

# **GUÍA DE PROGRAMACIÓN EN TI-BASIC**

## **-Manual de programación en Ti-Basic V 0.2-**

### **Autor Principal:**

**David F. Suescun Ramirez**  
Ingeniero Mecatrónico  
Universidad de San Buenaventura  
Bogotá, Colombia

### **Colaboradores:**

**Fabio Silber**  
Est. Ingeniería Electrónica  
Universidad Tecnológica Nacional  
Buenos Aires, Argentina

**Raúl Bores Monjote**  
Est. Ingeniero Eléctrico  
Instituto Tecnológico de Mérida  
Mérida, México

**Don Diego Gómez de la Vega**  
Est. Ingeniería Eléctrica  
Universidad de Carabobo  
Valencia, Venezuela

**Daisuke-Ing es Copyright © 2005-2007 David F. Suescun R.**

Este documento esta enmarcado dentro del proyecto titulado Daisuke-Ing<sup>©</sup> que contiene programas de ayuda para Ingeniería Mecatrónica e Ingenierías afines.

<http://www.daisuke.tk>

# TABLA DE CONTENIDOS

	pág.
<b>1. INTRODUCCIÓN .....</b>	<b>4</b>
<b>2. FUNDAMENTOS DE PROGRAMACIÓN .....</b>	<b>5</b>
2.1 ANÁLISIS.....	5
2.2 DISEÑO .....	6
2.2.1 <i>Diseño de Pantallas</i> .....	7
2.2.2 <i>Especificaciones del proceso</i> .....	7
2.2.3 <i>Validación de Datos</i> .....	8
2.3 DESARROLLO .....	8
2.4 PRUEBAS .....	9
2.5 DOCUMENTACIÓN .....	9
2.6 IMPLEMENTACIÓN .....	10
2.6.1 <i>Instalación</i> .....	10
2.7 SOPORTE .....	10
<b>3. ELEMENTOS BÁSICOS.....</b>	<b>11</b>
3.1 INICIO Y FIN DE UN PROGRAMA O FUNCIÓN .....	11
3.2 LÍNEA DE CÓDIGO .....	12
3.3 COMENTARIOS DE CÓDIGO.....	13
3.4 INDENTACIÓN O TABULACIÓN .....	13
<b>4. USO DE VARIABLES .....</b>	<b>14</b>
4.1 TIPOS DE VARIABLES DE TI-BASIC .....	14
4.2 ÁMBITO DE LAS VARIABLES.....	16
4.3 COMANDOS RELACIONADOS CON VARIABLES .....	17
4.4 VARIABLES LOCALES.....	18
4.5 VARIABLES DEL SISTEMA .....	19
4.6 CONSTANTES .....	20
<b>5. DIFERENCIA ENTRE PROGRAMA Y FUNCIÓN .....</b>	<b>21</b>
5.1 CUANDO USAR UN PROGRAMA.....	21
5.2 CUANDO USAR UNA FUNCIÓN.....	22
5.3 COMPARACIÓN ENTRE FUNCIÓN Y PROGRAMA .....	23
<b>6. INGRESO Y SALIDA DE DATOS .....</b>	<b>24</b>
6.1 EN PROGRAMAS .....	24
6.1.1 <i>Entrada de datos como argumentos</i> .....	24
6.1.2 <i>Pantalla Program I/O (Entrada/Salida)</i> .....	25
6.1.2.1 Entrada de datos en IO .....	25
6.1.2.1.1 Input.....	26
6.1.2.1.2 InputStr .....	28
6.1.2.1.3 Prompt .....	30
6.1.2.2 Salida de datos en IO.....	32
6.1.2.2.1 Output .....	32
6.1.2.2.2 Disp.....	33
6.1.2.2.3 Pause.....	34
6.1.3 <i>Cuadros de Diálogo</i> .....	35

6.1.4	PopUp.....	38
6.2	EN FUNCIONES .....	39
6.2.1	Entrada de datos como argumentos .....	39
6.2.2	Salida de datos con Return.....	40
<b>7.</b>	<b>MENÚS.....</b>	<b>41</b>
7.1	TOOLBAR.....	41
7.2	CUSTOM.....	44
<b>8.</b>	<b>CONTROL DE FLUJO.....</b>	<b>45</b>
8.1	CONDICIONAL IF.....	46
8.1.1	Para una sola instrucción.....	46
8.1.2	Para múltiples instrucciones .....	46
8.1.3	Para caso verdadero y falso .....	47
8.1.4	Para varios casos .....	48
8.2	CONDICIONAL WHEN .....	49
8.3	BUCLES .....	52
8.3.1	For.....	52
8.3.2	While.....	54
8.3.3	Loop.....	55
8.4	ETIQUETAS LBL Y GOTO .....	56
<b>9.</b>	<b>SUBROUTINAS.....</b>	<b>58</b>
9.1	SUBROUTINA LOCAL .....	59
9.2	SUBROUTINA GLOBAL .....	60
9.3	CONSIDERACIONES DE CARPETA.....	60
<b>10.</b>	<b>USO DE LIBRERÍAS .....</b>	<b>61</b>
10.1	ARCHING.....	61
10.2	FLIB .....	61
10.3	HAIL .....	62
10.4	IMPRIMIR .....	62
10.5	VERTEL .....	63
<b>11.</b>	<b>AYUDA .....</b>	<b>64</b>
<b>12.</b>	<b>HISTORIAL DE VERSIONES.....</b>	<b>65</b>
12.1	GUÍA DE PROGRAMACIÓN EN TI-BASIC .....	65
<b>13.</b>	<b>CRÉDITOS.....</b>	<b>66</b>
<b>14.</b>	<b>LICENCIA.....</b>	<b>67</b>
<b>15.</b>	<b>AVISO LEGAL .....</b>	<b>67</b>

# 1. INTRODUCCIÓN

En Internet es muy común encontrar manuales de programación para todo tipo de lenguajes, pero el material didáctico para el lenguaje Ti-Basic es muy reducido. Así nace la iniciativa de esta guía. Facilitar conocimientos de programación a los usuarios de las calculadoras Texas Instruments con procesador 68k de Motorola (ti-89, ti-89 Titanium, ti-92, ti-92 Plus y Voyage 200).

Los programas y funciones permiten expandir las capacidades de la calculadora y les dan la posibilidad a los usuarios de compartir soluciones específicas y generales con otros usuarios, de todo tipo de temas académicos (e inclusive juegos).

En esta guía se trataron los temas más importantes cuando se empieza a programar por primera vez. Esperamos sea de su agrado y ayuda. No dude en contactar a los autores ante cualquier inquietud (ver CRÉDITOS).

Aparte de esta guía se hacen las siguientes recomendaciones:

- Programar usando el editor de TI-BASIC para PC Daisuke-Edit:

<http://www.subterraneos.net/rapco/daisuke/modules/wfdownloads/singlefile.php?cid=4&lid=24>

- Usar devFlowcharter (versión 0.9.9 en adelante) para hacer el diseño de algoritmos y genera el código en TI-BASIC:

<http://sourceforge.net/projects/devflowcharter/>

- Si tiene un proyecto en mente primero busque en:

<http://www.ticalc.org>

y

<http://www.calculadoras.cl/foro/index.php>

Para ver el estado de arte en programas similares y no repetir desarrollos.

Para programadores avanzados se recomienda la Ti-TipList:

<http://www.technicalc.org/tiplist/en/html/index.htm>

## 2. FUNDAMENTOS DE PROGRAMACIÓN

Existen muchas formas de empezar a desarrollar un programa,. Uno de forma intuitiva puede hacer muchas cosas, pero estos desarrollos no se comparan con los que tienen planeación y estructuras establecidas de *cómo* y *cuándo*.

A continuación una de las metodologías de programación más difundida en la creación de sistemas informáticas; muy aplicable a TI-BASIC.

### METODOLOGÍA DEL CICLO DE VIDA

Dentro de las diferentes metodologías de programación contenidas en la ingeniería de software, el ciclo de vida es la más adecuada ya que es para programación diferente a la programación por casos (*Cases*) u orientada a objetos (OOP).

Para esta metodología se encuentra una gran gama de pasos a seguir. Esto difiere según el autor. Aún así, la esencia de la metodología se mantiene.

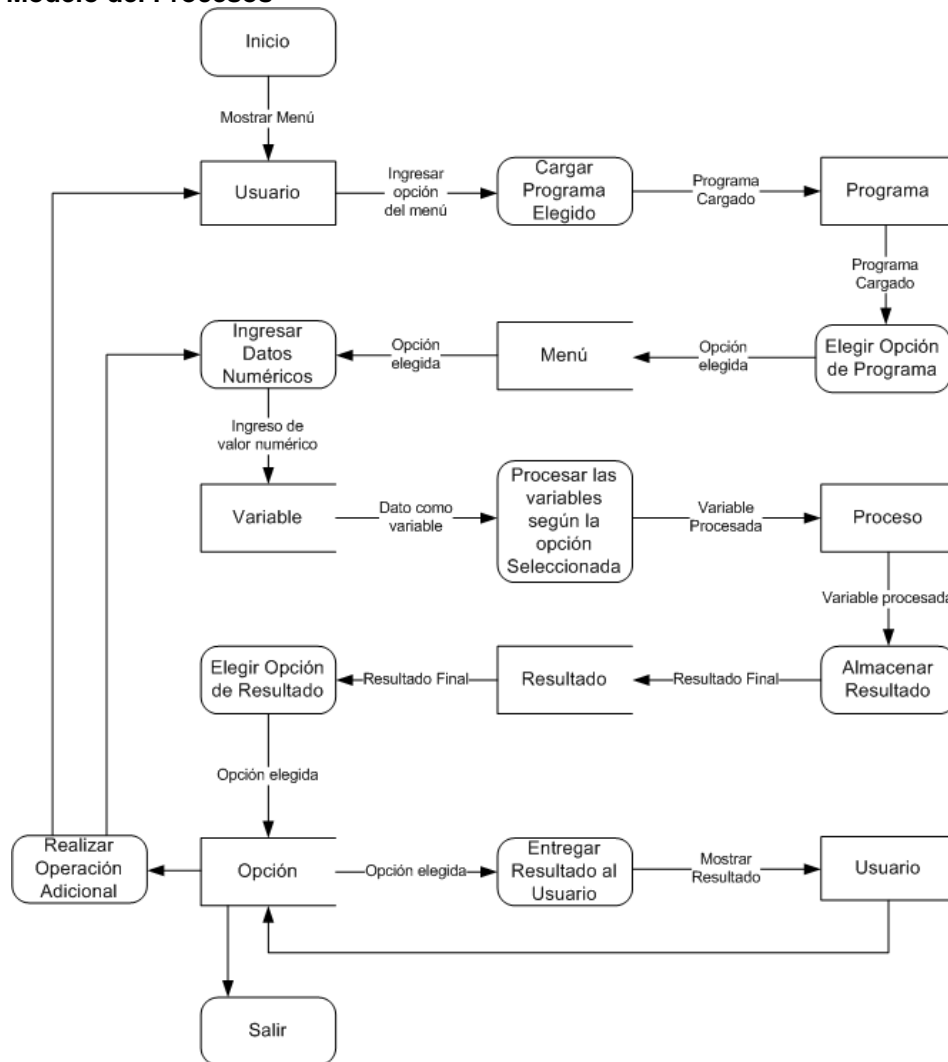
### 2.1 ANÁLISIS

Dentro de esta primera fase se establece el problema que se va a resolver mediante el software. Una vez establecido el problema se hace una recopilación exhaustiva de información para determinar las características y el funcionamiento que el sistema debe tener.

Con base a estos requerimientos se realiza un primer acercamiento al proceso que debe realizar el software para lograr su cometido, esto se ve reflejado en un modelo de funcionamiento que debe considerar al usuario final.

Esta guía plantea el siguiente modelo de procesos. Considérese que los rectángulos redondeados son procesos, los rectángulos normales son interfaces gráficas y los rectángulos sin línea en el lateral derecho son almacenamientos o bancos de datos de donde proviene la entrada para un proceso o en donde es almacenada la salida después de dicho proceso.

Figura 1 Modelo del Procesos



## 2.2 DISEÑO

En esta fase se produce un modelo que cumpla con los requerimientos entregados en la fase de análisis. Esto tiene dos enfoques, el operativo y el computacional.

En el operativo se consideran los métodos de ingreso y salida de datos así como el diseño de pantallas. La forma como el usuario percibe el software desarrollado.

En la parte computacional se deben determinar las variables que se van a manejar, así como el algoritmo que debe implementarse, para esto se hace uso de

diagramas de flujo<sup>1</sup> o pseudo-código, describiendo el lugar en donde cada variable será almacenada y cómo será procesada.

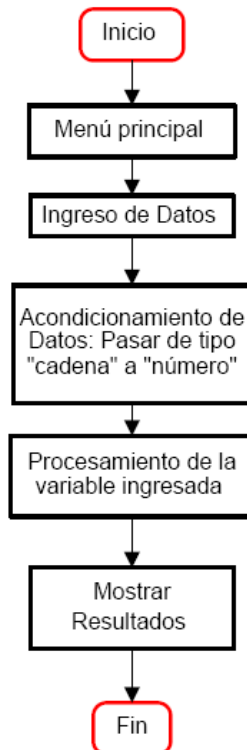
### 2.2.1 Diseño de Pantallas

Durante el proceso de diseño de pantallas se establece de qué forma se va a presentar la interfaz del programa al usuario. Esta interfaz comprende los menús, el ingreso y la salida de datos. Más adelante se explicarán los diferentes métodos que se pueden usar para realizar dichas interfaces.

### 2.2.2 Especificaciones del proceso

Para el ingreso, procesamiento e impresión de datos se sigue el siguiente esquema:

**Figura 2 Esquema general de Procesamiento de datos**



---

<sup>1</sup> Recomendamos el uso de devflowchater para este propósito ya que también genera el código en TI-BASIC: <http://sourceforge.net/projects/devflowcharter/>

### 2.2.3 Validación de Datos

“Consiste en el proceso de comprobar la precisión de los datos; conjunto de reglas que se pueden aplicar a un control para especificar el tipo y el intervalo de datos que los usuarios pueden especificar.”<sup>2</sup>

Cuando se hace un programa de carácter científico (matemáticas, ingenierías, física, etc...) es posible que los datos esperados como entradas no tengan siempre la misma naturaleza (números enteros, números reales, expresiones, funciones, ecuaciones, cadenas, listas, matrices, vectores, etc...).

Es por esto que se deben realizar comprobaciones de datos para asegurarle al usuario que los datos que está ingresando pueden ser procesados de la forma como el programa fue diseñado y codificado. No basta con buena información gráfica, tampoco se puede confiar en el “sentido común” ya que para el programador resulta demasiado evidente, pero se desconoce el enfoque que un usuario le pueda dar.

## 2.3 DESARROLLO

Esta fase también conocida como “Programación” o “Codificación” es en donde se traduce la información ya procesada a un lenguaje de programación.

Del buen desempeño de las partes encargadas de las fases de análisis y diseño depende la facilidad y rapidez con que se desarrolle esta fase.

Uno de los defectos del ciclo de vida es que en principio es lineal, no se esperan saltos hacia atrás, pero en esta fase es algo común a realizar, ya que pueden surgir errores que tengan lugar en la fase de análisis o diseño. Esto también aplica a las fases subsecuentes.

**Nota:** Se recomienda el uso del Editor de TI-BASIC: Daisuke-Edit para esta labor. Puede obtenerlo en:

<http://www.subterraneos.net/rapco/daisuke/modules/wfdonloads/singlefile.php?cid=4&lid=24>

---

<sup>2</sup> Validación de datos. Asistencia de Microsoft Office. Microsoft Latinoamérica. [en línea] <<http://office.microsoft.com/es-hn/assistance/HP010967143082.aspx>> [Fecha de Consulta: 09/05/06]



## 2.4 PRUEBAS

En esta fase se busca detectar errores, sean computacionales, de diseño o de análisis. Se comparan los resultados obtenidos con el software con los resultados esperados y se evalúa el comportamiento del sistema en cuanto a velocidad y rendimiento.

Esta prueba no debe confundirse con el proceso de depuración: durante la prueba se determina la existencia de errores (el hecho de que no se encuentre ninguno durante esta fase no prueba la inexistencia de errores), por otro lado, durante el proceso de depuración se determina la localización del error (o los errores).

**Nota:** Se recomienda el uso del emulador TiEMU para esta labor. Puede obtenerlo en [http://lpg.ticalc.org/prj\\_tiemu/](http://lpg.ticalc.org/prj_tiemu/)

## 2.5 DOCUMENTACIÓN

“La documentación de sistemas de información es el conjunto de información que nos dice qué hacen los sistemas, cómo lo hacen y para quién lo hacen.”<sup>3</sup>

La documentación tiene dos partes: En una se describe como fue realizado el software; se incluyen los resultados de las fases anteriores. Esta facilita el proceso de mantenimiento del software ya que le permite a nuevos equipos de desarrollo entender la labor realizada y les da pautas para continuar mejorando el sistema. Esta documentación se subdivide en una documentación interna y un manual técnico. La documentación interna se realiza poniendo comentarios dentro del código fuente, comentarios cortos y precisos del proceso que se realiza. El manual técnico contiene el análisis con el cuál se llega al problema, así como el algoritmo diseñado y el código fuente desarrollado.

La segunda parte de la documentación describe el funcionamiento del software y está dirigida a los usuarios finales. Esta es tan importante como el desarrollo mismo, ya que con esta se aumenta la probabilidad de que el software sea utilizado de la forma esperada, explotando todas las funciones y servicios que preste.

---

<sup>3</sup> Contreras, R. 2006. Análisis y diseño de sistemas de información El Ciclo de Vida del Desarrollo de Sistemas de Información I, Instituto Tecnológico de Morelia.

## 2.6 IMPLEMENTACIÓN

Este es el inicio de la vida útil del sistema desarrollado. Se deben abordar estrategias de distribución del producto y de aseguramiento de que los usuarios sean entrenados en el uso del mismo.

Muchas veces un cambio de sistema presenta problemas referentes a la conversión de sistemas anteriores al presentado, así como escepticismo para su uso. Problemas que deben solucionarse para lograr el éxito del sistema.

El mayor medio de distribución de programas para las calculadoras Texas Instruments es el portal TiCalc – <http://www.ticalc.org> -, pero existen otros como el Foro de Calculadoras.cl - <http://www.calculadoras.cl/foro/index.php> -

### 2.6.1 Instalación

Los programas codificados en Ti-Basic deben ser interpretados por la calculadora cuando estos son ejecutados. Este proceso puede realizarse una sola vez si el programa se ejecuta desarchivado y se archiva inmediatamente después o antes de apagar la calculadora. En cualquier otro caso, si el programa se archiva antes de ser ejecutado tendrá que ser interpretado cada vez que se ejecute. Si el programa se ejecuta sin archivar, y se deja desarchivado, el programa debe ser interpretado cada vez que prenda la calculadora.

**Nota:** Se recomienda leer la Guía de Creación de un Instalador del proyecto Daisuke-Ing disponible en:

<http://www.subterranos.net/rapco/daisuke/modules/wfdownloads/singlefile.php?cid=10&lid=25>

## 2.7 SOPORTE

Todo sistema esta sujeto a cambios, las principales razones pueden resumirse en los errores que los usuarios puedan encontrar en el sistema, así como un cambio en el hardware o periféricos que ocasionen incompatibilidad con el sistema, y por último, ampliaciones que pueda requerir el sistema, detalles que no hayan sido contemplados durante las fases anteriores.

Estos cambios infieren que se retorne a las fases iniciales completando así el ciclo.

El desarrollo de los programas usando una estructura bien establecida permite facilitar cualquier cambio que deba realizarse en caso de que nuevos sistemas operativos o modelos de calculadoras *Texas Instruments* surjan.

### 3. ELEMENTOS BÁSICOS

Existen ciertos elementos básicos que deben recalcarse antes de empezar a programar. Uno muy importante es el símbolo de “almacenamiento en una variable”: “→”, y la forma como este se debe usar; la estructura es la siguiente:

**Valor → Variable**

Lo que significa que el “Valor” es almacenado en la “Variable”. Debe tenerse en cuenta que el símbolo igual (=) se usa como igualdad en comparaciones o ecuaciones, y no como un método de almacenamiento de variables, por lo tanto algo como:

**Valor=Variable ó Variable=Valor**

No almacena nada.

Existen otras consideraciones que serán explicadas a lo largo de esta guía.

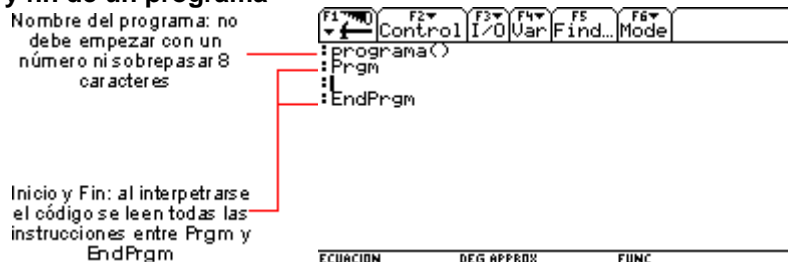
#### 3.1 INICIO Y FIN DE UN PROGRAMA O FUNCIÓN

Un programa es una serie de órdenes ejecutadas en orden secuencial (aunque algunas órdenes alteran el flujo del mismo). En general, todo lo que puede ejecutarse en la pantalla *Home* puede incluirse en un programa. La ejecución del programa continúa hasta llegar al final o hasta que se ejecuta la orden *Stop* o *Return*.

**Un programa:**

Un programa tiene la siguiente estructura:

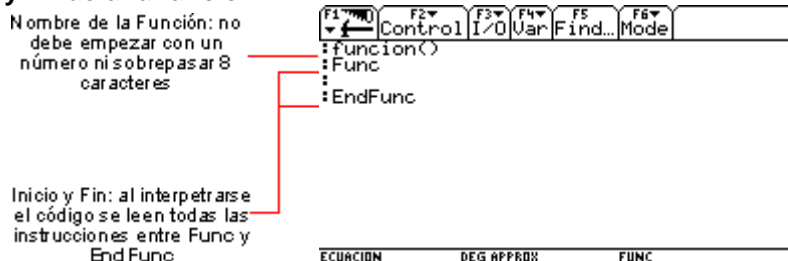
**Figura 3 Inicio y fin de un programa**



## Una Función:

Un programa tiene la siguiente estructura:

**Figura 4 Inicio y fin de una función**



## 3.2 LÍNEA DE CÓDIGO

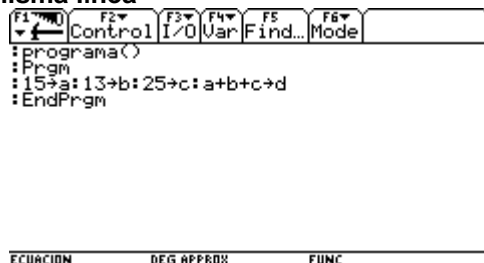
Cada línea de código en el lenguaje Ti-Basic empieza por dos puntos (:), es posible tener comandos en líneas separadas:

**Figura 5 Comandos en líneas separadas**



O tener varios comandos en la misma línea usando dos puntos (:) para dividirlos:

**Figura 6 Comandos en la misma línea**



### 3.3 COMENTARIOS DE CÓDIGO

El símbolo (©) permite introducir comentarios en el programa. Al ejecutarlo, se ignorarán todos los caracteres situados a la derecha de ©.

**Figura 7** Uso de comentarios



**Consejo:** Utilice comentarios para introducir información que resulte útil a quien lea la codificación del programa...Véase 2.5 Documentación p9...

### 3.4 INDENTACIÓN O TABULACIÓN

La indentación (también conocida como Tabulación) es el espacio en blanco que se pone anterior a un texto para identificar dentro de qué estructura se encuentra. Por ejemplo:

```
If 1>0 Then
  Text "Es mayor"
EndIf
```

Nótese el espacio antes de "Text...", a este espacio se le conoce como indentación.

La indentación es bastante útil para hacer mantenimiento o modificación a un programa o función. Dentro de las prácticas recomendadas de programación está su uso.

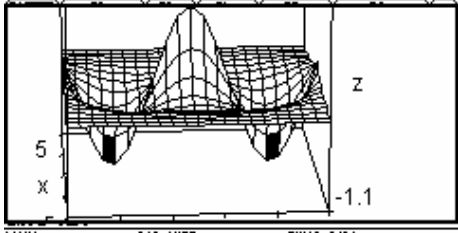
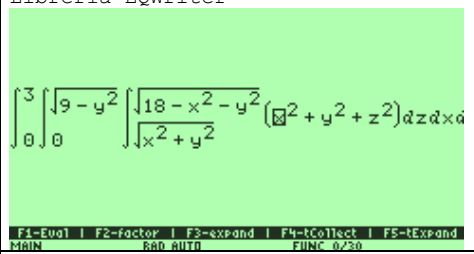
## 4. USO DE VARIABLES

### 4.1 TIPOS DE VARIABLES DE TI-BASIC

A continuación una tabla explicativa de los tipos de datos manejados por las calculadoras.

**Tabla 1 Tipos de Variables de TI-BASIC**

Objetos por tipo de la TI89, TI92, TI92plus, TI Voyage 200PLT			
#	Objetos	Tipo	Ejemplos
1	-Números reales en base 10 (Enteros, decimales, fraccionarios, notación científica) -Números enteros en base 2,16	"NUM"  *.89E *.9XE	1 1, 2.5 5/2 1.2E-3 55°20'30.14ω 0b11111111 (Binario) 0hfff (Hexadecimal)
2	-Cadenas (Sucesión de caracteres)	"STR" *.89S *.9XS	"X^3-6*X^2+11*X=6"
3	-Matrices y Vectores numéricos/simbólicos reales y/o complejos	"MAT" *.9XM *.89m	[[1,a][3+4*i, b]] [i, j, k]
4	-Nombres de variables simbólicas (deben estar sin asignarse a un objeto) -Expresiones simbólicas -Ecuaciones diferenciales -Ecuaciones algebraicas -Ecuaciones trascendentes -Sistemas de ecuaciones lineales numéricos y/o simbólicos -Sistemas de ecuaciones <b>no</b> lineales numéricos y/o simbólicos -Expresiones/Números complejos -Constantes simbólicas -Constantes lógicas	"EXPR" *.89E *.9XE	x y z t x1 x2 x3 x_ y_ X^3-6*X^2+11*X-6 yψψ+2*yψ+y=0 X^3-6*X^2+11*X=6 sen(x)+ln(x)=0 a*x+4*y=b and 5*x+c*y=-8 x^2+y^2=b and x*y=c 3+4*i e i Å ∞, -∞ false true
5	-Listas de expresiones matemáticas	"LIST" *.89L *.9XL	{x=i, x=-i, x=-1} {1,2,3,4,5}
6	-(Global Name) Nombres de <i>variables</i> asignadas a un objeto (objeto->nombre) y que no están predefinidos con la instrucción Local  El objeto se almacena en el explorador de archivos(VAR-LINK)  -(Local Name) Nombres de variables asignadas a un objeto (objeto->nombre) y que estan predefinidos con la instrucción Local o son los argumentos dentro de un algoritmo  Nombredefuncion/programa(argumento1,arg2,argN)	"VAR" *.89E *.9XE	base8->temp1 [Enter]  temp1 es una variable global que se puede leer en cualquier momento  :unitV2(v) :Func :Local x :(dotp(v,v))^(1/2)->x :return v/x :EndFunc  v & x son variables locales :xRoot(x,y) :Func :approx(y^(1/x)) :EndFunc  x & y son variables locales

7	- Funciones. Siempre retornara un objeto en la historia	"FUNC" *.89F *.9XF	:charPoly(m,vx) :Func :expand(det(m- vx*identity(colDim(m)))) :EndFunc
8	- Programa. La información se almacena en el explorador de archivos(VAR-LINK) o se visualiza en ciertas ventanas	"PRGM" *.89P *.9XP	:factQR(m) :Prgm :qr m,mq,mr :EndPrgm
9	-Datos estadísticos	"DATA" *.89C *.9XC	
10	-Archivo de texto	"TEXT" *.89T *.9XT	:Formulas de Física :Óptica ...
11	-Objeto grafico o Imagen	"PIC" *.89I *.9XI	
12	-El nombre de variable no contiene ningún objeto  -La variable definida como Local aun no se le asignado un objeto	"NONE"	Delvar base: base->templ: gettype(base) [ENTER] "NONE"  :Laplace(f,v1,v2) :Func :Local templ ... :return getType(templ) :EndFunc
13	-Base de datos grafica	"GDB"	
14	-programa escrito en lenguaje T.I.assembly	"ASM" *.89Z *.9XZ	Librería EQWriter  
15	-Tipo disponible por el usuario (Requiere lenguaje ensamblador) -O posible nuevo tipo en futuras versiones del O.S (A.M.S.)	"OTHER"	
16	Grupo de Archivos	*.89g	

Nota: TI-BASIC a diferencia de muchos otros lenguajes de programación puede usar cualquier variable para almacenar todo Tipo de datos sin necesidad de definirla ni restringirla sólo a un tipo. Es decir, es posible almacenar un número en una variable y después almacenar una cadena en la misma variable sin que esto represente un problema.

## 4.2 ÁMBITO DE LAS VARIABLES

Tabla 2 Ámbito de las variables

Ámbito	Descripción
<b>Variables del sistema (Global)</b>	<p>Variables de nombre reservado que se crean automáticamente para almacenar información sobre el estado de la TI-89 / Voyage™ 200 PLT. Por ejemplo, las variables de ventana (xmin, xmax, ymin, ymax, etc.) están disponibles de forma global para cualquier carpeta.</p> <ul style="list-style-type: none"> <li>• Es posible referirse a estas variables utilizando solamente el nombre de las mismas, independientemente de la carpeta que esté en uso.</li> <li>• Los programas no pueden crear variables del sistema, aunque pueden utilizar sus valores y, en la mayoría de los casos, almacenar nuevos valores.</li> </ul>
<b>Variables de carpeta</b>	<p>Variables que se almacenan en determinadas carpetas (Global)</p> <ul style="list-style-type: none"> <li>• Si se almacena sólo en un nombre de variable, la variable se almacenará en la carpeta actual. Por ejemplo:</li> </ul> <pre>5 → start</pre> <ul style="list-style-type: none"> <li>• Si sólo se hace referencia a un nombre de variable, dicha variable debe encontrarse en la carpeta actual. De lo contrario, no se encontrará (aun cuando la variable exista en una carpeta distinta).</li> <li>• Para almacenar o hacer referencia a una variable de otra carpeta, será preciso especificar un nombre de camino. Por ejemplo:</li> </ul> <pre>5 → class\start</pre> <p>En donde <i>class</i> es la carpeta y <i>start</i> la variable</p> <p>Después de interrumpir el programa, las variables de la carpeta creadas en el programa continúan existiendo y ocupando la memoria. De todas formas se pueden borrar usando el Comando <b>DelVar</b> explicado más adelante.</p>
<b>Variables locales</b>	<p>Variables provisionales que sólo existen mientras el programa se está ejecutando. Al interrumpir el programa, las variables locales se borran automáticamente.</p> <ul style="list-style-type: none"> <li>• Para crear variables locales en el programa, será preciso definir las utilizando la orden Local.</li> <li>• Las variables locales se consideran como únicas aunque exista una variable en la carpeta con el mismo nombre.</li> <li>• Las variables locales son muy útiles para almacenar temporalmente los valores que no se quieren guardar.</li> </ul>



### 4.3 COMANDOS RELACIONADOS CON VARIABLES

Tabla 3 Comandos relacionados con variables

Orden	Descripción
→	Almacena un valor en una variable.
<b>Archive</b>	Mueve las variables especificadas de la RAM a la memoria de archivo de datos del usuario.
<b>BldData</b>	Permite crear una variable de datos basada en la información gráfica introducida en Y=Editor, Window Editor, etc.
<b>CopyVar</b>	Copia el contenido de una variable.
<b>Define</b>	Define una variable de programa (subrutina) o de función dentro de un programa.
<b>DelFold</b>	Borra una carpeta. Primero deben borrarse todas las variables incluidas en dicha carpeta.
<b>DelVar</b>	Borra una variable.
<b>getFold</b>	Devuelve el nombre de la carpeta actual.
<b>getType</b>	Devuelve una cadena que indica el tipo de datos (EXPR, LIST, etc.) de la variable.
<b>Local</b>	Establece una o más variables como variables locales.
<b>Lock</b>	Bloquea una variable, de forma que no pueda modificarse o borrarse accidentalmente sin antes desbloquearla.
<b>MoveVar</b>	Desplaza una variable de una carpeta a otra.
<b>NewData</b>	Crea una variable de datos cuyas columnas consisten en una serie de listas.
<b>NewFold</b>	Crea una nueva carpeta.
<b>NewPic</b>	Crea una variable de imagen gráfica basada en una matriz.
<b>Rename</b>	Asigna un nuevo nombre a la variable.
<b>Unarchiv</b>	Desplaza las variables especificadas de la memoria de archivo de datos del usuario a la RAM.
<b>Unlock</b>	Desbloquea una variable bloqueada.

## 4.4 VARIABLES LOCALES

Las variables locales son variables temporales que sólo existen mientras la función se calcula o el programa se ejecuta. Siempre que sea posible, utilice variables locales para aquellas empleadas exclusivamente en un programa y que no necesiten almacenarse cuando el mismo finalice.

Para definir un variables como locales se debe usar la instrucción *Local* seguida de el listado de variables separadas por comas (.). Se puede definir una o varias variables locales.

### Ejemplo:

```
Define la variable i como variable local---- :Local i
                                           :For i,0,5,1
                                           : Disp i
                                           :EndFor
                                           :Disp i
```

Si establece la variable *i* como local, ésta se borrará automáticamente al interrumpir el programa para no agotar la memoria.

### Consideraciones:

- Las funciones gráficas de la calculadora no pueden acceder a variables definidas como locales.
- Todas las variables locales deben tener un valor inicial asignado antes de poder hacerse referencia a ellas.
- No puede utilizar una variable local para realizar cálculos simbólicos.
- No puedo hacer referencia a una variable local desde otro programa o función.
- Si usa una variable local que se ha definido anteriormente como global (antes de la ejecución del programa o función) el programa da prelación a la variable local, es decir, usa el valor asignado localmente y no el valor asignado globalmente. Si modifica la variable local, la global no será afectada.

## 4.5 VARIABLES DEL SISTEMA

El sistema tiene algunas variables que no podrán ser usadas para almacenar datos de forma arbitraria. Estas variables modifican valores de gráfica, regresiones, etc...

**Figura 8 Variables reservadas**

### TI-89 / TI-92 Plus Reserved Names

$\Delta t b1$	medx2	tc	zeye $\theta$
$\Delta x$	medx3	tmax	zeye $\phi$
$\Delta y$	medy1	tmin	zeyew
$\Sigma x$	medy2	tplot	zfact
$\Sigma x^2$	medy3	tstep	zmax
$\Sigma xy$	minX	u1(n) - u99(n)	zmin
$\Sigma y$	minY	u11 - ui99	znmax
$\Sigma y^2$	nc	$\bar{x}$	znmin
$\sigma x$	ncontour	xc	zplstep
$\sigma y$	ncurves	xfact	zplstrt
$\theta c$	nmax	xgrid	zscl
$\theta max$	nmin	xmax	zt $\theta de$
$\theta min$	nStat	xmin	ztmax
$\theta step$	ok	xres	ztmaxde
c1 - c99	plotStep	xscl	ztmin
corr	plotStrt	xt1(t) - xt1(t)	ztplotde
diftol	q1	$\bar{y}$	ztstep
dtime	q3	y1'(t) - y99'(t)	ztstepde
eqn	R <sup>2</sup>	y1(x) - y99(x)	zxgrid
errornum	r1( $\theta$ ) - r99( $\theta$ )	yc	zxmax
Estep	rc	yfact	zxmin
exp	regCoef	ygrid	zxres
eye $\theta$	regEq(x)	y11 - yi99	zxsc1
eye $\phi$	seed1	ymax	zygrid
eyew	seed2	ymin	zymax
fldpic	Sx	yscl	zymin
fldres	Sy	yt1(t) - yt99(t)	zyscl
main	sysData	z $\theta$ max	zzmax
maxX	sysMath	z $\theta$ min	zzmin
maxY	t $\theta$	z $\theta$ step	zzsc1
medStat	tblInput	z1(x,y) - z99(x,y)	
medx1	tblstart	zc	

Imagen extraída del documento Ti-TipList: <http://www.technicalc.org/tiplist/en/html/index.htm>

A estas deben sumarse:

$r_1, r_2, r_3, \dots, r_n$

$t_1, t_2, t_3, \dots, t_n$

$c_1, c_2, c_3, \dots, c_n$

$y_1, y_2, y_3, \dots, y_n$  (estas se pueden usar, pero son las variables de funciones de la ventana Graph)

## 4.6 CONSTANTES

Es posible que al evaluar y despejar ecuaciones aparezcan algunas constantes “@”. Para comprender plenamente las respuestas se debe conocer qué representa cada una. El sufijo aumentará automáticamente a medida que se realicen operaciones.

**Constantes @n# y @#:**

- Constantes @n# :

Las constantes @n1...@n255 representan cualquier número entero arbitrario, aunque esta notación identifica números enteros arbitrarios distintos.

Es común que las constantes aparezcan mientras resuelve ángulos o si existen funciones trigonométricas en la ecuación; esto se debe a que existen múltiples ángulos y respuestas que cumplen las condiciones.

- Constantes @# :

Las constantes @1...@ 255 representan cualquier número.

Así mismo, es común que aparezcan constantes mientras resuelve sistemas de ecuaciones; esto se debe a que algunas respuestas son paramétricas, por lo tanto se cumplen las condiciones para cualquier valor @#. Si desea una respuesta particular reemplace la constante por un valor numérico y evalúe de nuevo.

## 5. DIFERENCIA ENTRE PROGRAMA Y FUNCIÓN

Tanto los programas como las funciones son desarrollos de software muy útiles para ampliar la funcionalidad de nuestras calculadoras.

En esta sección explicamos con detalle cuales son las diferencias entre estos dos tipos de sistemas.

### 5.1 CUANDO USAR UN PROGRAMA

Un programa creado en *Program Editor* o *Daisuke-Edit* es muy similar a las FLASH APPS ya que cuenta con una interfaz gráfica.

Los programas (al igual que las funciones) son idóneos para realizar operaciones o tareas repetitivas, ya que sólo es necesario escribirlas una vez para poder utilizarlas tantas veces como sea necesario. Los programas ofrecen ciertas ventajas que las funciones no:

- Pueden crearse programas para ampliar las aplicaciones incorporadas en la TI-89 / Voyage™ 200 PLT, siendo su uso similar al de cualquier otra aplicación.
- Los programas permiten el uso de interfaces gráficas para facilitar el ingreso de datos y la forma como se muestran los resultados, ya que permite el uso de tablas, gráficas, imágenes, menús, diálogos, etc.... Las funciones carecen de esta ventaja.
- Los programas permiten el uso de estructuras Try... Else... EndTry para capturar errores y realizar validaciones y depuración de errores.
- Véase el numeral 5.3 para más información.

## 5.2 CUANDO USAR UNA FUNCIÓN

Una función creada en *Program Editor* o *Daisuke-Edit* es muy similar a las funciones e instrucciones utilizadas habitualmente en la pantalla Home.

Las funciones (al igual que los programas) son idóneas para realizar operaciones o tareas repetitivas, ya que sólo es necesario escribirlas una vez para poder utilizarlas tantas veces como sea necesario. No obstante, las funciones ofrecen ciertas ventajas que los programas no:

- Pueden crearse funciones que amplíen las incorporadas en la TI-89 / Voyage™ 200 PLT, siendo su uso similar al de cualquier otra función.
- Las funciones devuelven valores que pueden representarse gráficamente o introducirse en una tabla; los programas carecen de esta ventaja.
- Las funciones (no los programas) pueden utilizarse en expresiones. Por ejemplo:  $3 * \text{func1}(3)$  es válido, no  $3 * \text{prog1}(3)$ .
- Véase el numeral 5.3 para más información.

## 5.3 COMPARACIÓN ENTRE FUNCIÓN Y PROGRAMA

**Tabla 4** Tabla comparativa entre funciones y programas

Capacidad	Programa	Función
¿Permite el uso de argumentos?	Sí	Sí
¿Permite devolver valores a la pantalla de HOME?	No por sí sólo. Es posible usando programas externos.	Sí
¿Permite ser usado como una función para asignar un valor a una variable?	No por si sólo, pero puede usar variable globales para modificar una variable externa.	Sí
¿Permite el uso de variables locales?	Sí	Sí, de hecho todas las variables <b>deben</b> ser locales.
¿Permite el uso de variables globales?	Sí	No
¿Permite elementos gráficos?	Sí	No
¿Permite el uso de estructuras Try...Else..EndTry para atrapar y tratar errores?	Sí	No
¿Que estructuras son permitidas?	Permite todas las estructuras existentes de Ti-Basic	Cycle Define Exit For...EndFor Goto If...EndIf (en todas sus formas) Lbl Local Loop...EndLoop Return While...EndWhile
¿Qué funciones son permitidas?	Permite todas las funciones existentes de Ti-Basic	Pueden emplear todas las funciones incorporadas en la TI-89 / Voyage™ 200 PLT excepto: setFold, setGraph, setMode, setTable, switch
¿Permite llamar programas como subrutinas?	Sí	No
¿Permite definir programas?	Sí	No
¿Permite definir funciones?	Si	Si, pero sólo funciones locales, no globales.

## 6. INGRESO Y SALIDA DE DATOS

Los datos de entrada son importantes ya que esta es la información que el usuario del programa/función brinda para realizar el procesamiento de datos.

Dependiendo de la naturaleza y complejidad de cada desarrollo se pueden escoger diferentes métodos de pedirle información al usuario. Lo mismo pasa con la salida de datos; estos son igualmente importantes ya que son el resultado obtenido con el programa/función.

A continuación los métodos básicos de ingreso y salida de datos para programas y funciones.

### 6.1 EN PROGRAMAS

#### 6.1.1 Entrada de datos como argumentos

Los programas pueden recibir “argumentos” como datos de entrada. Los argumentos son variables que se definen dentro del programa de tal forma que el usuario pueda meter la información de la siguiente forma:

```
carpeta\prog(Arg_1, Arg_2, Arg_3, ..., Arg_n)
```

en donde Arg\_# es cada argumento. Y los escribe desde la ventana HOME sin necesidad de usar interfaces gráficas de ingreso de datos.

El código en TI-BASIC para que ese programa pueda usar esos argumentos debe ser así:

```
:prog(a,b,c,...,n)
:Prgm
...
:EndPrgm
```

El orden en que se escriban los argumentos desde la venta HOME, es el mismo orden en que se cargan las variables al interior del programa; de esa forma:

```
a=Arg_1
b=Arg_2
c=Arg_3
...
n=Arg_n
```



**Consideraciones:**

- En TI-BASIC, a diferencia de otros lenguajes de programación, no se debe especificar el tipo de variable de cada argumento.
- Los programas de TI-BASIC permiten un número fijo de argumentos, por lo tanto, si se definen **n** argumentos dentro del programa, el usuario debe escribir **n** argumentos para poder ejecutarlo. No existe la posibilidad de *argumentos opcionales*.
- Todas las variables usadas como argumentos se convierten automáticamente en variables locales, por lo tanto deben tenerse las consideraciones de las variables locales.
- Todos los argumentos se definen dentro del paréntesis que debe escribirse después del programa. No es posible crear argumentos que se escriban de forma distinta.

**6.1.2 Pantalla Program I/O (Entrada/Salida)**

La Pantalla Program I/O (E/S en español) es un recurso que la calculadora tiene para pedir y mostrar información durante la ejecución de un programa. Tiene un aspecto similar al HOME pero no tiene una línea de comando.

A Continuación se explican los métodos básicos de ingreso y Salida de datos usando las funciones de la Pantalla Program I/O.

**Nota:** Cuando se usan estos métodos, debe asegurarse de que el programa vuelva a la pantalla HOME al finalizar la ejecución del mismo; para hacer esto puede usar la instrucción **DispHome** antes de la línea de finalización de programa **EndPrgm**.

**6.1.2.1 Entrada de datos en IO**

La ventaja de los ingresos de datos usando la pantalla Program I/O es que cuentan con cierta validación de datos dependiendo del comando empleado. Son ingresos sencillos, únicos (una vez ingresados el usuario no se puede devolver por sí solo) y la interfaz gráfica puede ser complementada con otros comandos.

### 6.1.2.1.1 Input

Interrumpe el programa momentáneamente, muestra un texto (definido por el programador) en la pantalla Program I/O, espera a que se introduzca una expresión, y almacena dicha expresión en una variable (también definida por el programador).

El comando tiene la siguiente estructura:

```
:Input [promptCadena], var
```

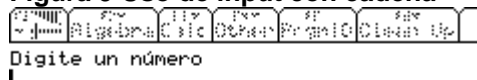
En donde *promptCadena* es una argumento opcional. Debe ser de tipo cadena (*string*) y es el texto que muestra antes de pedir el valor de la variable.

*var* es la variable en donde la información digitada por el usuario será almacenada.

#### Ejemplo 1:

```
:Input "Digite un número",a
```

**Figura 9 Uso de input con cadena**



ECUACION DEG APPROX FUNC 0/30

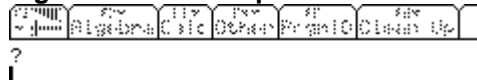
Esta línea almacenará el *el número ingresado* en la variable *a*.

#### Ejemplo 2:

Si omite *promptCadena*, aparece el indicador "?".

```
:Input a
```

**Figura 10 Uso de input sin cadenas**



ECUACION DEG APPROX FUNC 2/30

Esta línea almacenará el *el número ingresado* en la variable *a*.

**Consideraciones:**

- *Input* procesa toda la información digitada como expresiones simbólicas y variables, por lo tanto, si necesita recibir una cadena (texto) el usuario tendrá que usar comillas, ej. "info ingresada". Para este caso se recomienda el uso de *InputStr*.
- *Input* tiene un sistema de validación de datos que revisa la correcta sintaxis de los valores ingresados usando este comando. Esto puede ser útil si el programador no quiere hacer rutinas de validación.
- El usuario no puede cambiar el valor después de ingresarlo. El programador tendría que dar la posibilidad, a través de código, de que el usuario se pueda devolver a cambiar el valor.

### 6.1.2.1.2 InputStr

Interrumpe el programa momentáneamente, muestra un texto (definido por el programador) en la pantalla Program I/O, espera a que se introduzca una expresión, y almacena dicha expresión en una variable (también definida por el programador).

El comando tiene la siguiente estructura:

```
:InputStr [promptCadena], var
```

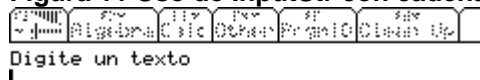
En donde *promptCadena* es un argumento opcional. Debe ser de tipo cadena (*string*) y es el texto que muestra antes de pedir el valor de la variable.

*var* es la variable en donde la información digitada por el usuario será almacenada.

#### Ejemplo 1:

```
:InputStr "Digite un texto",a
```

**Figura 11 Uso de InputStr con cadena**



ECUACION      DEG APPROX      FUNC 0/30

Esta línea almacenará el *texto ingresado* en la variable *a*.

#### Ejemplo 2:

Si omite *promptCadena*, aparece el indicador "?".

```
:InputStr a
```

**Figura 12 Uso de InputStr sin cadenas**



ECUACION      DEG APPROX      FUNC 2/30

Esta línea almacenará el *texto ingresado* en la variable *a*.

**Consideraciones:**

- *InputStr* procesa toda la información digitada como cadenas, por lo tanto, si necesita recibir una cadena (texto) el usuario **no** tendrá que usar comillas, ej. info ingresada. Si lo que necesita es ingresar expresiones algebraicas o numéricas utilice *Input*.
- El usuario no puede cambiar el valor después de ingresarlo. El programador tendría que dar la posibilidad, a través de código, de que el usuario se pueda devolver a cambiar el valor.

### 6.1.2.1.3 Prompt

Presenta el indicador *var?* en la pantalla Program I/O para cada variable de la lista de argumentos. Almacena la expresión que se introduzca en la variable correspondiente.

El comando tiene la siguiente estructura:

```
:Prompt var[,var2][,var3]
```

En donde *var* es la variable en donde la información digitada por el usuario será almacenada. El comando permite el uso de múltiples variables, es decir, con una sola línea puede pedir información para varias variables.

#### Ejemplo 1:

```
:Prompt var
```

**Figura 13 Uso de Prompt para una variable**



ECUACION DEG APPROX FUNC 0/30

Esta línea almacenará el *la información ingresada* en la variable *var*.

#### Ejemplo 2:

```
:Prompt var,var2
```

**Figura 14 Uso de Prompt para dos variables**



ECUACION DEG APPROX FUNC 3/30

Esta línea almacenará el *la información ingresada* en las variables correspondientes: *var* y *var2*.

**Consideraciones:**

- *Prompt* procesa toda la información digitada como expresiones simbólicas y variables, por lo tanto, si necesita recibir una cadena (texto) el usuario tendrá que usar comillas, ej. "info ingresada". Para este caso se recomienda el uso de *InputStr*.
- *Prompt* tiene un sistema de validación de datos que revisa la correcta sintaxis de los valores ingresados usando este comando. Esto puede ser útil si el programador no quiere hacer rutinas de validación.
- Es muy útil cuando las variables tienen un nombre familiar a la temática del programa, por lo tanto con el indicador "var?" el usuario identifica la información que se le está pidiendo.
- El usuario no puede cambiar el valor después de ingresarlo. El programador tendría que dar la posibilidad, a través de código, de que el usuario se pueda devolver a cambiar el valor.

### 6.1.2.2 Salida de datos en IO

La ventaja de la salida de datos usando la pantalla Program I/O es que puede mostrar tanto textos como expresiones algebraicas, listas y matrices (imágenes no).

#### 6.1.2.2.1 Output

Presenta exprOCadena (una expresión o cadena de caracteres) en la pantalla Program I/O en las coordenadas (fila, columna). No detiene el flujo del programa, es decir, el programa se sigue ejecutando sin parar.

El comando tiene la siguiente estructura:

```
: Output fila,columna,exprOCadena
```

En donde:

fila: la fila en donde empieza exprOCadena

columna: la columna en donde empieza exprOCadena

exprOCadena: La expresión o cadena a mostrar

#### Ejemplo:

```
: Output 2,10,"Coordenada 2,10"
```

**Figura 15 Uso de Output con una cadena**



EQUATION      DEG APPROX      FUNC 1/30

Esta línea mostrará *la información deseada* en la coordenada (Fila, Columna).



### 6.1.2.2.2 Disp

Presenta exprOCadena (una expresión o cadena de caracteres) en la pantalla Program I/O. No detiene el flujo del programa, es decir, el programa se sigue ejecutando sin parar.

El comando tiene la siguiente estructura:

```
: Disp exprOCadena
```

En donde:

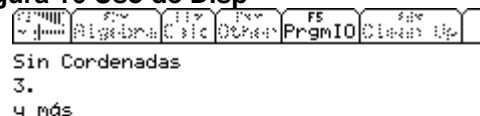
exprOCadena: La expresión o cadena a mostrar

A diferencia de **Output** con **Disp** no es necesario preocuparse por la ubicación, el mismo ubica el siguiente renglón y lo usa. En caso de no caber en pantalla se desplaza la pantalla verticalmente. Además de esta ventaja, permite mostrar más de un dato usando una sola línea: se consigue separando cada expresión o cadena con comas. Ver Ejemplo.

#### Ejemplo:

```
: Disp "Sin Cordenadas",1+2,"y más"
```

Figura 16 Uso de Disp



ECUACION DEG APPROX FUNC 20/30

En el ejemplo se usó Disp para mostrar tres datos: dos cadenas y una expresión. Nótese que la expresión es evaluada antes de ser mostrada y que cada coma representa un renglón nuevo.

#### Consideraciones:

- Si la expresión o cadena es muy larga no podrá verse por completo, ya que el comando no permite desplazarse horizontalmente. Por lo tanto, debe considerarse el largo de la información a mostrarse con este método.

### 6.1.2.2.3 Pause

Presenta exprOCadena (una expresión o cadena de caracteres) en la pantalla Program I/O. Detiene el flujo del programa, es decir, el programa para al ejecutar este comando.

El comando tiene la siguiente estructura:

: Pause exprOCadena

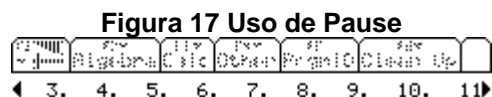
En donde:

exprOCadena: La expresión o cadena a mostrar

A diferencia de **Output** con **Pause** no es necesario preocuparse por la ubicación, el mismo ubica el siguiente renglón y lo usa. En caso de no caber en pantalla se desplaza la pantalla verticalmente. Este comando no permite mostrar más de un dato en la misma línea pero tiene una funcionalidad extra: si el dato mostrado es muy largo permite desplazamiento horizontal (**Disp** y **Output** no).

#### Ejemplo:

:Pause {1,2,3,4,5,6,7,8,9,10,11,12}



ECUACION DEG APPROX FUNC 1/30 12:00

En el ejemplo se usó **Pause** para mostrar una lista de 12 elementos. Nótese que la lista no cabe en una sola pantalla y se permite el desplazamiento horizontal para verla completa.

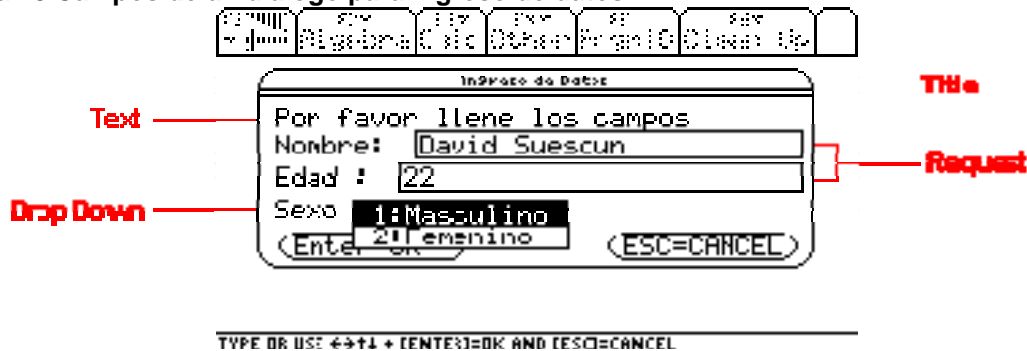
#### Consideraciones:

- Por el hecho de detener el flujo del programa el usuario está obligado a presionar ENTER o ESC cada vez que se use Pause en el programa.

### 6.1.3 Cuadros de Diálogo

Esta opción de ingreso de datos es la más flexible ya que permite mostrar e ingresar información y seleccionar opciones. A continuación una imagen explicativa de los campos de un diálogo:

Figura 18 Campos de un diálogo para ingreso de datos



*Dialog...EndDlog* genera un recuadro de diálogo cuando se ejecuta el programa.

El bloque puede ser de una línea o una serie de líneas. Las opciones válidas de bloque son: *Text*, *Request*, *DropDown* y *Title*.

- **Title**

Con *Title* se asigna un título al diálogo.

```
:Title "Cadena a usar de título"
```

- **Text**

*Text* imprime una cadena ya sea para mostrar información requerida en el momento de pedir los datos o para imprimir resultados.

Para mostrar información:

```
:Text "texto a mostrar"
```

Para mostrar un variable **var** tipo cadena:

```
:Text var
```

Para mostrar un variable **var2** tipo expresión:

```
:Text string(var2)
```

Para mostrar mezclas de texto e imprimir resultados

```
:Text "var2="&string(var2)& " unidades"
```

- **Request**

*Request* genera un campo en donde el usuario puede ingresar el dato solicitado, que será almacenado en la variable indicada como un dato cadena<sup>4</sup>.

Para ingresar cadenas

```
:Request "Cadena",var
```

Para ingresar número o expresiones

```
:Request "Expresión",var,0
```

Los datos almacenados por este comando son de tipo cadena, por lo tanto, si se ingresa un “número” este será almacenado como STRING y no como EXPR así que debe ser convertido, para hacer esto debe usar el comando **exp.**, la estructura es la siguiente:

```
: expr (var)→var
```

- **DropDown**

*DropDown* genera una lista desplegable para determinar las opciones disponibles; la opción seleccionada le asigna un valor numérico a una variable determinada.

Usando una lista fija

```
:DropDown "Texto",{ "Opción 1","Opción 2","Opción 3"},var
```

Usando una variable **lista** tipo lista de elementos tipo cadena

```
:DropDown "Texto",lista,var
```

Las variables en un recuadro de diálogo pueden tener valores que se mostrarán como los valores por omisión (o iniciales). Lo que significa que se puede hacer precarga de valores en caso de que sea necesario, y es posible actualizar los campos con los valores ingresados por el usuario, así, cuando desee modificar los datos ingresados puede usar los mismos datos que ya había ingresado sin necesidad de hacerlo de nuevo.

- **Variable del Sistema: Ok**

Los diálogos cuentan con dos opciones aparte de las programadas: *Enter* y *ESC*. Cada uno modifica la variable *ok*, *Enter* le asigna 1 y *ESC* le asigna 0. Esto permite validar el ingreso de datos o realizar acciones distintas para cada caso.

<sup>4</sup> De forma predeterminada el programa cambia su teclado a alfa, pero agregando “,0” al final de la instrucción *request* se habilita el teclado numérico

Los bloques también tienen desventajas frente a otras formas de ingreso de datos. La más importante para tener en consideración es la de dimensión y validación.

El número de caracteres máximos no es un valor fijo ya que el tipo de letra usado en las calculadoras es proporcional (no fijo), por lo que se deben realizar pruebas en los diferentes modelos para asegurarse de que no existan errores de dimensión. Para tener certeza de la compatibilidad entre modelos, se desarrollaron tres programas que aumentaban consecutivamente el número de caracteres hasta llegar al punto en donde un error de dimensión apareciera. Para este propósito se usaron cadenas con la letra "M" ya que es la que mayor cantidad de píxeles requiere sobre el eje horizontal. Concluyendo:

Ti-89, Ti.89 Titanium

Título: 25 Caracteres.

*DropDown*: 18 Caracteres por elemento.

Ti 92, Ti-92 Plus y voyage 200

Título: 48 Caracteres.

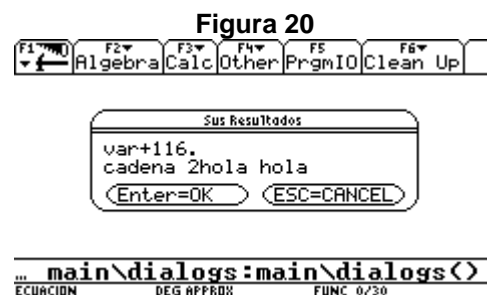
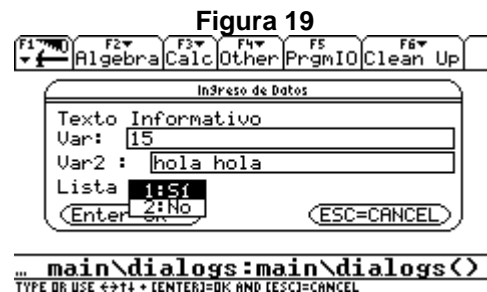
*DropDown*: 30 Caracteres por elemento.

En cuanto al número de elementos en un dialogo, tras realizar pruebas, se determinó que siete elementos es el número óptimo para no crear problemas de incompatibilidad entre los cinco modelos de calculadoras.

Estos datos fueron experimentales y no proporcionados por la Texas Instruments.

### Ejemplo:

```
:Dialog
:Title "Ingreso de Datos"
:Text "Texto Informativo"
:Request "Var",varnum,0
:Request "Var2 ",varchar
:DropDown "Lista",{"Sí","No"},var3
:EndDlog
:expr (varnum)→varnum
:varnum+1→varnum
:Dialog
:Title "Sus Resultados"
:Text "var+1"&string(varnum)
:Text "cadena 2"&varchar
:EndDlog
```



Los elementos **Text** y **Request** pueden usarse fuera de la estructura **Dialog...EndDlog** cuando se necesita mostrar un solo texto o pedir un solo datos. Esto ahorra líneas de código y tiene la misma interfaz, sólo que de un elemento y sin título.

#### 6.1.4 PopUp

Presenta un menú desplegable que contiene las cadenas de caracteres de elementoLista, espera a que se seleccione un elemento, y almacena el número seleccionado en var.

```
:PopUp elementoLista, var
```

Los elementos de elementoLista deben ser cadenas de caracteres:

```
{elemento1Cadena, elemento2Cadena, elemento3Cadena, ...} → elementoLista
```

Si var ya existe y tiene un número de elemento válido, dicho elemento se muestra como la opción por omisión.

elementoLista debe contener al menos una opción.

A diferencia de otros métodos como el de diálogos, **PopUp** no modifica la variable de sistema **ok** así que si el usuario no selecciona ninguna opción y presiona ESC, el programa podría fallar al tratar de leer **var**, por lo que se recomienda cargar un valor a la variable **var** antes de usarla en un **PopUp**, puede ser 0 para diferenciarlo de las opciones posible que van de 1 en adelante.

#### Ejemplo:

```
:0→var
: PopUp {"Opción 1","Opción 2","Opción 3"},var
```



```
1:Opción 1
2:Opción 2
3:Opción 3
```

```
"","Opción 2","Opción 3"},var
TYPE OR USE <=>+ * (ENTER)=OK AND (ESC)=CANCEL
```

Dependiendo que opción seleccione, var será cargada con 1,2 o 3. Si el usuario presiona ESC **var** valdrá 0, ya que ese fue el valor con que se cargo **var** antes de usar el **PopUp**.

## 6.2 EN FUNCIONES

### 6.2.1 Entrada de datos como argumentos

Las funciones pueden recibir “argumentos” como datos de entrada. Los argumentos son variables que se definen dentro de la función de tal forma que el usuario pueda meter la información de la siguiente forma:

```
carpeta\funcion(Arg_1, Arg_2, Arg_3, ..., Arg_n)
```

en donde Arg\_# es cada argumento. Y los escribe desde la ventana HOME sin necesidad de usar interfaces gráficas de ingreso de datos.

El código en TI-BASIC para que esa función pueda usar esos argumentos debe ser así:

```
:prog(a,b,c,...,n)
:Func
...
:EndFunc
```

El orden en que se escriban los argumentos desde la venta HOME, es el mismo orden en que se cargan las variables al interior del programa; de esa forma:

```
a=Arg_1
b=Arg_2
c=Arg_3
...
n=Arg_n
```

#### Consideraciones:

- En TI-BASIC, a diferencia de otros lenguajes de programación, no se debe especificar el tipo de variable de cada argumento.
- Las funciones de TI-BASIC permiten un número fijo de argumentos, por lo tanto, si se definen **n** argumentos dentro de la función, el usuario debe escribir **n** argumentos para poder ejecutarlo. No existe la posibilidad de *argumentos opcionales*.
- Todas las variables usadas como argumentos se convierten automáticamente en variables locales, por lo tanto deben tenerse las consideraciones de las variables locales.

- Todos los argumentos se definen dentro del paréntesis que debe escribirse después de la función. No es posible crear argumentos que se escriban de forma distinta.

### 6.2.2 Salida de datos con Return

El único método de salida de datos que tiene una función es usando **Return**.

```
:Return [expresión]
```

Devuelve expresión como el resultado de la función. Se utiliza en un bloque Func...EndFunc o en un bloque Prgm...EndPrgm. En un programa no devuelve un dato sino que sale del programa.

Es posible usar varias veces Return en una función para devolver distintos valores, pero en las buenas prácticas de programación recomiendan usar un solo **Return** al final de la función.

#### Ejemplo:

```
:ejemplo(a,b)
:Func
:Local result
:If a>b Then
: a-b→result
:Else
: b-a→result
:EndIf
:Return result
:EndFunc
```

Figura 22 Uso de Return



La función del ejemplo revisa cuál de los dos argumentos es mayor y realiza **mayor-menor** y asigna el resultado a una variable local **result** y al final de la función devuelve el resultado usando **Return result**.



## 7. MENÚS

Los menús son interfaces gráficas que facilitan el acceso a opciones y funcionalidades de los programas y la calculadora en sí (así como para otros sistemas como los PC, los celulares, etc...)

A continuación se explican los dos tipos de menú usados en TI-BASIC:

### 7.1 TOOLBAR

*ToolBar* crea un menú en la barra de herramientas. El bloque puede ser de una sola opción o una sucesión de opciones separadas por renglones.

Las opciones pueden ser *Title* o *Item*. *Item* debe tener etiquetas. *Title* también debe tener una etiqueta si no contiene ningún *Item*. Este bloque permite el uso de imágenes en lugar de texto para mostrar las opciones. La imagen no debe exceder 16 píxeles de alto.

En la siguiente gráfica se muestran las partes de un menú típico del proyecto Daisuke-Ing. Se estableció “Archivo” como la primera opción (botón de función F1) y la opción para salir en el último botón de función disponible (varía según el número de opciones).

Figura 23 Ejemplo de Menú ToolBar



Estas etiquetas se definen dentro del código como *Lbl etiqueta*; al seleccionar una de las opciones del menú se ejecuta el comando *Goto etiqueta* transfiriendo el control del programa hasta *etiqueta*.

La estructura es la siguiente:

```
:Loop
:
: ToolBar
: Title "Opción 1",etiql
: Title "Opción 2"
: Item "Sub-opción 1",etiql2
: Item "Sub-opción 2",etiql3
: EndTbar
: Exit
:
: @Opción 1
: Lbl etiql
: CÓDIGO 1
: Cycle
:
: @Sub-opción 1
: Lbl etiql2
: CÓDIGO 2
: Cycle
:
: @Sub-opción 2
: Lbl etiql3
: CÓDIGO 3
: Cycle
:
:EndLoop
```

Para la estructura se usó el **ToolBar** dentro de un bucle **Loop**, se usó de esta forma para evitar el uso de Goto's.

En la **Opción 1** se usó la etiqueta **etiql** porque no tiene sub-elementos, por otro lado, se puede apreciar que la **Opción 2** no tiene etiqueta ya que tiene dos sub-elementos y son estos las que deben usarlas.

En el final del **ToolBar**, después de **EndTbar** se usó un **Exit**, ya que si se el programa se encuentra mostrando el **ToolBar** y el usuario presiona ESC esa será la acción a realizar, si no estuviese el **Exit** haría la misma acción que si hubiera seleccionado la **Opción 1**.

Consejos: Para poner el fondo del menú ToolBar en blanco (de forma predeterminada muestra el home) la forma más práctica es usando: `:ClrIO:Disp` justo antes, mire el ejemplo.

**Ejemplo:**

```

: Loop
: ClrIO:Disp
: Toolbar
: Title "Opción 1",etiql
: Title "Opción 2"
: Item "Sub-opción 1",etiql2
: Item "Sub-opción 2",etiql3
: EndTbar
: Exit
: @Opción 1
: Lbl etiql
: Text "Opción 1"
: Cycle
: @Sub-opción 1
: Lbl etiql2
: Text "Sub-Opción 1"
: Cycle
: @Sub-opción 2
: Lbl etiql3
: Text "Sub-Opción 2"
: Cycle
:
: EndLoop

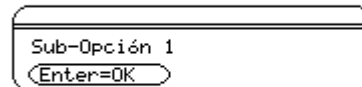
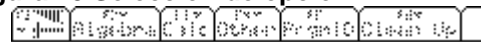
```

**Figura 24 Menú Toolbar**


---

 TYPE OR USE <=>+/-+ENTER=OK AND ESC=CANCEL
 

---

**Figura 25 Selección de opción**


---

 ECUACION DEG APPROX FUNC 3/30
 

---

**Consideraciones:**

- Mientras se ejecuta el Toolbar el programa sólo reconoce teclas para selección de opciones y ESC, no se pueden programar teclas para realizar otras acciones.
- No se puede usar código dentro de **Toolbar...EndTbar** para modificarlo durante ejecución. Por ejemplo si se quiere una opción especial para alguna condición no se podrá usar If.

## 7.2 CUSTOM

Este menú no se ejecuta dentro de un programa sino desde el HOME. Configura una barra de herramientas que se activa al pulsar 2ND+CUSTOM . Es muy similar a la instrucción ToolBar, excepto que los enunciados Title e Item no pueden tener etiquetas.

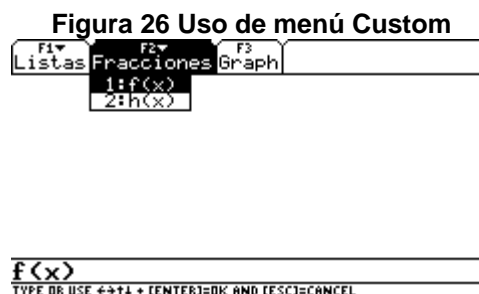
```
:Custom
:bloque
:EndCustm
```

Nota: 2ND+CUSTOM actúa como un conmutador. La primera vez llama al menú y la segunda vez lo cierra. El menú también se cierra cuando se cambia de aplicación.

Sirve para copiar el texto del elemento en la línea de comandos del HOME. Los programadores pueden usar los menú Custom para crear “accesos directos” a sus programas y funciones. Debe tener en cuenta que sólo puede haber un menú custom definido a la vez. Y será reemplazado en el momento que otro programa trate de definir uno nuevo.

### Ejemplo:

```
:prueba()
:Prgm
:Custom
:Title "Listas"
:Item "Lista1"
:Item "Puntajes"
:Item "L3"
:Title "Fracciones"
:Item "f(x)"
:Item "h(x)"
:Title "Graph"
:EndCustm
:EndPrgm
```



Sólo pueden definirse dentro de un programa y no una función.

## 8. CONTROL DE FLUJO

El control del flujo es la manera que tiene un lenguaje de programación de provocar que el flujo de la ejecución avance y se ramifique en función de los cambios de estado de los datos. La ramificación, iteración, selección y llamadas a subrutina son formas de control de flujo.

### Operadores Lógicos:

El control de flujo dependerá de los condicionales que introduzca el programador y es vital que preste mucha atención a la hora de su ingreso. Los diferentes tipos de condicionales que el programador podrá ingresar son:

Operador	Verdadero Si:	Ejemplo
>	Mayor que	$a > 8$
<	Menor que	$a < 0$
$\geq$	Mayor o igual que	$a + b \geq 100$
$\leq$	Menor o igual que	$A + 6 \leq b + 1$
=	Igual	List1=list2
≠	Distinto de	mat1≠mat2

### Operadores Booleanos:

El programador también puede vincular distintas condiciones con operadores booleanos de forma tal que el condicional final sea una combinación de condicionales aislados.

Operador	Verdadero Si:	Ejemplo
and	Ambas son condiciones son verdaderas	$a > 0$ and $a \leq 10$
or	Al menos una condición es verdadera	$a < 0$ or $b + c > 10$
xor	Una prueba es verdadera y la otra falsa	$a + 6 < b + 1$ xor $c < d$
not	Cuando la condición final es falsa	$6 < 5$

A continuación se explicarán distintos métodos para realizar control de flujo de datos.

## 8.1 CONDICIONAL IF

Comando que sirve para elegir entre una acción u otra dependiendo de una o varias condiciones establecidas por el programador.

### 8.1.1 Para una sola instrucción

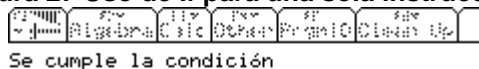
PARA EFECTUAR UNA SOLA ACCIÓN EN CASO DE CUMPLIR LA CONDICIÓN REQUERIDA POR EL PROGRAMADOR SE DEBE USAR LA SIGUIENTE ESTRUCTURA:

```
:...
:If condición
:   Instrucción a realizar
:...
```

**Ejemplo:**

```
:
:4→x
:If x>3
:  Pause "Se cumple la condición"
:
```

**Figura 27** Uso de If para una sola instrucción



En el ejemplo se carga la variable **x** con **4**, por lo tanto cumple la condición **x>3** y muestra el mensaje estipulado. Si se hubiese cargado la variable **x** con **3** o un número menor, no hubiese mostrado el mensaje.

### 8.1.2 Para múltiples instrucciones

Para efectuar más de una acción en caso de cumplir la condición requerida por el programador se debe usar la siguiente estructura:

```
:
:If condición Then
:  Instrucción 1
:  Instrucción 2
:  ...
:  Instrucción n
:EndIf
```

:  
**Ejemplo:**

```
:
:If 5>4 or 3>4 Then
: Disp "Instrucción 1"
: Disp "Instrucción 2"
: Disp "...
: Pause "Instrucción n"
:EndIf
:
```

**Figura 28 Uso de If para una varias instrucciones**



En el ejemplo ponen dos condiciones: 1. **5>4** y 2. **3>4**. Las condiciones están vinculadas con el operador booleano **or**. La primera condición resulta en verdadero, la segunda resulta en falso. Como la condición final requiere que cumple sólo una de las condiciones la condición final es verdadera, por lo tanto ejecuta las “n” instrucciones. Si el operador booleano hubiese sido **and** la condición final sería falsa y no hubiera ejecutado las instrucciones contenidas entre If.. EndIf.

### 8.1.3 Para caso verdadero y falso

Cuando se deben ejecutar un set de instrucciones en caso de que la condición sea verdadera y otro set cuando la condición sea falsa, es posible hacer esto usando la siguiente estructura:

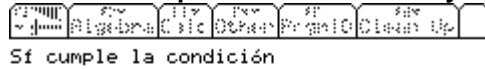
```
:
:If condición Then
: ...
: Set de instrucciones en caso de ser verdadero
: ...
:Else
: ...
: Set de instrucciones en caso de ser falso
: ...
:EndIf
:
```

**Ejemplo:**

```

:
:4→x
:If x>3 Then
: Pause "Sí cumple la condición"
:Else
: Pause "NO cumple la condición"
:EndIf
:

```

**Figura 29 Uso de If para caso verdadero y falso**

MAIN DEG APPROX FUNC 2/30 END

En el ejemplo se asigna a x un valor de 4, por lo tanto cumple la condición  $x > 3$  y ejecuta el correspondiente código. Si x tuviese un valor de 3 o menor no cumpliría la condición.

**8.1.4 Para varios casos**

Cuando se necesitan diferentes acciones para diferentes casos (condiciones), es posible lograrlo usando la siguiente estructura:

```

:If condición-1 Then
:  acción-a1
:  acción-a2
:...
:  acción-an
:ElseIf condición-2 Then
:  acción-b1
:  acción-b2
:...
:  acción-bn
:...
:ElseIf condición-n Then
:  acción-x1
:  acción-x2
:...
:  acción-xn
:EndIf

```

Ahora si el programador además quiere efectuar una acción en caso de no cumplirse ninguna de las condiciones, solo deberá anexarse este código dentro del if (en vez de sólo **EndIf**)

```

:Else
:  acción-y1
:  acción-y2
:...
:  acción-yn
:EndIf

```



**Ejemplo:**

```

:
:4→x
:If x=0 Then
: Pause "x es igual a 0"
:ElseIf x=1 Then
: Pause "x es igual a 1"
: x+1→x
:ElseIf x=2 Then
: Pause "x es igual a 2"
:Else
: Pause "x es distinto a 0,1,2"
:EndIf
:

```

**Figura 30 Uso de If para varios casos**

En este ejemplo asignamos un valor de 4 a la variable x, por lo tanto no cumple las condiciones **x=0** ni **x=1** ni **x=2**, por lo tanto ejecuta el código para cualquier otra condición: **Else**.

**Consideraciones:**

- Sólo cumplirá UNA condición. Si cumple la primera y modifica una variable en donde teóricamente cumpliría otras de las condiciones el caso no será evaluado. Ejemplo: En el ejemplo si  $x=1$  entra a ese caso y convierte  $x=2$ , el código `Pause "x es igual a 2"` no será ejecutado.

**8.2 CONDICIONAL WHEN**

Este condicional/función hace casi lo mismo que un condicional de estructura If..Else ya que permite definir una condición y obtener un resultado cuando sea verdadera y otro cuando sea falsa. Tiene además la opción de definir un tercer resultado cuando el valor de la condición sea desconocido. Los posibles resultados deben ser expresiones, variables o boléanos, no permite usar otro tipo de código u acción. Tiene el mismo funcionamiento de una función, así que devuelve un valor ya sea para usarlo en un condicional, como argumento, para asignárselo a una variable o para definir una función. Tiene la siguiente estructura:

```
when(condición,verdadero[,falso][,desconocido])
```

En donde:

**condición:** La condición que determina el resultado

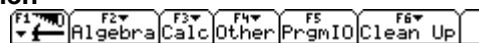
**verdadero:** el valor a devolver si la condición es verdadera

**falso:** el valor a devolver si la condición es falsa. Este argumento es opcional.

**desconocido:** cuando se desconoce el valor de la condición. Este argumento es opcional.

**Ejemplo 1:**

:  $2 \rightarrow x$   
 :when ( $x \geq 2$ , "Sí", "No", "Desconocido")

**Figura 31** Uso de la función/condicional When

■  $2 \rightarrow x$  2.  
 ■ { "Si",  $x \geq 2$   
   "No", else "Si"  
   "Desconocido", UNDEF  
**when( $x \geq 2$ , "Si", "No", "Desconoci...**  
 ECUACION DEG APPROX FUNC 2/30

**Ejemplo 2:**

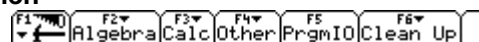
:  $1 \rightarrow x$   
 :when ( $x \geq 2$ , "Sí", "No", "Desconocido")

**Figura 32** Uso de la función/condicional When

■  $1 \rightarrow x$  1.  
 ■ { "Si",  $x \geq 2$   
   "No", else "No"  
   "Desconocido", UNDEF  
**... $x \geq 2$ , "Si", "No", "Desconocido">**  
 ECUACION DEG APPROX FUNC 2/30

**Ejemplo 3:**

: undef  $\rightarrow x$   
 :when ( $x \geq 2$ , "Sí", "No", "Desconocido")

**Figura 33** Uso de la función/condicional When

■ undef  $\rightarrow x$  undef  
 ■ { "Si",  $x \geq 2$   
   "No", else "Desconocido"  
   "Desconocido", UNDEF  
**... $x \geq 2$ , "Si", "No", "Desconocido">**  
 ECUACION DEG APPROX FUNC 2/30

En los tres ejemplos se usó la misma estructura pero un diferente valor para  $x$ , de esta forma se pueden apreciar los tres posibles casos. Note que como resultados devolvimos cadenas, también es posible devolver otros tipos de datos, lo que es de gran ayuda para crear gráficas, evaluar expresiones, generar diálogos dinámicos, etc...

Nótese que es una función con un condicional, por lo tanto una estructura como:

```
:when (condición, verdadero, falso, desconocido) → var
```

Reemplaza rápidamente una estructura como:

```
:If condición=true Then
: verdadero→var
:ElseIf condición=false Then
: falso→var
:Else
: desconocido→var
:EndIf
```

Que debe ser la forma interna de procesar **when**.

Por otro lado la función **when** permite anidarse, es decir, usar **when** dentro de **when**, lo que permite evaluar más de un caso, una estructura básica sería:

```
:when (condición1, verdadero, when (condición2, verdadero2, falso2))
```

En este caso se omitió el último argumento “desconocido” para los dos casos por que la idea es mostrar como se anida. Este proceso se puede realizar cuantas veces sea necesario y es muy útil para definir funciones con diferentes valores para diferentes rangos de datos.

Anidando **when** también es posible reemplazar estructuras **If..Elseif..Elseif... Else...EndIf** de cualquier cantidad de casos.

## 8.3 BUCLES

Los bucles le permiten al programador repetir una a mas acciones las veces que el desee, sin tener que repetir el código. Su uso es ideal para problemas iterados, para realizar búsquedas u ordenamientos o muchas acciones donde el programador note un cierto patrón repetitivo dentro del código o si desea que el programa se ejecute más de una vez para hacer diferentes corridas sin salir del mismo.

Un factor muy importante al usar bucles es que todos necesitan puntos de salida, esto significa que una condición debe presentarse para que el bucle termine. De no ser así el programa entrará en una repetición infinita que “bloquea” la calculadora, consumiendo recursos y gastando batería. En caso de que esto pase se debe para la ejecución usando la tecla ENTER. Tenga en cuenta que se puede salir de todos los bucles con el comando Exit.

El comando Cycle obliga a reiniciar el bucle. También aplica para los diferentes tipos de bucles.

### 8.3.1 For

La ventaja de este comando, respecto al resto de bucles, es que este permite inicializar una variable que funciona como contador, aumentándolo según un incremento definido cada vez que se realiza un ciclo (Cycle). Su uso es ideal para los casos en donde se conozca el número de repeticiones o se necesite hacer un barrido a listas o matrices. Tiene la siguiente estructura:

```
:For variable, inicio, fin, incremento
: acción-1
: acción -2
: ....
: acción -n
:EndFor
```

En donde:

**variable:** será la variable usada como contador

**inicio:** el valor inicial del contador

**fin:** el valor final del contador. Sale del bucle al alcanzar este valor.

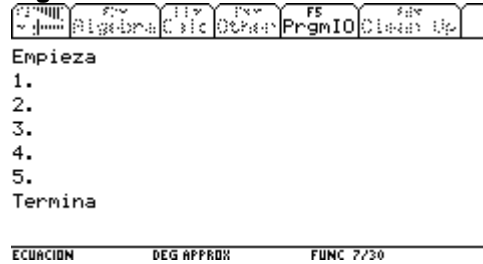
**incremento:** el valor en qué la **variable** incrementa hasta alcanzar **fin**

**Ejemplo 1:**

```

:
:{1,2,3,4,5}→lista
:Disp "Empieza"
:For i,1,5,1
: Disp lista[i]
:EndFor
:Disp "Termina"
:

```

**Figura 34 Uso del bucle For**

En este ejemplo usamos la variable *i* para leer las posiciones de una lista, aprovechamos que la variable incrementa de 1 en 1 hasta 5 para hacerlo.

**Consideraciones:**

- En TI-BASIC la línea `For variable, inicio, fin, incremento` se lee cada vez que repite el bucle, por lo tanto, si cambiamos el valor de la variable durante la ejecución del bucle podemos adelantar o atrasar en estado del bucle. Lo mismo para el incremento y fin, no aplica para inicio. Esto aparte de darle cierto dinamismo al bucle también trae más consideraciones, vea la siguiente.
- Es usual que los programadores usen como *fin* del `for` la longitud de una lista, así: `dim(lista)`. Esto es bueno cuando la lista se ha generado durante la ejecución y no conocemos su longitud a la hora de programar, pero también significa que cada vez que hace un ciclo debe ejecutar esa función, eso compromete la velocidad y el rendimiento del programa. En este caso se recomienda usar una variable temporal para tener un valor fijo cada vez que hace el ciclo, la estructura sería así:

```

:dim(lista)→vartemp
:For i,1,vartemp,1

```

### 8.3.2 While

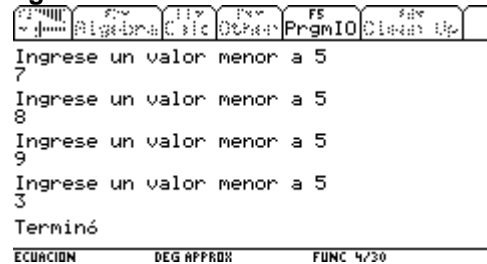
Este comando repetirá las acciones siempre y cuando se cumpla cierta condición (similar a las del if) ingresada por el programador. Su estructura es la siguiente:

```
:While condición
: acción-1
: acción-2
: ....
: acción-n
:Endwhile
```

#### Ejemplo:

```
:6→x
:While x>5
: input "Ingrese un valor menor a :5",x
:endwhile
:Disp "Terminó"
```

Figura 35 Uso de While



El bucle se repite hasta que no se ingrese un número menor a 5.

#### Consideraciones:

- Dado que en **While** la condición se evalúa desde que entra al bucle, se deben tener valores asignados a las variables usadas para evaluar la condición. Si en el ejemplo no se hubiese hecho **x=6** antes del **While** hubiera salido un error: "A test did not result to TRUE or FALSE".

### 8.3.3 Loop

Este comando es un ciclo que se repetirá infinitas veces a menos que se especifique la salida del mismo (se logra con el comando Exit). Con el uso de este bucle es posible usar múltiples salidas para diferentes condiciones en distintas partes del código, además permite “saltar” al inicio del bucle fácilmente con el comando Cycle. El bucle tiene la siguiente estructura

```
:Loop
: acción-1
: acción-2
: ....
: acción-n
: Código de corte
:EndLoop
```

#### Ejemplo:

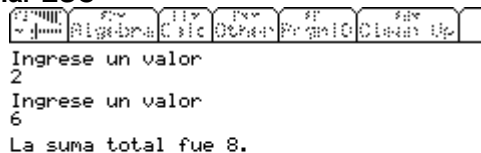
```
:0→suma
:Loop
: Input "Ingrese un valor",aux
: suma+aux→suma
: Dialog
: Text "¿Desea continuar?"
: EndDlog
: if ok=0
: Exit
:EndLoop
:Pause "La suma total fue "&string(suma)
```

Figura 36 Uso del bucle Loop

Figura 37



Figura 38 Acción al presionar ESC



El ingreso de valores se realiza hasta que en el diálogo de **Text** no se presione ESC, de esa forma se asigna 0 a la variable **ok** y ejecuta el comando de salida **Exit**.

## 8.4 ETIQUETAS LBL Y GOTO

Transfiere el control de un programa a la etiqueta *Nombre de etiqueta*.

*Nombre de etiqueta* debe estar definido en el mismo programa utilizando la instrucción Lbl.

La estructura es la siguiente:

```
:Lbl etiqueta
:
:...
:
:Goto etiqueta
```

La etiqueta puede definirse antes o después de invocar el GoTo.

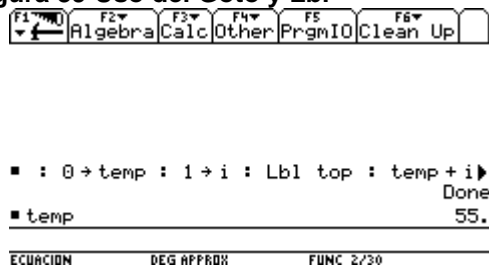
Estos comandos se han masificado mucho entre los programadores de TI-BASIC por su facilidad de uso. Esto no significa que sea una práctica recomendada.

La programación con Goto y Lbl es una programación spaghetti, no tiene un control de flujo fácilmente representable, es uno más bien enredado. En esta guía se motiva el uso de estructuras lógicas como los bucles y los condicionales para evitar el uso de Goto y Lbl, no sólo por que hacen que los programas se vuelvan difíciles de mantener y actualizar, sino por que también disminuye el rendimiento al necesitar más recursos y tiempo para saltar de ubicación a etiqueta.

### Ejemplo:

```
:0→temp
:1→i
:Lbl TOP
:temp+i→temp
:If i<10 Then
: i+1→i
: Goto TOP
:EndIf
:Disp temp
```

Figura 39 Uso del Goto y Lbl



En este ejemplo se hace un “bucle” con Goto y Lbl para alcanzar una condición. Se repite 10 veces, al finalizar muestra del resultado de incrementar i de uno en uno y sumarlo al acumulado total de temp, alcanzando un valor de 55.



Nótese que es una mejor práctica usar un bucle de los ya explicados para realizar esta misma acción, a continuación las alternativas:

Se cambió el 10 por 1000 y se midieron los tiempos de cada una

#### 1. Con **While**

```
:0→temp
:1→i
:temp+i→temp
:While i<1000
: i+1→i
: temp+i→temp
:EndWhile
:Disp temp
```

29 segundos

#### 2. Con **For**

```
:0→temp
:For i,1,1000
: temp+i→temp
:EndFor
:Disp temp
```

23 segundos

#### 3. Con **Loop**

```
:0→temp
:1→i
:Loop
:temp+i temp
:If i<1000 Then
: i+1→i
: Cycle
:EndIf
:Exit
:EndLoop
:Disp temp
```

30 segundos

#### 4. Con **Goto**: 33 segundos

Estas son sólo alternativas. Todas fueron probadas y el resultado fue el mismo. Cada programador tiene su propio estilo y pensamiento, estos factores se reflejan en la forma como codifica.

#### Consideraciones:

- Cada vez que se usa Goto, la calculadora vuelve a leer todo el código desde el principio en busca de la etiqueta, esto consume recursos y disminuye la velocidad del proceso. Observe los tiempo debajo de cada alternativa; se cambio el limite de 10 por 1000 y se contabilizaron las alternativas para hacer una comparación real de velocidad. Usando Goto la misma rutina tomó 33 segundos.

## 9. SUBROUTINAS

Existen varias formas de aprovechar un bloque de código para realizar acciones que se necesiten más de una vez. La primera opción son los bucles... Véase 8.3 BUCLES... que son de gran utilidad si el proceso se debe repetir en un mismo momento específico del proceso general, por otro lado, si el bloque a reutilizar se necesita en un instante diferente del proceso los bucles no son una alternativa. Algunos programadores optan por armar una red de Goto's y Lables (Lbl)... Véase 8.4 ETIQUETAS LBL Y GOTO... pero no es una práctica recomendable.

Si la acción a repetir se necesita en momentos no lineales del flujo de datos se la mejor alternativa es el uso de subrutinas. Estas consisten en funciones y programas con propósitos particulares. Con estas podemos ahorrar espacio al no duplicar bloques idénticos de código, además permiten programar modularmente, lo que facilita la detección y depuración de errores.

Las subrutinas no sólo sirven para procesos que se requieran más de una vez, también son útiles para reducir el tamaño de programas muy grandes; además del tamaño reducido también facilita una programación más ordenada, aún cuando sean procesos que se requieran una única vez.

Su estructura de uso es muy sencilla: Una subrutina se llama de la misma forma que se llaman funciones y programas desde el HOME:

```
:carpeta\program(args1,args2,...,argsn)
:carpeta\funcion(args1,args2,...,argsn)
```

Valga aclarar que se puede usar el resultado de una función para asignar un valor a una variable:

```
:carpeta\funcion(args1,args2,...,argsn)→var PERMITIDO
```

Pero no se puede usar un programa para asignar un valor a una variable:

```
:carpeta\program(args1,args2,...,argsn)→var NO PERMITIDO
```

Los programas y funciones usadas como subrutinas tienen las mismas ventajas y desventajas de cada uno de su tipo... Véase 5 DIFERENCIA ENTRE PROGRAMA Y FUNCIÓN...

Entendida su utilidad y el fin con el que la vaya a usar se debe tomar una decisión: Definir la subrutina localmente (dentro del programa/función principal, de uso

privativo) o de forma global (como un archivo independiente, asequible desde varios programas/funciones). A continuación una breve descripción de cada estructura y algunas consideraciones y beneficios de cada una.

## 9.1 SUBROUTINA LOCAL

Por lo general los usuarios desean programas de un solo archivo, esto debido a la facilidad de transferencia y de uso. Definir subrutinas locales ayuda a lograrlo.

Una subrutina local es aquella que se define dentro de un programa/función principal, por lo tanto solo puede ser usada por el (la) mismo(a), por eso decimos que son de uso privativo (Privado). Aún así, si definimos una subrutina dentro de un programa pero no la definimos como **Local** quedará disponible como global después de la ejecución del programa. La estructura es la siguiente:

### Programa:

```
:ProgPrin()
:Prgm
: Local subprgm
: Define subprgm(ar1,...arn)=Prgm
: .
: .
: Código
: .
: .
: EndPrgm
: .
: .
: .
:EndPrgm
```

### Función:

```
:ProgPrin()
:Prgm
: Local subfunc
: Define subfunc(ar1,...arn)=Func
: .
: .
: Código
: .
: .
: EndFunc
: .
: .
: .
:EndPrgm
```

Se observa claramente que el comando usado para definir la subrutina es **Define**. En el ejemplo definimos las subrutinas dentro de un programa. En el caso de las funciones es posible definir una función dentro de una función, pero nunca un programa dentro de una función.

Nótese también que se usó **Local** para definir la subrutina como tal, pero esta no es la única forma de definir una subrutina para uso exclusivo (al declarar subrutinas en funciones si es necesario que sea local). La otra forma de hacerlo es borrando la subrutina antes de terminar la ejecución del programa principal usando el comando **Delvar** seguido del nombre de la subrutina. Las subrutinas locales corren más rápido que las globales según los consejos de la Ti-TipList<sup>5</sup>

---

<sup>5</sup> <http://www.technicalc.org/tiplist/en/html/index.htm>

## 9.2 SUBROUTINA GLOBAL

Cuando se desarrolla una serie de programas para teorías particulares es común encontrar procesos idénticos para más de un programa. Una forma de optimizar el desarrollo es aislando el proceso en una subrutina para evitar duplicar código.

Una subrutina global es aquella que se define tal cuál hacemos programas y funciones normalmente; en un archivo independiente. Esto significa que la subrutina será asequible desde cualquier otro archivo de la calculadora (este es el caso de las librerías)... Véase 10 USO DE LIBRERÍAS...

## 9.3 CONSIDERACIONES DE CARPETA

De forma preestablecida se crean programa en la carpeta **main**, pero si todo el mundo hace sus desarrollos usando esa carpeta como la ubicación de sus programas el usuario tendrá problemas de organización con sus archivos.

Algo muy importante al empezar un desarrollo es preestablecer el posible tamaño del mismo; si se prevé un desarrollo de múltiples archivos es recomendable usar una carpeta única para los programas.

Si finalmente decide usar una carpeta distinta a **main** por favor tenga en cuenta las siguientes consideraciones:

- A la hora de llamar las subrutinas globales debe incluir tanto la carpeta como el nombre de la subrutina: `carpeta/subrutin()`.
- Si está usando subrutinas locales sólo necesita el nombre sin carpeta para llamar la subrutina: `subrutin()`
- Es posible definir en que carpeta trabajar al comienzo del programa principal usando el comando `setFold(carpeta)`. Tenga en cuenta que es posible reestablecer la carpeta inicial usando una variable temporal de la siguiente forma: `setFold(stats)→antifold` y reestableciendo “antifold” antes de finalizar el programa principal de esta forma: `setFold(#antifold)`.
- Revise cuidadosamente que las variables quedan definidas para su carpeta antes de distribuir el programa, es posible un fallo Terminal si algún archivo queda en una carpeta equivocada.

## 10. USO DE LIBRERÍAS

Las librerías son programas y archivos adicionales que permiten sumar funciones y funcionalidades extras a los programas y funciones.

El uso de las librerías además de ampliar el horizonte de ti-Basic ayuda a evitar repeticiones de código, optimizando el uso de la memoria de la calculadora y facilitando el desarrollo de programas.

Si decide usar alguna de ellas por favor úsela en la carpeta original que su desarrollador le haya designado, evite duplicar archivos.

### 10.1 ARCHING

Archiving es un compendio de rutinas que pueden ser usados en programas nuevos con el fin de sumarles funcionalidad. Se desarrollaron herramientas con cinco enfoques diferentes: Ecuaciones, Programación Multiplataforma, Rutinas Operativas, validación de datos e imágenes.

Archiving traduce: Archivos de Ingeniería, la aplicación de sus funciones tiene un enfoque ingenieril. Fue desarrollada por David Suescun con la finalidad de crear un “framework” para el proyecto Daisuke-Ing.

La librería con la documentación completa está disponible en:

<http://www.subterranos.net/rapco/daisuke/modules/wfdownloads/singlefile.php?cid=10&lid=6>

### 10.2 FLIB

Esto es una librería avanzada para los programadores de TI-BASIC, incluyendo muchas funciones tanto básicas como sofisticadas, compatibilidad con todas las versiones de ROM y de hardware así como ningún requisito de kernel.

Version 3.2, 04 - 02 - 2004

Copyright 2000-2004 by François LEIBER

Este programa se distribuye bajo la licencia GLP. Está disponible en:

<http://www.ticalc.org/archives/files/fileinfo/117/11770.html>

La Documentación de FLIB 3.2, por François LEIBER fue traducida a español está disponible en: <http://www.ticalc.org/archives/files/fileinfo/394/39440.html>

## 10.3 HAIL

Esta útil librería permite a usuarios de la calculadora trabajar expresiones matemáticas, simbólicas, listas y matrices en una interfaz Pretty Print.

A los programadores les ofrece una alternativa de ingreso de datos, más amigable e intuitiva, así como una alternativa para mostrar resultados que incluyan expresiones grande, listas, matrices, etc..

La librería fue desarrollada por Samuel Stearly y está disponible en:

<http://www.geocities.com/sstear70/exw.html>

Parte de su manual fue traducido a español por Raúl Bores Monjote (lzerw), incluye la información que un programador necesita, más no información para un usuario normal. Está disponible en:

<http://www.subterranos.net/rapco/daisuke/modules/wfdwnloads/singlefile.php?cid=10&lid=28>

## 10.4 IMPRIMIR

Imprimir es una librería desarrollada por Fabio Silber (cosmefulanito04) que sirve para mostrar cadenas en Pretty Print (la característica de la calculadora para mostrar expresiones tal cual las anotamos en papel) sin simplificar la expresión. Esto es muy práctico en programas paso a paso en donde se necesitan expresiones no simplificadas.

### Estructura simple de uso:

```
:main\imprimir(string)
```

Al ingresar una cadena string, el programa lo convierte automáticamente en una expresión matemática y si esta supera el largo de la pantalla se habilita la posibilidad de recorrerla de izquierda a derecha.

### Estructuras alternativas:

```
imprimir(string,num)
```

Con num igual a 0: Funciona igual que con el prototipo simple.

Con num igual a 1 o mayor a 2: Funcionara igual que con el prototipo simple pero además muestra el resultado en HOME. (Similar a Copyto\_h por Samuel Stearly)

Con num igual a 2: Envía un texto al HOME.

Esta útil y pequeña librería está disponible en:

<http://www.ticalc.org/archives/files/fileinfo/398/39896.html>

## **10.5 VERTEL**

Esta librería es muy similar a Flib, su manual también fue traducido al español por Raúl Bores Monjote (Izerw), puede conseguirlo en:

<http://www.subterranos.net/rapco/daisuke/modules/wfdwnloads/singlefile.php?cid=10&lid=27>

## 11. AYUDA

Esta guía fue escrito con la colaboración de programadores y usuarios de la comunidad de Calculadoras.cl, por tanto, en el foro de esta pagina: url <http://www.calculadoras.cl/foro/index.php> puede hacer preguntas para resolver sus dudas.

También pueden usar la página oficial del proyecto Daisuke-Ing para comunicar sus dudas e inquietudes: <http://www.daisuke.tk>

En caso de quererse comunicar con uno de los autores en particular, por favor remítase al capítulo de Créditos.



## 12. HISTORIAL DE VERSIONES

### 12.1 GUÍA DE PROGRAMACIÓN EN TI-BASIC

A continuación las fechas y características de las versiones existentes de esta guía de programación en el lenguaje Ti-Basic.

#### **Versión 0.2**

Desarrollo realizado en Julio de 2007

##### **General**

- Se terminaron las referencias de las librerías. Gracias a Izerw por su aporte!

#### **Versión 0.1**

Desarrollo realizado en Junio de 2007

##### **General**

- Se tomó el modelo de documentación de Daisuke-Ing para hacer la estructura inicial del documento.
- Se estableció una primera estructura de contenidos a ser abordados en la guía.
- Se tomaron fragmentos del documento de “ESTÁNDARES DE PROGRAMACIÓN DE DAISUKE-ING<sup>©</sup>” escrita por David Suescun para la sección de fundamentos de programación y otros capítulos.
- 

##### **Cosas por hacer**

- Redactar una guía especializada de uso de cadenas, listas, matrices, tablas y gráficas.

## 13. CRÉDITOS

### Autor Principal:

David F Suescun Ramirez (uniquekaiser)

[DaisukeIng@gmail.com](mailto:DaisukeIng@gmail.com)

[desarrollo@daisuke.tk](mailto:desarrollo@daisuke.tk)

<http://www.daisuke.tk>

### Colaboradores:

Fabio Silber (Cosmefulanito)

[cosmefulanito04@yahoo.com.ar](mailto:cosmefulanito04@yahoo.com.ar)

Gran contribución en el capítulo 8 redactando la primera versión del mismo.

Raúl Bores Monjiote (Izerw)

[izerw1@gmail.com](mailto:izerw1@gmail.com)

Traductor de los manuales de Hail y Vertel.

Don Diego Gómez de la Vega (DonDiegogv)

[dondiegogv@hotmail.com](mailto:dondiegogv@hotmail.com)

Colaboración en el capítulo 9 redactando la primera versión del mismo.

### Programas Usados:

Daisuke-Edit

<http://www.daisuke.tk>

devFlowcharter 0.9.9

<http://sourceforge.net/projects/devflowcharter/>

TiEmu Versión 2.81b

[http://lpg.ticalc.org/prj\\_tiemu/index.html](http://lpg.ticalc.org/prj_tiemu/index.html)

Ti-Connect 1.6

<http://education.ti.com>

### Agradecimientos Especiales:

- La comunidad de <http://calculadoras.cl>

### Licencia del Documento

GNU GPL

<http://www.gnu.org/licenses/gpl.html>

## 14. LICENCIA

Todos los programas y funciones que constituyen Daisuke-Ing<sup>©</sup> se publican bajo la licencia GNU GPL. Se incluye como anexa en este documento. También disponible en:

<http://www.gnu.org/licenses/gpl.html>

## 15. AVISO LEGAL

Debido a que el programa se licencia libre de costo, no existe garantía para el programa, hasta lo permitido por las leyes aplicables. Excepto cuando se establezca de otra forma por escrito los poseedores del copyright y/u otras partes proveen el programa "como está" sin garantía de ninguna clases, ya sea expresa o implícita, incluyendo, pero no limitándose a, la garantía implícita de uso y utilidad para un propósito particular. El riesgo completo acerca de la calidad y eficiencia del programa es suyo. Si el programa se mostrara defectuoso, usted asumirá todo el coste del servicio necesario y de la reparación o corrección.

En ningún caso, a no ser que se requiera por las leyes aplicables o se acuerde por escrito, podrá ningún poseedor de *copyright* o cualquier otra parte que haya modificado y/o redistribuido el programa ser responsable ante usted por daños o perjuicios, incluyendo cualquier daño general, especial, incidental, o consecuente que se derive del uso o incapacidad de uso de este programa (incluyendo, pero no limitándose a la pérdida de datos o producción de datos incorrectos o pérdidas sufridas por usted o una tercera parte, o una incapacidad del programa para operar junto a otros programas), incluso si el poseedor del *copyright* u otra parte había sido avisado de la posibilidad de tales daños.

Este programa se puede redistribuir y modificar de forma libre. No está permitido modificarlo y publicarlo como suyo.

Si encuentra alguna forma de mejorar el programa, comuníquela para que la mejora sea incluida dentro de la versión oficial y más usuarios se puedan beneficiar de esta.

Por favor remítase a la licencia para más información.