Author........Zeda Elnara
e-mail........xedaelnara@gmail.com
Project.......Grammer
Version.......2.29.04.12 (major version followed by the date)
Last Update...29 April 2012
Language......English
Programming...Assembly
Size..........1-Page app

To follow the progress of Grammer, check out these boards where Grammer is actively updated:
http://www.omnimaga.org/index.php?board=199.0
http://www.cemetech.net/forum/viewforum.php?f=71

Si vous parlez en français, persalteas (de Tout-82 et Éspace-TI) faisait un tuto pour le Grammer. A ce moment vous pouvez le trouver ici:
http://tiemulation.kegtux.org/Grammertutorial.htm

(Je suis désolé pour ma grammaire, je ne parle pas français)

Updates will probably not occur frequently for the summer of 2012 because I won't have internet.

If you have questions or suggestions, feel free to email me or post in the forums!

Intro

Grammer is a powerful programming language for the TI-83+/84+/SE calculators. Unlike TI-BASIC, it is not designed to do math and as such, Grammer math is fairly limited in some respects. This also means, however, that it uses a math system with its own tricks and optimisations. If you are going to learn how to effectively use Grammer, you will first need the Grammer interprter on your calculator (this document assumes you have the latest version of the Grammer App). After that, you should become familiar with Grammer's:

- Number system
- Math
- Pointers
- Drawing
- Data structures (sprites, arrays)

Getting Started

First, send Grammer 2 to your calculator. If you have this document, I assume you have the App.
Next, run the app on your calc. Grammer is now installed. If you want to make a program, you have to remember two very important things:

Start the program with **.0:**. This lets Grammer know it is a Grammer program
End the program with **Stop**. This lets Grammer know to stop executing code.
Now, before I explain all the technical stuff, could you to create this program and run it?

```
:.0:Return
:ClrDraw
:Text(0,0,"YUM
:DispGraph
:Stop
```

It should look like this:

Number System

The Grammer number system works like this:

- **Numbers are integers 0 to 65535**. This means no fractions, or decimals. There are a few commands that handle higher numbers.

Let's look at this closer. First, why 65535? That is easy to answer. Grammer uses 16-bit math. In binary, the numbers are 0000000000000000b to 1111111111111111b. Convert that to decimal (our number system) and you get 0 to 65535. *If you don't understand binary or hexadecimal, see the* ***Binary Lesson!*** *section. Understanding binary and hex is not necessary, but it will help you understand why certain commands act the way they do. Plus, it can help you figure out some advanced tricks.*

Now, let's look at some scenarios. If you overflow and add 1 to 65535, it loops back to zero. Similarly, if you subtract 1 from 0, you loop back to 65535. Then you can see that 65530+11=5. You also now know that -1=65535. So what happens if you multiply 11*65535? Believe it or not, you will get -11 which is 65536-11=65525.*(If you ever want to go into more advanced math in college, hold on to this info for when you get into Abstract Algebra. You are working with the ring* $Z_{2^{16}}$*).*

Division, unfortunately is not as nice as multiplication, addition, or subtraction. 3/-1 will give you 0 because it is 3/65535. Don't worry, though, there are ways to get around this, just read the next section.

Math

Grammer math is very different from the math that you are used to. The main points are:

- Math is done right to left
- All functions in Grammer are math.

For the first point, let's look at an example of the math string 3+9/58-3*14+2^2:

3+9/58-3*14+**2^2**

3+9/58-3***14+4**

3+9/58-**3*18**

3+9/**58-54**

3+**9/4**

3+2

5

Parentheses are not part of Grammer math, but to give you better control, you have two main options: Use a space or a colon between chunks of math to compute each chunk in order. In most situations, you should use a colon. So say you want to do ((3+4)*(6+3))/6. You have 3 chunks that you can compute in order:

3+4:*6+3:/6

7:*6+3:/6

7*6+3:/6

7*9:/6

63:/6

63/6

10

Pretty cool, right? That is actually even more optimised than using parentheses! Now, if you are like me, you might not like the look of that missing remainder of 3 in that division. 63/6 is 10.5, not 10, but you know Grammer rounds down. But guess what? There is a system variable called θ' that will contain the remainder after division and other such extra information after math.

You probably want to know what math operations you have access to and what they do, so here you go:

*a***+***b*	add	This adds two numbers together. If there is overflow, θ'=1, else it is 0.
*a***-***b*	subtract	This subtracts two numbers. If there is overflow, θ'=65535, else it is zero
*a*******b*	multiply	This multiplies two numbers. Overflow is stored in θ'.
*a***/***b*	divide	This divides two numbers. The remainder is stored in θ'.
*a***/** *b*	signed divide	This divides two signed numbers. There is a space after the /. Example, 65533/65535=3.
a^2	squared	This squares a number. Overflow is stored in θ'.
-*a*	negative	This returns the negative of whatever follows it. Essentially, this is 65536-n.
min(*a***,***b*	minimum	This returns the smaller of two values
max(*a***,***b*	maximum	This returns the larger of two values
sin(*a*	sine	This returns the sine of a number as a value between -128 and 127. The eriod is 256, not 360. For example, sin(32) is like sin(45) in normal math. If you need help, take the actual sin(45) and multiply by 128. Rounded, this is 91. So, sin(32)=91.

cos(*a*	cosine	This computes the cosine of a number. See the notes on sin(.
e^(*a*	2^	This returns 2 to the power of *a*. For example, e^(3 returns 8.
gcd(*a*,*b*	GCD	Returns the greatest common divisor of two numbers.
lcm(*a*,*b*	LCM	Returns the lowest common multiple of two numbers.
*a***>Frac**	Factor	θ' contains the smallest factor of the number. Output is *a* divided by that number. For example, 96>Frac will output 48 with θ'=2. You can use this to test primality.
√(*a*	square root	Returns the square root of the number, rounded down. θ' contains a remainder.
√('*a*	Rounded sqrt	Returns the square root rounded to the nearest integer
abs(*a*	absolute val	Returns the absolute value of a number. If *a*>32767, it is treated as negative.
rand	random	Returns a random integer between 0 and 65535
randInt(*a*,*b*	rand integer	Returns a random integer between *a* and *b-1*
a **nCr** *b*	n choose r	Returns *a* choose *b*. In mathematics, this is typically seen as n!/((n-r)!r!). I had to invent an algorithm for this to avoid factorials because otherwise, you could not do anything like 9 nCr 7 (9!>65535).
!*a*	Is 0?	If the following expression results in 0, this returns 1, else it returns 0.
a **and** *b*	bit AND	Computes bitwise AND of two values. Remember the *Binary Lesson!* ?
a **or** *b*	bit OR	Computes bitwise OR of two values.
a **xor** *b*	bit XOR	Computes bitwise XOR of two values.
not(*a*	bit invert	Inverts the bits in the value.
*a***=***b*	equal	If *a* and *b* are equal, this returns 1, else it returns 0.
*a***≠***b*	not equal	If *a* and *b* are not equal, this returns 1, else it returns 0.
*a***>***b*	greater	If *a* is greater than *b*, this returns 1, else it returns 0.
*a***≥***b*	greater or equal	If *a* is greater than or equal to *b*, this returns 1, else it returns 0.
*a***<***b*	less	If *a* is less than *b*, this returns 1, else it returns 0.
*a***≤***b*	less or equal	If *a* is less than or equal to *b*, this returns 1, else it returns 0.

Pointers and More

To understand pointers, you have to understand how memory works. First, every byte of memory has what is called an address. An address, is a pointer to that byte. For example, the first byte of memory is at address 0, the second byte is at address 1, et cetera. On the calcs, there are 65536 bytes of memory addressed at a time. The last 32768 bytes are RAM. This is where your program and everything in it is stored. This has some powerful implications. If you have a pointer to a section of code and you tell Grammer to start executing there, it can jump there immediately. If you have a pointer to a string, you can use that pointer to draw the string or use it. This means you don't have to create any external variables for your strings or sprites. If you want to create an appvar for save data, having the pointer to that lets you edit the data, save to it, or read from it. Pointers are powerful As such, you will probably be using a bunch of them, so you should use pointer vars:

Pointer Vars-(or just "vars") are two byte values. These can hold a 16-bit number, so these are well suited to holding pointers. These are all the letters A to Z and θ and A' to Z' and θ'. Also, for readability, you can use the lowercase letters instead of A', for example.
How do you store to these? Like all BASIC programmers know, use →. For example:

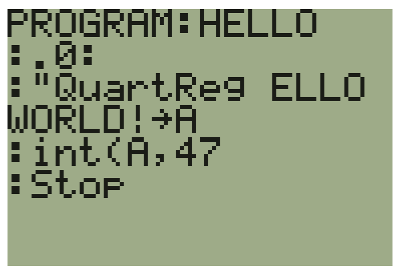
```
:3→A
```

Likewise, for a string:

```
:"Hello World→A
```

Don't be fooled, that does not store the string anywhere. It just stores where the string is located in A. So if you change the byte at A, you will change the "H" in your program. If you want to try it, run this program and then check the source again.

```
:.0:
:"HELLO WORLD→A
:int(A,47             ;This changes the byte to byte 47. Bytes are 0 to 255.
:Stop
```



Now you know that byte 47 corresponds to the token "QuartReg ". Be careful, though, not all tokens are one byte. Lowercase letters are a notorious example of two-byte tokens.

Where else are pointers useful? The best example is actually with labels or finding programs. Anything that requires searching, actually. For example, in Grammer, to goto a label, you would do **Goto Lbl "HI** and that would jump to the label named **.HI**. You can also get a pointer to this label. If you need to jump to that label often, this saves lots of time by not having to search for the label every time. Remember, everything is math in Grammer:

```
:.0:
:Lbl "HI→A
:Repeat getKey(15       ;getKey(15 returns 1 if clear is being pressed
:prgmA                  ;This calls the subroutine pointed to by A
:End
:Stop
:.HI
:B+1→B
:Text('0,0,B            ;Displays the number B at (0,0)
:End                    ;Always at the end of a subroutine
```

Now I want to take time to finally start explaining some code.

Labels can be a bit more descriptive than in BASIC. You can use up to 42 bytes for a name, technically, but try to maintain readability. You can also use pretty much whatever tokens you want in a label name. For example, I had a label named ICircle(that I had draw inverted filled circles. **Lbl** takes a string argument where the string is the name of a label. It outputs the pointer to the line after the label.

Goto and **prgm** let you redirect your program. **Goto** jumps to the code and **prgm** will execute the code and return once it gets to End. Both of these take a pointer to figure out where to go.

Repeat works like it does in TI-BASIC. It first executes the code between **Repeat** and **End** and then it tests the condition. If the result is 0, it repeats the procedure. If it is anything else, it stops looping and continues.

getKey gives you two ways to read keys. Key codes are different in Grammer from TI-BASIC. The first way is to use it like you would in TI-BASIC. For example, getKey→A stores the keypress in A. You can also use getKey(15 to quickly test a specific key. This is great if you want to use multiple key presses.

Command List

Here are a bunch of commands that do not fit in the drawing or math category. This is a whole lot of info (about 9 pages), but I made some examples later on so that you can figure out how to use these better :)

Operators

→ This stores the last computed value to a variable. For example:

```
:Return→A'
```

That will store the value output from Return to A'.
Likewise, you can store to some OS variables such as real vars (A through Theta) and Strings (Str1 to Str256). For real vars, use the prefix i (the imaginary number) and Strings just use their string number. For example, 3→iA will store to the OS realvar A. "Hello→Str2 will strore the string "Hello" to Str2. With Strings, if you use the Str1 token followed by a number like 55, you will be accessing the hacked string 155. Str0 is the same as string 10, remember, so Str00 is the same as Str100.

As another note, when storing to an OS string, if you want to use it as a Grammer string, you need to have ' follow the string name (it adds a newline token to the end of the string).

To store 32-bit results, you will need 2 vars. →AB will store θ' in A and Ans in B. So to store a 32-bit result from multiplication: A*D→AB.

// This is used to start a comment. The comment goes to the end of the line. A commented line is skipped. As a note, the user can include a comment after code.

" This starts a string. The output is a pointer to the string that can be used later to reference it.

π If you put a pi symbol before a number, the number is read as hexadecimal. For example, π3F would be read as 63.

! This has several uses. The first is to work like the not() token in TI-BASIC. So for example, 3=4 would return 0 because it is not true. However, !3=4 would return 1. Likewise, !3=3 would return 0. The other use is with loops. For example, **If A=3** will test if A is 3 and if it is, it executes the code. However, **!If A=3** will execute the code if A is **not** 3. See If , If... Then... End, While, Repeat, and Pause If.

i This is the imaginary i. Use this to access OS real vars. For example, to read OS var A and store it to Grammer var A:

```
:iA→A
```

And to store a Grammer var to an OS var:

```
:B'→iA
```

This does not read decimals or imaginary pasrts of a number.

E This indicates the start of a binary string. This is the exponential E.

Loops/Conditionals

If If the expression following is not 0, the line following it will be executed. The line is skipped if it is 0. Conditions are computed to the next newline. For example, you can have → and : in an If statement in Grammer.

```
:If A=B         ;Since A=B is false, the following line is skipped
:9→A
```

Or also:

```
:If 3=B→A:*14:-14   ;This is the full statement
:Text(0,0,"Yay!
```

If... Then... End This is similar to If except if the statement results in 0, any code between amd including Then and End will be skipped. This works like the TI-BASIC command. For example:

```
:If 3=4               ;3=4 returns 0
:Then
:3→A
:9→B
:16→C
:End
```

For(The arguments for this are:

For(Var,Start,End

Var is the name of a var

Start is the starting value to load to the var

End is the max value to load to the var
Alternatively, you can use:

For(Val

Val is the number of times to execute the loop. 0=65536

What this does is it loads the initial **Start** value into **Var**. It executes code until it reaches an End statement, then it incrememnts the var. If incrementing goes higher than **End**, the loop finishes and code continues, otherwise it executes the loop again. So for an example:

```
:For(R,0,48
:Circle(32,48,R,1
:DispGraph
:End
:Stop
```

Pause If This will pause so long as the condition is true for example, to pause until a key is pressed, **Pause If !getKey**

Alternatively, using !Pause If will pause while the condition is false. So to pause until enter is pressed, do **!Pause If 9=getKey**

While While loops are like If statements addicted to crack-- they just keep coming back. An If statement is content with just checking if the result is true (true=1), but a while loop will not only execute the code up to End if it is true, but it will loop back to try it again! To give you an idea, this will keep looping until Clear is pressed, and while it is at it, it will increment A and decrement B:

```
:0→A→B
:While getKey≠15
:A+1→A
:B-1→B
:End              ;This tells the While loop to End and restart!
```

Alternatively, **!While** will only execute the code if the statement is **not** true.

Repeat This is a loop that is kind of the opposite of a While loop. This will repeat the code up to an End until the statement is true. So for example, to wait until clear is presed:

```
:Repeat getKey=15
:End
```

!Repeat checks if the statement is false in order to end. For example, to remain in the loop while Enter is being pressed:

```
:!Repeat getkey=9
:End
```

Control

Return This returns a pointer to the next line of code.

Goto This is unlike the BASIC Goto command. This jumps to a pointer as opposed to a label. For example:

```
:Return→L
:<<code>>
:Goto L        ;This jumps to the line after "Return→L"
```

Lbl This returns the pointer of a label. The argument is a pointer to the label name. For exaple, **Lbl "HI** will search for .HI in the program code. Also, you can specify which variable the label is in. For example, if you wanted to jump to a laqbel in another program, you can add a second argument as the name of the var. For example, to find the label HI in prgmBYE:

```
 Lbl "HI","BYE
```

Pause This will pause for approximately x/100 seconds. So **Pause 66** will pause for about .66 seconds.

prgm This is used to execute a sub routine.

Func The arguments are:

Func*Pointer[,Counter*

This will automatically execute the subroutine pointed to by <<pointer>> based on *Counter*. *Counter* is based on an internal counter, not based on actual timings like seconds or milliseconds. The default is 128. So for example:

```
:FuncLbl "DISP
:Repeat getKey(15
:<<do stuff>>
:End
:Stop
:.DISP
:DispGraph
:End
```

That will do DispGraph several times per second automatically.

Asm(This can be used to run an assmebly program. For technical info, see this section. Unsquished ASM programs are not yet supported.

AsmPrgm This allows you to input asm code in hex. (C9 is needed)

ln(This will let you jump forwards or backwards a given number of lines. For example:

```
:ln(3
:"NOT
:"Executed
:"YAY :D
```

Or to jump backwards:

```
:"YAY :D
:"Erm...
:"Yeah...
```

```
:ln(-3
```

Input/Computing

getKey This returns a value from 0 to 56 that is the current key press. You can use this chart for values.
Also, **getKey(** will allow you to see if a key is being pressed. For example, getKey(9 will return 1 if enter is pressed

Input This allows you to input a string. The pointer to the string is returned. (this is not a permanent location, the data will be overwritten the next time Input is used) To get a value input from the user, you can use expr(:

```
:expr(Input →A
```

That will store the result to A

Ans This will return the value of the previous line

expr(This will compute a string as a line of code (useful with Input)

inString(This is similar to the TI-BASIC command. This will return the location of a sub-string. The inputs are where to start searching and the string to search for:

inString(SearchStart,SearchString

So an example would be:

```
:Lbl "DATA→A
:inString(A,"How→B
:.DATA
:HELLOHowdyWoRlD!
```

The size of the input string is returned in θ' and if there was no match found, 0 is returned.

length(This will return the size of a variable (in RAM or Archive) as well as the pointer to the data in θ'. For example, to get the size of the appvar Data:

```
:length("UData→A
```

If the var is not found, -1 is returned.

length(' This is used to search for a line. For example, if you want to find a specific line number in a program, this is what you would use. The syntax:

length('*StartSearch*,*Size*,*LineNumber*,[*LineByte*

StartSearch is where to begin the search
Size is how many bytes to search in. 0 will search all RAM.
LineNumber is the line number you are looking for
LineByte is an optional argument for what byte is considered a new line.

The output is the location of the string and θ' has the size of the string. If the line is not found, the last line is returned instead.

solve(*command subset*

CopyVar solve(0,"VarName1","VarName2"[,size[,offset

This will copy the program named by VarName1 from RAM or archive to a new program named by VarName2. If Varname2 already exists, it will be overwritten. So for example, to copy Str6 to Str7:

```
:solve(0,"DStr6","DStr7
```

This returns the pointer to the new var and the size of the var is in θ'

The last arguments are optional. Size lets you choose how many bytes are copied (instead of just copying the whole var). You can also add an offset argument to tell where to start reading from.

CopyDataI solve(1,loc_i,loc_f,size

This copies data from loc_i to loc_f. (Forward direction)

CopyDataD solve(2,loc_i,loc_f,size

This copies data from loc_i to loc_f. (Backward direction)

ErrorHandle solve(3,Pointer

This will allow your program to have a custom error handler. **Pointer** is 0 by default (meaning Grammer will handle it). Otherwise, set it to another value and grammer will redirect the program to that location. The error code is returned in Ans. For Example:

```
:solve(3,Lbl "ERR
:<<code>>
:.ERR
:If =1           ;Means there was a memory error
:Stop
:End
```

Ans and θ' are put back to normal when the error handler completes.

Errors:

0=ON

1=Memory

CallError solve(4,Error#

This will execute the error code of a Grammer error. For example, to make a Memory error:

```
:solve(4,1
```

Using Error 2, you can input a string for a custom error:

```
:solve(4,2,"Uh-Oh!
```

Physics

R▶Pr(This will clear the particle buffer.

R▶Pθ(This will recalculate the particle positions and draw them. If you want to change the particle buffer, just add a pointer argument. If you want to use a program, for example, as a buffer:

```
:Get("EBUF→A
:R▶Pθ(A-2
```

P▶Rx(This will add a particle to the buffer. Just use the pixel coordinate position. For example:

```
:P▶Rx(2,2
```

P▶Ry(This will cahnge the particle effect. 0 is normal sand, 1 is boiling.

P▶Rx(' This will convert a rectangular region of the screen to particles. The inputs are:

P▶Rx(' *Y,X,Height,Width*

This scans the area for pixels that are turned on and adds them to the current particle buffer.

Miscellaneous

conj(**Warning: I have no knowledge of musical lingo, so excuse my mistakes**

This is a sound command with three inputs. The syntax is:

conj(Note,Octave,Duration

Notes are:

0 =C	1 =C#	2 =D	3 =D#
4 =E	5 =F	6 =F#	7 =G
8 =G#	9 =A	10=A#	11=B

Octave is 0 to 6

Duration is in 64th notes. So for example, a 32nd dot note use 3/64th time. Duration is thus 3.

conj(' This sound routine has several inputs:

conj('*Duration,'Period*

conj('*Duration,DataLoc,Size*

This reads data for the period directly to save time (intead of converting numbers on the fly). Size is the size of the data in words, not bytes.

Drawing

Drawing in Grammer has a few similarities to TI-BASIC, but not many. The first concept I want to tell you about is that of drawing buffers. These include the graph screen and other chunkc of memory that you use like the graph screen. A drawing buffer is 768 bytes and there are two that do not use user RAM that you can use in Grammer. Their pointers are π9872 and π9340. The first is called AppBackUpScreen by assembly programmers and the second is what we know as the graph screen. The graph screen is default. You can also use π86EC, but Grammer uses that for scratch work for a few commands as well. Now, to the actual drawing!

In all cases, (0,0) is the upper left corner of the screen. If you want to try to draw a circle, be prepared for how awesome this will be to if you were a BASIC coder:

```
:.0:
:ClrDraw
:Repeat getKey(15
:randInt(0,96→X
:randInt(0,64→Y
:randInt(0,64→R
:Circle(Y,X,R,3     ;Draws a circle with an inverted outline at (Y,X)
:DispGraph
:End
:Stop
```

If you tried it, you will see that drawing in Grammer is very fast compared to BASIC. One of the reasons is that Grammer does not update the LCD witht he contents of the graph screen automatically. This means you have to do it yourself with **DispGraph** and this means that you can do a lot of drawing without showing your users the behind the scenes stuff.

Now let's say you want to draw to set another buffer as default. This is where you use Disp:

```
:Disp π9872
```

Now, whenever you draw or update the LCD, that is the buffer that will be used. This means you can preserve the graph screen. Alternatively, most drawign commands have an optional argument to draw toa specific buffer.

Grayscale

Grayscale is a nice little feature in Grammer that can be used if you do it correctly. Now that you know how buffers work, you can give this a try. First, you need two buffers-- the gray buffer and the black buffer. Define these using **Disp °** and **Disp '**, respectively (both found in the Angle menu). For example, to add the gray buffer, use **Disp °π9872** and now whenever you use DispGraph, it will update one cycle of gray. Because the LCD does not support grayscale naturally, you will need to update the LCD often and regularly. This is what I managed to do (and yes, I drew this myself :3)

A pixel is seen as gray if it is ON in the gray buffer, but OFF in the black buffer. If a pixel is on in the black buffer, it will just appear black. Because grayscale is so important, I will give an example program that draws in grayscale:

```
:.0:
:π9872→Z
:Disp °Z
:ClrDraw
:ClrDrawZ
:0→X→Y
:Repeat getKey(15)
:Pxl-Change(Y,X         ;Just drawing the cursor
:Pxl-Change(Y,X,Z       ;Pixel changing it on both buffers
:DispGraph
:Pxl-Change(Y,X
:Pxl-Change(Y,X,Z
;If getKey(54
:Pxl-On(Y,X,Z
:If getKey(9
:Pxl-On(Y,X
:If getKey(54:+getKey(56    ;If [2nd] or [Del]
:Pxl-Off(Y,X
:If getKey(9:+getkey(56     ;If [Enter] or [Del]
:Pxl-Off(Y,X,Z
:X+getKey(3
:-getKey(2
:If <96
:→X
:Y+getKey(1
:-getKey(4
:If <64
:→Y
:End
:Stop
```

Sprites

Using sprites is a pretty advanced technique, so don't expect to understand everything here.

Sprites are pretty much mini pictures. They are a quick way to get detailed objects that move around making them a powerful graphics tool. In Grammer, the main sprite commands are **Pt-On(** and **Pt-Off(** and both have differences and advantages over the other.

Sprite Data

Sprite data is in the form of bytes or hexadecimal and you will want to understand binary to hex conversions for this. For example, to draw an 8x8 circle, all the pixels on should be a 1 in binary and each row needs to be converted to hex:

```
0 0 1 1 1 1 0 0  =3C
0 1 0 0 0 0 1 0  =42
1 0 0 0 0 0 0 1  =81
1 0 0 0 0 0 0 1  =81
1 0 0 0 0 0 0 1  =81
1 0 0 0 0 0 0 1  =81
0 1 0 0 0 0 1 0  =42
0 0 1 1 1 1 0 0  =3C
```

So the data would be 3C4281818181423C in hexadecimal.

Sprite Logic

There are 5 forms of sprite logic offered by Grammer, currrently. These tell how the sprite should be drawn and can all be useful in different situations.

Overwrite:

For an 8x8 sprite, this will erase the 8x8 area on the screen and draw the sprite.

AND:

This leaves the pixel on the screen on if and only if the sprites pixel is on and the pixel on the screen is on.

OR:

This will turn a pixel on on the screen if it is already on or the sprite has the pixel on. This never erases pixels.

XOR:

If the sprites pixel is the same state as the one on the screen, the pixel is turned off, otherwise, it is turned on. For example, if both pixels are on, the result is off.

Erase:

Any pixels that are on in the sprite are erased on screen. The pixels that are off in the sprite do not affect the pixels on the graph buffer.

Pt-On(

This is used to display sprites as tiles. This means it displays the sprite very quickly, but you can only draw to every 8 pixels.

Pt-Off(

This is a slightly slower sprite routine, but it allows you to draw the sprite to pixel coordinates.

Drawing Commands

These are the drawing commands. Some of these have alternate syntaxes that do very different things. This section alone is six pages.

Graphics

DispGraph

Displays the graph screen. You can display another buffer by using a pointer. For example, DispGraphπ9872

Circle(

The syntax is:

Circle(**Y**,**X**,**R[**,Method**[**,pattern**[**,buffer

This draws a circle using **Y** and **X** as pixel coordinates and **R** as the radius of the circle in pixels. **Method** is how to draw the circle:

1-Black border (Default)
2-White border
3-Inverted border

Pattern is a number from 0 to 255 that will be used as a drawing pattern. For example, 85 is 01010101 in binary, so every other pixel will not be drawn. Use 0 for no pattern. If the bit is 0, the pixel will be drawn, if it is 1, it won't be drawn. **Buffer** is the buffer to draw to (useful with grayscale).

Pt-Off(

This is used to draw sprites to pixel coordinates. It is limited in some ways, compared to the Pt-On(command, but more flexible in others. The syntax is:

Pt-Off(Method,DataPointer,Y,X,**[**~~Width~~,**[**Height**[**,Buffer

Method is how the sprite is drawn:

0-Overwrite
  This overwrites the graph screen data this is drawn to.
1-AND
  This draws the sprite with AND logic
2-XOR
  This draws the sprite with XOR logic
3-OR
  This draws the sprite with OR logic
5-Erase
  Where there are normally pixels on for the sprite, this draws them as pixels off.

DataPointer is a pointer to the sprite data
Y is the pixel Y-coordinate
X is the pixel X-coordinate
~~*Width*~~ is 1. More options may be due in the future, but for now, just put 1 :) The default is 1.
Height is the number of pixels tall the sprite is. 8 is default
***By adding 8 to the Method, the data will be read as hexadecimal**

Pt-On(

This also draws sprites, but only to 12 columns (every 8 pixels). This is slightly faster than Pt-Off(and has the advantage of variable width. It also has the DataSwap option that isn't present with the Pt-Off(command. Here is the syntax of the command:

Pt-On(Method,DataPointer,Y,X,**[**Width,**[**Height[,Buffer

Method-This is how the sprite is drawn:

- 0-Overwrite
- 1-AND
- 2-XOR
- 3-OR
- 4-DataSwap
  - This swaps the data on the graph screen with the sprite data. Doing this twice results in no change.
- 5-Erase
- 6-Mask
  - This will display a masked sprite.
- 7-Gray
  - This draws a frame of a 3 level gray sprite

DataPointer is a pointer to the sprite data

Y is the pixel Y-coordinate

X is a value from 0 to 11.

Width is how wide the sprite is. 1=8 pixels, 2=16 pixels,.... Default is 1.

Height is the number of pixels tall the sprite is. Default is 8.

***By adding 8 to the Method, the data will be read as hexadecimal**

Line('

This is used to draw lines. The syntax for this command is:

Line('x1,y1,x2,y2[,Method[,Buffer

So it is two sets of pixel coordinates and then the **Method**:

- 0=White
- 1=Black
- 2=Invert

If **Method** is ommitted, it uses 1 as the default.

Buffer is the buffer to draw to.

Text(

Text(has a lot of neat features in Grammer. First, Grammer uses its own font default. It works like the homescreen in that it draws in 24 columns (the homescreen draws in 16 columns). The font is 4x6 and is fixed width. However, here are the neat aspects and how to use the command. Using the Output(command will let you choose other font styles such as plotting to pixel coordinates.

-To draw text, simply do this:

```
:Text(Y,X,"Text      ;"Text" can be a pointer to a string
```

-To draw a number, use the ' symbol:

```
:Text('Y,X,99
```

-To draw a number in a specific base (use 2 to 32), add another argument:

```
:Text('Y,X,99,16     ;drawn in hexadecimal (so it shows 63)
```

-To draw at the end of the last text drawn, use a degree symbol to replace coordinates:

```
:Text(°"Text
```

-Likewise, you can do this with numbers:

```
:Text('°99,2         ;draws 99 in binary
```

-*Another* feature is using /Text(or Text(ʳ for typewriter text mode (that is the superscript r found at [2nd][APPS]). This will display characters with a delay. The delay is chosen with [Fix Text(](). This will even display the individual letters in a token as if it is being typed. Here is an example:

```
:/Text(Y,X,"HELLO
```

-And you can use numbers and other operators, too!
-Another thing that is nice is that text wraps to the next line and if it goes off the bottom, it wraps to the top.
-To display a char by number, the arguments are:

```
:Text(Y,X,'#
```

The ' operator tells it to draw char(#).
-To draw text as an ASCII string, use ° before the string. For example:

```
:Text(Y,X,°"HIrandM(WORLD
```

That will display the text "HI WORLD" because randM(corresponds to the space char in the ASCII set.

To display 32-bit number display. The upper and lower 16-bits must be in a pVar. An example where B is the upper 16-bits and C' is the lower 16-bits:

```
:Text('0,0,BC'
```

Using the Text(command with no arguments returns the X position in Ans and the Y position in θ'.

If you want to draw to coordinates based on the last drawn coordinates, you can do something like this:

```
:Text(+3,+0,"Hello
```

But instead of +0, just leave it empty like this:

```
:Text(+3,,"Hello
```

Line(

This is used to draw rectangles. The syntax for this command is:

Line(x,y,Height,Width,Method

x is a value from 0 to 95 and is the x pixel coordinate to begin drawing at
y is a value from 0 to 63 and is the y pixel coordinate to begin drawing at

Height is a value from 1 to 64 is the number of pixels tall the box will be
Width is a value from 1 to 96 is the number of pixels tall the box will be
Method is what kind of fill you want:

0-White. This turns off all of the pixels of the rectangle
1-Black. This turns on all of the pixels of the rectangle
2-Invert. This inverts all of the pixels of the rectangle
3-Black border. Draws a black perimeter not changing the inside
4-White border. Draws a white perimeter not changing the inside
5-Inverted border. Draws an inverted perimeter not changing the inside
6-Black border, White inside.
7-Black border, Inverted inside.
8-White border, Black inside.
9-White border, Inverted inside.
10-Shifts the contents in that rectangle up
11-Shifts the contents in that rectangle down

Pxl-On(

This turns a pixel on using coordinates (y,x). To draw to a specific buffer, add its pointer as a last argument.

Pxl-Off(

This turns a pixel off using coordinates (y,x). To draw to a specific buffer, add its pointer as a last argument.

Pxl-Change(

This inverts a pixel using coordinates (y,x). To draw to a specific buffer, add its pointer as a last argument.

ClrDraw

This clears the graph screen buffer and resets the text coordinates. Optionally, you can clear a specific buffer by putting its pointer directly after. For example, ClrDrawπ9872

ClrHome

This clears the home screen buffer and resets the cursor coordinates

Shade(

This sets the contrast to a value from 0 to 39. 24 is normal and this is not permanent. An example is:

```
:Shade(30
```

Horizontal

This draws a horizontal line on the graph. The syntax is
Horizontal y[,method,[,Buffer
y is a value from 0 to 63
method is how to draw the line:

0=draws a white line
1=draws a black line
2=draws an inverted line

Buffer is the buffer to draw to.

Vertical

This draws a vertical line on the graph. The syntax is:

Vertical x[,method[,Buffer

x is a value from 0 to 95

method is how to draw the line:
0=draws a white line
1=draws a black line
2=draws an inverted line
Buffer is the buffer to draw to.

Tangent(

This is used to shift the screen a number of pixels. The syntax is:

Tangent(#ofShifts,Direction[,Buffer

# of shifts is the number of pixels to shift the graph screen
Direction is represented as a number:
1 = Down
2 = Right
4 = Left
8 = Up
You can combine directions by adding the values. For example, Right and Up would be 10 because 2+8=10

Disp

This will let you change the default graph buffer. For example, if you don't want to use the graph screen, you can put this at the start of the program:

:Disp π9872

Also, if you are using grayscale, you can use the following:
Disp ' will set the black buffer.
Disp ° will set the gray buffer.

Pt-Change(

This command is used to draw tilemaps. There is currently one method, but more should be added in the future. Here is the syntax:

Pt-Change(0,MapData,TileData,MapWidth,MapXOffset,MapYOffset,TileMethod

-**MapData** is a pointer to the map data
-**TileData** is a pointer to the tile set
-**MapWidth** is the width of the map (at least 12)
-**MapXOffset** is the X offset into the map data
-**MapYOffset** is the Y offset into the map data
-**TileMethod** is how the sprite will be drawn (see Pt-On()

Fill(

0-Black
This fills the screen buffer with black pixels
1-Invert
This inverts the screen buffer
2-Checker1
This fills the screen buffer with a checkered pattern
3-Checker2
This fills the screen buffer with another checkered pattern
4,x-LoadBytePatternOR
copies a byte to every byte of the buffer data with OR logic

5,x-LoadBytePatternXOR
copies a byte to every byte of the buffer data with XOR logic
6,x-LoadBytePatternAND
copies a byte to every byte of the buffer data with AND logic
7,x-LoadBytePatternErase
copies a byte to every byte of the buffer data with Erase logic
8,x-BufCopy
x points to another buffer. The current buffer gets copied there
9,x-BufOR
x points to another buffer. This gets copied to the current buffer with OR logic.
10,x-BufAND
x points to another buffer. This gets copied to the current buffer with AND logic.
11,x-BufXOR
x points to another buffer. This gets copied to the current buffer with XOR logic.
12,x-BufErase
x points to another buffer. This gets copied to the current buffer by erasing.
13,x-BufSwap
x points to a buffer. This swaps the current buffer with the other.
14,x-CopyDownOR
The current buffer is copied x pixels down to itself with OR logic
15,x-CopyDownAND
The current buffer is copied x pixels down to itself with OR logic
16,x-CopyDownXOR
The current buffer is copied x pixels down to itself with OR logic
17,x-CopyDownErase
The current buffer is copied x pixels down to itself with OR logic
18,x-CopyUpOR
The current buffer is copied x pixels up to itself with OR logic
19,x-CopyUpAND
The current buffer is copied x pixels up to itself with OR logic

20,x-CopyUpXOR

The current buffer is copied x pixels up to itself with OR logic

21,x-CopyUpErase

The current buffer is copied x pixels up to itself with OR logic

22,type-FireCycle

This burns the contents if the screen for one cycle. If **type** is 0, white fire is used, if it is 1, black fire is used.

23,Type,Y,X,Width,Height-Fire Cycle 2

Type is the same as FireCycle and the other inputs are the same as Pt-On(where X and Width go by every 8 pixels.

Data Structures

Grammer doesn't really have any data structures which is both good and bad. Bad because it makes you have to think a little more about how to approach a problem, but good in that it allows you to create precisely what you need. This is where you will need commands to create variables, insert or remove data, and edit the data. I will also try to explain how to create some basic data structures like arrays and matrices. First, here are the commands you have to work with:

Memory Access

Get(— This uses a string for the name of an OS var and returns a pointer to its data.
-If the variable does not exist, this returns 0
-If it is archived, the value returned will be less than 32768
-θ' contains the flash page the variable is on, if it is archived, otherwise θ' is 0
As an example, Get("ESPRITES→A' would return a pointer to the data of prgmSPRITES in A'.

(— Use this to read a byte of data from RAM

{ — Use this two read a two byte value from RAM (little endian)

int(— Use this to write a byte of data to RAM.

iPart(— Use this to write a word of data to RAM, little endian (a word is 2 bytes). For example, to set the first two bytes to 0 in prgmHI:

```
:Get("EHI→A
:iPart(A,0
```

Send(— Use this to create Appvars or programs of any size (so long as there is enough memory). For example, to create prgmHI with 768 bytes:

```
:Send(768,"EHI
```

Programs must be prefixed with "E", protected programs "F" and appvars "U"
Also, you can use lowercase letters if you want :)

[
[[
[(— This allows you to write multiple bytes to a RAM location. For example, to write some bytes to the address pointed to by A:

```
:A[1,2,3,4
```

To store some values as words, you can use ° after the number. These will be stored little endian. For example:

```
:A[1,2,3°,4
```

In order to store all values as words, use **[[** instead:

```
:A[[1,2,3,4
```

To directly store hexadecimal, use **[(**. For example:

```
:A[(3C7EFFFFFFFF7E3C
```

IS>(— This is used to read memory. The argument is one of the pointer vars. It reads the byte pointed to by the pvar and then the pvar is incremented (so consecutive uses will read consecutive bytes). For example, to display the hex of the first four bytes of a var:

```
:Get("EPROG→Z
:Text('0,0,IS>(Z,16     ;The bold is the Text( arguments.
:Text('°,IS>(Z,16
:Text('°,IS>(Z,16
:Text('°,IS>(Z,16
```

Archive Follow this with a var name to archive the var. For example, to archive prgmPROG, do this:

```
:Archive "EPROG
```

Unarchive Use this like Archive, except this unarchives the var

Delvar Use this like Archive, except this will delete a var

sub(Use this to remove data from a variable. the syntax is:

```
:sub(#ofBytes,Offset,"Varname
```

For example, to delete the first 4 bytes of program Alpha:

```
:sub(4,0,"EAlpha
```

augment(This is used to insert data into a var. The syntax is:

```
:augment(#ofbytes,Offset,"VarName
```

For example, to insert 4 bytes at the beginning of appvar Hello

```
:augment(4,0,"UHello
```

Now let's make an array! First you need to know what you want. Do you want to have 2-byte pieces of data or 1-byte? I like using 1 byte, so here is what we do:

```
:.0:
:Send(256,"VDat→Z     ;We create a TempProg with 256 bytes of data called Dat.
:Z[rand,rand,rand     ;write 3 random bytes.
:ClrDraw
:For(3
:Text('°Is<(Z         ;Display the value at byte Z. Also increments Z.
:Text('°",
:DispGraph
:End
:Stop
```

That didn't really need a 256-byte vaariable, but I figured I would show how to make one. Anyways, what that did was make a 256-byte tempprog (which the OS automatically deletes once control is returned to the OS and you are on the homescreen). Then, we stored 3 random values to the first three bytes, then we displayed those values with commas after each number. If you want to use that 256 bytes for a matrix, instead, you can make it a 16x16 matrix and access elements using a formula. For example, to read (Y,X):

```
:(Z+X+Y*16
```

That means that the data is stored in rows. That is why we take the row number and multiply by 16 (that is the number of elements per row). This happens to be the syntax that tilemaps are stored (stored in rows).

Binary Lesson!

This document was designed to explain the basics of Decimal (our number system), Hexadecimal (base16, the ASM number system), and binary (machine code, 0's and 1's). Again, this is going to be very basic. Check the internet if you want to learn more.

Converting from Dec to Hex:

1) Divide the number by 16. The remainder is the first number. If it is 0 to 9, just keep that. If it is 10 to 15, use letters A to F.
2) If the number is 16 or larger, still, divide by 16.
3) Repeat step 1 and 2 until finished.

Here is an example of 32173 converted to Hex:

32173/16= 2010 13/16 Remainder=13 "D"

2010/16= 125 10/16 Remainder=10 "A"

125/16= 7 13/16 Remainder=13 "D"

7/16= 7/16 Remainder=7 "7"

So the number is **7DAD**h

Converting from Hex to Dec:

I will start this with an example of 731h:

1*16^0 1

3*16^1 48

7*16^2 1792

Add them all up to get 1841. Did you see the pattern with the 16^n?

Binary.

Converting to and from binary is pretty similar. Just replace all the 16's with 2's and you will have it.

Octal.

Replace all the 16's with 8's.

Other Bases.

Replace the 16's with whatever number you want.

Here is some cool knowledge for spriting. Each four binary digits represents one hexadecimal digit. For example:

00110101 corresponds to 35. This makes it super easy to convert a sprite which is binary to hexadecimal! You only need the first 16 digits, so here you go:

0000 = 0	□□□□	0100 = 4	□■□□	1000 = 8	■□□□	1100 = C	■■□□
0001 = 1	□□□■	0101 = 5	□■□■	1001 = 9	■□□■	1101 = D	■■□■
0010 = 2	□□■□	0110 = 6	□■■□	1010 = A	■□■□	1110 = E	■■■□
0011 = 3	□□■■	0111 = 7	□■■■	1011 = B	■□■■	1111 = F	■■■■

Charts

Key Codes

You can use this as a guide to the key values ouput by getKey in Grammer. For example, Clear=15

```
    ______________________________________
   /         TI-84 Plus Silver Edition     \
   |   ________Texas Instruments________    |
   |   |                               |    |
   |   |                               |    |
   |   |                               |    |
   |   |                               |    |
   |   |                               |    |
   |   |                               |    |
   |   |                               |    |
   |   |_______________________________|    |
   |    _ _     _ _     _ _     _ _     _ _ |
   |  / 5 3 \ / 5 2 \ / 5 1 \ / 5 0 \ / 4 9 \|
   |                                _       |
   |    _ _     _ _     _ _      __|4|__    |
   |  / 5 4 \ / 5 5 \ / 5 6 \   |2_   _3|   |
   |    _ _     _ _     _ _        |1|      |
   |  / 4 8 \ / 4 0 \ / 3 2 \               |
   |    _ _     _ _     _ _     _ _     _ _ |
   |  / 4 7 \ / 3 9 \ / 3 1 \ / 2 3 \ / 1 5 \|
   |    _ _     _ _     _ _     _ _     _ _ |
   |  / 4 6 \ / 3 8 \ / 3 0 \ / 2 2 \ / 1 4 \|
   |    _ _     _ _     _ _     _ _     _ _ |
   |  / 4 5 \ / 3 7 \ / 2 9 \ / 2 1 \ / 1 3 \|
   |    _ _     _ _     _ _     _ _     _ _ |
   |  / 4 4 \ / 3 6 \ / 2 8 \ / 2 0 \ / 1 2 \|
   |    _ _     _ _     _ _     _ _     _ _ |
   |  / 4 3 \ / 3 5 \ / 2 7 \ / 1 9 \ / 1 1 \|
   |    _ _     _ _     _ _     _ _     _ _ |
   |  / 4 2 \ / 3 4 \ / 2 6 \ / 1 8 \ / 1 0 \|
   |    _ _     _ _     _ _     _ _     _ _ |
   |  / _ _ \ / 3 3 \ / 2 5 \ / 1 7 \ / 0 9 \|
    \_________________Brie__________________/
```

Also, there are the diagonal directions:

5=Down+Left

6=Down+Right

7=Up+Left

8=Up+Right

16=All directions mashed

Data Types

00=Real	log(	
01=List	A	
02=Matrix	B	
03=EQU	C	
04=String	D	
05=Program	E	
06=ProtProg	[	F
07=Picture	]	G
08=GDB	{	H
09=Unknown	}	I
10=Unknown Equ	J	
11=New EQU	K	
12=Complex	$^{-1}$	L
13=Complex List	^{2}	M
14=Undefined	N	
15=Window	^{3}	O
16=ZSto	(	P
17=Table Range	)	Q
18=LCD	2	R
19=BackUp	3	S
20=App	4	T
21=Appvar	5	U
22=TempProg	6	V
23=Group	7	W

- Format is not compatible
- Do not use.
- Symbol Var. Compatible with each other.
- Named Var. Compatible with each other.

Examples

Here are some code examples to hopefully help you. Some are really short snippets of code, others are longer pieces.

Movement

Here is code that changes X and Y based on key presses.

```
:X+getKey(3
:min(-getKey(2,95→X
:Y+getKey(1
:min(-getKey(4,63→Y
```

Particles

```
:.0:
:0→X→Y
:Repeat getKey(15
:R▶Pθ(
:If getKey(9
:P▶Rx(Y,X
:X+getKey(3
:min(-getKey(2,95→X
:Y+getKey(1
:min(-getKey(4,63→Y
:End
:Stop
```

Thanks

I have to give special thanks to Yeongjin Nam for his work on writing a better tutorial for Grammer and as well Louis Becquey (persalteas) for his work on writing a french readme/tutorial. Both of them have also made many vlauable suggestions that have helped make Grammer what it is right now. Thanks much!

I would also like to thank the sites that have let me get the word out about this project, especially Omnimaga and Cemetech.