Project.....Grammer
Program.....GRAMMER
Author.....Zeda Elnara (ThunderBolt)
E-mail.....xedaelnara@gmail.com
Size.....4633 bytes
Language.....English
Programming.....Assembly
Version.....1.05.07.11
Last Update.....5 July 2011

Grammer Tutorial

Introduction

Grammer is a programming language designed to be powerful, fast, efficient, and somewhat safe. However, with all of that said, though it isn't as tough to learn as Assembly, it will probably be a little more difficult than BASIC. One similarity to BASIC is that Grammer is interpreted. This makes it slower than assembly, but it is smaller and more safe.

This tutorial will be designed to show Grammer users all the cool features and hacks that Grammer provides, so have fun!

Terminology

Pointer

The point of a pointer is to tell the parser where data is. This is the same thing as an address in assembly. All of the code and data is stored somewhere in memory, and a **pointer** tells Grammer where the data or code is. When you are using a string, for example, Grammer uses a pointer that tells it where the string data is.

Sprites

A sprite is just an image. In Grammer, sprites are a multiple of 8 pixels wide up to 96 and up to 64 pixels tall.

Strings

Strings can be any form of data. They can be text, sprite data, label names, even code.

Pointer Vars

Pointer Vars are the letters A through Z and Theta as well as there primes. For example, S and S' are two different pointer vars. These are used to store pointers or values. Pointer vars are 16 bits, so only integers from 0 to 65535 are stored to them.

Subject 1-Numbers

In Grammer, values are integers from 0 to 65535. This is one of the key differences between Grammer and TI-BASIC. While this may make some things difficult, this has a few uses and effects not found in BASIC. First, what happens when you go beyond 65535? The numbers simply loop back to 0! So what happens when you go below 0? You loop back starting at 65535! This is called modular math, so if you understand this section, you will actually be understanding an important concept in number theory. The uses of this will come later...

Subject 2-Math

In Grammer, math does not follow Order Of Operations and has some limiting factors (like only working with 16-bit integers). However, there are exploits that can be useful for this. Anyway, math is done from right to left, so as an example, we will look at 3*4+6/4-2:

3*4+6/4-2
3*4+6/**2**
3*4+**3**
3*7
21

The other math symbols are ^{2} and the negative sign. The tricks, however, come from Θ'. There are several operations that modify Θ' as a way to return additional information. For example, when adding, if the result exceeds 65535, 1 is stored to Θ', otherwise it becomes 0:

+: If the result is greater than 65535, Θ' is 1, otherwise it is 0.
-: If the result is less than 0, Θ' is 65535, otherwise it is 0.
*: Θ' is the upper 16 bits allowing for a 32-bit result.
/: Θ' is the remainder
^{2}: Same as multiplication

If you wanted to do **If 24=((A*8)+(B*12))**, you would do this:

```
:A*8
:+B*12
:If =24
```

However, if you want to keep it all on one line (because anything in an If or While statement does not modify Ans), you can use a space instead of a newline:

```
:If A*8 +B*8 =24
```

Subject 3-Logic

Logical operators are used to compare values. If the logic is true, the result is 1 and if it is false, the result is 0. As an example, 3>4 is false because 3 is not greater than 4, so this returns a 0. This is useful for If statements.

Subject 4-If statements

Subject 5-Loops

Subject 6-Labels

Subject 7-Strings

Subject 8-Sprites

Subject 9-Sub Routines

Grammer Examples

I figured the easiest way to learn was through examples, so here are some codes to play with and be sure to read up on the commands! Just click the commands in the code to jump to the link!

Simple Loop 0

This will loop until enter is pressed!

```
:Return→A
:If 9≠getKey
:Goto A
```

Label Abuse 0

```
:.HELLO          ;The label HELLO
:"HELLO          ;Stores the pointer in Ans to this string
:If getKey=15
:"BYE            ;Replaces pointer in Ans if clear is pressed
:→Z              ;Stores pointer in Z
:Goto Lbl Z      ;Finds the label named by string Z and jumps to it
:.BYE            ;The label BYE
:Stop
```

Sprites!

Just use the arrows to move the sprite around and press clear to exit.

```
:Lbl "SPRITE→D
:0→B →C
:While A≠15
:Pt-On(4,D,B,C,1,8     ;Swaps the data on the screen with the sprite data
:DispGraph             ;Displays the graph
:Pt-On(4,D,B,C,1,8     ;Re-swaps the data
:getKey→A
:B                     ;Just to get it in Ans
:If A=1
:+8                    ;Adds 8 to Ans
:If A=4
:-8
:If =64                ;Checks if Ans is 64
:56
:If >56                ;Uses the trick that negative values are large :D
:0
:→B                    ;Stores Ans to B
:C                     ;same process as with B
:If A=3
:+1
:If A=2
:-1
:If =12
:11
```

```
:If >11
:0
:→C
:End
:.SPRITE
:01234567
:Stop
```

Commands

*Not like BASIC, "Ans" is always the last computed value, not the value from the previous line.

→

This stores Ans to a variable. For example:

```
:Return→A'
```

That will store the value output from Return to A'

//

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 so long as there is a space or colon before the //. Examples of valid comments are:

```
:////This is a comment, so 3→A does noting to A.
:1→A
```

or:

```
:1→A //Hi!
```

or:

```
:6→A://Rawr.
```

.

This is used to start a label name. Anything in the label is ignored, so commands can be used in labels. For example:

```
:.Circle(
```

Or:

```
:.Circle(0
```

"

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

If x

If "x" is not 0, the line following it will be executed. The line is skipped if "x" is 0. "x" can be any operation resulting in a number. For example:

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

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
```

Return

This returns a pointer to the next line of code in Ans

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 x

This returns the pointer of a label. **x** is a pointer to the label name. See the examples at the beginning to see how Lbl can be used.

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
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
```

While

While loops are like If statements addicted to cocaine-- 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!
```

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
```

DispGraph

Displays the graph screen

getKey

This returns a value from 0 to 56 that is the current key press. You can use this chart for values.

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'.

prgm

This is used to execute a sub routine.

Circle(Y,X,R,Method

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
2-White border
3-Inverted border

*__Method__ has changed and will likely change again

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

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 :)
Height is the number of pixels tall the sprite is

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

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

DataPointer is a pointer to the sprite data
Y is the pixel Y-coordinate
X is a value from 0 to 11. Multiply this by 8 to get the pixel coordinate it will draw to.
Width is how wide the sprite is. 1=8 pixels, 2=16 pixels, et cetera
Height is the number of pixels tall the sprite is

Line('

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

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

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

0=White
1=Black
2=Invert

*This is still buggy, so I would avoid this...

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.

Pxl-On(

This turns a pixel on using coordinates (y,x)

Pxl-Off(

This turns a pixel off using coordinates (y,x)

Pxl-Change(

This inverts a pixel using coordinates (y,x)

ClrDraw

This clears the graph screen buffer and resets the text coordinates

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
```

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

Vertical

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

```
Vertical x,method
```

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

Text(

This is used to draw text to the graph screen. The font is 4 pixels wide and 6 pixels tall and this will wrap text to the next line if it goes off the edge or it wraps it back to the top of the screen if it goes off the bottom.

There are several syntaxes. The most basic is:

```
Text(y,x,"Text
```

y is a pixel coordinate

x is a value from 0 to 23, drawing to 24 columns.

"Text is a string or a pointer to a string

If you want to draw after the last text drawn, you can replace the **y,x,** with a degree symbol. For example:

```
:Text(0,0,"RAWR
:Text(°"HELLO
```

If you want to draw a number, the syntax is:

```
Text('y,x,Value
```

So for example:

```
:Text('0,0,337
```

There is also an option to draw the value in another base:

```
Text('y,x,Value,base
```

So to draw the value in binary, you can do:

```
:Text('0,0,337,2
```

If you want to use the degree symbol, simply replace the **y,x**, again:

Text('°Value

Text('°Value,base

So to use a few examples:

```
:Text(0,0,"QWERTY.
:Text('°35,6                    ;Draws 35 in base 6
:Text(°" IS A FISHBOT
```

That will display "QWERTY.55 IS A FISHBOT" on the graph screen.

To better fine tune your control of text output, there is yet another optional argument to limit the number of displayed characters. This is not available for numerical display (yet), but an example is:

```
:"sin(→A
:Text(6,0,A,3
:Text(12,0,A,2
:Text(18,0,A,1
```

Math

- **/** is used to divide two numbers. The remainder is stored in theta prime.
- ***** is used to multiply two values. The lower 16 bits are stored to "Ans" and the upper 16 bits (for the 32-bit value) are stored in theta prime.
- **-** is used to subtract two numbers. Numbers below 0 are calculated as if 65536 was added. For example, 3-6 would result in -3 which is 65536-3 or 65533. If the number goes below 0, theta prime is 1, else it is 0.
- **+** is used to add two numbers. If the number exceeds 65535, 65536 is subtracted from it and theta prime is 1. Otherwise, theta prime is 0. For example, 65534+99 would return 97, and theta prime as 1.
- ² multiplies a number by itself
- √**(** takes the square root of the number
- √**('** takes the rounded square root of the number
- **sin(** takes the sine of a number. This has a period of 256 and returns a value from -127 to 127.
- **cos(** takes the cosine of a number. This has a period of 256 and returns a value from -127 to 127.
- **abs(** returns the absolute value of a number. If the number is greater than or equal to 32768 (2^15), this returns 65536 minus the number. For example, **abs(65533)** would return 65536-65533=3.
- **min(** returns the lower of two values. For example, **min(3,A)** returns 3 if A is larger than 3 or the value of A if A is less than 3.
- **max(** returns the larger of two values.
- **gcd(** returns the Greatest Common Divisor of two numbers
- **lcm(** returns the Least Common Multiple of two numbers
- **nCr** will perform the operation **n choose r**.

Logic

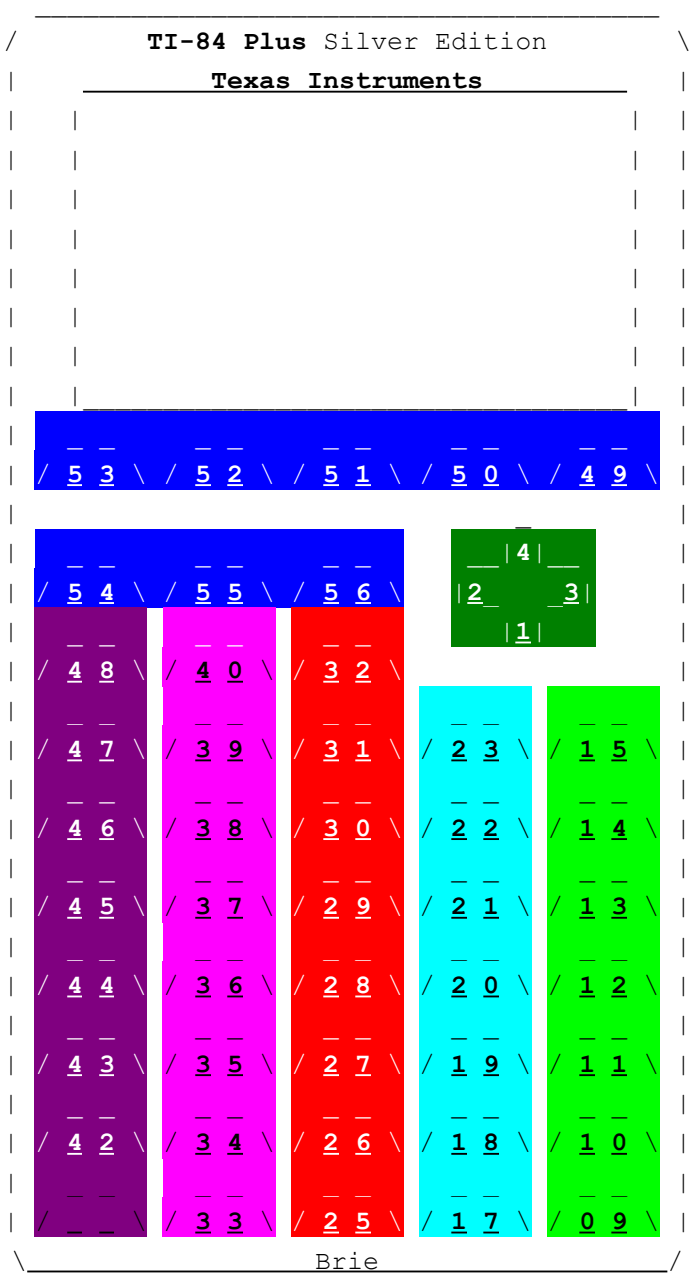
This will be explained in terms of "x (logic) y" where x and y are values

- **=** returns 1 if x is equal to y. Otherwise, it returns 0.
- **<** returns 1 if x is less than y. Otherwise, it returns 0.
- **>** returns 1 if x is greater than y. Otherwise, it returns 0.
- **≤** returns 1 if x is less than or equal to y. Otherwise, it returns 0.

- ≥ returns 1 if x is greater than or equal to y. Otherwise, it returns 0.
- ≠ returns 1 if x is not equal to y. Otherwise, it returns 0.

getKey Values

Creepily enough (I just checked), this is almost the exact size of my real calc O.O That shows how much I use it... Anywho, you can use this as a guide to the key values ouput by getKey in Grammer. For example, Clear=15



Questions and Answers

Since I started this project 10 May 2011 (2 days into summer vacation), I haven't been able to get any kind of feedback, so I will make up some fun questions for now :D

Q: I want to crash my calculator. How can Grammer help me do this?

A: Um, at the moment, the best way is to write data to random spots in memory.

Q: Can I have an example of that?

A: Sure, I guess...

```
:Return→A
:int(rand,-1      ;the -1 is an optimisation for 65535 :)
:Goto A
```

I cannot guarantee that this will crash, but it should make things volatile!

Q: I found a bug! What do I do?!

A: Pick it up gently and inspect it. Try to find what brought the bug to you and let me know so that I can try to find it, too :)

Q: I have some ideas! Would you like to hear them?

A: Yes! I might not be able to implement them or I might have reasons to not implement them, but if I can, I would love to! If you can think of a syntax, too, that would be great!

Q: Can I take a look at the source?

A: Sure, I don't mind! However, if you want to release a modified version, please inform the end user that it is modified and how these modifications change program flow. For example, if you change Grammer to handle only 8 bit values, I would be pretty confused when I get a bunch of bug reports about not being able to use Goto and Return and whatnot properly :)

Q: <<Your question goes here>>

A: <<My response goes here>>

Grammer Diary

2 July 2011 (19:26)

An EnPro fan e-mailed me with a request for a sprite command that could draw to pixel coordinates. Because of this, I have gotten to work on a better sprite command and I decided to test drive it with Grammer... and it works! It isn't as complete as I would like it to be (It doesn't have an option for width or DataSwap), but it is much more user friendly than before. It can only draw sprites 8 pixels wide and it does not clip the sprite. I left the Width argument in case I can later add code to use it.

15 June 2011 (07:43)

I modified the sine/cosine routine to speed it up slightly. I originally got the gist of the algorithm from Axe code, but I have since remodeled it and optimised it so much that it only loosely resembles its original form.

14 June 2011 (17:27)

After playing around with my other hobby (math), I wrote an algorithm to compute **nCr**. The cool part about it is that it is a legitimate use of some of the math that I research, so I feel useful ^-^. For the curious, this is a polynomial time algorithm (no computing factorials). This is what the algorithm looks like in Grammer Code as a subroutine. Inputs are N and R and the result is output in D:

```
:N-R→N
:If <R
:N→A R→N A→R        ;Cuts down computation time
:1→C →D
:For(A,1,R
:C*N
:/A→C
:+D→D
:N+1→N
:End
:End                ;Exits the subroutine
```

13 June 2011 (17:09)

Added the Repeat command as another loop command.

13 June 2011 (08:06)

Added the For(command as a new loop command.

12 June 2011 (22:24)

I added code to convert tokens to ASCII for cases like text display or searching for a var. This means that named vars with lowercase letters (like some appvars) can now be accessed and tokens are now displayed properly with the text command. Also, I added the lcm(command and made a few mini games that still need finishing touches.

11 June 2011 (14:08)

The prerelease never happened, but I have added in several new Text(syntaxes to allow for displaying numbers, I added the Full command to allow manipulation of the processor speed (set to 15MHz, 6MHz, or toggle), I added ClrDraw, ClrHome, min(, max(, abs(, rand, gcd(, Shade(, Horizontal, Vertical

6 June 2011 (12:51)

Preparing for a pre release of Grammer... Also, I added a modified circle routine that is fast O.O

4 June 2011 (16:45)

I decided to add in one of my square root routines as well as Pxl-On, Pxl-Off, and Pxl-Change. I also made it so that the user could obtain the rounded square root (rounded to the nearest whole) and I made the pixels not draw if they were off screen. The first thing I did was make some circle sub routines to play with :) Now I think I will try to make the sin/cos routines a little more accurate by adding rounding to that, too...

1 June 2011 (19:48)

I randomely decided to add **prgm** as a method of executing sub routines. I also added **Pt-Off(** as a "concept command" for a sprite routine that draws variable size sprites to pixel coordinates. Currently it has a weird sprite data syntax and the sprites need to be a multiple of 8 pixels wide, but it does draw to pixel coordinates.

17 May 2011 (21:50)

I have been thinking about adding assembly support like this:
AsmPrgm will execute a simple hex opcode
AsmComp(will use multiple line support with whatever goodies I add. I have a program that can compress and execute code like this that has comment support. Maybe I will add support for labels and some instructions...
AsmComp(will load 5 pointer vars in a row (like R,S,T,U,V) into the register pairs af,bc,de,hl,ix and then use an argument to execute a call in a jump table. It will then return the values of the registers in the pointer vars. This will likely be used only by me as a way to debug new routines.

These are only tentative. The first will probably be added, but if the other two are added, they will likely be modified often and won't remain backward compatible.

17 May 2011 (19:58)

I have been working on other projects, but I came back to this today. I decided to try and make a text drawing routine that could draw to pixel columns (instead of every fourth one). I started thinking of ways to do this when I thought that making a general sprite routine would work better. The problem? I've never made a sprite routine that worked on pixel columns. So I started coming up with ideas for how to draw this and I thought that I could make a general sprite routine for any size sprite that could be mapped to pixel coordinates. To do that, I made a mask routine for masking the sprite data and the screen data. However, I side tracked myself and ended up using the mask routine as a way to make rectangle routines, so all that I accomplished was that. I wrote it from scratch and it should be faster than the one I made for BatLib because I draw the boxes as sprites, so I do not need to draw whole rows. Plus, it works unlike some of the OS routines and it has 10 fill methods.

14 May 2011 (16:11)

After just getting back from a baby shower 15 minutes ago, I have added reading and writing bytes and words. I plan to add the same ability for nibbles, too, but for now, this should be great for modifying sprites and game data.

14 May 2011 (11:00 ish)

I have now added While loops (that can be nested) as well as the Get(command to start referencing OS variable data. I have also been putting some work into the tutorial, but that is going to be a tad difficult. Pretty soon I will need to add in some rectangle routines and text routines

13 May 2011 (almost noon)

I have been trying to tease out a bug that causes a crash every so often. During program execution, there is no problem, but when it exits, it crashes. Since some LCD stuff was going crazy, I wrote an LCD routine to display the graph buffer and while I was at it, I decided to make a routine to test the ON key (to use as a delay for the LCD writing). Now, I have a way to break out of a program, but it didn't fix the crashing. I am going to see if it has anything to do with the stack pointer...

--A half hour later--

... and this is why I need to practice with mnemonics. I had the hex comment correct, but I had the mnemonic incorrect... For some reason, I was calling the routine to display the graph buffer instead of jumping to it. I am going to see if there is another bug there, though.

--Another half hour later--

... and this time it is corrected in full. What happened was in an If statement, if the result was true, it jumped to the start of the program (where the address of the start is pushed). I should have either jumped to a spot 11 bytes ahead or used ret. I went with ret, this time. It worked before because the code structure was a little different than it is now.

13 May 2011

After working out some kinks in the program flow and fixing up how string arguments are handled, labels, strings, and commands with arguments will work properly, now. I thought labels were working properly yesterday, but when I decided to do **Lbl "HELLO→A,**it didn't work. After some debugging, I figured out that it was looking for .HELLO→ as the label. I only figured out this was an issue when I was testing out the new sprite command and I tried referencing a label to source data from it. I did **Pt-On(0,Lbl "HELLO",0,0,1,8** and it was completely malfunctioning. I then found out that there was also the problem that:

- Arguments weren't able to be read one after the other because the program counter wasn't being updated properly
- The ending quote was being parsed as a starting quote.

I tried fixing the parsing first and my first few attempts failed. I ended up doing things like making it so that only the last argument was read until I figured out that I would need to include code before each consecutive read to update the program counter (instead of making it update when it read a comma).

Fixing the string and label problem wasn't too bad and only required a few bytes of code.

So, yeah, the details of the new sprite command are that it uses Pt-On(, has six drawing methods. It uses byte data instead of nibble data

12 May 2011 (later)

Finished the Lbl command so now data referencing can be started! After *finally* finishing up that SearchString routine (it was a lot simpler than I thought it would be...), users can now use labels up to 764 bytes long, but really, I should cap that to 10 or something x.x

12 May 2011

Today I have added few more things including:

Addition (with 1-bit carry)
Subtraction (with 1-bit carry)
Multiplication (with 16-bit carry)
"Squared" symbol (^{2}) works (with 16-bit carry)
Logical operators like **>** and **=**
Negative **(-)**
sin(
cos(
If
If ... Then... End
getKey
End
"

I am also starting work on the **Lbl** command which will be very useful to have and I also want to get started on graphics commands. If I finish **Lbl**, I will be able to pretty easily implement some sprite commands *cough*

11 May 2011

Okay, actual progress was made besides planning and whatnot. The basic outline of the program was started and I made my first real use of an assembler. Thanks Kerm for your DCS SDK! I have finally managed to figure out how to compile assembly source code and have it packaged as a program! Now I need to figure out how to turn it into an App...

The current syntax allows the user to put the name of the var with Grammer code in Ans (as a string) and then do Asm(prgmGRAMMER to start the Grammer parser. It currently has Ans as a debugging tool (it displays a value) and it can convert numbers to 2 byte integers. I also added division, commenting, storing to and reading variables, DispGraph, Goto, and Return

10 May 2011

Progress on Grammer started after about a month of debating and ideas.