Dathanna

Ronald T. Kneusel

July 11, 2021

Contents

1 What Is Dathanna?

Dathanna (Irish for "colors") is a simple esolang designed for the Apple II computer. Dathanna programs live on the Apple's low-resolution graphics screen using pairs of colors stacked vertically to specify commands.

As a language, Dathanna borrows the idea of a "playfield" from Befunge-93, but program flow is always left to right, row by row. The playfield has 19 rows and 40 columns, meaning the longest possible Dathanna program is 760 instructions. While program flow is row by row, left to right, top to bottom, the branch instructions specify row and column, so programs are free to jump from place to place within the playfield. The restricted playfield means that Dathanna, like Befunge-93, is not Turing complete.

Dathanna processes data using a stack of at most 16 floating-point values. Additionally, there is a single array of 16 values for off-stack data storage, which may be used as an array or as individual variables. Dathanna implements the full range of arithmetic operators and math functions supported by Applesoft BASIC, in which the interpreter is written. The disk image includes a Dathanna-specific editor to make entering programs (marginally) easier.

Dathanna runs on a native Apple II, but a design goal was to use a fast emulator to run code that would be too slow for real hardware. Programs too inefficient to run on the original hardware are

now possible via high-speed emulation, thereby opening up a new world of possibilities, especially if the emulated environment has some access to the main operating system.

This document will help you configure an emulator to run Dathanna before presenting the language itself. Next comes the editor and some example code to get you started.

2 Emulator Environment

Dathanna comes as a standard, DOS 3.3 order disk image (`dathanna.dsk`). Virtually all Apple II emulators use this format. I use Applewin,

`https://github.com/AppleWin/AppleWin`

which is native on Windows and runs perfectly on Linux via `wine`, which is how I use it. Booting `dathanna.dsk` looks like this,

The interpreter is in `DATHANNA` and the editor in, surprisingly, `EDITOR`. `BACKUP` is a copy of `DATHANNA` if the original is accidentally altered. The $\star$ before the names indicate that the files are locked. To change them, you need to `UNLOCK` them first. Files marked as `A` are Applesoft BASIC programs. Those marked with a `B` are binary files, either Dathanna source code or, for `XORSHIFT`, machine language routines. `XORSHIFT` provides a better pseudorandom number generator than the one built into Applesoft. It's also faster. The program `HELLO` is automatically run when the Apple II boots, or, via the emulator, the disk image is loaded into drive 1, and the Apple icon, using the original Apple colors, is clicked.

Most emulators allow control of the speed at which they run. The idea here is to run the emulator as fast as possible. For Applewin, this is accomplished by clicking the joystick/speaker icon and, at the bottom, moving the "Emulation Speed Control" slider to "Fastest." When running Dathanna

programs, go as fast as possible. If you want to modify the language or editor, I recommend using the authentic machine speed as BASIC listings will fly by too quickly otherwise.

3 The Dathanna Language

A typical Dathanna program looks like this,

which is why the language is named "colors."

Instructions are pairs of colored blocks stacked vertically, beginning with column 0 on the left. Therefore, in the example above, the first instruction is a magenta block over a brown block.

The top block of an instruction is the primary command. The bottom block is an argument to the command. Magenta is `PUSH` with the bottom color specifying the value to push on the stack. As there are sixteen low-resolution colors, there are sixteen possible values that may be pushed on the stack,

As with Befunge, to get larger values on the stack, multiple operations are required. Note, the stack holds floating-point values, not integers.

The first instruction above now makes sense: push 8 on the stack. The second pushes 9. The third, orange and blue, implements a different operation. Orange is `MATH` with the argument, blue, specifying multiplication. So, the first three instructions above end with $8 \times 9 = 72$ on the stack.

Instructions are evaluated left to right until the end of row 0. The instruction pointer then moves back to column 0 and graphics row 2, Dathanna's row 1, to execute the orange/black instruction (orange: `MATH`, black: `+`). To end a program, use black as the top color.

In Dathanna, colors specify three different things: commands, arguments/modifiers, and numbers. The editor described below shows all three values to simplify entering programs. It's unnecessary to memorize dozens of color combinations, though even casual use will make it easy to read many commands.

Dathanna uses a stack with a maximum of 16 floating-point values. Dathanna also sports 16 memory locations accessible as individual variables (aqua: `PUTV`, white: `GETV`) or as a 16-element vector (light green: `STORE`, yellow: `FETCH`). At present, Dathanna has no access to the underlying Apple hardware (e.g., graphics, sound, disk, etc.)

Figure 1 lists all Dathanna commands. The *Color* and *Top* columns identify the command. The *Bottom* column describes the lower color and what it does. *Any* means any color. In those cases, the color is either ignored (e.g., `HALT`) or represents a number (leftmost column of Figure 1). TOS is short for "top of stack," meaning the top stack value. For `STORE` and `FETCH`, Forth-style stack effect comments are used to indicate the value (`v`) and the index into memory (`idx`). The first five math operations, and `mod`, expect two values on the stack. The others accept only one value.

Dathanna has two branch commands, but under `COMPARE`. The first is unconditional; it changes the program counter to the row and column on the stack. The second is conditional and uses the top stack value as a flag. If the flag is zero, the instruction does not branch; otherwise, it does. The remaining `COMPARE` instructions compare the top two stack items and leave a flag on the stack.

A Dathanna idiom is to perform a comparison, then push the destination row and column and finally, use `ROT` to get the flag value in the proper place before executing `IF`.

Running a Dathanna program is simple. First, run `DATHANNA` then enter the name of the program at the `?` prompt,

```
] RUN DATHANNA
? WORLD
```

Dathanna clears the screen and runs the code,

```
HELLO, WORLD!
```

4 Using the Editor

Dathanna programs are 1024 byte blocks of the Apple II's memory, the portion shown when low-resolution graphics are enabled. To create a program, use the editor,

```
] RUN EDITOR
```

which presents a blank editor screen showing the playfield and a cursor initially at the top left,

	Color	*Top*	*Bottom*
0	black	`HALT`	terminate program (*any*)
1	magenta	`PUSH`	number to TOS (*any*)
2	dark blue	`INPUT`	number to TOS (orange)
			character to TOS (light green)
3	purple	`COMPARE`	< (gray 1)
			<= (medium blue)
			> (light blue)
			>= (brown)
			= (orange)
			<> (gray 2)
			`( r c -- )` branch (purple)
			`( r c f -- )` if (aquamarine)
4	gray 1	`STACK`	dup (magenta)
			drop (dark blue)
			swap (purple)
			rot (gray 1)
			over (medium blue)
			nip (light blue)
7	light blue	`PRINT`	TOS as number (orange)
			TOS as character (light green)
9	orange	`MATH`	+ (black)
			− (magenta)
			* (dark blue)
			/ (purple)
			^ (dark green)
			sqrt (gray 1)
			sin (medium blue)
			cos (light blue)
			tan (brown)
			abs (orange)
			exp (gray 2)
			log (pink)
			int (light green)
			rnd (yellow)
			atn (aquamarine)
			mod (white)
12	light green	`STORE`	`( v idx -- )` (*any*)
13	yellow	`FETCH`	`( idx -- v)` (*any*)
14	aquamarine	`PUTV`	TOS to var number (*any*)
15	white	`GETV`	var number to TOS (*any*)

Figure 1: Dathanna commands

which is shown at the bottom of the screen as row 0, column 0. Note, the row is Dathanna's row, not the graphics row; therefore, it only updates every two graphics rows.

Use `?` to see a quick command summary,

Use I, J, K, and M to move the cursor. When the cursor is in position, use the spacebar to mark that location the current cursor color. Change the cursor color with A and Z. As you change the color, notice the text at the bottom of the screen change along with the color. For example, when the cursor is purple, the second text line reads,

meaning the cursor's numeric value is 3, command value is `COMPARE`, and argument value is `SWAP`, `/`, or `BRANCH`. If the top color is purple and the bottom is purple, the command is `BRANCH`, an unconditional branch the row and column on the stack. However, if the top color is gray 1 and the bottom color is purple, the command is `SWAP`. Likewise, if the top color is orange, then a purple bottom color is division, all from Figure 1.

The editor is best used by setting first the top color of a command using the list on the left side of the colon, followed by setting the bottom color using the right side of the colon.

For example, to push 13 on the stack and print it as a character, which is how the Apple II outputs a newline, you need two commands. In the editor, do the following,

- A or Z to get the cursor to magenta for `PUSH`.
- Spacebar to mark the cursor position as magenta.
- M to move down to the bottom part of the instruction.
- A until the `VAL` is 13. Then spacebar to mark the cursor position.
- K to move right, I to move up to the top of the next instruction.

- Z to change the cursor color to light blue for `PRINT`. Spacebar.
- M to the bottom of the instruction and A to light green to output the TOS as a character.

Use `S` to save a program to disk. This literally stores the low-resolution graphics memory as a binary file. To load a program, use `L`. To erase from the cursor position to the left of a row, use `E`. Finally, to exit, press the escape key.

5 Some Examples

Dathanna includes several example programs. To see the files, use `CATALOG` at the `]` prompt followed by `GR` and `BLOAD` filename, or use the editor.

5.1 WORLD

Print `HELLO, WORLD!`. No loops, just stack manipulations to get the proper ASCII value on the stack. First is `H`, 72, which is duplicated, and subsequent ASCII values found by offsets from the previous value.

5.2 RANDOM

Asks the user for a number, n, and prints a random integer from $[1, n]$ to simulate an n-sided die. This is a good one to start with. Load it into the editor and move the cursor below each command, changing the color with A and Z to match the top then bottom to help you understand how the commands are arranged.

5.3 COUNTER

Counts forever printing the number, one per line. Use this as an example of an unconditional branch, the purple/purple command at the end of the row. Notice the magenta/black and magenta/magenta commands before purple/purple to push 0 then 1 on the stack. The branch is to row 0, column 1 to continue the loop. The first instruction is magenta/black to push 0 on the stack.

5.4 LOOP

Similar to `COUNTER` but uses a conditional loop to stop after ten numbers. Note the trick of using `ROT` to move the flag to the proper place on the stack for `IF` at the end of the row.

5.5 ALPHA

Print the letters of the alphabet followed by a newline (ASCII 13). Walkthrough this one in the editor to see what each of the three blocks of code is doing.

5.6 PI

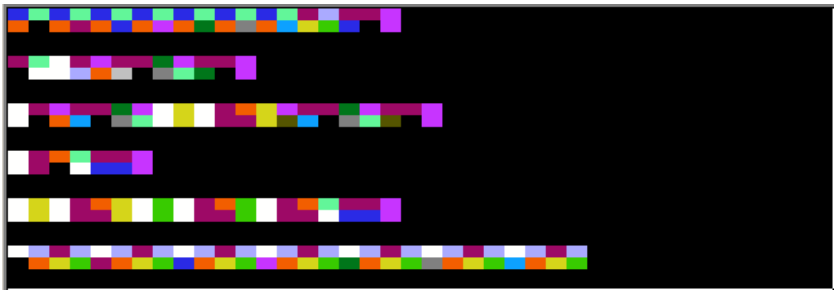
Estimate π with random numbers. The code implements a loop picking two values in $[0, 1)$ and asking if the sum of their squares is less than 1.0 or not. If so, count it. As the loop runs, four times the ratio between the number of counted points and the number of points sampled will approach π. An Applesoft version is in `XORPI`.

Both programs use the 32-bit xorshift pseudorandom number generator in `XORSHIFT` which is loaded at memory location `$5F00`. Dathanna loads source code beginning at memory location `$6000`.

A C example using 32-bit xorshift to estimate π is only accurate to 3 or 4 digits after 1 billion samples, so expect about 2 digits at best from the programs here.

The 32-bit xorshift routine generates an unsigned 32-bit integer. The Apple II's floating-point format cannot handle that at full precision, so Dathanna uses only the first 24-bits of the value returned to generate a float in the range $[0, 1)$.

5.7 GNOME



This program asks the user for seven floating-point numbers then uses gnome sort (also called stupid sort) to sort the numbers and print them in numerical order.

Gnome sort is a simplest $O(n^2)$ sort and needs only a single loop. In Python, gnome sort is,

```
def stupid(A):
    p = 0
    while p < len(A):
        if (p == 0) or (A[p] >= A[p-1]):
            p += 1
        else:
            t = A[p]
            A[p] = A[p-1]
            A[p-1] = t
```

```
p -= 1
```

The Dathanna version is implemented as,

ROW 0 Input seven numbers to the array (A), 0..6.

ROW 2 P=0; If P==7, goto ROW 10, else goto ROW 4.

ROW 4 If P==0, goto ROW 6, if A[P] $\geq$ A[P-1], goto ROW 6, else goto ROW 8.

ROW 6 P=P+1, goto ROW 0, check for P==7.

ROW 8 Swap A[P] and A[P-1]; P=P-1, goto ROW 0, check for P==7.

ROW 10 Print the sorted list and exit.