AXE PARSER

Created By
Kevin Horowitz

Version 0.3.0 Gamma
June 16, 2010

Complete User's Guide

Table of Contents

Overview

Summary

So what exactly is Axe Parser? Axe is a revolutionary new programing language for the TI-83/84 series calculators. It is designed with game creation in mind, but the applications are limitless. It offers an alternative to the restrictions of BASIC but without the complexity of pure Assembly. An Axe program can compile into a no-stub executable or to your favorite shell and therefore does not require any external application to run your program.

Axe vs. Other Languages

	BASIC	xLIB/Celtic	BBC Basic	Assembly	Axe
Language Difficulty	Easy	Easy	Medium	Hard	Medium
Speed	Slow	Medium	Fast	Very Fast	Very Fast
Editable On-calc?	Yes	Yes	Special Editor Required	Not Reasonably	Yes
Execution	Interpreted	Interpreted	Interpreted	Compiled	Compiled
Sprite Support	No	Yes	Yes	Yes	Yes
External Variables Required To Run	Pictures, Vars, Lists, Strings, etc.	Same as BASIC plus 16kb App	49kb App	None	None
Shell Compatibility	Yes	Some	No	Yes	Yes
Specialty	Math	Games	Variety	Everything	Games
Source Code Visible	Always	Always	Always	Optional	Optional

There is no one programming language for everything, each has advantages and disadvantages. It is up to you to decide what best fits your needs. If your top priorities are speed, ease of use, and the ability to do lots of things, then Axe Parser is definitely the way to go.

Freedoms & Limits

Axe Parser allows you to do what most calculator programming languages do not. You are able to draw sprites, access external variables (like appvars), multiplayer linking, grayscale support, interrupts, sound, multi-key press detection, contrast adjustment, and much much more. All of this usually at the same speed as regular assembly. You even have the option of inserting your own asm code directly into the program.

Unfortunately, with every new freedom, there is a price to pay. Like assembly, but unlike the other languages, a bad crash in Axe will usually cause a RAM clear and you will lose most of your data. If the crash is really really bad, its possible but unlikely, that the archive could get corrupted as well. For this reason, it is recommended that you begin programming on an emulator until you get used to the language enough to where you don't make those kinds of mistakes. I recommend wabbitemu which can be downloaded at RevSoft.org. Another option is to remember to archive your source code and backup all your programs before you start experimenting with Axe Parser.

Another thing to keep in mind is that the programs are not as optimized as pure assembly. They will be on average one and a half to two times larger than if the program were actually optimized by an experienced asm programmer. However, there are many optimizations that can be done using Axe itself, and that will be discussed in a later section.

Axe is the ultimate tool for a good execution (of programs)

Using Axe Parser

Writing & Editing Programs

To start making your programs, you follow the exact same procedure as if you were starting a BASIC program. You go to [PRGM], New, and then type in the name. This is your **source code** so make sure you pick a name that's not the same as what you plan to name your final program. For instance if you plan to make “MARIO” you might want to name your source “MARIOSRC”.

The next thing you absolutely need is an Axe Header. You must start the first line with a period, which is a comment in Axe, followed by the name you want for the compiled program. If you want a program description, you can type a space and then the description on that same line, but this is optional. In the above example your program might look like this:

```
PROGRAM:MARIOSRC
:.MARIO A fun platformer
:
```

That's basically all you need to get started. This program should show up on the list of programs in the Axe Compile menu. You can compile from both RAM and Flash.

Compiling Programs

To compile a program, go ahead and open up the Axe Application. In “Options” you can select what type of shell you want to compile for. If you don't want a shell, you can run your programs using the “asm(“ command in the catalog.

Next, select “Compile” to see a list of all Axe source code files on your calculator. Select the program you want to compile and hit “Enter”. Compiling usually takes about one second for every 4000 bytes of source on an 84+ so don't be surprised if its almost instant. If you get an error, check the error codes section to diagnose the problem. In future versions, Axe will be able to scroll to the error as well.

Once compiling is complete, you should see the executable in your programs list. Remember it is recommended that you archive the source before actually running the file to prevent loss of data in case of a crash.

Error Messages

ERR: LBL MISSING	You called a label or subroutine that does not exist.
ERR: STACK FULL	There are too many parenthesis in a single expression.
ERR: MAX SYMBOLS	There are too many static pointers or labels.
ERR: DUPLICATE	The label or static pointer already exists.
ERR: BLOCK	Either an “End” is missing you you have too many “End”s.
ERR: HEXIDECIMAL	The hexadecimal number is invalid or not the right size.
ERR: BAD NUMBER	The number is too large or small.
ERR: BAD NAME	Invalid name for a label or static pointer
ERR: UNDEFINED	The static pointer you are using does not exist.
ERR: OUT OF MEM	The ram is full and there is no more room to write the program.
ERR: BAD SYMBOL	Very common error. You have either used an unsupported command or you used the wrong data for a supported command e.g. A number where a variable is expected.
ERR: FILE NAME	You named the output file the same as your source.
ERR: PIC MISSING	The picture that is to be absorbed does not exist.
ERR: UNKNOWN	Something is wrong with the parser. Report bug immediately!

Programming

Numbers & Basic Math

Alright, now to the fun stuff! Okay, this is one of the most important differences between Axe and BASIC. Numbers in Axe are all 16-bit **integers** meaning that there's no such thing as fractions and decimals. What the 16-bit part of it means is that a number can only hold a value between **0** and **65,535**. This is called the **unsigned** number system meaning that there is no sign: all the numbers are positive.

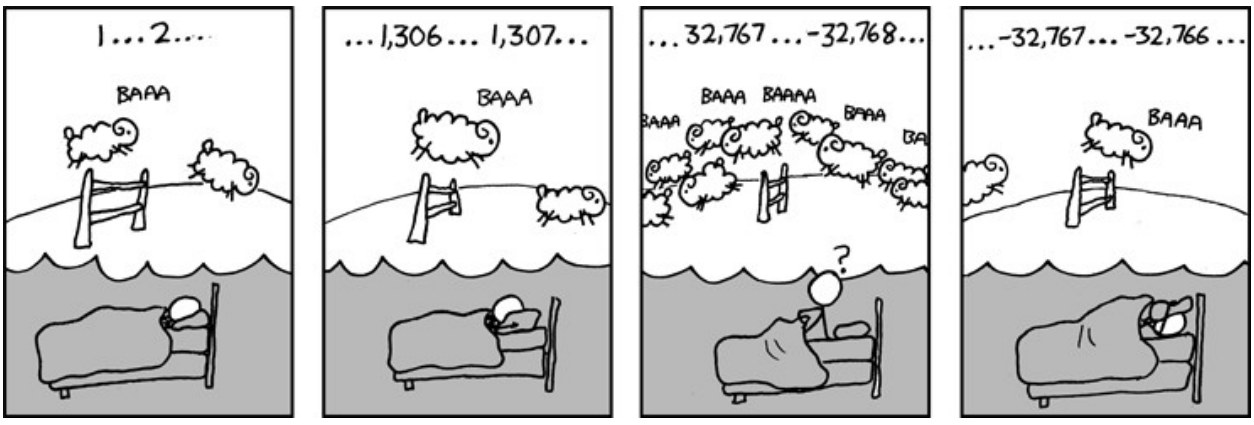
Now wait a minute you say, what if I want to use negative numbers? Well in that case, you want to use **signed** representation. The way that works is that we cut our range in half and say that all the numbers on one side are positive and numbers on the other side are negative. So numbers from 0 to 32767 we still say are positive but now the numbers from 32768 to 65535 we say are actually -32768 to -1. So our new range is **-32,768** to **32,767**

Remember, both representations are really the exact same number. It's just a different way of representing it. So -1 really is the same number as 65535. And I'm not just making this up, the mathematics works this way. If you add 65535 to a number you get exactly the same result as if you subtracted 1! How the heck does that happen? That brings me to my next point which is **modular arithmetic**.

Remember, we can only hold values between 0 and 65535 right? Well what happens if we keep adding and **overflow** the maximum value? Let's count by constantly adding 1:

```
0,1,2,3,...,65533,65534,65535,0,1,2,3...
```

Do you see what happened? Once you pass the maximum, you loop around back to 0 again. Signed representation does the same thing, best illustrated by this xkcd comic:



Now lets get into some Axe code to apply our new-found knowledge. This will all appear to be very similar to BASIC, but remember, the number system works in a completely different way. Lets make a program that finds the sum of the first 100 numbers:

```
PROGRAM:A                   You have to add ▸Dec to
:.SUM100                    display numbers because
:0→S                        there are multiple ways
:For(N,1,100)               to display data in Axe.
:S+N→S                      Numbers display using
:End                        unsigned representation.
:Disp S▸Dec
```

Next up is multiplication and division. Multiplication is nice because mathematically, its just repeated addition, so it works the same for signed and unsigned numbers. Division on the other hand can give different results depending on if you want to do it signed or unsigned. Signed division, or any signed math in general, you can perform by typing the operation twice. For instance:

```
-10/5 = 13105           Parser sees: 65526/5
-10//5 = -2             Parser sees: -10/5
```

Although the math is basically what you would expect, what you might not expect is that Axe always does the **order of operations** from left to right always. So watch carefully how the parser will read this:

```
4-3*8+2/3               Subtraction first
1*8+2/3                 Multiplication
8+2/3                   Addition
10/3                    Division
3                       Always rounds down.
```

Of course, you can us parenthesis whenever you need them. The reason I don't automatically add the parenthesis for you to do the usual order of operations is because I want you to be aware of how many parenthesis you use. The less you need, the more optimized the code is and the faster and smaller it will be. You'll see more about this in the optimization section.

Another thing I should mention is that the carrot key "^" no longer means "power". It now is the modulus operator. It means the remainder left over after a division. You can learn more about this command and the other advanced math commands in the commands list section.

Pointers

You must understand pointers! If you've never used a programming language with pointers before, pay VERY close attention. Pointers are such powerful and useful tools that you'll be using them in virtually all of your programs. We'll start slow.

What is **RAM**? You probably know its just a bunch of memory, 65536 bytes of it to be exact on the TI-83/84. But how do you access all of that RAM? How do you read and write to it? We just follow the simplest procedure; give every byte in RAM its own **address**. The 0th byte has the address 0, the 1st byte has the address 1, all the way up to the last byte with the address 65535. Get used to starting your counts at zero by the way.

So lets make some data. We'll start with the classic string:

```
PROGRAM:A
:.HELLO
:"Hello World"→Str1
:Disp Str1
```

Its just like BASIC right? Wrong! It just *appears* to behave like BASIC. Let's look at what's actually going on. First of all, Str1 is a **pointer** which is a number, not an actual string. The sentence "Hello World" gets stored in the executable at some address and the pointer Str1 is just a number that tells us where that string is located in the program. The Display routine really takes a number as an argument. Then it takes that pointer, finds the address it points to, and that's how it knows where to find the string to display.

But since a pointer is just a number, we can do math with it! What do you think would be the effect of doing this modification?

```
PROGRAM:A
:.HELLO
:"Hello World"→Str1
:Disp Str1+6
```

This just outputs "World". That's because the string is stored in order in consecutive addresses. Lets pretend the "H" was located at address 100. That means "e" is at 101, the first "l" is at 102, etc. So by the time we get to the address 106, we're at "W". That's why the display routine skips the hello part.

In Axe, Str1 and the other calculator variables are called **static pointers**. What that means is once you declare them, their value cannot change for the rest of the program. The allowed static pointer variables are Str, Pic, and GDB. You can use these for whatever you want since they're all just numbers anyway, but its a naming convention to use Str for strings, Pic for sprites, and GDB for data. None of these are the BASIC variables by the way, they're entirely new variables that just share the same name for convenience.

Another thing is that Axe allows you to name everything with up to 2 numbers/letters instead of just a single number. The following are all unique valid names:

```
Str1        Str01       Str66
Pic0        Pic9Z       Pic8C
GDB4        GDB45       GDB3X
```

There are other ways to enter data, one of those is by using **hexadecimal**. You don't really have to worry about how it works, but its just a more compact way to write a number since there's 16 characters for each digit. This is convenient for sprites since you can use a tool to convert them. One such tool is included in the package.

So now lets draw a happy face sprite.

```
PROGRAM:A                       Pt-On Draws sprites and
:.HAPPY                         takes the position on the
:[3C7EDBFFBDDB663C]→Pic1        screen and a pointer to
:ClrDraw                        the sprite as arguments.
:Pt-On(44,28,Pic1)              We then have to update
:DispGraph                      the screen to see it.
```

Data & Arrays

So how do we actually manipulate data? First, let's take care of reading. We have to tell the calculator what byte to read, so we need to give it the address. Let's look at how we can read a list of numbers:

```
PROGRAM:A                       ΔList is data in the form
:.LISTREAD                      of a list of numbers.
:ΔList(5,2,6,11,4)→GDB1
:For(A,0,4)                     The imaginary 'i' is a
:Disp {A+GDB1}▶Dec,i            newline character.
:End
```

We use the curly brackets to indicate that we want the byte that's at the address of the pointer, not the the value of the pointer itself. This is what the pointer is said to "point to" in a process called **referencing**.

We can similarly store values into these addresses:

```
PROGRAM:A                       Remember, bytes stored in
:.LISTWRIT                      memory can only hold
:Δlist(0,0,0,0,0)→GDB1          values between 0 and 255
:For(A,0,4)                     unsigned or -128 to 127
:A*2→{A+GDB1}                   signed.  If you want to
:Disp {A+GDB1}▶Dec,i            have the full range, see
:End                            the {}r command.
```

Even though you wrote over data in the program, that data isn't going to be saved when you exit. That's because when you run a program, it actually executes a copy of the program and when its done executing there's no **write-back** meaning that the temporary copy just disappears and doesn't replace the original.

Its really pointless to have to store this list, called an **array** in the program itself since its data isn't actually used and it gets written over anyway. Instead, the calculator actually has some special addresses called **free ram** that you're allowed to use for whatever you want. The addresses to the largest of these locations are in the constants L1-L6. You'll have to see the command list for more details about which ones are best to use. I'll use L1 and L2 in my examples.

Lets say you need to control 20 sprites on the screen and each sprite needs an X and Y coordinate that has to be stored and updated. Here is an example of what you can do:

```
PROGRAM:A                       The easiest way to get
:.DRAW20                        random numbers is to take
:[3C7EDBFFBDDB663C]→Pic1        the modulus with X to get
:                               a random number between 0
:.Initialize XY                 and X-1.
:For(A,0,19)
:rand^88→{A+L1}                 Also, when you're using
:rand^54→{A+L2}                 free ram, it DOES NOT use
:End                            any program ram like in
:                               BASIC.  A list or matrix
:.Display Spites                in free ram takes 0
:For(A,0,19)                    memory whereas the same
:Pt-On({A+L1},{A+L2},Pic1)      data in BASIC can be
:End                            pretty huge.
:DispGraph
```

Matrices, or more generally **multidimensional arrays** are a little trickier. Imagine a 2 dimensional array that we want to store in L1. We have to fit this into a one dimensional array so the best way to do it is by combining it from left to right and top to bottom.

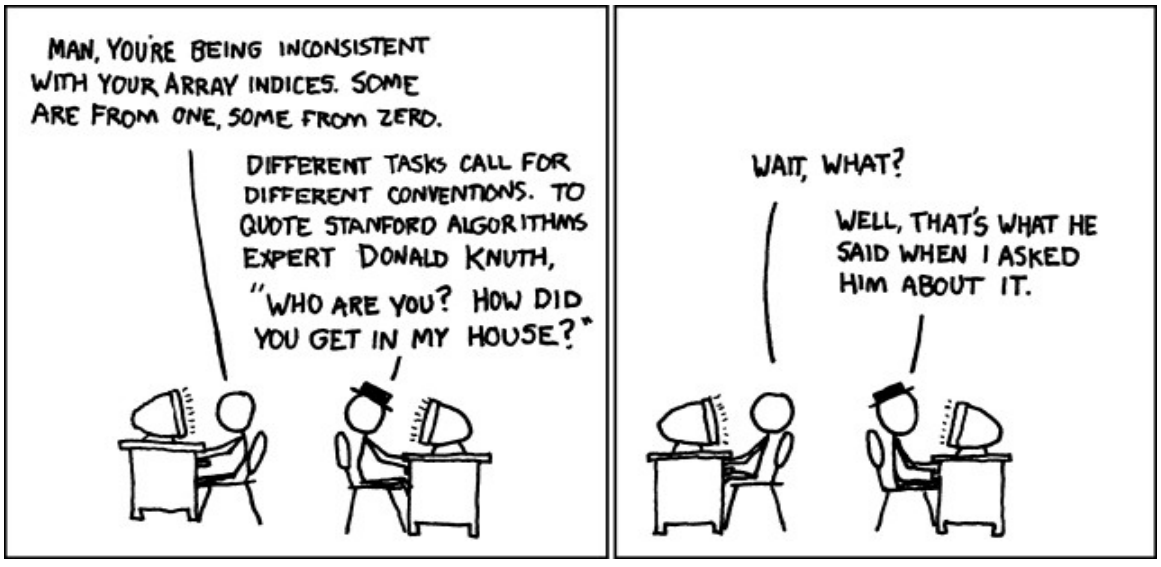
```
1  2  3
A  B  C
X  Y  Z
L  O  L
```

```
1  2  3  A  B  C  X  Y  Z  L  O  L
```

If we know the row "R" and column "C" both starting at 0, we can find the data in that position by using this simple math trick: `:{R*3+C+L1}`

So in general, you can use this formula, which is already in order of operations:

```
<Row#>*<Total Columns> + <Column#> + <Start of Data>
```



Read & Write to Files

There are times when you need to save information that will continue to be there even after you quit the program like high scores for instance. To do that, you can use **appvars** which are files with data. Unfortunately, unlike free ram, appvars move around and thus can not just be located with a static pointer. Instead you have to search the **symbol table** which is just a list of all the files on the calculator to find where the data actually is.

To do this, you must first create a static pointer to the name of the variable you plan to search for. You also need a prefix. For appvars, the prefix is the 'v' located at [2nd] [8] on the keypad. The lowercase 'v' will NOT work. Next, you use the getCalc() command to search for that name and return a pointer to the data. You'll want to save that pointer somewhere so you can use it throughout the program when you work with the data. Lets open up the appvar “MyScore” and change the first 2 bytes to your latest score if its higher than the previous score:

```
PROGRAM:A                       If statements don't need
:.HISCORE                       the "Then" anymore.
:"vMyScore"→Str1
:getCalc(Str1)→P                We use the ʳ modifier to
:If P                           store the full 16-bit
:If S>{P}ʳ                      number instead of just
:S→{P}ʳ                         the first 8-bits.
:End
:End
```

Notice we also have to make sure the appvar exists before we can use it. That's what the “If P” is for. It just makes sure P is non-zero. If it were zero, it couldn't be accessed for whatever reason. Also, programs can be accessed the same way but by using the prefix “prgm” instead.

There are other commands to create, archive, and unarchive these variables. You can read about them in the commands list.

An Example Program

Let's make one of the first video games ever created: Pong! Fist our simple header:

```
PROGRAM:PONGSRC                 Simple Description
:.PONG MY FIRST AXE GAME
```

Next step is to define all our data. There will be 2 strings and 2 sprites:

```
:"PONG"→Str1                    Our Title
:"SCORE:"→Str2                  For the score
:[000000000000FFFF]→Pic1        The paddle
:[0000182C3C180000]→Pic2        The ball
```

Now a very primitive title screen. It will just say "PONG" and pause for a second.

```
:DiagnosticOff                  You should almost always
:ClrHome                        include the DiagnosticOff
:Output(6,3,Str1)               command in start of your
:Pause 1000                     programs because it makes
                                them much cleaner.
```

Now to initialize our variables. The ball's X position will be inflated for more precision.

```
:0→S-1→D                        S=Score, D=YSpeed
:44→Z*256→X                     Z=Paddle_X, X=Ball_X*256
:10→Y                           Y=Ball_Y, V=XSpeed
:sub(HT)                        HT will be a subroutine
                                that is called when the
                                ball hits the paddle and
                                it sets the new speed.
```

Setup the main loop of the program. Also take care of all key presses.

```
:Repeat getKey(15)              By testing individual
:If getKey(2) and (Z≠0)         keys rather than all of
:Z-2→Z                          them at once, it makes
:End                            the program a lot faster
:If getKey(3) and (Z≠88)        and more flexible.
:Z+2→Z
:End
```

Now, we update the new position of the ball.

```
:X+V→X                          Position plus speed
:Y+D→Y                          equals new position
```

Here is the collision detection. Bounce if you hit a wall or the paddle and change the speed. Quit if the ball goes off of the screen.

```
:If Y>70                        Quit when reaching edge.
:Goto D                         D is the ending label.
:End
:If Y=0                         Bounce off the roof and
:sub(HT)                        change direction.
:End
:If Y=54 and                    If the paddle and ball
 (abs(X/256-Z)<8)               are close enough, bounce
:sub(HT)                        and increase the score.
:S+1→S
:End
:If X/256=0 or (X/256=88)       Bounce off of a wall and
:-V→V+X→X                       update the position
:End                            again.
```

Time to draw the sprites and update the screen:

```
:ClrDraw                        The buffer only updates
:Pt-On(Z,54,Pic1)               to the screen when you
:Pt-On(X/256,Y,Pic2)            call DispGraph
:DispGraph
```

We're done with our loop, so lets add the quit code:

```
:End                            End the loop.
:Lbl D
:ClrHome                        What used to be "Stop" in
:Disp Str2,S▶Dec,i              BASIC is now "Return"
:Return
```

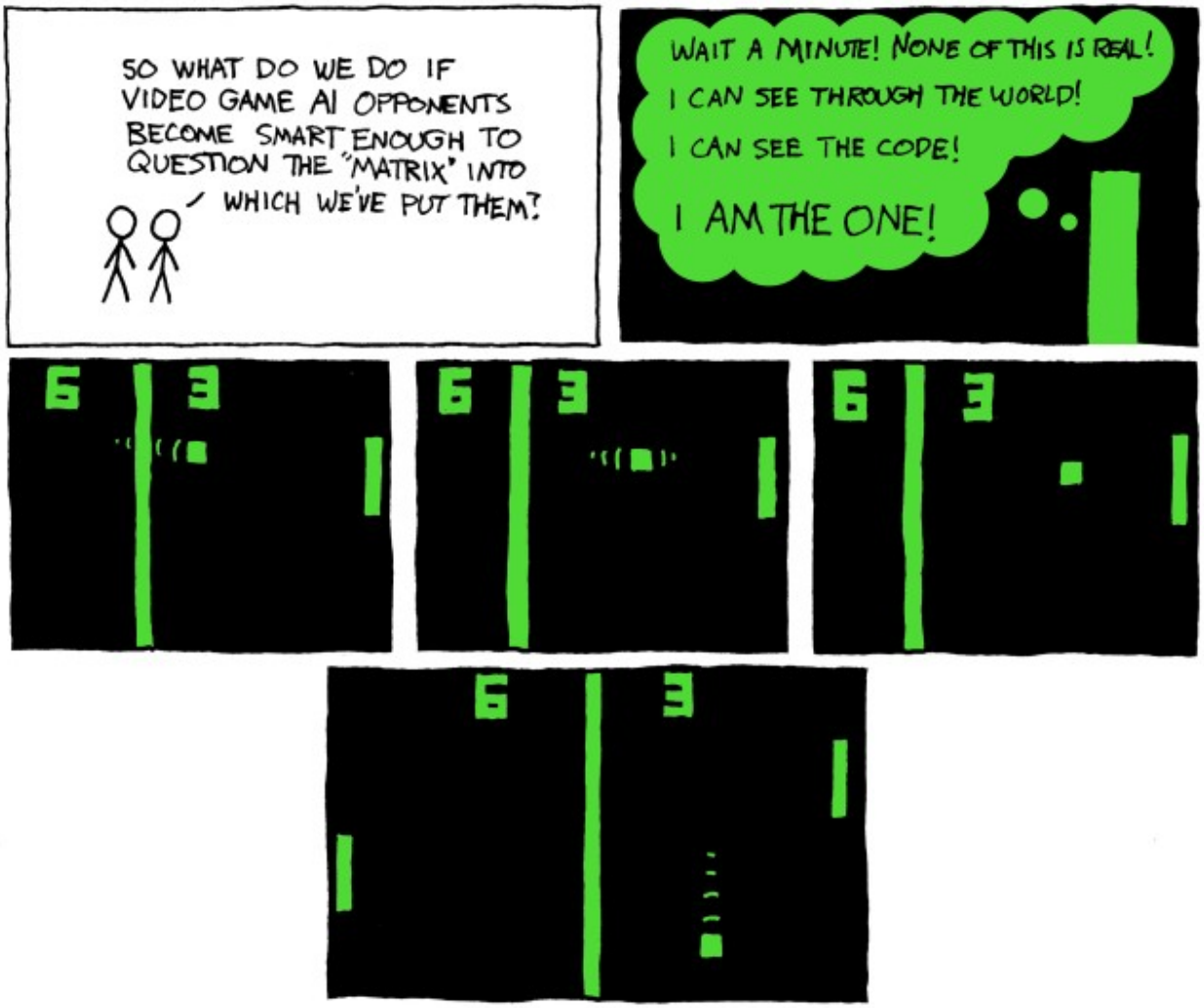
Finally, we'll add our "Hit" subroutine to bounce the ball:

```
:Lbl HT                         Subroutines should end
:rand^512-256→V                 with "Return" but the end
:-D→D                           of the code automatically
                                returns anyway, so no
                                reason to add it twice.
```

That's it! So just run it through the parser and you should see a new “PONG” program show up on your program list. If you didn't use a shell, you can run it by using the command asm(). The result is a fun little game that should keep you occupied for many years to come! (Hopefully this estimate is an exaggeration)

Anyway, the outline here for program flow is a pretty standard practice for game programming. You have in order: the Header, data declarations, title screen, variable setting, the main loop, quitting code, and then all your subroutines. There are a lot more example programs included with the parser, you can check out their source for a better look at exactly how its done.

Good luck with your projects and happy programming!



Optimization Tricks

THE FIRST LAW OF OPTIMIZATION: 99% OF THE TIME, IF THE COMPILED CODE IS SMALLER, ITS ALSO FASTER. Basically, you'll have to play around with different commands. Start a new program and type out the command you are trying to optimize and compile it. Check the size of the compiled program and then try another way of doing it until you find the smallest size.

Evaluate all constant expressions before compiling. It may be more convenient to look at "3*8+Pic1" as the 3rd sprite, but you should definitely write "24+Pic1" instead because its better for the size and speed.

If you're not going to use a subroutine more than once, just put it inline. Even if it might make the program more organized, it is adding at least 4 bytes to the file size.

Conversely, when you do find yourself repeating the same actions twice or more, its probably best to put it in a subroutine. You can always test the program both ways and see if it makes it any smaller.

If the last line of your program is Return, then remove it. Even if it was part of a subroutine, the return is automatically added at the end for you so there's no need to have 2 returns at the end. Saves 1 byte.

If you have constants in your expression, try putting them at the end if at all possible.

Unoptimized	Optimized
`:2*A`	`:A*2`
`:1+A`	`:A+1`

Try to make expressions use the least amount of parenthesis as possible. Use left to right order of operations to your advantage.

Unoptimized	Optimized
`:A*(B+(C*D))`	`:C*D+B*A`

When initializing variables, initialize variables with the same value together. Even if the variables are within 2 of each other, its still worth it to do this.

Unoptimized	Optimized
`:0→A:0→B:0→C`	`:0→A→B→C`
`:10→A:11→B:13→C`	`:10→A+1→B+2→C`

Never make an if or while statement end with "not equal to zero". Instead, you can omit that part since not being zero is implied. Likewise, don't end an if, repeat, or while statement with "equals zero".

Unoptimized	Optimized
`:If A≠0`	`:If A`
`:If A=0`	`:!If A`
`:While A=0`	`:Repeat A`

If speed is super super important, you can chain divisions by 2 to quickly divide by higher powers of two. It could increase the file size, but it will definitely be much faster. For instance, a faster way to do: A/8 is: A/2/2/2.

Animated sprites should be done with pointers. Don't use a huge lists of if-then statements to decide which sprite to display. Just define all the sprites in order one after another and reference them by offset from the first sprite.

Unoptimized	Optimized

`:If A=0:Pt-On(X,Y,Pic0):End`
`:If A=1:Pt-On(X,Y,Pic1):End`
`:If A=2:Pt-On(X,Y,Pic2):End`
	`:Pt-On(X,Y,A*8+Pic0)`

There is a procedure called “Tail Call Optimization” which can save bytes and speed when you have a subroutine call another subroutine right before it returns. Instead, you can just goto to that subroutine and steal its “Return” instead of coming back and using the original one.

Unoptimized	Optimized
`:sub(A)` `:Return`	`:Goto A`

Other Info & Resources

The best way I think to learn more about the commands at the moment is to look through the commands list file and play around with modifying the example programs. Just experiment with different things.

For specific help about specific questions, please I would prefer if you did not email me, instead I have a forum dedicated to the parser at Omnimaga.org. The Axe Parser Forum is currently located at http://axe.omnimaga.org. Its also a great resource to look through the topics there because you might find some very useful information that I probably forgot about or skipped in this tutorial. I usually have a new update there on a weekly basis.

If you want to contact me through email for some other reason, my email is:
compfreakkev@yahoo.com

I'm especially interested in bugs if you find any. Although, you can also report that on the Axe Forums. Make sure you have the latest version of Axe Parser before reporting.

Credits

First off, thanks to the Omnimaga community for their extraordinary help with alpha testing, giving me very helpful suggestions and feedback, and keeping me motivated to work on this project. I couldn't have do it without you guys.

Special Thanks:

Brendan Fletcher (calc84maniac) For his super savvy assembly knowledge and help with creating efficient routines for the parser.

Alex Marcolina (BuilderBoy) For the logo graphics and really showing Axe's full potential with awesome programs and screen shots.

Drew DeVault (SirCmpwn) For helping others with his detailed tutorials online. Also makes really cool programs.

Brandon Wilson (BrandonW) Helped hugely with the many of the awesome features that I would never be able to do without his l33t hacking skills.

And of course giving credits to xkcd for the comics used in this tutorial. Its my favorite web comic. You'll probably like it too if you're nerdy (which clearly you are if you're reading a tutorial about how to program on a calculator).