

Chapter 21

用户RPL语言编程

用户RPL语言是最常用于编程计算器的编程语言。程序组件可以通过在程序容器«»之间以适当的顺序包含在行编辑器中。由于计算机用户在RPN模式编程中有更多的经验，本章中的大部分示例都将以RPN模式呈现。此外，为便于输入编程命令，我们建议您将系统标志117设置为SOFT菜单。一旦在RPN模式下调试和测试，程序在ALG模式下的工作效果相同。如果您更喜欢在ALG模式下工作，只需学习如何在RPN中进行编程，然后将操作模式重置为ALG以运行程序。有关ALG模式下用户RPL编程的简单示例，请参阅本章的最后一页。




编程的一个例子

在本指南的前几章中，我们提出了许多可用于各种应用程序的程序（例如，程序CRMC和CRMT，用于创建多个列表中的矩阵，在第10章中介绍）。在本节中，我们将介绍一个简单的程序，介绍与计算器编程相关的概念。我们将编写的程序将用于定义函数 $f(x) = \sinh(x) / (1 + x^2)$ ，它接受列表作为参数（即，x可以是数字列表，如第8章所述）。在第8章中，我们指出加号，作为列表的串联运算符，而不是产生逐项总和。相反，您需要使用ADD运算符来实现列表的逐项求和。因此，要定义上面显示的函数，我们将使用以下程序：

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

要键入程序，请按照以下说明操作：

Keystroke sequence:












⏏ « »
['] ALPHA ⏏ (X) ⏏ STO
ALPHA ⏏ (X)
⏏ MTH   
/ SPC ALPHA ⏏ (X) ⏏ x^2




Produces:














«
'x' STO
x
SINH
1 x SQ

Interpreted as:


启动RPL计划
将级别1存储到变量x中
将x放在第1级
计算1级的sinh
输入1并计算x2

 MTH 	ADD	Calculate $(1+x^2)$,
	/	then divide
 ALPHA  X 	'x'	
 PRG   	PURGE	Purge variable x
		Program in level 1

要保存程序，请使用:  ALPHA  G 

Press  恢复变量菜单，并通过在级别1 (   ) 中输入参数值然后按  来评估 $g(3.5)$ 。结果是 1.2485, 即 $g(3.5) = 1.2485$ 。尝试获取 $g(\{1\ 2\ 3\})$ ，输入显示的第1级列表       and pressing . 如果您的CAS设置为EXACT模式，结果现在是 $\{\text{SINH}(1)/2\ \text{SINH}(2)/5\ \text{SINH}(3)/10\}$ ，如果您的CAS设置为APPROXIMATE模式，结果将为 $\{0.5876.. \ 0.7253... \ 1.0017...\}$.

全局和局部变量和子程序

上面定义的程序 , 可以显示为

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




用  .














请注意，程序使用变量名称 x 来存储通过编程步骤 'x' STO 放置在堆栈1级的值。在程序执行时，变量 x 作为先前存储的任何其他变量存储在变量菜单中。计算完该函数后，程序将清除（擦除）变量 x ，使其在完成程序评估后不会显示在变量菜单中。如果我们不在程序中清除变量 x ，那么在程序执行后我们可以使用它的值。因此，在该程序中使用的变量 x 被称为全局变量。使用 x 作为全局变量的一个含义是，如果我们先前定义了一个名为 x 的变量，它的值将被程序使用的值替换，然后在程序执行后从变量菜单中完全删除。

因此，从编程的角度来看，全局变量是a程序执行后用户可以访问的变量。有可能

可以在程序中使用仅为该程序定义的局部变量，并且在程序执行后将无法使用。之前的程序可以修改为：

« → x « x SINH 1 x SQ ADD / » »

箭头符号 (→) 是通过将右移键  与 0 键  key, i.e.,  → 组合而获得的。另外，请注意，在主程序中还有一组额外的编程符号 («») 表示存在子程序，即« x SINH 1 x SQ ADD / »主程序以组合→x开始，表示将堆栈1级的值分配给局部变量x。然后，编程流程在子程序内继续进行，方法是将x放入堆栈中，评估SINH (x)，将1放入堆栈，将x放入堆栈，平方x，将x加1，并将堆栈级别2分开 (SINH (x)) 通过堆栈级别1 (1 + x²)。程序控制然后传递回主程序，但第一组结束编程符号 (») 和第二组之间不再有命令，因此，程序终止。堆栈中的最后一个值，即SINH (x) / (1 + x²)，作为程序输出返回。程序的最后一个版本中的变量x永远不会占用变量菜单中变量之间的位置。它在计算器内存中运行，不会影响变量菜单中任何类似命名的变量。因此，在这种情况下，变量x被称为程序的局部变量，即局部变量。

Note: 要修改程序，请将程序名放在堆栈中(  )，然后使用  。使用箭头键 (   )移动程序。使用退格键/删除键删除任何不需要的字符。要添加程序容器 (即«»), 请使用 «», 因为这些符号成对出现，您必须在子程序的开头和结尾输入它们，并使用删除键删除其中一个组件以生成所需的计划，即：

« → x « x SINH 1 x SQ ADD / » ».

完成编辑程序后按`ENTER`。修改后的程序存储回变量 `■`。

全局变量范围

从程序开发的角度来看，您在HOME目录或任何其他目录或子目录中定义的任何变量都将被视为全局变量。但是，此类变量的范围，即目录树中可访问变量的位置，将取决于树中变量的位置（请参阅第2章）。

确定变量范围的规则如下：全局变量可供定义它的目录和附加到该目录的任何子目录访问，除非所考虑的子目录中存在具有相同名称的变量。该规则的后果如下：

- 除非在目录或子目录中重新定义，否则可以从HOME中的任何目录访问HOME目录中定义的全局变量。
- 如果在目录或子目录中重新定义变量，则此定义优先于当前目录上的任何其他定义。
- 运行引用给定全局变量的程序时，程序将使用调用程序的目录中的全局变量的值。如果调用目录中不存在具有该名称的变量，程序将搜索当前目录上的目录，直到HOME目录，并在当前目录上方的最近目录中使用与所考虑的变量名对应的值。

可以从该目录或其任何子目录访问在给定目录中定义的程序。



所有这些规则可能会让新计算器用户感到困惑 它们都可以简化为以下建议：创建具有有意义名称的目录和子目录来组织数据，并确保在正确的子目录中拥有所需的所有全局变量。

局部变量范围

局部变量仅在程序或子程序中有效。 因此，它们的范围仅限于定义它们的程序或子程序。 局部变量的一个例子是FOR循环中的索引（在本章后面描述），例如 « → n x « 1 n FOR j x NEXT n →LIST » »

PRG菜单

在本节中，我们将计算器的系统标志117设置为SOFT菜单，显示PRG（编程）菜单的内容。 使用此标志设置PRG菜单中的子菜单和命令将显示为软菜单标签。 这有助于在组合程序时在行编辑器中输入编程命令。

要访问PRG菜单，请使用击键组合  PRG。在PRG菜单中，我们识别以下子菜单（按  移动到PRG菜单中的下一个子菜单集合）：











以下是这些子菜单及其子菜单内容的简要说明：

- STACK: 用于操作RPN堆栈元素的函数
- MEM: 与内存操作相关的函数
- DIR: 与操作目录相关的函数
- ARITH: 用于操纵存储在变量中的索引的函数
- BRCH: 具有程序分支和循环功能的子菜单的集合
- IF: IF-THEN-ELSE-END结构用于分支
- CASE: 用于分支的CASE-THEN-END构造

START: START-NEXT-STEP构造用于分支
FOR: 循环的FOR-NEXT-STEP构造
DO: DO-UNTIL-END构造for循环
WHILE: 循环的WHILE-REPEAT-END构造
TEST: 比较运算符, 逻辑运算符, 标志测试函数
TYPE: 用于转换对象类型, 拆分对象等的函数
LIST: 与列表操作相关的功能
ELEM: 用于操作列表元素的函数
PROC: 将过程应用于列表的功能
GROB: 用于处理图形对象的函数
PICT: 在图形屏幕中绘制图片的功能
CHARS: 字符串操作的函数
MODES: 修改计算器模式的功能
FMT: 要更改数字格式, 请使用逗号格式
ANGLE: 更改角度测量和坐标系
FLAG: 设置和取消设置标志并检查其状态
KEYS: 定义和激活用户定义的密钥 (第20章)
MENU: 定义和激活自定义菜单 (第20章)
MISC: 其他模式更改 (蜂鸣声, 时钟等)
IN: 程序输入的功能
OUT: 程序输出的功能
TIME: 与时间相关的功能
ALRM: 报警操作
ERROR: 错误处理的功能
IFERR: IFERR-THEN-ELSE-END构造用于错误处理
RUN: 用于运行和调试程序的函数

浏览RPN子菜单

从击键组合  **PRG** 开始, 然后按相应的软菜单键(e.g., ). 如果要访问该子菜单中的子菜单(e.g.,  within the  sub-menu), 子菜单中, 请按相应的键。 要在子菜单中向上移动, 请按  键, 直到在(e.g.,  within the  sub-menu) or to the PRG menu (i.e., ).

子菜单列出的功能

以下是子菜单列出的PRG子菜单中的功能列表。

STACK	MEM/DIR	BRCH/IF	BRCH/WHILE	TYPE
DUP	PURGE	IF	WHILE	OBJ→
SWAP	RCL	THEN	REPEAT	→ARRAY
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	ERRO	NEXT
TICKS	LASTARG	HALT
		KILL
TIME/ALRM	ERROR/IFERR	OFF
ACK	IFERR	
ACKALARM	THEN	
STOALARM	ELSE	
RCLALARM	END	
DELALARM		
FINDALARM		

PRG菜单中的快捷方式

上面列出的PRG菜单的许多功能都可以通过其他方式获得：

- 键盘中提供了比较运算符 (\neq , \leq , $<$, \geq , $>$)
- **MODE** 键提供的输入功能，可以激活MODES子菜单中的许多功能和设置。
- **TIME** 访问TIME子菜单中的功能。
- 功能STO和RCL（在MEM / DIR子菜单中）可通过键 **STOP** and **RCL** 使用。
- 功能RCL和PURGE（在MEM / DIR子菜单中）可通过TOOL菜单 (**TOOL**) 获得。
- 在BRCH子菜单中，在按下任何子菜单键之前按下左移键 (**←**) or (**→**) 将创建与所选子菜单键相关的构造。 这仅适用于RPN模式下的计算器。 示例如下所示：

← 

```

1:
IF ◀
THEN
END

```

IF CASE START FOR DO WHILE


← 

```

CASE ◀
THEN
END

```

IF CASE START FOR DO WHILE

← 

```

IF ◀
THEN
ELSE
END

```

IF CASE START FOR DO WHILE

← 

```

2:
1:
THEN
END

```

IF CASE START FOR DO WHILE

← 

```

2:
1:
START ◀
NEXT

```

IF CASE START FOR DO WHILE

← 

```

2:
1:
FOR ◀
NEXT

```

IF CASE START FOR DO WHILE

← 

```

2:
1:
START
STEP

```

IF CASE START FOR DO WHILE

← 

```

2:
1:
FOR ◀
STEP

```

IF CASE START FOR DO WHILE

← 

```

1:
DO ◀
UNTIL
END

```

IF CASE START FOR DO WHILE

← 

```

1:
WHILE ◀
REPEAT
END

```

IF CASE START FOR DO WHILE




请注意，插入提示符(◀) 在每个构造的关键字之后可用，因此您可以在正确的位置开始键入。

常用命令的击键序列

以下是用于访问PRG菜单中用于数值编程的常用命令的击键序列。 这些命令首先按菜单列出：

STACK

DUP
SWAP
DROP

 PRG **STACK** **DUP**
 PRG **STACK** **SWAP**
 PRG **STACK** **DROP**

NEW ORDER

PURGE
ORDER

 PRG **NEW ORDER** **PURGE**
 PRG **NEW ORDER** **ORDER**


STACK IF

IF
THEN
ELSE
END

 PRG **STACK IF** **IF**
 PRG **STACK IF** **THEN**
 PRG **STACK IF** **ELSE**
 PRG **STACK IF** **END**

STACK CASE

CASE
THEN
END

 PRG **STACK CASE** **CASE**
 PRG **STACK CASE** **THEN**
 PRG **STACK CASE** **END**

STACK START

START
NEXT
STEP

 PRG **STACK START** **START**
 PRG **STACK START** **NEXT**
 PRG **STACK START** **STEP**

STACK FOR

FOR
NEXT
STEP

 PRG **STACK FOR** **FOR**
 PRG **STACK FOR** **NEXT**
 PRG **STACK FOR** **STEP**

STACK DO

DO
UNTIL
END

 PRG **STACK DO** **DO**
 PRG **STACK DO** **UNTIL**
 PRG **STACK DO** **END**

8801 WHILE

WHILE
REPEAT
END

← PRG 8801 WHILE WHILE
← PRG 8801 WHILE REPEAT
← PRG 8801 WHILE END

1131

==
AND
OR
XOR
NOT
SAME
SF
CF
FS?
FC?
FS?C
FC?C

← PRG 1131 /
← PRG 1131 NXT 000
← PRG 1131 NXT 001
← PRG 1131 NXT 002
← PRG 1131 NXT 003
← PRG 1131 NXT 004
← PRG 1131 NXT 005
← PRG 1131 NXT NXT 006
← PRG 1131 NXT NXT 007
← PRG 1131 NXT NXT 008
← PRG 1131 NXT NXT 009
← PRG 1131 NXT NXT 010
← PRG 1131 NXT NXT 011

1132

OBJ→
→ARRY
→LIST
→STR
→TAG
NUM
CHR
TYPE

← PRG 1132 032 →
← PRG 1132 I → 033
← PRG 1132 I → 034
← PRG 1132 I → 035
← PRG 1132 I → 036
← PRG 1132 I → 037
← PRG 1132 NXT NUM
← PRG 1132 NXT CHR
← PRG 1132 NXT TYPE

1133 1134

GET
GETI
PUT
PUTI
SIZE
HEAD
TAIL

← PRG 1133 1134 038
← PRG 1133 1134 039
← PRG 1133 1134 040
← PRG 1133 1134 041
← PRG 1133 1134 042
← PRG 1133 1134 NXT HEAD
← PRG 1133 1134 NXT TAIL

Test Prog

REVLIST
SORT
SEQ

← PRG Test Prog REVLIST
← PRG Test Prog NXT SORT
← PRG Test Prog NXT SEQ

Modes Angle

DEG
RAD

← PRG NXT Modes Angle DEG
← PRG NXT Modes Angle RAD

Modes Menu

CST
MENU
BEEP

← PRG NXT Modes Menu CST
← PRG NXT Modes Menu MENU
← PRG NXT Modes Menu BEEP

Unit

INFORM
INPUT
MSGBOX
PVIEW

← PRG NXT Unit INFORM
← PRG NXT Unit INPUT
← PRG NXT Unit MSGBOX
← PRG NXT Unit PVIEW

Run

DEBUG
SST
SST↓
HALT
KILL

← PRG NXT NXT Run DEBUG
← PRG NXT NXT Run SST
← PRG NXT NXT Run SST↓
← PRG NXT NXT Run HALT
← PRG NXT NXT Run KILL

用于生成数字列表的程序

请注意，PRG菜单中的功能不是可用于编程的唯一功能。事实上，计算器中的几乎所有功能都可以包含在程序中。因此，您可以使用，例如，

MTH菜单中的功能。 具体来说，您可以使用通过MTH / LIST菜单提供的列表操作功能，如SORT，ΣLIST等。

作为额外的编程练习，并尝试上面列出的击键序列，我们在此提出了三个用于创建或操纵列表的程序。 节目名称和列表如下：

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 »

这些操作的运作如下：

- (1) **LISC:** 创建一个n个元素的列表，所有元素都等于常量c。 操作：输入n，输入c，按 
Example: 5  6.5   creates the list: {6.5 6.5 6.5 6.5 6.5}
- (2) **CRLST:** 创建一个从n1到n2的数字列表，增量为Δn，即， $\{n_1, n_1+\Delta n, n_1+2\cdot\Delta n, \dots, n_1+N\cdot\Delta n\}$, where $N=\text{floor}((n_2-n_1)/\Delta n)+1$. 操作：输入n1，输入n2，输入Δn，按 
Example: .5  3.5  .5   produces: {0.5 1 1.5 2 2.5 3 3.5}
- (3) **CLIST:** 创建一个列表，其中包含元素的累积总和，即原始元素 list is $\{x_1, x_2, x_3, \dots, x_N\}$, then CLIST creates the list:


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

操作：将原始列表放在1级，按 .

Example: {1 2 3 4 5}   produces {1 3 6 10 15}.

通常，程序是程序容器E和»之间包含的任何计算器指令序列。


顺序编程的例子

In general, a program is any sequence of calculator instructions enclosed between the program containers  and ». 子程序可以作为程序的一部分包含在内。 本指南前面提到的例子（例如，在第3章和第8章中）6可以基本上分为两类：（a）通过定义函数生成的程序；（b）模拟一系列堆栈操作的程序。 接下来描述这两种类型的程序。 这些程序的一般形式是输入？过程输出，因此，我们将它们称为顺序程序。


通过定义函数生成的程序

这些是通过使用函数DEFINE( DEF) 与以下形式的参数生成的程序：

'function_name(x₁, x₂, ...) = expression containing variables x₁, x₂, ...'

该程序存储在名为function_name的变量中。 通过使用 function_name. 将程序调用到堆栈时。 该计划显示如下：

« → x₁, x₂, ... 'expression containing variables x₁, x₂, ...' ».

要在RPN模式下评估一组输入变量x₁, x₂, ... 的函数，请按适当的顺序将变量输入堆栈（即先x₁，然后是x₂，然后是x₃等），然后按 标有function_name的软菜单键。计算器将返回函数function_name (x₁, x₂, ...) 的值。

示例：宽矩形通道的Manning方程。

例如，考虑以下等式，使用Manning方程计算宽矩形明渠中的单位放电（每单位宽度的放电）q：





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

其中Cu是一个常数，取决于所用单位的系统[国际体系（SI）单位Cu = 1.0，英国体系（ES）单位Cu = 1.486]，n是曼宁的阻力系数，取决于通道衬砌的类型和其他因素，y0是流动深度，S0是作为无量纲分数给出的通道床斜率。

Note: 曼宁系数的值n在表格中可用作无量纲数，通常在0.001到0.5之间。Cu的值也可以在没有尺寸的情况下使用。但是，应注意确保y0的值具有适当的单位，即S.I中的m和E.S中的ft。q的结果以相应系统的适当单位返回，即S.I中的m² / s和E.S中的ft² / s。因此，Manning方程在尺寸上不一致。

假设我们想要创建函数q（Cu，n，y0，S0）来计算这种情况下的单位放电q。
使用表达式

$$'q(Cu,n,y0,S0)=Cu/n*y0^{(5./3.)*\sqrt{S0}'$$

作为函数DEFINE的参数。 请注意，等式中的指数5./3.表示由小数点引起的实数比。 如果需要，按  键恢复变量列表。 此时，软菜单键标签中会有一个名为  的变量。 要查看q的内容，请使用  。 通过定义函数 q（Cu，n，y0，S0）生成的程序如下所示：

$$« \rightarrow Cu\ n\ y0\ S0\ 'Cu/n*y0^{(5./3.)*\sqrt{S0}' \gg.$$

这将被解释为“按顺序输入Cu，n，y0，S0，然后计算引号之间的表达式。”例如，计算Cu = 1.0，n = 0.012，y0 = 2 m和S0的q 在RPN模式下使用 = 0.0001，使用：

1  0.012  2  0.0001  

The result is 2.6456684 (or, q = 2.6456684 m²/s).

您还可以将输入数据与单个堆栈行中的空格分开，而不是使用(ENTER)。

模拟一系列堆栈操作的程序

在这种情况下，假设操作序列中涉及的术语存在于堆栈中。首先打开程序容器，然后输入程序 (P) «». 接下来，输入要执行的操作序列。输入所有操作后，按(ENTER) 完成程序。如果这是一次性程序，您可以在此时按 (EVAL) 以使用可用的输入数据执行程序。如果它是一个永久性程序，则需要将其存储在变量名中。

描述这种类型的程序的最佳方式是一个例子：

示例：矩形通道的速度头。

假设我们想要计算宽度为b的矩形通道中的速度头 h_v ，其流动深度为y，带有放电Q。The

比能量计算为 $h_v = Q^2 / (2g(by)^2)$ ，其中g是加速度重力 ($g = 9.806 \text{ m/s}^2$ in S.I. units or $g = 32.2 \text{ ft/s}^2$ in E.S. units). If we were to calculate h_v for $Q = 23 \text{ cfs}$ (cubic feet per second = ft^3/s), $b = 3 \text{ ft}$, and $y = 2 \text{ ft}$, we would use: $h_v = 23^2 / (2 \cdot 32.2 \cdot (3 \cdot 2)^2)$. 以交互方式使用RPN模式计算器，我们可以将此数量计算为：

2 (ENTER) 3 (×) (←) x^2 3 2 (·) 2 (×)
2 (×) 2 3 (←) x^2 (▶) (÷)

Resulting in 0.228174, or $h_v = 0.228174$.

为了将这个计算作为一个程序放在一起，我们需要在堆栈中按照它们在计算中使用的顺序输入数据 (Q, g, b, y)。就变量Q, g, b和y而言，刚刚执行的计算被写为（不要键入以下内容）：

y (ENTER) b (×) (←) x^2 g (×) 2 (×) Q (←) x^2 (▶) (÷)

Note: SQ 是键击序列产生的功能 $\leftarrow x^2$.

将程序保存到名为hv的变量中:

\leftarrow ALPHA \leftarrow H ALPHA \leftarrow V STOP

软键菜单中应该有一个新变量 \leftarrow (按下 \leftarrow VAR 查看变量列表。) 可以使用 EVAL 函数评估堆栈中剩余的程序。 结果应该是 0.228174..., 和以前一样。此外, 该程序可供将来使用变量 \leftarrow . For example, for $Q = 0.5 \text{ m}^3/\text{s}$, $g = 9.806 \text{ m/s}^2$, $b = 1.5 \text{ m}$, and $y = 0.5 \text{ m}$, use:

0.5 \leftarrow 9.806 \leftarrow 1.5 \leftarrow 0.5 \leftarrow

Note: \leftarrow 在这里用作 \leftarrow 输入数据输入的替代。

The result now is 2.26618623518E-2, i.e., $h_v = 2.26618623518 \times 10^{-2} \text{ m}$.

Note: 由于在 \leftarrow 中编程的等式在尺寸上是一致的, 我们可以在输入中使用单位。

如前所述, 本节介绍的两种类型的程序是顺序程序, 在程序流程遵循单一路径的意义上, 即 INPUT \rightarrow OPERATION \rightarrow OUTPUT. 使用菜单 \leftarrow PRG \leftarrow 中的命令可以实现程序流程的分支。关于程序分支的更多细节如下所示。

程序中的交互式输入






在上一节所示的顺序程序示例中, 用户并不总是清楚在程序执行之前必须将变量放入堆栈的顺序。对于程序 \leftarrow 的情况, 写为


$\leftarrow \rightarrow \text{Cu } n \text{ } y0 \text{ } S0 \text{ 'Cu/n*y0}^{(5/3)}*\sqrt{S0}' \text{ } \leftarrow$,

始终可以将程序定义调回到堆栈中( ) 以查看必须输入变量的顺序，即 $\rightarrow Cu \ n \ y0 \ S0$. 但是，对于程序  的情况，它的定义

$$\ll * SQ * 2 * SWAP SQ SWAP / \gg$$

没有提供必须输入数据的顺序的线索，当然，除非您对RPN和用户RPL语言非常有经验。

检查程序结果作为公式的一种方法是在堆栈中输入符号变量而不是数值结果，并让程序对这些变量进行操作。为了使这种方法有效，计算器的CAS（计算器代数系统）必须设置为symbolic and exact模式。这是通过使用   , 来实现的，并确保删除选项 _Numeric and _Approx 中的复选标记。按   返回正常的计算器显示。按下  显示变量菜单。

我们将使用后一种方法来检查使用程序 得到的公式结果如下：我们知道程序有四个输入，因此，我们使用符号变量S4，S3，S2和S1来 表示输入时的堆栈级别：

Next, press . The resulting formula may look like this

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

如果您的显示器未设置为教科书样式，或者像这样，

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

如果选择了教科书样式。 由于我们知道函数SQ（）代表x²，我们将后面的结果解释为








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

它表示公式中不同堆栈输入级别的位置。通过将此结果与我们编程的原始公式进行比较，即，

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

我们发现必须在堆栈级别1 (S1) 中输入y，在堆栈级别2 (S2) 中输入b，在堆栈级别3 (S3) 中输入g，在堆栈级别4中输入Q (S4)。

提示输入字符串

这两种识别输入数据顺序的方法效率不高。但是，您可以通过使用变量名称提示他或她来帮助用户识别要使用的变量。从用户RPL语言提供的各种方法来看，最简单的方法是使用输入字符串和函数INPUT ( PRG      



结果是一个堆栈提示用户输入a的值并将光标放在提示符前面：a：输入a的值，比如35，然后按 **ENTER**。结果是输入字符串：a：堆栈级别1中的35



带有输入字符串的函数

如果您要使用这段代码来计算函数， $f(a) = 2 * a^2 + 3$ ，您可以将程序修改为如下所示：

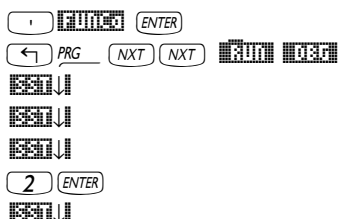
```
« "Enter a: " { "←a: " {2 0} V }
INPUT OBJ→ → a « '2*a^2+3' » »
```

将此新程序保存在名称 'FUNCa' (FUNCTION of a):

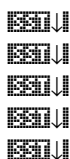
按 **2nd** **F1** 运行程序。当提示输入 **2nd** **F1** 的值时，例如2，然后按 **ENTER**。结果只是代数 $2a^2 + 3$ ，这是一个不正确的结果。计算器提供调试程序的功能，以识别程序执行期间的逻辑错误，如下所示。

调试程序

为了弄清楚它为什么不起作用，我们在计算器中使用DEBUG函数如下：

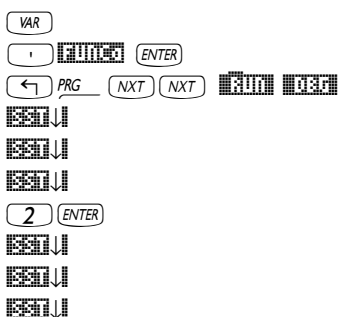


Copies program name to stack level 1
Starts debugger
Step-by-step debugging, result: "Enter a:"
Result: {" ←a:" {2 0} V}
Result: user is prompted to enter value of a
Enter a value of 2 for a. Result: "←a:2"
Result: a:2



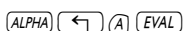
Result: empty stack, executing $\rightarrow a$
Result: empty stack, entering subprogram «
Result: '2*a^2+3'
Result: '2*a^2+3', leaving subprogram »
Result: '2*a^2+3', leaving main program»

进一步按 软菜单键不再产生输出，因为我们已经逐步完成整个程序。这个运行调试器没有提供任何关于为什么程序没有为 $a = 2$ 计算 $2a^2 + 3$ 的值的消息。为了看看子程序中 a 的值是多少，我们需要再次运行调试器 评估子计划内的一个。请尝试以下方法：



Recovers variables menu
Copies program name to stack level 1
Starts debugger
Step-by-step debugging, result: "Enter a:"
Result: {" $\leftarrow a$:" {2 0} V}
Result: user is prompted to enter value of a
Enter a value of 2 for a. Result: " $\leftarrow a$:2"
Result: a:2
Result: empty stack, executing $\rightarrow a$
Result: empty stack, entering subprogram «

此时我们在子程序subprogram « '2*a^2+3' » 内，它使用局部变量 a 。要查看使用的价值：



这确实表明了局部变量 $a = 2$

让我们在这点上杀掉调试器，因为我们已经知道了我们将得到的结果。要终止调试器，请按 。您收到 <I> Interrupted 消息，确认已杀死调试器。按 恢复正常的计算器显示。

Note: 在调试模式下，每次按 显示屏的左上角都会显示正在执行的程序步骤。可以使用名为 在PRG菜单中的 子菜单下。这可以用于立即执行从主程序内调用的任何子程序。 的应用示例将在稍后显示。

修复程序

对于程序产生数值结果失败的唯一可能的解释似乎是在代数表达式'2 * a ^ 2 + 3'之后缺少命令→NUM， 让我们通过添加缺少的EVAL函数来编辑程序。
编辑后的程序应如下所示：

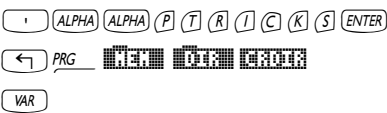
```
« "Enter a: " { "←a: " { 2 0 } V } INPUT
OBJ→ → a « '2*a^2+3' →NUM » »
```

将其再次存储在变量FUNCa中，并以a = 2再次运行程序。这次，结果为11，即2 * 2² + 3 = 11。

输入两个或三个输入值的字符串

在本节中，我们将在目录HOME中创建一个子目录，以保存一个，两个和三个输入数据值的输入字符串示例。 这些将是通用输入字符串，可以包含在任何未来的程序中，负责根据每个程序的需要更改变量名称。

让我们开始创建一个名为PTRICKS（Programming TRICKS）的子目录，以保存我们以后可以借用的编程花絮，以便在更复杂的编程练习中使用。 要创建子目录，首先要确保移动到HOME目录。 在HOME目录中，使用以下按键创建子目录PTRICKS：



输入目录名称'PTRICKS'
创建目录
恢复变量列表

程序可能具有3个以上的输入数据值。 当使用输入字符串时，我们希望一次将输入数据值的数量限制为5，原因很简单，一般来说，我们只能看到7个堆栈级别。 如果我们使用堆栈级别7为输入字符串赋予标题，并将堆栈级别6留空以便于读取显示，我们只有堆栈级别1到5来定义输入变量。

输入两个输入值的字符串程序

两个输入值的输入字符串程序，例如a和b，如下所示：

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

通过修改INPTa的内容可以轻松创建该程序。将程序存储到变量INPT2中。

应用：评估两个变量的函数

考虑理想气体定律， $pV = nRT$ ，where p = gas pressure (Pa), V = gas volume(m^3), n = number of moles (gmol), R = universal gas constant = $8.31451 \text{ J/(gmol} \cdot \text{K)}$, and T = absolute temperature (K).

我们可以将压力 p 定义为两个变量 V 和 T 的函数，对于给定质量的气体， $p(V, T) = nRT / V$ ，因为 n 也将保持不变。假设 $n = 0.2 \text{ gmol}$ ，那么编程的程序是

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

我们可以通过键入以下程序来定义函数


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

并将其存储到变量INPT2中。

下一步是添加输入字符串，该字符串将提示用户输入 V 和 T 的值。要创建此输入流，请修改INPT2中的程序以读取：

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

将新程序存回变量INPT2。按F5运行该程序。在输入字符串中输入 $V = 0.01 \text{ m}^3$ 和 $T = 300 \text{ K}$ 的值，然后按

 . 结果是49887.06_J / m ^ 3。J / m ^ 3的单位相当于帕斯卡 (Pa) , 它是S.I.系统中的优选压力单位。

Note: 因为我们故意在函数定义中包含单位, 所以输入值必须在输入中附加单位才能产生正确的结果。

三个输入值的输入字符串程序

三个输入值的输入字符串程序, 例如a, b和c, 如下所示:

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

通过修改INPT2的内容使其看起来如上图所示, 可以轻松创建该程序。然后, 生成的程序可以存储在名为INPT3的变量中。通过该程序, 我们完成了输入字符串程序的集合, 这些程序允许我们输入一个, 两个或三个数据值。将这些程序作为参考并复制和修改它们以满足您编写的新程序的要求。

应用: 评估三个变量的函数

假设我们想要编程包括摩尔数理想气体定律, N, 作为附加的变量, 即, 我们要定义的函数

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

并修改它以包含三变量输入字符串。将此函数组合在一起的过程与先前在定义函数p (V, T) 时使用的过程非常相似。 生成的程序如下所示:

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

将此结果存储回变量 . 要运行程序, 请按 .

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
p |FUNC|INFTa|
```

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

通过输入表格输入

函数 **INFORM** (**←** **PRG** **→** **NXT** **TABLE**) 可用于为程序创建详细的输入表单。功能**INFORM**需要五个参数，顺序如下：

1. 标题：描述输入表格的字符串
2. 字段定义：包含一个或多个字段定义 $\{s_1\ s_2\ \dots\ s_n\}$ 的列表，其中每个字段定义 s_i 可以具有以下两种格式之一：
 - a. 一个简单的字段标签：一个字符串
 - b. 形式 $\{\text{"label"}\ \text{"helpInfo"}\ type_0\ type_1\ \dots\ type_n\}$ 的规范列表。
“label”是字段标签。“helpInfo”是一个详细描述字段标签的字符串，类型规范是字段允许的变量类型列表（有关对象类型，请参见第24章）。
3. 字段格式信息：单个数字 col 或列表 $\{col\ tabs\}$ 。在此规范中， col 是输入框中的列数， $tabs$ （可选）指定标签与表单中字段之间的制表位数。该列表可以是一个空列表。默认值为 $col = 1$ 且 $tabs = 3$ 。
4. 重置值列表：如果在使用输入表单时选择了~~TABLE~~选项，则列表包含重置不同字段的值。
5. 初始值列表：包含字段初始值的列表。

第4项和第5项中的列表可以是空列表。此外，如果没有为这些选项选择任何值，则可以使用NOVAL命令 ( PRG  NXT  ).

激活功能INFORM后，如果输入选项，则会得到一个零，或者在指定顺序和数字1的字段中输入值的列表，即在RPN堆栈中：

2:	{ v ₁ v ₂ ... v _n }
1:	1

因此，如果堆栈级别1中的值为零，则不执行任何输入，而此值为1时，输入值在堆栈级别2中可用。

例1 - 作为一个例子，考虑以下程序，INFP1（输入形式程序1）通过Chezy公式计算开放通道中的放电 $Q = C(R \cdot S)^{1/2}$ ，其中C是Chezy系数，通道表面粗糙度的函数（典型值80-150），R是通道的水力半径（长度），S是通道床的斜率（无量纲数，通常为0.01到0.000001）。以下程序通过函数INFORM定义输入表单：

```
« " 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 »
```

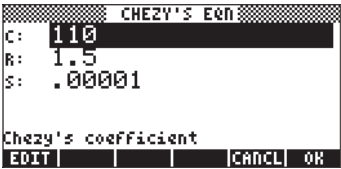
在程序中，我们可以识别输入的5个组件，如下所示：

1. 标题: " CHEZY'S EQN"
2. 字段定义: 有三个，标签为"C: ", "R: ", "S: ", 信息字符串"Chezy coefficient", "Hydraulic radius", "Channel bed slope", 仅接受数据 所有三个字段都输入0（实数）：

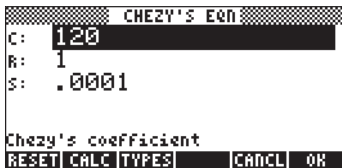
```
{ { "C:" "Chezy's coefficient" 0 } { "R:" "Hydraulic
radius" 0 } { "S:" "Channel bed slope" 0 } }
```


3. 字段格式信息: {} (空列表, 因此使用默认值)
4. 重置值列表: {120 1 .0001}
5. 初始值清单: {110 1.5 .00001}

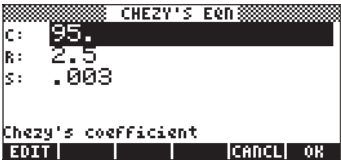
将程序保存到变量INFP1中。按运行程序。加载初始值的输入表单如下:




要查看重置这些值的效果, 请使用  (选择全部重置以重置字段值)。

现在, 输入三个字段的值, 例如, C = 95, R = 2.5和S = 0.003, 在输入每个新值后按 在这些替换之后, 输入表单将如下所示:



现在, 要将这些值输入程序, 请再次按 这将激活INFORM函数, 在堆栈中产生以下结果:



因此，我们演示了函数INFORM的使用。要了解如何在计算中使用这些输入值，请按如下方式修改程序：

```
« "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 »
```

在INFORM命令之后，上面显示的程序步骤包括使用IF-THEN-ELSE-END构造的决策分支（在本章的其他地方详细描述）。程序控制可以根据堆栈级别1中的值发送到两种可能中的一种。如果此值为1，则控制权将传递给命令：

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

这些命令将计算Q的值并为其添加标记（或标签）。另一方面，如果堆栈级别1中的值为0（在使用输入框时输入 **0.00001** 时发生），程序控制将传递给命令：


"Operation cancelled" MSGBOX

这些命令将生成一个消息框，指示操作已取消。

Note: 功能MSGBOX属于PRG / OUT子菜单下的输出功能集合。命令IF, THEN, ELSE, END 在 PRG/BRCH/IF 子菜单下可用。函数OBJ→, →TAG 在PRG/TYPE 子菜单下可用。函数 DROP 在PRG/STACK 子菜单下可用。函数 → and →NUM 在键盘上可用。

示例2 - 为了说明在函数INFORM的参数中使用第3项（字段格式信息），将程序INFP1中使用的空列表更改为{2 1}，意思是2，而不是默认的3列，只有一个 标签和值之间的制表位。将此新程序存储在变量INFP2中：

```
« " 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 »
```




运行程序  产生以下输入形式：



示例3 - 将字段格式信息列表更改为{3 0}并将修改后的程序保存到变量INFP3中。 运行此程序以查看新的输入表单：



创建一个选择框

函数CHOOSE ( PRG   CHOOSE) 允许用户在程序中创建一个选择框。 此函数需要三个参数：

1. 提示（描述选择框的字符串）
2. 选择定义列表{c1 c2 ... cn}。 选择定义ci可以具有以下两种格式中的任何一种：
 - a. 一个对象，例如数字，代数等，将在选择框中显示，也将是选择的结果。
 - b. 列表{object_displayed object_result}，以便在选择框中列出object_displayed，如果选择此选项，则选择object_result作为结果。
3. 一个数字，指示默认选项的选项定义列表中的位置。 如果此数字为0，则不突出显示默认选项。

如果使用 **ENTER** 则CHOOSE功能的激活将返回零，或者，如果做出选择，则选择所选择的选择（例如，v）和数字1，即，在RPN堆栈中：

2:	v
1:	1

示例1 - 用于计算明渠流量中的速度的Manning方程包括系数Cu，其取决于所使用的单位系统。 如果使用S.I. (Systeme International) , Cu = 1.0, 而如果使用E.S. (英文系统) , Cu= 1.486。 以下程序使用选择框让用户通过选择单位系统来选择Cu的值。 将其保存到变量CHP1（选择程序1）：

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

运行此程序（按**ENTER**）显示以下选择框：



取决于您是选择S.I.单位还是E.S. 单位，函数CHOOSE在堆栈级别2中放置值1或值1.486，在级别1中放置1。如果取消选择框，则CHOICE返回零（0）。
函数CHOOSE返回的值可由其他程序命令操作，如修改后的程序CHP2所示：

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



此新程序中CHOOSE功能之后的命令表示基于堆栈级别1通过IF-THEN-ELSE-END构造的值的决定。 如果堆栈级别1中的值为1，则命令“Cu” →TAG 将在屏幕中生成标记结果。 如果堆栈级别1中的值为零，则为











命令 “Operation cancelled” MSGBOX将显示一个消息框，指示操作已取消。

识别程序中的输出





识别数字程序输出的最简单方法是“标记”程序结果。 标签只是附加到数字或任何对象的字符串。 该字符串将是与该对象关联的名称。 例如，在早期，当调试程序INPTa（或INPT1）和INPT2时，我们获得了标记数字输出的结果，例如：a: 35。

标记数字结果

要标记数字结果，您需要将数字放在堆栈级别2中，将标记字符串放在堆栈级别2中，然后使用→TAG function (← PRG  →) 例如，要生成标记结果B: 5。请使用：






   ""       →

将标记的数字结果分解为数字和标记

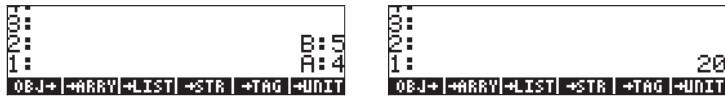
要将标记结果分解为其数值及其标记，只需使用函数 OBJ→ (← PRG   →). 使用→OBJ分解标记数字的结果是将数值放在堆栈级别2中，将标记放置在堆栈级别1中。如果您只想使用数值，那么您将使用退格键  删除标记。 例如，分解标记数量B: 5（见上文）将产生：



“取消标记”已标记的数量

“去标记”意味着从标记的数量中提取对象。 通过按键组合访问此功能：    . 例如，给定标记量a: 2，DTAG返回数值2。

Note: 对于带有标记数量的数学运算，计算器将在操作前自动“扣留”数量。例如，下面的左侧图显示了在RPN模式下按(×)键之前和之后的两个标记数量：



标记输出的示例

例1 - 标记函数FUNCa的输出

让我们修改前面定义的函数FUNCa，以产生标记输出。使用(↵) (F1) 将FUNCa的内容调回堆栈。原始功能程序读取

```
« "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 » »
```

使用(↵) (F1) 将程序存回FUNCa。接下来，按(F2)运行程序。出现提示时输入值2，然后按(ENTER)。结果现在是标记结果F: 11。








示例2 - 标记函数FUNCa的输入和输出

在此示例中，我们修改程序FUNCa，以便输出不仅包括已评估的函数，还包含带有标记的输入的副本。使用(↵) (F1) 将FUNCa的内容调用到堆栈：

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



























将其修改为：

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



(回想一下, 使用  , 可以使用SWAP功能。使用   将程序存回 。接下来, 按  运行程序。出现提示时输入值2, 然后按 。结果现在有两个带标记的数字a: 2。在堆栈级别2和F: 11。在堆栈级别1。

Note:因为我们使用输入字符串来获取输入数据值, 所以局部变量a实际上存储了一个标记值 (: a: 2, 在上面的示例中)。因此, 我们不需要在输出中标记它。我们需要做的就是上面的子程序中在SWAP函数之前放置一个a, 并将标记的输入放在堆栈中。需要指出的是, 在执行函数计算时, 标记输入a的标签自动丢弃, 并且在计算中仅使用其数值。

要逐步查看函数FUNCa的操作, 可以使用DEBUG函数, 如下所示:

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

例3 - 标记函数p (V, T) 的输入和输出

在本例中，我们修改了程序以便输出标记输入值和标记结果。使用将程序内容调用到堆栈：

```
«“Enter V, T, and n:“ {“↵ :V:↵ :T:↵ :n:“ {2 0} V } INPUT
OBJ→ →V T n ‘(8.31451_J/(K*mol))* (n*T/V)’ »
```



将其修改为：


```
«“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: 请注意，我们已经将函数p (V, T, n) 的计算和标记放在子程序[内部程序符号集合中包含的指令序列之前，之后调用输入变量VT n) «»]。这是必要的，因为没有程序符号将输入变量的两个列表分开 (V T N«V T n) ，程序将假定输入命令

→V T N V T n

需要六个输入值，而只有三个可用。结果将是生成错误消息和程序执行中断。

要将上述子程序包含在修改后的程序定义中，将要求您在子程序的开头和结尾使用。由于程序符号成对出现，无论何时调用_«> 您都需要删除子程序开头的结束程序符号 (») 和子程序结束时的开始程序符号 («) 。

要在编辑程序时删除任何字符，请将光标放在要删除的字符的右侧，然后使用退格键。

使用   将程序存回到变量p中。接下来，按 。Enter values of V = 0.01_m^3, T = 300_K, and n = 0.8_mol, when prompted. Before pressing  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|INFTa|
```















执行程序后，堆栈将如下所示：

```
4:      V: (.01_m^3)
3:      T: (300_K)
2:      n: (.8_mol)
1:      P: (199548.24_ J / m^3)
INFP1| p |FUNCa|INFTa|
```

总结：这里显示的三个示例中的共同线程是使用标记来标识输入和输出变量。如果我们使用输入字符串来获取输入值，那么这些值已经预先标记，并且可以轻松地调用到堆栈中以进行输出。使用→TAG命令可以识别程序的输出。

使用消息框

消息框是一种更好的方式来显示程序的输出。 计算器中的消息框命令是通过使用  PRG    获得的。 消息框命令要求放置在框中的输出字符串在堆栈级别1中可用。要查看MSGBOX命令的操作，请尝试以下练习：

```
 " (ALPHA)  T (ALPHA)  ::  /  .  2
 _ (ALPHA)  R (ALPHA)  A (ALPHA)  D
 PRG   
```

结果是以下消息框：



按 **OK** 取消消息框

您可以使用消息框通过使用标记输出（转换为字符串）作为MSGBOX的输出字符串来从程序输出。 要将任何标记结果或任何代数或非标记值转换为字符串，请使用 **PRG** **→STR** 中可用的函数→STR。

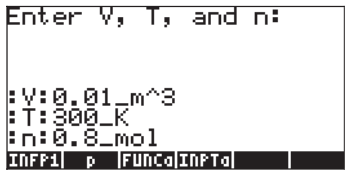
使用消息框进行程序输出

最后一个示例中的函数 **→STR** 可以修改为：

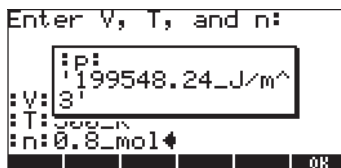
```
« “Enter V, T and n:” {“↵”:V:“↵”:T:“↵”:n:“ {2 0} V } INPUT  
OBJ→ →V T n « V T n‘(8.31451_J/(K*mol))* (n*T/V)’ EVAL “p”  
→TAG →STR MSGBOX » »
```

按 **↵** **→STR** 将程序存回到变量p中。 按 **→STR** 运行程序。 出现提示时，输入V = 0.01_m ^ 3, T = 300_K和n = 0.8_mol的值。

与早期版本的**→STR**一样，在按 **ENTER** 输入之前，堆栈将如下所示：



第一个程序输出是一个包含字符串的消息框：



按 **2ND** **QUIT** 取消消息框输出。堆栈现在看起来像这样：



在消息框中包含输入和输出

我们可以修改程序，这样不仅输出，而且输入都包含在消息框中。对于程序 **MSGBOX** 的情况，修改后的程序将如下所示：



```
« "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 » »
```

请注意，您需要在子程序中的每个变量名 **V**、**T** 和 **n** 之后添加以下代码：

→STR "↵" +

要在第一次输入这段代码，请使用：






您会注意到在键入击键序列后   在堆栈中生成一个新行。

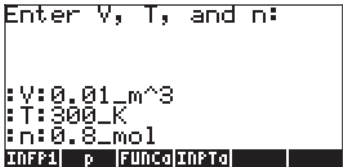
需要包含的最后一个修改是在子程序最后调用函数后三次输入加号。

Note: 此程序中的加号 (+) 用于连接字符串。 连接只是连接单个字符串的操作。

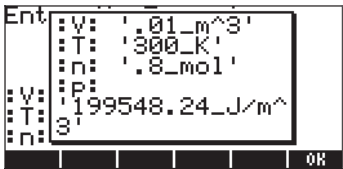
要查看该程序的运行情况：


- 使用   将程序存回到变量p中。
- 按  运行程序。
- 出现提示时，输入 $V = 0.01_m^3$ ， $T = 300_K$ 和 $n = 0.8_mol$ 的值。

与早期版本的[p]一样，在按[ENTER]输入之前，堆栈将如下所示：



第一个程序输出是一个包含字符串的消息框：



Press  取消消息框输出。

将单元合并到程序中

正如您已经能够从本章中提供的不同版本程序 2.3.1 的所有示例中看到的那样，将单位附加到输入值可能是一个繁琐的过程。您可以让程序本身将这些单元附加到输入和输出值。我们将通过再次修改程序 2.3.1, 来说明这些选项，如下所示。

Recall the contents of program 2.3.1 召回程序 2.3.1 的内容，并将其修改为如下所示：

Note: 注意：我已经将程序任意分成几行以便于阅读。这不一定是程序在计算器堆栈中显示的方式。但是，命令序列是正确的。此外，回想一下，字符 \leftarrow 不会显示在堆栈中，而是产生一个新行。

```
« “Enter V,T,n [S.I.] : “ {“ $\leftarrow$  :V:“ $\leftarrow$  :T:“ $\leftarrow$  :n: “ {2 0} V }  
INPUT OBJ $\rightarrow$   $\rightarrow$  V T n  
« V ‘1_m^3’ * T ‘1_K’ * n ‘1_mol’ *  $\rightarrow$  V T n  
« V “V”  $\rightarrow$  TAG  $\rightarrow$  STR “ $\leftarrow$  ” + T “T”  $\rightarrow$  TAG  $\rightarrow$  STR “ $\leftarrow$  ” + n “n”  $\rightarrow$  TAG  
 $\rightarrow$  STR “ $\leftarrow$  ” +  
‘(8.31451_J/(K*mol)) * (n*T/V)’ EVAL “p”  $\rightarrow$  TAG  $\rightarrow$  STR + + +  
MSGBOX » » »
```

该程序的新版本包括附加级别的子程序（即，第三级程序符号«», 以及使用列表的一些步骤，即，

V ‘1_m^3’ * { } + T ‘1_K’ * + n ‘1_mol’ * + EVAL \rightarrow 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' : 然后将对应于V的S.I.单元置于堆栈级别1中，将V的标记输入移动到堆栈级别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' * : 计算T的值，包括S.I.单位
- 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.

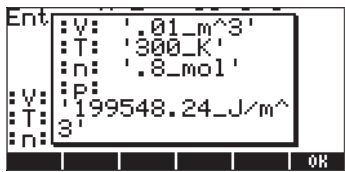
要查看此版本的程序，请执行以下操作：

- [↶][p]将程序存储回变量p。
- 按[p]运行程序。
- 出现提示时输入V = 0.01，T = 300和n = 0.8的值（现在不需要单位）。

在按 **ENTER** 输入之前，堆栈将如下所示：



按 **ENTER** 运行程序。输出是一个包含字符串的消息框：



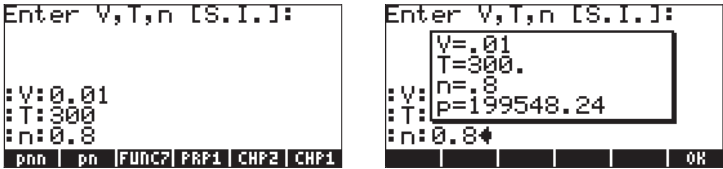
按 **EXIT** 取消消息框输出。

消息框输出没有单位

让我们再次修改程序**8.31451**以消除整个程序的使用。无单元程序将如下所示：

```
« "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 » »
```

当输入数据V = 0.01, T = 300和n = 0.8运行时，产生消息框输出：



Press **EXIT** 取消消息框输出。

关系和逻辑运算符

到目前为止，我们主要使用顺序程序。用户RPL语言提供允许分支和循环程序流的语句。其中许多根据逻辑陈述是否真实做出决定。在本节中，我们将介绍用于构造此类逻辑语句的一些元素，即关系和逻辑运算符。





关系运算符

关系运算符是用于比较两个对象的相对位置的运算符。例如，仅处理实数，关系

运算符用于表示关于两个或多个实数的相对位置。根据所使用的实际数字，这样的陈述可以为真（由计算器中的数值1表示），或者为假（在计算器中由数值0表示）。

可用于编程计算器的关系运算符是：

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'

键盘中提供除==（可通过键入  ==  创建）之外的所有运算符。它们也可以在  PRG  中找到。

由关系运算符连接的两个数字，变量或代数形成一个逻辑表达式，它可以取值 true（1.），false（0.），或者根本不能被计算。要确定逻辑语句是否为 true，请将语句置于堆栈级别1，然后按 (EVAL)。 Examples:

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



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

在下一个例子中，假设变量m未初始化（它没有给出数值）：

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

评估语句的结果与原始语句相同的事实表明该语句不能唯一地进行求值。

逻辑运算符

逻辑运算符是用于连接或修改简单逻辑语句的逻辑粒子。 计算器中可用的逻辑运算符可以通过按键序列轻松访问:  **PRG**  **NXT** .

可用的逻辑运算符是: **AND**, **OR**, **XOR** (异或) , **NOT**和**SAME**。 运算符将产生真或假的结果, 具体取决于受影响的逻辑语句的真值。 运算符**NOT** (否定) 适用于单个逻辑语句。 所有其他都适用于两个逻辑语句。

将一个或两个语句的所有可能组合与应用某个逻辑运算符的结果值一起制表会产生所谓的运算符的真值表。 以下是计算器中可用的每个标准逻辑运算符的真值表:

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

计算器还包括逻辑运算符**SAME**。这是一个非标准的逻辑运算符，用于确定两个对象是否相同。如果它们相同，则返回值1 (**true**)，否则返回值0 (**false**)。例如，在RPN模式下，以下练习返回值0：

`'SQ(2)' [ENTER] 4 [ENTER] SAME`

请注意，使用**SAME**意味着对“相同”一词的非常严格的解释。因此，SQ (2) 与4不同，尽管它们在数值上都评估为4。

程序分支

程序流的分支意味着程序在两个或更多个可能的流路径中做出决定。用户RPL语言提供了许多可用于程序分支的命令。包含这些命令的菜单可通过按键序列访问：



此菜单显示程序构造的子菜单



程序构造**IF ... THEN..ELSE ... END**，**CASE ... THEN ... END**将被称为程序分支构造。其余的结构，即**START**，**FOR**，**DO**和**WHILE**，适用于控制程序内的重复处理，并称为程序循环结构。后面的类型的程序结构将在后面的部分中更详细地介绍。

IF分支

在本节中，我们使用构造IF ... THEN ... END和IF ... THEN ... ELSE ... END提供示例。

IF ... THEN ... END构造

IF ... THEN ... END是IF程序构造中最简单的。 这种结构的一般格式是：

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

该结构的操作如下：

- 1. 评估logical_statement。
- 2. 如果logical_statement为true，则执行program_statements并在END语句后继续执行程序流程。
- 3. 如果logical_statement为false，则跳过program_statements并在END语句后继续执行程序流程。

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



The functions     在该菜单中可由用户选择性地键入。或者，要直接在堆栈上生成IF ... THEN ... END构造，请使用：



这将在堆栈中创建以下输入：



用光标◀ 在IF语句前面，提示用户输出程序执行时将激活IF构造的逻辑语句。
示例：键入以下程序：

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

并将其保存在名称“f1”下。按下VAR并确认变量菜单中确实存在变量, 验证以下结果：

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

这些结果证实了IF ... THEN ... END构造的正确操作。

如所写的，程序计算函数f1 (x) = x2，如果x <3（否则不输出）。

The IF...THEN...ELSE...END 构造

IF ... THEN ... ELSE ... END构造允许基于logical_statement的真值的两个备选程序流路径。这种结构的一般格式是：

IF logical_statement THEN program_statements_if_true ELSE
program_statements_if_false END.

该结构的操作如下：

1. 评估logical_statement。
2. 如果logical_statement为true，则执行program statements_if_true并在END语句后继续执行程序流程。
3. 如果logical_statement为false，则执行program statements_if_false并在END语句后继续执行程序流程。

要生成一个IFTHEN... ELSE ... END直接在堆栈上构造，使用：

 PRG   
这将在堆栈中创建以下输入：



示例：键入以下程序：

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

并将其保存在名称'f2'下。按下 **VAR** 并确认变量菜单中确实存在变量 **f2**

验证以下结果：

0 **f2** Result: 0 1.2 **f2** Result: 1.44

3.5 **f2** Result: -2.5 10 **f2** Result: -9

这些结果证实了IF ... THEN ... ELSE ... END构造的正确操作。程序，如编写，计算功能

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

Note: 对于这种特殊情况，有效的替代方案是使用表单的IFTE函数: 'f2(x) = IFTE(x<3,x^2,1-x)'

Nested IF...THEN...ELSE...END 结构

在大多数计算机编程语言中，IF ... THEN ... ELSE ... END结构可用，用于程序表示的一般格式如下：

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

在设计包含IF构造的计算器程序时，您可以手动编写IF构造的伪代码，如上所示。例如，对于程序 **f2**，您可以编写

```

IF x<3 THEN
    x2
ELSE
    1-x
END

```

虽然这个简单的构造在你的函数只有两个分支时工作正常，你可能需要嵌套 IF ... THEN ... ELSE ... END 构造来处理具有三个或更多分支的函数。例如，考虑一下这个功能

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

下面是一个使用评估此功能的可能方式 IF ... THEN ... ELSE ... END 结构：

```

IF x<3 THEN
    x2
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
END

```

像这样的复杂IF结构称为一组嵌套的IF ... THEN ... ELSE ... END结构。

基于上面显示的嵌套IF结构，评估f3 (x) 的一种可能方法是编写程序：

```
« → 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 » »
```

将程序存储在变量**IF3** 中，并尝试以下评估：

```
1.5 IF3      Result:  2.25 (i.e., x2)
2.5 IF3      Result:  6.25 (i.e., x2)
4.2 IF3      Result: -3.2 (i.e., 1-x)
5.6 IF3      Result  -0.631266... (i.e., sin(x), with x in radians)
12  IF3      Result: 162754.791419 (i.e., exp(x))
23  IF3      Result: -2. (i.e., -2)
```

CASE结构

CASE结构可用于编码几个可能的程序通量路径，如前面介绍的嵌套IF结构的情况。此构造的一般格式如下：





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

在评估此构造时，程序会测试每个logical_statements，直到找到一个为true。程序执行相应的

program_statements, 并将程序流传递给END语句后面的语句。

CASE, THEN和END语句可通过使用  PRG   进行选择性输入。.

如果您在BRCH菜单中, 即, ( PRG ) 您可以使用以下快捷键输入CASE构造 (光标的位置由符号 ◀表示):


-  : 启动案例构造, 提供提示: CASE ◀ THEN END END
-  : 通过添加粒子完成CASE线THEN ◀ END




示例 - 使用CASE语句编写程序f3 (x) 该函数由以下5个表达式定义:




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

使用用户RPL语言中的CASE语句, 我们可以将此函数编码为:

« → 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 » »

将程序存储到名为. 的变量中。然后, 尝试以下练习:

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



5.6		<i>Result:</i> -0.631266... (i.e., sin(x), with x in radians)
12		<i>Result:</i> 162754.791419 (i.e., exp(x))
23		<i>Result</i> -2. (i.e., -2)

如您所见，f3c产生与f3完全相同的结果。程序的唯一区别是使用的分支结构。对于函数f₃(x)的情况，它需要五个表达式来定义，CASE结构可能比一些嵌套的IF ... THEN ... ELSE ... END结构更容易编码。

程序循环

程序循环是允许程序重复执行许多语句的结构。例如，假设您要计算从0到n的整数的平方和，即

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


要计算此总和，您所要做的就是使用公式编辑器中的   键并加载求和的限制和表达式（求和的例子见第2章和第13章）。但是，为了说明编程循环的使用，我们将使用我们自己的用户RPL代码计算这个总和。可以使用四种不同的命令来编写用户RPL中的程序循环，这些命令是START，FOR，DO和WHILE。START和FOR命令使用索引或计数器来确定循环执行的次数。命令DO和WHILE依赖于逻辑语句来决定何时终止循环执行。以下各节将详细介绍循环命令的操作。





START构造

START构造使用索引的两个值来重复执行多个语句。START结构有两个版本：START ... NEXT和START ... STEP。当索引增量等于1时使用START ... NEXT版本，并且当用户确定索引增量时使用START ... STEP版本。

START结构中涉及的命令可通过以下方式获得:



在BRCH菜单 () 中，以下按键可用于生成START结构（符号表示光标位置）：

-  : 启动START ... NEXT构造: START ◀ NEXT
-  : 启动START ... STEP构造: START ◀ STEP

START...NEXT 结构

本声明的一般形式是:


```
start_value end_value START program_statements NEXT
```

因为对于这种情况，增量为1，为了使循环结束，您应该确保start_value < end_value。 否则你将产生所谓的无限（永无止境）循环。

示例 - 计算上面定义的总和S.

START ... NEXT构造包含一个索引，其值是用户无法访问的。 由于需要计算索引本身（在本例中为k），我们必须创建自己的索引(k, in this case) 每次循环执行时我们将在循环内递增。 计算S的可能实现是程序：

```
« 0. DUP → n S k « 0. n START k SQ S + 1. 'k' STO+ 'S' STO  
NEXT S "S" → TAG » »
```

输入程序，并将其保存在名为  的变量中。.

以下是该程序如何工作的简要说明：

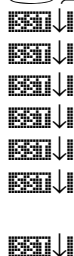
1. 这个程序需要一个整数作为输入。因此，在执行之前，该数字（n）处于堆栈级别1.然后执行该程序。
2. 输入零，将n移动到堆栈级别2。
3. 命令DUP，可以输入为 $\boxed{\text{ALPHA}} \boxed{\text{ALPHA}} \boxed{\text{D}} \boxed{\text{U}} \boxed{\text{P}} \boxed{\text{ALPHA}}$ ，复制堆栈级别1的内容，向上移动所有堆栈级别，并将副本放置在堆栈级别1中。因此，执行DUP后，n为在堆栈级别3中，零填充堆栈级别1和2。
4. 该段代码→n S k分别将n，0和0的值存储到局部变量n，S，k中。我们说变量n，S和k已被初始化（S和k为零，n为用户选择的任何值）。
5. 代码段0. n START标识START循环，其索引将取值0,1,2, ..., n
6. 和s在代码段中增加k2: k SQ S +
7. 索引k在以下代码中增加1: 1。 k +
8. 此时，S和k的更新值分别在堆栈级别2和1中可用。代码段'k'STO将堆栈级别1的值存储到局部变量k中。S的更新值现在占用堆栈级别1。
9. 代码段'S'STO将堆栈级别1的值存储到局部变量k中。堆栈现在是空的。
10. 粒子NEXT将索引增加1并将控制发送到循环的开始（步骤6）。
11. 重复循环，直到循环索引达到最大值n。
12. 程序的最后一部分调用S的最后一个值（求和），标记它，并将其置于堆栈级别1中，以便用户将其视为程序输出。

要逐步查看正在运行的程序，可以按如下方式使用调试器（使用n = 2）。让SL1表示堆栈级别1：

$\boxed{\text{VAR}} \boxed{2} \boxed{[']} \boxed{\text{ENTER}}$

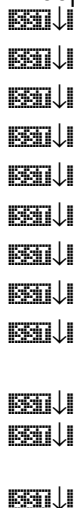
Place a 2 in level 2, and the program name, 'S1', in level 1

← PRG NXT NXT



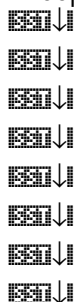
Start the debugger. SL1 = 2.
 SL1 = 0., SL2 = 2.
 SL1 = 0., SL2 = 0., SL3 = 2. (DUP)
 Empty stack ($\rightarrow n S k$)
 Empty stack (« - start subprogram)
 SL1 = 0., (start value of loop index)
 SL1 = 2.(n), SL2 = 0. (end value of loop index)
 Empty stack (START – beginning of loop)

-- loop execution number 1 for $k = 0$







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













-- loop execution number 2 for $k = 1$








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


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

-- loop execution number 3 for $k = 2$

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

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

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

分步列表已完成。在 $n = 2$ 时运行程序的结果是S: 5。

还要检查以下结果: VAR

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

START...STEP 结构

The general form of this statement is:

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

循环索引的start_value, end_value和increment可以是正数或负数。对于增量 > 0, 只要索引小于或等于end_value, 就会执行。对于增量<0, 只要索引大于或等于end_value, 就会执行。

示例 - 生成值列表

假设您要生成x值从x = 0.5到x = 6.5的增量为0.5的列表。您可以编写以下程序:

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

并将其存储在变量

在这个程序中, xs = 循环的起始值, xe = 循环的结束值, dx = 循环的增量值。程序将xs, xs + dx, xs +2 dx, xs +3 dx, ...的值放在堆栈中。然后, 它计算使用这段代码生成的元素数量: $xe - xs - dx / ABS 1. +$

最后, 程序将列表与放置在堆栈中的元素放在一起。

- 检查程序调用 0.5 2.5 0.5 产生列表 {0.5 1. 1.5 2. 2.5}.
- 要查看分步操作, 请使用程序DBUG作为简短列表, 例如:

VAR 1 SPC 1.5 SPC 0.5 ENTER
 ['] F1 F2 ENTER
 ← PRG NXT NXT F10 F12

Enter parameters 1 1.5 0.5
 Enter the program name in level 1
 Start the debugger.

用 F10 进入程序并查看每个命令的详细操作。

FOR构造

与START命令的情况一样，FOR命令有两个变体：

FOR ... NEXT构造，用于循环索引增量为1，FOR ... STEP构造，用于由用户选择的循环索引增量。然而，与START命令不同，FOR命令确实要求我们提供循环索引的名称（例如，j，k，n）。我们不必担心自己增加索引，就像在使用START的示例中所做的那样。与索引对应的值可用于计算。

FOR构造中涉及的命令可通过以下方式获得：

← PRG F10 F12

在BRCH菜单 (← PRG F10) 中，可以使用以下按键来生成FOR结构（符号 ← 表示光标位置）：

- ← F10: Starts the FOR...NEXT construct: FOR ← NEXT
- → F10: Starts the FOR...STEP construct: FOR ← STEP

FOR ... NEXT构造

本声明的一般形式是：

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

要避免无限循环，请确保start_value < end_value.









示例 - 使用FOR ... NEXT构造计算求和S以下程序计算求和

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

使用FOR ... NEXT循环:

« 0 → n S « 0 n FOR k k SQ S + 'S' STO NEXT S "S" →TAG » »

将此程序存储在变量中。 验证以下练习: 

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

您可能已经注意到该程序比存储在 中的程序简单得多。 无需初始化k，或在程序中增加k。 程序本身负责产生这样的增量。

FOR ... STEP构造

本声明的一般形式是:


start_value end_value FOR loop_index program_statements
increment STEP



循环索引的start_value，end_value和increment可以是正数或负数。 对于增量> 0，只要索引小于或等于end_value，就会执行。 对于增量<0，只要索引大于或等于end_value，就会执行。 程序语句至少执行一次（例如，1 0 START 1 1 STEP返回1）












示例 - 使用FOR ... STEP构造生成数字列表

输入程序：

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

并将其存储在变量中.


- 检查程序调用 0.5  2.5  0.5   产生列表 {0.5 1. 1.5 2. 2.5}.
- 要查看分步操作，请使用程序DEBUG作为简短列表，例如：

 1  1.5  0.5 
[']  
 PRG    

Enter parameters 1 1.5 0.5

Enter the program name in level 1

Start the debugger.

用进入程序并查看每个命令的详细操作。

DO构造

该命令的一般结构是：

DO program_statements UNTIL logical_statement END

DO命令启动执行program_statements的无限循环，直到logical_statement返回FALSE (0)。logical_statement必须包含其值在program_statements中更改的索引的值。

示例1 - 此程序在屏幕的左上角生成一个计数器，在无限循环中加1，直到按键（按任意键）停止计数器： « 0 DO DUP 1 DISP 1 + UNTIL KEY
END DROP »

发生击键时，Command KEY的计算结果为TRUE。

示例2 - 使用DO ... UNTIL ... END构造计算求和S









以下程序计算总和

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

使用DO ... UNTIL ... END循环:

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


将此程序存储在变量  中。验证以下练习: 

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












示例3 - 使用DO ... UNTIL ... END构造生成列表

输入以下程序


«→ 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 » »

并将其存储在变量中 .

- 检查程序调用0.5  2.5  0.5   产生列表 {0.5 1. 1.5 2. 2.5}.
- 要查看分步操作, 请使用程序DBUG作为简短列表, 例如:

 1  1.5  0.5 
[']  
 PRG    

Enter parameters 1 1.5 0.5
Enter the program name in level 1
Start the debugger.

Use  进入程序并查看每个命令的详细操作。

WHILE构造

该命令的一般结构是：

```
WHILE logical_statement REPEAT program_statements END
```

当logical_statement为true（非零）时，WHILE语句将重复program_statements。如果没有，程序控制将在END之后立即传递给语句。program_statements必须包含一个循环索引，该循环索引在下次重复开始时检查logical_statement之前被修改。与DO命令不同，如果logical_statement的第一次评估为false，则永远不会执行循环。

例1 - 使用WHILE ... REPEAT ... END构造计算求和S。









以下程序计算总和

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

使用WHILE ... REPEAT ... END循环：

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


将此程序存储在变量  中。验证以下练习： VAR

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














示例2 - 使用WHILE ... REPEAT ... END构造生成列表

输入以下程序



```
« → 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 » » »
```

并将其存储在变量中 .






- 检查程序调用 0.5  2.5  0.5   产生列表 {0.5 1. 1.5 2. 2.5}.
- 要查看分步操作，请使用程序DEBUG作为简短列表，例如：

 1  1.5  0.5 
  
     

Enter parameters 1 1.5 0.5
Enter the program name in level 1
Start the debugger.




Use   进入程序并查看每个命令的详细操作。

错误和错误捕获




PRG / ERROR子菜单的功能提供了操作计算器中的错误和捕获程序中的错误的方法。PRG / ERROR子菜单可通过      获得，包含以下功能和子菜单：

```
2:
1:
DOERR|ERRN|ERRM|ERRD|LASTA|IFERR
```

DOERR

此函数执行用户定义错误，从而使计算器的行为就像发生了特定错误一样。该函数可以将整数，二进制整数，错误消息或数字零（0）作为参数。例如，在RPN模式下，输入    会产生以下错误消息：

错误：内存清除

如果输入  11h   则会产生以下消息：错误：未定义的FPTR名称

如果输入“TRY AGAIN” **ENTER** **|||||**,会产生以下消息: TRY AGAIN
最后, **0** **ENTER** **|||||**,产生消息: Interrupted (中断)

ERRN

此函数返回表示最近错误的数字。例如,如果您尝试 **0** **/x** **ON** **|||||**则会得到数字 # 305h。这是表示错误的二进制整数: Infinite Result (无限结果)

ERRM

此函数返回表示最近错误的错误消息的字符串。例如,在 Approx 模式下,如果您尝试 **0** **/x** **ON** **|||||**,则会得到以下字符串: “Infinite Result” (无限结果)

ERRO

此函数清除最后一个错误编号,以便之后在 Approx 模式下执行ERRN将返回 # 0h。例如,如果您尝试 **0** **/x** **ON** **|||||** **|||||**, you get # 0h. 此外,如果您尝试 **0** **/x** **ON** **|||||** **|||||**,则会得到空字符串“ ”。






LASTARG

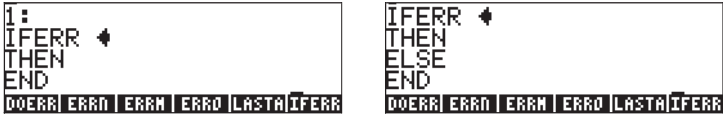
此函数返回最近执行的命令或函数的参数的副本。例如,在RPN模式下,如果使用 **3** **÷** **2** **ENTER**,然后使用函数LASTARG (**|||||**),您将获得堆栈中列出的值3和2。另一个例子,在RPN模式下,如下 **5** **TAN** **ENTER**。在这些条目生成5之后使用LASTARG。

Sub-menu IFERR

The **|||||** 子菜单提供以下功能:



这些是IFERR ... THEN ... END构造或IFERR THEN ... ELSE ... END构造的组成部分。两个逻辑结构都用于在程序执行期间捕获错误。在  子菜单中，输入  , or  , 将IFERR结构组件放入堆栈中，为用户填写缺失的条件做好准备，即




两个错误捕获结构的一般形式如下：

IF trap-clause THEN error-clause END


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

这些逻辑结构的操作类似于IF ... THEN ... END和IF ... THEN ... ELSE ... END结构的操作。如果在执行trap子句期间检测到错误，则执行error-clause。否则，执行normal-clause。

例如，考虑以下程序() 将两个矩阵A和b作为输入，并检查陷阱子句中是否存在错误：A b / (RPN模式，即A / b)。如果有错误，则程序调用函数LSQ (Least Squares，见第11章) 来求解方程组：

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






尝试使用参数A = [[2, 3, 5], [1, 2, 1]] and b = [[5], [6]]. 这两个参数的简单划分会产生错误：/错误：Invalid Dimension (无效的维度) .

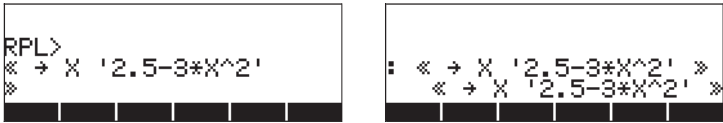
但是，使用程序的错误捕获构造,具有相同的参数会产生：[0.262295..., 0.442622...].

代数模式下的用户RPL编程

虽然前面介绍的所有程序都是在RPN模式下生成和运行的，但在代数模式下，您可以使用函数RPL>在用户RPL中键入程序。 该功能可通过命令目录获得。例如，尝试在代数模式下创建以下程序，并将其存储到变量P2中：

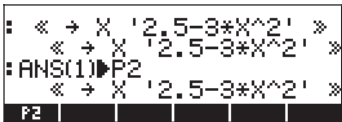
```
« → X '2.5-3*X^2' »
```

首先，从命令目录()激活RPL>功能。 在ALG模式下激活的所有功能都有一对附加在其名称上的括号。 RPL>函数也不例外，除非我们在屏幕中键入程序之前必须删除括号。 使用箭头键 ( ) 和删除键 () 从RPL> () 语句中删除括号。 此时您将准备好输入RPL程序。 下图显示了按下  键之前和之后的程序的RPL>命令。




要存储程序，请使用STO命令，如下所示：

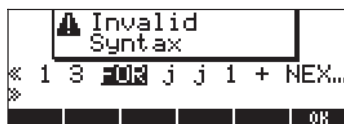
```
 ANS  ALPHA  2 
```



参数X = 5的程序P2的评估显示在下一个屏幕中：



虽然您可以在代数模式下编写程序，而不使用函数RPL>，但当您按下  时，某些RPL结构将产生错误消息，例如：



然而，使用RPL，在代数模式下加载此程序时没有问题：

