PDFTest1

Overview

By Aaron Janse

AsciiDots is an esoteric programming language based on ascii art. In this language, "dots", represented by periods (`.`), travel down ascii art paths and undergo operations.

Samples

Hello world:

```
 .-$"Hello, World!"
```

Quine:

```
 ($'.-#40-$_a#-#35-$_a#-#39-$_a#)
```

Counter:

```
     /1#-.
     |
   /-+-$#\
   | |   |
  [+]<1#-*
   |     |
   \--<--/
      |
      0
      #
      |
      .
```

Semi-compact factorial calculator:

```
 /---------*--~-$#-&
 | /--;---\| [!]-\
 | *------+--*#1/
 | | /1#\ ||
[*]*{-}-*~<+*?#-.
 *-------+-</
 \-#0----/
```

Code-golfed counter (15 bytes):

```
/.*$#-\
\{+}1#/
```

Implementation

The original implementation is written in Python and is [https://github.com/aaronduino/asciidots on Github] It can be tried out online at [https://asciidots.herokuapp.com asciidots.herokuapp.com]

Program Syntax

Basics

Starting a program

`.` (a period), or • (a bullet symbol), signifies the starting location of a "dot", the name for this language's information-carrying unit. Each dot is initialized with both an address and value of `0`.

Ending a program

Interpretation of a dots program ends when a dot passes over an `&`. It also ends when all dots die (i.e. they all pass over the end of a path into nothingness)

Comments

Everything after `` ` ` `` (two back ticks) is a comment and is ignored by the interpreter

Paths

`|` (vertical pipe symbol) is a vertical path that dots travel along
− is a horizontal path that dots travel along

"Note": Only one path should be adjacent to a starting dot location, so that there is no question where it should go

Here's an example program that just starts then ends (note that programs aren't always written and run top-to-bottom):

```
. `` This is where the program starts
| `` The dot travels downwards
| `` Keep on going!
& `` The program ends
```

Think as these two paths as mirrors:
/
\

So... here's a more complex program demonstrating the use of paths (it still just starts then ends):

```
/-&          `` This is where the program ends!
|
\-\ /-\
  | | |
```

```
/-/ | \-\
\---/   |
        |
        \-. `` Here's where the program starts
```

==== Special Paths ====

+ is the crossing of paths (they do not interact)

> acts like a regular, 2-way, horizontal, path, except dots can be inserted into the path from the bottom or the top. Those dots will go to the right
< does likewise except new dots go to the left
^ (caret) does this but upwards
v (the lowercase letter 'v') does likewise but downwards

Here's a way to bounce a dot backwards along its original path using these symbols:

```
/->-- `` Input/output comes through here
| |
\-/
```

But there is an easier way to do that:

(reflects a dot backwards along its original path. It accepts dot coming from the left, and lets them pass through to the right
) does likewise but for the opposite direction

* duplicates a dot and distributes copies including the original dot to all attached paths except the origin of dot

Here's a fun example of using these special paths. Don't worry—we'll soon be able to do more than just start then end a program.

```
  /-\ /-& `` End
  | | |
  \-+-v
    | | /-\
(-<-/ | | |
  |   \-<-/
  \-\
    |
    .    `` Start
```

Addresses and Values

@ sets the address to the value after it following the direction of the line
# does the same except it sets the value

Interactive Console

$ is the output console. If there are single/double quotation marks (' or "), it outputs the text after it until there are closing quotation marks. # and @ are substituted with the dot's value and address, respectively

When _ follows a `$`, the program does not end printing with a [https://en.wikipedia.org/wiki/Newline newline].

When not in quotes, if a `a` comes before a `#` or `@` symbol, the value is converted to ascii before it is printed

Here's how to set and then print a dot's value:

```
. `` This dot is the data carrier
| `` Travel along these vertical paths
# `` Set the value...
3 ``   ... to 3
| `` Continue down the path
$ `` Output to the console...
# ``   ... the dot's value
```

Here's our hello world again:

```
.-$"Hello, World!"
```

Here's how to print that character 'h' without a newline:

```
.-$_"h"
```

And this prints '%' using the ascii code 37:

```
.-#37-$a#
```

`?` is input from the console. It prompts the user for a value, and pauses until a value is entered in. It only runs after a `#` or `@` symbol

```
. `` Start
|
# `` Get ready to set the value
? `` Prompt the user
|
$
# `` Print that value to the console
  `` Since the only dot goes off the end of the path, it dies. Since no
```

Control Flow

~ (tilde) redirects dots going through it horizontally to the upward path if a dot waiting at the bottom has a value "not" equal to than `0`. Otherwise, the dot continues horizontally. If an exclamation point (`!`) is under it, then it redirects the dot upwards only if the value of the dot waiting "is" equal to zero.

`!` acts like a pipe. Special function described above

This example prompts for a value then prints to the console whether the user provided value is equal to zero:

```
  /-$"The value is not equal to zero"
  |
.-~-$"The value is equal to zero"
  |
```

```
?
#
|
.
```

Operations

`[]` multiplies the value that passes through vertically by the value that runs into it horizontally. When a dot arrive here, it waits for another dot to arrive from a perpendicular direction. When that dot arrives, the dot that arrived from the top or bottom has its value updated and it continues through the opposite side. The dot that passed through horizontally is deleted.
`{}` does likewise except it multiplies the value that enters horizontally by the value that enters vertically. The resulting dot exits horizontally

Other operations work similarly but with a different symbol in the middle. This is the key to these symbols:
`*`: multiplication
`/`: division
`÷`: also division
`+`: addition
`−`: subtraction
`%`: modulus
`^`: exponent
`&`: boolean AND
`!`: boolean NOT
`o`: boolean OR
`x`: boolean XOR
`>`: greater than
`≥`: greater than or equal to
`<`: less than
`≤`: less than or equal to
`=`: equal to
`≠`: not equal to

Boolean operations return a dot with a value of `1` if the expression evaluates to true and `0` if false.

These characters are only considered operators when located within brackets. When outside of brackets, symbols like `*` perform their regular functions as described earlier.

Example:

```
`` Simple subtraction:
``   (3 - 2 = 1)

   #
   $
   |
  [-]-2#-.
   |
   3
   #
   |
   .
```

Add two user inputted values together then output the sum:

```
.-#?-{+}-$#
       |
.-#?--/
```

Warps

A warp is a character that teleports, or 'warps', a dot to the other occurrence of the same letter in the program.

Define warps at the beginning of the file by listing them after a `%$`. The `%$` must be at the beginning of the line.

Example:

```
%$A

.-#9-A `` Create a dot, set its value to 9, then warp it

A-$#   `` Print the dot's value (9)
```

Here's a fun example of using warps (although it is not very useful in this case)

```
%$A

#  /-)
$  |
\>-A
 \-3#-.

A-\
\-/
```

Libraries

Dots supports libraries! A library is a program that defines a character (usually a letter).

Using Libraries

A library can be imported by starting a line with `%!`, followed with the file name, followed with a single space and then the character that the library defines.

By default, all copies of the character lead to the same ([https://en.wikipedia.org/wiki/Singleton_pattern singleton]) library code. This can cause some unexpected behavior if the library returns an old dot, since that old dot will come out of the char that "it" came from.

Here's an example of importing the standard `for_in_range` library (located in the `libs` folder) as the character `f`:

```
%!for_in_range.dots f
```

The way to use a library varies. Inputs and outputs of the library are through the alias character.

For the `for_in_range` library, the inputs are defined as follows: The dot coming from the "'left"' side sets the starting value of the counter. The dot coming from the "'right"' side sets the end value of the counter.

And the outputs are as follows: A dot for each number within the range defined by the inputs is output from the "'top"'. When the loop is complete (the end value has been reached), a dot is output from the "'bottom"'.

Here is an example of outputting all the numbers between `1` and `100` to the console, then stopping the program:

```
%!for_in_range.dots f

          #
          $
          |
.-*-#1---f-\
  \-#100-+-/
          |
          &
```

Creating Libraries

Each library defines a character that will act as a warp to & from the library.

That can be done like so:

```
%^X `` X could be replaced with a different character, if so desired
```

It is recommended that you create warps for different sides of the char. Just look at the example code for the `val_to_addr.dots` library:

Here's the code for a library that accepts a dot coming from the left, sets its value to its address, and then outputs it to the right:

```
%^X
%$AB

B-X-A


A-*----@{+}-#0-B
  |       |
  \------/
```

Interpretation

Each tick, the dots will travel along the lines until they hit a character that acts as a function of multiple dots (i.e. an operation character or a ~ character). The dot will stop if it goes on a path that it has already traversed in the same tick

Due to the fact that dots may be moving backwards down a line, if a number or system value (e.g. `?`) is seen without a preceding `@` or `#`, it will be ignored, along with any `@` or `#` immediately thereafter

More Examples

Hello, World!

```
.-$"Hello, World!"
```

Counter:

```
     /1#-.
     |
   /-+-$#\
   | |   |
  [+]<1#-*
   |     |
   \--<--/
      |
      0
      #
      |
      .
```

Find prime numbers:

```
%$T
```

```
         .
         |
         #
         3
         |
         @
         1
         |
/--*--*-<--\
|  |  |   /+----\
|  #  |   v+-0@-~-\
|  2  | /->~*{%}/ |
|  |  | 1  |\-+---/
|  |  | @  ^\ |
\-{+}-+-*  01 |
       | |  ## |
       | v--*+-/
       | |  ||
     /-* |  *+--\
     | T |  ||  |
```

```
    # $ # /~/  |
    0 # 1 */   |
    | | | |    |
    \->-+-~----<-#$-2#-.
        \-/


 /--------\
 T        |
 *--------~
 |        |
 \-*----@[=]
   |      |
   \--#1--/
```

Print the Fibonacci Sequence:

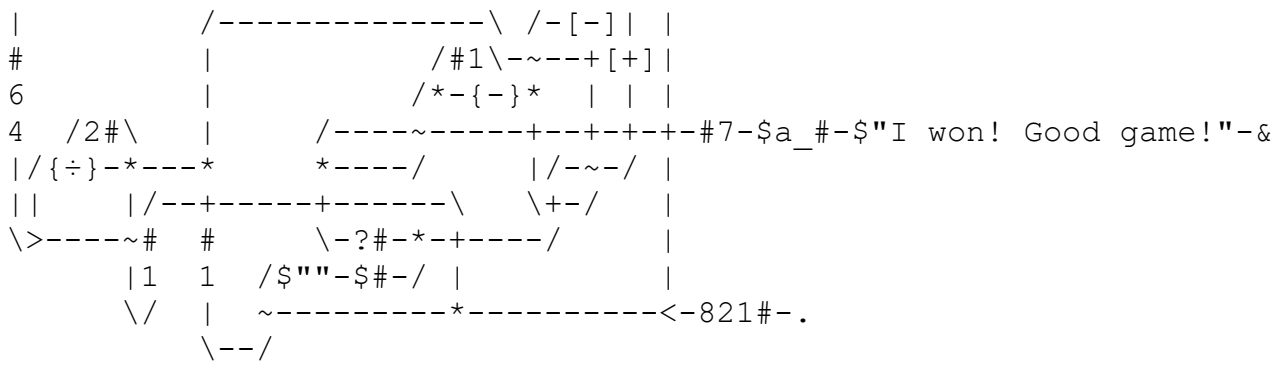
```
.-#1\
. /->\
>[+] |
\-*#$/
```

Print powers of 2:

```
 .-#2\
 /---v
[]2#
 -#$/
```

And a game!

```
/-""$-.
|
\--$"Pick a number between 1 and 255 (inclusive)"\
/-----------------------------------------------/
\--$"I will correctly guess that number after no more than 8 tries"\
/-----------------------------------------------------------""$/
\--$"After each of my guesses, respond with: "\
/---------------------------------------------/
\--$"     '2' if I guess too high,"\
/--------------------------------/
\--$"     '1' if I guess too low,"\
/-------------------------------/
\--$"  or '0' if I guess correctly"\
/--------------------------------/
|
|                              /->-\
```



External Resources

[https://github.com/aaronduino/asciidots Main Github repo]

It got a writeup on Motherboard! [https://motherboard.vice.com/en_us/article/a33dvb/asciidots-is-the-coolest-looking-programming-language]

Category:Languages Category:2017 Category:Turing_complete Category:Two-dimensional_languages