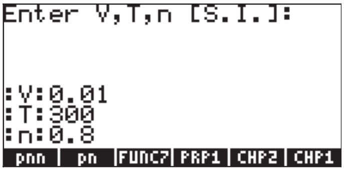
Press OK to cancel message box output.

Message box output without units

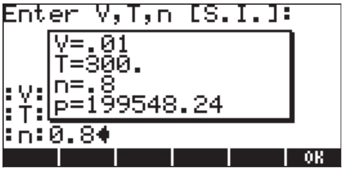
Let's modify the program FUNC once more to eliminate the use of units throughout it. The unit-less program will look like this:

```
« "Enter V,T,n [S.I.]: " {"↵ :V:↵ :T:↵ :n: " {2 0} V }
INPUT OBJ→ →V T n
« V DTAG  T DTAG  n DTAG →  V T n
« "V=" V →STR + "↵ "+ "T=" T →STR + "↵ "+ "n="  n →STR +
"↵ "+
'8.31451*n*T/V' EVAL →STR "p=" SWAP + + + + MSGBOX » » »
```

And when run with the input data V = 0.01, T = 300, and n = 0.8, produces the message box output:



```
Enter V,T,n [S.I.]:


:V:0.01
:T:300
:n:0.8
pnn | pn |FUNC7|PRP1|CHP2|CHP1
```



```
Enter V,T,n [S.I.]:
   V=.01
   T=300.
:V:n=.8
:T:p=199548.24
:n:0.8
|  |  |  |  | OK
```

Press OK to cancel the message box output.

Relational and logical operators

So far we have worked mainly with sequential programs. The User RPL language provides statements that allow branching and looping of the program flow. Many of these make decisions based on whether a logical statement is true or not. In this section we present some of the elements used to construct such logical statements, namely, relational and logical operators.

Relational operators

Relational operators are those operators used to compare the relative position of two objects. For example, dealing with real numbers only, relational operators are used to make a statement regarding the relative position of two or more real numbers. Depending on the actual numbers used, such a statement can be true (represented by the numerical value of 1. in the calculator), or false (represented by the numerical value of 0. in the calculator).

The relational operators available for programming the calculator are:

Operator	**Meaning**	**Example**
==	"is equal to"	'x==2'
≠	"is not equal to"	'3 ≠ 2'
<	"is less than"	'm<n'
>	"is greater than"	'10>a'
≥	"is greater than or equal to"	'p ≥ q'
≤	"is less than or equal to"	'7≤12'

All of the operators, except == (which can be created by typing ↱ = ↱ =), are available in the keyboard. They are also available in ↰ PRG TEST.

Two numbers, variables, or algebraics connected by a relational operator form a logical expression that can take value of true (1.), false (0.), or could simply not be evaluated. To determine whether a logical statement is true or not, place the statement in stack level 1, and press EVAL (EVAL). Examples:

'2<10' EVAL, result: 1. (true)

'2>10' EVAL, result: 0. (false)

In the next example it is assumed that the variable m is not initialized (it has not been given a numerical value):

'2==m' EVAL, result: '2==m'

The fact that the result from evaluating the statement is the same original statement indicates that the statement cannot be evaluated uniquely.

Logical operators

Logical operators are logical particles that are used to join or modify simple logical statements. The logical operators available in the calculator can be easily accessed through the keystroke sequence: ← PRG TEST NXT.

The available logical operators are: AND, OR, XOR (exclusive or), NOT, and SAME. The operators will produce results that are true or false, depending on the truth-value of the logical statements affected. The operator NOT (negation) applies to a single logical statements. All of the others apply to two logical statements.

Tabulating all possible combinations of one or two statements together with the resulting value of applying a certain logical operator produces what is called the truth table of the operator. The following are truth tables of each of the standard logical operators available in the calculator:

p	NOT p
1	0
0	1

p	q	p AND q
1	1	1
1	0	0
0	1	0
0	0	0

p	q	p OR q
1	1	1
1	0	1
0	1	1
0	0	0

p	q	p XOR q
1	1	0
1	0	1
0	1	1
0	0	0

The calculator includes also the logical operator SAME. This is a non-standard logical operator used to determine if two objects are identical. If they are identical, a value of 1 (true) is returned, if not, a value of 0 (false) is returned. For example, the following exercise, in RPN mode, returns a value of 0:

‘SQ(2)’ ENTER 4 ENTER SAME

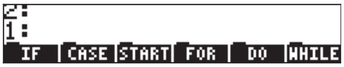
Please notice that the use of SAME implies a very strict interpretation of the word “identical.” For that reason, SQ(2) is not identical to 4, although they both evaluate, numerically, to 4.

Program branching

Branching of a program flow implies that the program makes a decision among two or more possible flow paths. The User RPL language provides a number of commands that can be used for program branching. The menus containing these commands are accessed through the keystroke sequence:

← PRG BRCH

This menu shows sub-menus for the program constructs



The program constructs IF…THEN..ELSE…END, and CASE…THEN…END will be referred to as program branching constructs. The remaining constructs, namely, START, FOR, DO, and WHILE, are appropriate for controlling repetitive processing within a program and will be referred to as program loop constructs. The latter types of program constructs are presented in more detail in a later section.

Branching with IF

In this section we presents examples using the constructs IF…THEN…END and IF…THEN…ELSE…END.

The IF…THEN…END construct

The IF…THEN…END is the simplest of the IF program constructs. The general format of this construct is:

```
IF logical_statement THEN program_statements END.
```

The operation of this construct is as follows:

1. Evaluate logical_statement.
2. If logical_statement is true, perform program _statements and continue program flow after the END statement.
3. If logical_statement is false, skip program_statements and continue program flow after the END statement.

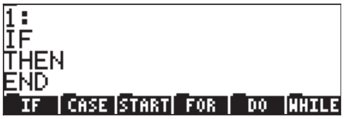
To type in the particles IF, THEN, ELSE, and END, use:

← PRG BRCH IF

The functions IF THEN ELSE END are available in that menu to be typed selectively by the user. Alternatively, to produce an IF…THEN…END construct directly on the stack, use:

← PRG BRCH ← IF

This will create the following input in the stack:



```
1:
IF
THEN
END
```

With the cursor ⬅ in front of the IF statement prompting the user for the logical statement that will activate the IF construct when the program is executed.
Example: Type in the following program:

```
« → x « IF 'x<3' THEN 'x^2' EVAL END "Done" MSGBOX » »
```

and save it under the name 'f1'. Press VAR and verify that variable f1 is indeed available in your variable menu. Verify the following results:

0 f1	Result: 0	1.2 f1	Result: 1.44
3.5 f1	Result: no action	10 f1	Result: no action

These results confirm the correct operation of the IF…THEN…END construct. The program, as written, calculates the function $f_1(x) = x^2$, if $x < 3$ (and not output otherwise).

The IF…THEN…ELSE…END construct

The IF…THEN…ELSE…END construct permits two alternative program flow paths based on the truth value of the logical_statement. The general format of this construct is:

```
IF logical_statement THEN program_statements_if_true ELSE
            program_statements_if_false END.
```

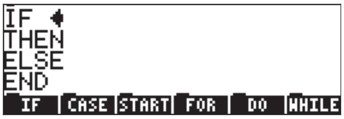
The operation of this construct is as follows:

1. Evaluate logical_statement.
2. If logical_statement is true, perform program statements_if_true and continue program flow after the END statement.
3. If logical_statement is false, perform program statements_if_false and continue program flow after the END statement.

To produce an IF…THEN…ELSE…END construct directly on the stack, use:

← PRG BRCH → IF

This will create the following input in the stack:


```
IF
THEN
ELSE
END
```

Example: Type in the following program:

`« → x « IF 'x<3' THEN 'x^2' ELSE '1-x' END EVAL "Done" MSGBOX » »`

and save it under the name 'f2'. Press VAR and verify that variable f2 is indeed available in your variable menu. Verify the following results:

0 f2	Result: 0	1.2 f2	Result: 1.44
3.5 f2	Result: -2.5	10 f2	Result: -9

These results confirm the correct operation of the IF…THEN…ELSE…END construct. The program, as written, calculates the function

$$f_2(x) = \begin{cases} x^2, if\ x < 3 \\ 1-x,\ otherwise \end{cases}$$

Note: For this particular case, a valid alternative would have been to use an IFTE function of the form: 'f2(x) = IFTE(x<3,x^2,1-x)'

Nested IF…THEN…ELSE…END constructs

In most computer programming languages where the IF…THEN…ELSE…END construct is available, the general format used for program presentation is the following:

```
IF logical_statement THEN
      program_statements_if_true
ELSE
      program_statements_if_false
END
```

In designing a calculator program that includes IF constructs, you could start by writing by hand the pseudo-code for the IF constructs as shown above. For example, for program f2, you could write

```
IF x<3 THEN
      x^2
ELSE
      1-x
END
```

While this simple construct works fine when your function has only two branches, you may need to nest IF…THEN…ELSE…END constructs to deal with function with three or more branches. For example, consider the function

$$f_3(x) = \begin{cases} x^2, if\ x < 3 \\ 1-x,\ if\ 3 \le x < 5 \\ \sin(x), if\ 5 \le x < 3\pi \\ \exp(x), if\ 3\pi \le x < 15 \\ -2, elsewhere \end{cases}$$

Here is a possible way to evaluate this function using IF… THEN … ELSE … END constructs:

```
IF x<3 THEN
      x^2
ELSE
      IF x<5 THEN
            1-x
      ELSE
            IF x<3π         THEN
                  sin(x)
            ELSE
                  IF x<15 THEN
                        exp(x)
                  ELSE
                        -2
                  END
            END
      END
END
```

A complex IF construct like this is called a set of <u>*nested*</u> IF … THEN … ELSE … END constructs.

A possible way to evaluate f3(x), based on the nested IF construct shown above, is to write the program:

```
« → x « IF ‘x<3‘ THEN ‘x^2‘ ELSE IF ‘x<5‘ THEN ‘1-x‘ ELSE IF
‘x<3*π‘ THEN ‘SIN(x)‘ ELSE IF ‘x<15‘ THEN ‘EXP(x)‘ ELSE –2 END
END END END EVAL » »
```

Store the program in variable F3 and try the following evaluations:

1.5 F3	*Result*:	2.25 (i.e., x^2)
2.5 F3	*Result*:	6.25 (i.e., x^2)
4.2 F3	*Result*:	-3.2 (i.e., 1-x)
5.6 F3	*Result*	-0.631266… (i.e., sin(x), with x in radians)
12 F3	*Result*:	162754.791419 (i.e., exp(x))
23 F3	*Result*:	-2. (i.e., -2)

The CASE construct

The CASE construct can be used to code several possible program flux paths, as in the case of the nested IF constructs presented earlier. The general format of this construct is as follows:

```
CASE
Logical_statement₁ THEN program_statements₁ END
Logical_statement₂ THEN program_statements₂ END
.
.
.
Logical_statement THEN program_statements END
Default_program_statements (optional)
END
```

When evaluating this construct, the program tests each of the *logical_statements* until it finds one that is true. The program executes the corresponding *program_statements*, and passes program flow to the statement following the END statement.

The CASE, THEN, and END statements are available for selective typing by using ↰ PRG BRCH CASE .

If you are in the BRCH menu, i.e., (↰ PRG BRCH) you can use the following shortcuts to type in your CASE construct (The location of the cursor is indicated by the symbol ⬅):

- ↰ CASE: Starts the case construct providing the prompts: CASE ⬅ THEN END END

- ↱ CASE: Completes a CASE line by adding the particles THEN ⬅ END

Example – program $f_3(x)$ using the CASE statement
The function is defined by the following 5 expressions:

$$f_3(x) = \begin{cases} x^2, \mathit{if}\ x < 3 \\ 1-x,\ \mathit{if}\ 3 \le x < 5 \\ \sin(x), \mathit{if}\ 5 \le x < 3\pi \\ \exp(x), \mathit{if}\ 3\pi \le x < 15 \\ -2, \mathit{elsewhere} \end{cases}$$

Using the CASE statement in User RPL language we can code this function as:

```
« → x « CASE ‘x<3‘ THEN ‘x^2‘ END ‘x<5‘ THEN ‘1-x‘ END ‘x<3*π‘
THEN ‘SIN(x)‘ END ‘x<15‘ THEN ‘EXP(x)‘ END –2 END EVAL » »
```

Store the program into a variable called f3. Then, try the following exercises:

1.5	f3	*Result*:	2.25 (i.e., x^2)
2.5	f3	*Result*:	6.25 (i.e., x^2)
4.2	f3	*Result*:	-3.2 (i.e., 1-x)

5.6 [illegible] *Result*: -0.631266… (i.e., sin(x), with x in radians)
12 [illegible] *Result*: 162754.791419 (i.e., exp(x))
23 [illegible] *Result* -2. (i.e., -2)

As you can see, f3c produces exactly the same results as f3. The only difference in the programs is the branching constructs used. For the case of function $f_3(x)$, which requires five expressions for its definition, the CASE construct may be easier to code than a number of nested IF … THEN … ELSE … END constructs.

Program loops

Program loops are constructs that permit the program the execution of a number of statements repeatedly. For example, suppose that you want to calculate the summation of the square of the integer numbers from 0 to n, i.e.,

$$S = \sum_{k=0}^{n} k^2$$

To calculate this summation all that you have to do is use the ↱ Σ key within the equation editor and load the limits and expression for the summation (examples of summations are presented in Chapters 2 and 13). However, in order to illustrate the use of programming loops, we will calculate this summation with our own User RPL codes. There are four different commands that can be used to code a program loop in User RPL, these are START, FOR, DO, and WHILE. The commands START and FOR use an index or counter to determine how many times the loop is executed. The commands DO and WHILE rely on a logical statement to decide when to terminate a loop execution. Operation of the loop commands is described in detail in the following sections.

The START construct

The START construct uses two values of an index to execute a number of statements repeatedly. There are two versions of the START construct: START…NEXT and START … STEP. The START…NEXT version is used when the index increment is equal to 1, and the START…STEP version is used when the index increment is determined by the user.

Commands involved in the START construct are available through:

← PRG BRCH START START

Within the BRCH menu (← PRG BRCH) the following keystrokes are available to generate START constructs (the symbol indicates cursor position):

- ← START: Starts the START…NEXT construct: START ⬅ NEXT
- → START: Starts the START…STEP construct: START ⬅ STEP

The START…NEXT construct
The general form of this statement is:

```
start_value end_value START program_statements NEXT
```

Because for this case the increment is 1, in order for the loop to end you should ensure that `start_value < end_value`. Otherwise you will produce what is called an infinite (never-ending) loop.

Example – calculating of the summation S defined above
The START…NEXT construct contains an index whose value is inaccessible to the user. Since for the calculation of the sum the index itself (k, in this case) is needed, we must create our own index, k, that we will increment within the loop each time the loop is executed. A possible implementation for the calculation of S is the program:

```
« 0. DUP →n S k « 0. n START k SQ S + 1. ‘k‘ STO+ ‘S‘ STO
NEXT S “S” →TAG » »
```

Type the program in, and save it in a variable called S1.

Here is a brief explanation of how the program works:

1. This program needs an integer number as input. Thus, before execution, that number (n) is in stack level 1. The program is then executed.
2. A zero is entered, moving n to stack level 2.
3. The command DUP, which can be typed in as ALPHA ALPHA D U P ALPHA, copies the contents of stack level 1, moves all the stack levels upwards, and places the copy just made in stack level 1. Thus, after DUP is executed, n is in stack level 3, and zeroes fill stack levels 1 and 2.
4. The piece of code `→ n S k` stores the values of n, 0, and 0, respectively into local variables n, S, k. We say that the variables n, S, and k have been <u>initialized</u> (S and k to zero, n to whatever value the user chooses).
5. The piece of code `0. n START` identifies a START loop whose index will take values of 0, 1, 2, …, n
6. The sum S is incremented by k^2 in the piece of code that reads: `k SQ S +`
7. The index k is incremented by 1 in the piece of code that reads: `1. k +`
8. At this point, the updated values of S and k are available in stack levels 2 and 1, respectively. The piece of code `‘k‘ STO` stores the value from stack level 1 into local variable k. The updated value of S now occupies stack level 1.
9. The piece of code `‘S‘ STO` stores the value from stack level 1 into local variable k. The stack is now empty.
10. The particle NEXT increases the index by one and sends the control to the beginning of the loop (step 6).
11. The loop is repeated until the loop index reaches the maximum value, n.
12. The last part of the program recalls the last value of S (the summation), tags it, and places it in stack level 1 to be viewed by the user as the program output.

To see the program in action, step by step, you can use the debugger as follows (use n = 2). Let SL1 mean stack level 1:

VAR 2 ['] S1 ENTER	Place a 2 in level 2, and the program name, ‘S1’, in level 1

← PRG NXT NXT RUN DBG	Start the debugger. SL1 = 2.
SST↓	SL1 = 0., SL2 = 2.
SST↓	SL1 = 0., SL2 = 0., SL3 = 2. (DUP)
SST↓	Empty stack (-> n S k)
SST↓	Empty stack (« - start subprogram)
SST↓	SL1 = 0., (start value of loop index)
SST↓	SL1 = 2.(n), SL2 = 0. (end value of loop index)
SST↓	Empty stack (START – beginning of loop)

--- loop execution number 1 for k = 0

SST↓	SL1 = 0. (k)
SST↓	SL1 = 0. (SQ(k) = k^2)
SST↓	SL1 = 0.(S), SL2 = 0. (k^2)
SST↓	SL1 = 0. (S + k^2)
SST↓	SL1 = 1., SL2 = 0. (S + k^2)
SST↓	SL1 = 0.(k), SL2 = 1., SL3 = 0. (S + k^2)
SST↓	SL1 = 1.(k+1), SL2 = 0. (S + k^2)
SST↓	SL1 = 'k', SL2 = 1., SL3 = 0. (S + k^2)
SST↓	SL1 = 0. (S + k^2) [Stores value of SL2 = 1, into SL1 = 'k']
SST↓	SL1 = 'S', SL2 = 0. (S + k^2)
SST↓	Empty stack [Stores value of SL2 = 0, into SL1 = 'S']
SST↓	Empty stack (NEXT – end of loop)

--- loop execution number 2 for k = 1

SST↓	SL1 = 1. (k)
SST↓	SL1 = 1. (SQ(k) = k^2)
SST↓	SL1 = 0.(S), SL2 = 1. (k^2)
SST↓	SL1 = 1. (S + k^2)
SST↓	SL1 = 1., SL2 = 1. (S + k^2)
SST↓	SL1 = 1.(k), SL2 = 1., SL3 = 1. (S + k^2)
SST↓	SL1 = 2.(k+1), SL2 = 1. (S + k^2)
SST↓	SL1 = 'k', SL2 = 2., SL3 = 1. (S + k^2)

SST↓	SL1 = 1. (S + k^2) [Stores value of SL2 = 2, into SL1 = ‘k’]
SST↓	SL1 = ‘S’, SL2 = 1. (S + k^2)
SST↓	Empty stack [Stores value of SL2 = 1, into SL1 = ‘S’]
SST↓	Empty stack (NEXT – end of loop)

--- loop execution number 3 for k = 2

SST↓	SL1 = 2. (k)
SST↓	SL1 = 4. (SQ(k) = k^2)
SST↓	SL1 = 1.(S), SL2 = 4. (k^2)
SST↓	SL1 = 5. (S + k^2)
SST↓	SL1 = 1., SL2 = 5. (S + k^2)
SST↓	SL1 = 2.(k), SL2 = 1., SL3 = 5. (S + k^2)
SST↓	SL1 = 3.(k+1), SL2 = 5. (S + k^2)
SST↓	SL1 = ‘k’, SL2 = 3., SL3 = 5. (S + k^2)
SST↓	SL1 = 5. (S + k^2) [Stores value of SL2 = 3, into SL1 = ‘k’]
SST↓	SL1 = ‘S’, SL2 = 5. (S + k^2)
SST↓	Empty stack [Stores value of SL2 = 0, into SL1 = ‘S’]
SST↓	Empty stack (NEXT – end of loop)

--- for n = 2, the loop index is exhausted and control is passed to the statement following NEXT

SST↓	SL1 = 5 (S is recalled to the stack)
SST↓	SL1 = “S”, SL2 = 5 (“S” is placed in the stack)
SST↓	SL1 = S:5 (tagging output value)
SST↓	SL1 = S:5 (leaving sub-program »)
SST↓	SL1 = S:5 (leaving main program »)

The step-by-step listing is finished. The result of running program S1 with n = 2, is S:5.

Check also the following results: VAR

3 S1	Result: `S:14`	4 S1	Result: `S:30`
5 S1	Result: `S:55`	8 S1	Result: `S:204`
10 S1	Result: `S:385`	20 S1	Result: `S:2870`
30 S1	Result: `S:9455`	100 S1	Result: `S:338350`

The START…STEP construct
The general form of this statement is:

```
start_value end_value START program_statements increment
NEXT
```

The start_value, end_value, and `increment` of the loop index can be positive or negative quantities. For `increment > 0`, execution occurs as long as the index is less than or equal to `end_value`. For `increment < 0`, execution occurs as long as the index is greater than or equal to `end_value`.

Example – generating a list of values
Suppose that you want to generate a list of values of x from x = 0.5 to x = 6.5 in increments of 0.5. You can write the following program:

```
« → xs xe dx « xs DUP xe START DUP dx + dx STEP DROP xe xs
- dx / ABS 1 + →LIST » »
```

and store it in variable GLIST.

In this program , xs = starting value of the loop, xe = ending value of the loop, dx = increment value for loop. The program places values of xs, xs+dx, xs+2·dx, xs+3·dx, … in the stack. Then, it calculates the number of elements generated using the piece of code: `xe xs - dx / ABS 1. +`

Finally, the program puts together a list with the elements placed in the stack.

- Check out that the program call 0.5 ENTER 2.5 ENTER 0.5 ENTER GLIST produces the list {0.5 1. 1.5 2. 2.5}.
- To see step-by-step operation use the program DBUG for a short list, for example:

VAR 1 SPC 1.5 SPC 0.5 ENTER	Enter parameters 1 1.5 0.5
['] GLIS2 ENTER	Enter the program name in level 1
← PRG NXT NXT RUN DBG	Start the debugger.

Use SST↓ to step into the program and see the detailed operation of each command.

The FOR construct

As in the case of the START command, the FOR command has two variations: the FOR…NEXT construct, for loop index increments of 1, and the FOR…STEP construct, for loop index increments selected by the user. Unlike the START command, however, the FOR command does require that we provide a name for the loop index (e.g., j, k, n).   We need not to worry about incrementing the index ourselves, as done in the examples using START. The value corresponding to the index is available for calculations.

Commands involved in the FOR construct are available through:

← PRG BRCH FOR

Within the BRCH menu (← PRG BRCH) the following keystrokes are available to generate FOR constructs (the symbol ⬅ indicates cursor position):

- ← FOR: Starts the FOR…NEXT construct: FOR ⬅ NEXT
- → FOR: Starts the FOR…STEP construct: FOR ⬅ STEP

The FOR…NEXT construct

The general form of this statement is:

```
start_value end_value FOR loop_index program_statements
NEXT
```

To avoid an infinite loop, make sure that `start_value < end_value`.

Example – calculate the summation S using a FOR…NEXT construct
The following program calculates the summation

$$S = \sum_{k=0}^{n} k^2$$

Using a FOR…NEXT loop:

```
« 0 →n S « 0 n FOR k k SQ S + ‘S‘ STO NEXT S “S” →TAG » »
```

Store this program in a variable [illegible]. Verify the following exercises: VAR

3 [illegible]	Result: `S:14`	4 [illegible]	Result: `S:30`
5 [illegible]	Result: `S:55`	8 [illegible]	Result: `S:204`
10 [illegible]	Result: `S:385`	20 [illegible]	Result: `S:2870`
30 [illegible]	Result: `S:9455`	100 [illegible]	Result: `S:338350`

You may have noticed that the program is much simpler than the one stored in [illegible]. There is no need to initialize k, or to increment k within the program. The program itself takes care of producing such increments.

The FOR…STEP construct

The general form of this statement is:

```
start_value end_value FOR loop_index program_statements
increment STEP
```

The start_value, end_value, and `increment` of the loop index can be positive or negative quantities. For `increment > 0`, execution occurs as long as the index is less than or equal to `end_value`. For `increment < 0`, execution occurs as long as the index is greater than or equal to `end_value`. Program statements are executed at least once (e.g., `1 0 START 1 1 STEP` returns 1)

Example – generate a list of numbers using a FOR…STEP construct
Type in the program:

```
« → xs xe dx « xe xs - dx / ABS 1. + → n « xs xe FOR x x
dx STEP n →LIST » » »
```

and store it in variable GLIS2.

- Check out that the program call 0.5 ENTER 2.5 ENTER 0.5 ENTER GLIS2 produces the list {0.5 1. 1.5 2. 2.5}.
- To see step-by-step operation use the program DBUG for a short list, for example:

VAR 1 SPC 1.5 SPC 0.5 ENTER	Enter parameters 1 1.5 0.5
['] GLIS2 ENTER	Enter the program name in level 1
← PRG NXT NXT RUN DBG	Start the debugger.

Use SST↓ to step into the program and see the detailed operation of each command.

The DO construct

The general structure of this command is:

```
DO program_statements UNTIL logical_statement END
```

The DO command starts an indefinite loop executing the program_statements until the logical_statement returns FALSE (0). The `logical_statement` must contain the value of an index whose value is changed in the `program_statements`.

Example 1 - This program produces a counter in the upper left corner of the screen that adds 1 in an indefinite loop until a keystroke (press any key) stops the counter: `« 0 DO DUP 1 DISP 1 + UNTIL KEY END DROP »`

Command KEY evaluates to TRUE when a keystroke occurs.

Example 2 – calculate the summation S using a DO…UNTIL…END construct

The following program calculates the summation

$$S = \sum_{k=0}^{n} k^2$$

Using a DO…UNTIL…END loop:

```
« 0. → n S « DO n SQ S + 'S' STO n 1 – 'n' STO UNTIL 'n<0' END
S "S" →TAG » »
```

Store this program in a variable S2. Verify the following exercises: VAR

3 S2	Result: `S:14`	4 S2	Result: `S:30`
5 S2	Result: `S:55`	8 S2	Result: `S:204`
10 S2	Result: `S:385`	20 S2	Result: `S:2870`
30 S2	Result: `S:9455`	100 S2	Result: `S:338350`

Example 3 – generate a list using a DO…UNTIL…END construct
Type in the following program

```
« → xs xe dx « xe xs – dx / ABS 1. + xs → n x « xs DO
'x+dx' EVAL DUP 'x' STO UNTIL 'x≥xe' END n →LIST » » »
```

and store it in variable GLIS2.

- Check out that the program call 0.5 ENTER 2.5 ENTER 0.5 ENTER GLIS2 produces the list {0.5 1. 1.5 2. 2.5}.
- To see step-by-step operation use the program DBUG for a short list, for example:

VAR 1 SPC 1.5 SPC 0.5 ENTER	Enter parameters 1 1.5 0.5
['] GLIS2 ENTER	Enter the program name in level 1
← PRG NXT NXT RUN DBG	Start the debugger.

Use SST↓ to step into the program and see the detailed operation of each command.

The WHILE construct

The general structure of this command is:

```
WHILE logical_statement REPEAT program_statements END
```

The WHILE statement will repeat the `program_statements` while `logical_statement` is true (non zero). If not, program control is passed to the statement right after END. The `program_statements` must include a loop index that gets modified before the `logical_statement` is checked at the beginning of the next repetition. Unlike the DO command, if the first evaluation of logical_statement is false, the loop is never executed.

Example 1 – calculate the summation S using a WHILE…REPEAT…END construct

The following program calculates the summation

$$S=\sum_{k=0}^{n}k^2$$

Using a WHILE…REPEAT…END loop:

```
« 0. →n S « WHILE 'n≥0' REPEAT n SQ S + 'S' STO n 1 - 'n' STO
END S "S" →TAG » »
```

Store this program in a variable S4. Verify the following exercises: VAR

3	S4	Result: `S:14`	4	S4	Result: `S:30`
5	S4	Result: `S:55`	8	S4	Result: `S:204`
10	S4	Result: `S:385`	20	S4	Result: `S:2870`
30	S4	Result: `S:9455`	100	S4	Result: `S:338350`

Example 2 – generate a list using a WHILE…REPEAT…END construct

Type in the following program

```
« → xs xe dx « xe xs - dx / ABS 1. + xs → n x « xs WHILE
'x<xe' REPEAT 'x+dx' EVAL DUP 'x' STO END n →LIST » » »
```

and store it in variable GLIST.

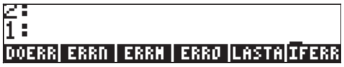
- Check out that the program call 0.5 ENTER 2.5 ENTER 0.5 ENTER GLIST produces the list {0.5 1. 1.5 2. 2.5}.
- To see step-by-step operation use the program DBUG for a short list, for example:

VAR 1 SPC 1.5 SPC 0.5 ENTER	Enter parameters 1 1.5 0.5
['] GLIST ENTER	Enter the program name in level 1
← PRG NXT NXT RUN DBG	Start the debugger.

Use SST↓ to step into the program and see the detailed operation of each command.

Errors and error trapping

The functions of the PRG/ERROR sub-menu provide ways to manipulate errors in the calculator, and trap errors in programs. The PRG/ERROR sub-menu, available through ← PRG NXT NXT ERROR , contains the following functions and sub-menus:



DOERR

This function executes an user-define error, thus causing the calculator to behave as if that particular error has occurred. The function can take as argument either an integer number, a binary integer number, an error message, or the number zero (0). For example, in RPN mode, entering 5 ENTER DOERR, produces the following error message: *Error: Memory Clear*

If you enter #11h ENTER DOERR, produces the following message: *Error: Undefined FPTR Name*

If you enter "TRY AGAIN" ENTER DOERR, produces the following message: *TRY AGAIN*

Finally, 0 ENTER DOERR, produces the message: *Interrupted*

ERRN

This function returns a number representing the most recent error. For example, if you try 0 1/x ON ERRN, you get the number #305h. This is the binary integer representing the error: *Infinite Result*

ERRM

This function returns a character string representing the error message of the most recent error. For example, in Approx mode, if you try 0 1/x ON ERRM, you get the following string: *"Infinite Result"*

ERR0

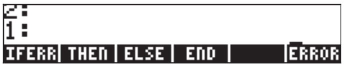
This function clears the last error number, so that, executing ERRN afterwards, in Approx mode, will return # 0h. For example, if you try 0 1/x ON ERR0 ERRN, you get # 0h. Also, if you try 0 1/x ON ERR0 ERRM, you get the empty string " ".

LASTARG

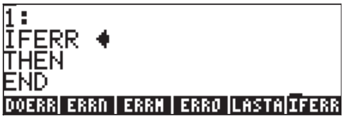
This function returns copies of the arguments of the command or function executed most recently. For example, in RPN mode, if you use: 3 ÷ 2 ENTER, and then use function LASTARG (LASTA), you will get the values 3 and 2 listed in the stack. Another example, in RPN mode, is the following: 5 TAN ENTER. Using LASTARG after these entries produces a 5.

Sub-menu IFERR

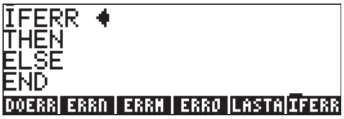
The IFERR sub-menu provides the following functions:



These are the components of the IFERR … THEN … END construct or of the IFERR … THEN … ELSE … END construct. Both logical constructs are used for trapping errors during program execution. Within the ERROR sub-menu, entering ↰ IFERR, or ↱ IFERR, will place the IFERR structure components in the stack, ready for the user to fill the missing terms, i.e.,



```
1:
IFERR ◆
THEN
END
```



```
IFERR ◆
THEN
ELSE
END
```

The general form of the two error-trapping constructs is as follows:

IF trap-clause THEN error-clause END

IF trap-clause THEN error-clause ELSE normal-clause END

The operation of these logical constructs is similar to that of the IF … THEN … END and of the IF … THEN … ELSE … END constructs. If an error is detected during the execution of the trap-clause, then the error-clause is executed. Otherwise, the normal-clause is executed.

As an example, consider the following program (ERR1) that takes as input two matrices, A and b, and checks if there is an error in the trap clause: A b / (RPN mode, i.e., A/b). If there is an error, then the program calls function LSQ (Least SQuares, see Chapter 11) to solve the system of equations:

```
« → A b « IFERR A b / THEN LSQ END » »
```

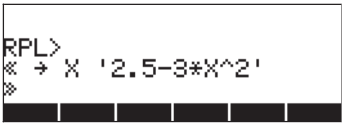
Try it with the arguments A = [[2, 3, 5] , [1, 2, 1]] and b = [[5] , [6]]. A simple division of these two arguments produces an error: */Error: Invalid Dimension.*

However, with the error-trapping construct of the program, ERR1, with the same arguments produces: [0.262295…, 0.442622…].

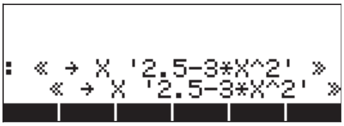
User RPL programming in algebraic mode

While all the programs presented earlier are produced and run in RPN mode, you can always type a program in User RPL when in algebraic mode by using function RPL>. This function is available through the command catalog. As an example, try creating the following program in algebraic mode, and store it into variable P2: `« → X '2.5-3*X^2' »`

First, activate the RPL> function from the command catalog (→ _CAT). All functions activated in ALG mode have a pair of parentheses attached to their name. The RPL> function is not exception, except that the parentheses must be removed before we type a program in the screen. Use the arrow keys (◀ ▶) and the delete key (◀) to eliminate the parentheses from the RPL>() statement. At this point you will be ready to type the RPL program. The following figures show the RPL> command with the program before and after pressing the ENTER key.



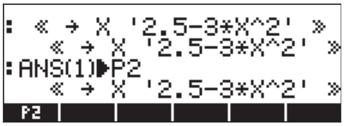
```
RPL>
« → X '2.5-3*X^2'
»
```



```
: « → X '2.5-3*X^2' »
  « → X '2.5-3*X^2' »
```

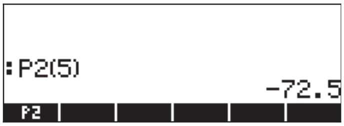
To store the program use the STO command as follows:

← ANS STO▶ ALPHA P 2 ENTER



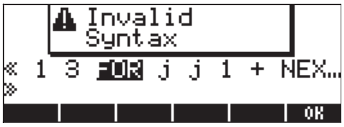
```
: « → X '2.5-3*X^2' »
  « → X '2.5-3*X^2' »
:ANS(1)▶P2
  « → X '2.5-3*X^2' »
```

An evaluation of program P2 for the argument X = 5 is shown in the next screen:



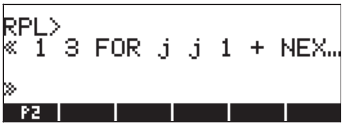
```
:P2(5)
                  -72.5
```

While you can write programs in algebraic mode, without using the function RPL>, some of the RPL constructs will produce an error message when you press ENTER, for example:

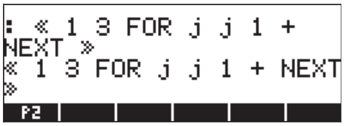


```
⚠ Invalid
  Syntax
« 1 3 FOR j j 1 + NEX…
»
```

Whereas, using RPL, there is no problem when loading this program in algebraic mode:



```
RPL>
« 1 3 FOR j j 1 + NEX…
»
```



```
: « 1 3 FOR j j 1 +
NEXT »
« 1 3 FOR j j 1 + NEXT
»
```