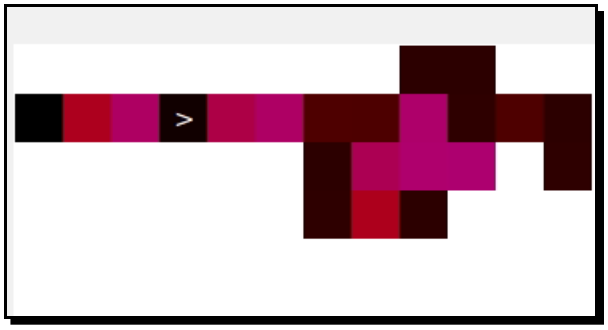
ENTRY // PixelLang Esoteric Language #1 // More Instructions [−] [×]

Date: 2019-09-06 **Author:** Ian Rash **Index:** #1

We covered Start, End, and Output Char in the last section, next we want to use Direction and Jump. Let's create a version of Hello World that does some crafty maneuvering to only use a single output char instruction for each character of "Hello World!". First we need to make the instructions we need to use.

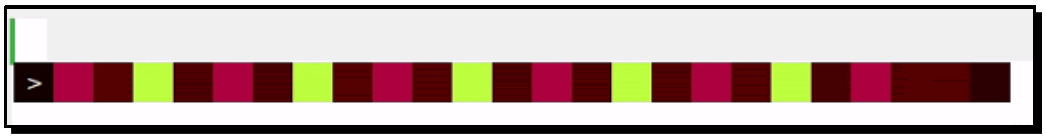
```
require "./src/pixel_lang_crystal"

Start.make_color(:right, 0)                  "140000" : String
# One of each char
OutputChar.make_color('H')                   "b00048" : String
OutputChar.make_color('e')                   "b00065" : String
OutputChar.make_color('l')                   "b0006c" : String
OutputChar.make_color('o')                   "b0006f" : String
OutputChar.make_color(' ')                   "b00020" : String
OutputChar.make_color('W')                   "b00057" : String
OutputChar.make_color('r')                   "b00072" : String
OutputChar.make_color('d')                   "b00064" : String
OutputChar.make_color('!')                   "b00021" : String
# Jumps to help us move around
Jump.make_color(0)                           "500000" : String
Jump.make_color(4)                           "500004" : String
# Directions
Direction.make_color(:up)                    "300000" : String
Direction.make_color(:down)                  "300002" : String
Direction.make_color(:left)                  "300003" : String
Direction.make_color(:right)                 "300001" : String
Direction.make_color(:turn_left)             "300004" : String
Direction.make_color(:turn_right)            "300005" : String
Direction.make_color(:reverse)               "300006" : String

End.make_color                               "0" : String
```

Then we just need to come up with an interesting layout. I had to specifically be aware of the OutputChar L, because it's used three times, and I need to go a different route each time.



I often times use Jump to allow a single directional line of code have multiple meanings.

For example, take this program, which only hits the pink spaces one way, and the yellow spaces other. This programs runs indefinitely.



Insert

Insert is instruction number 0x8, and it's 20bit argument is stacked onto the I stack of any Piston that executes it. It's used to introduce constants into your program.

Move

Move is an instruction which takes two register arguments, and moves the value from the source to the destination. When specifying a register, you also need to specify a register option which can change how the register is interacted with.

For example, a MOV(MA(0) -> MB(0)) will move the value in MA into MB, where MOV(MA(1) -> MB(0)) will move a random value where MA is the max of that number, and move it into MB.

Another good example would be MOV(I(0) -> MA(0)) which pops a value off I and puts it into MA, and MOV(I(2) -> MA(0)) which only peeks the value.

Register options can be very useful!

Let's give Move a try.

In the program above, 100, 200, and then 300 are placed on the I stack using Insert, then moved to output using MOV(I(0) -> O(0)), and a space is added between. We can see when the I stack runs out, it always returns 0 if there is no engine input to consume.

MOV can also be used with register options to change register behaviour. For example, if MA is equal to 1234 and MOV(MA(1) -> O(0)) is called, output will be a random number between 0 and 1234. Reading from I(2) doesn't pull the item off the I stack, while I(0) does.

Move also has two other options, swap and reverse. Swap swaps the values of two registers. The order it does that is very important, since register reads can trigger changes (like I or O for example). When swapping values, the source value is gotten first, then the destination value, then the source is set first, then the destination is set. This is important to understand especially when using the instruction MOV(I(0) -> I(0)), which will swap the top two values on the I stack.

You can also reverse the source and destination, this option isn't particularly useful, it's just to allow for more interesting colors to be made.

Arithmetic

One of my favorite instructions, Arithmetic allows you to build simple mathematical expressions. Arithmetic takes two source registers, an operation, and a destination register. Here are some examples.

```
Arithmetic.make_color(:ma, 0, :+, :mb, 0, :mav, 0)      "a00208" : String
Arithmetic.make_color(:o, 2, :/, :i, 2, :mbv, 0)        "af1e98" : String
Arithmetic.make_color(:mb, 0, :*, :mbv, 0, :o, 0)       "a41338" : String
Arithmetic.make_color(:mbv, 0, :-, :ma, 0, :i, 0)       "a60830" : String
Arithmetic.make_color(:i, 0, :%, :o, 0, :mbv, 0)        "ac4718" : String
```

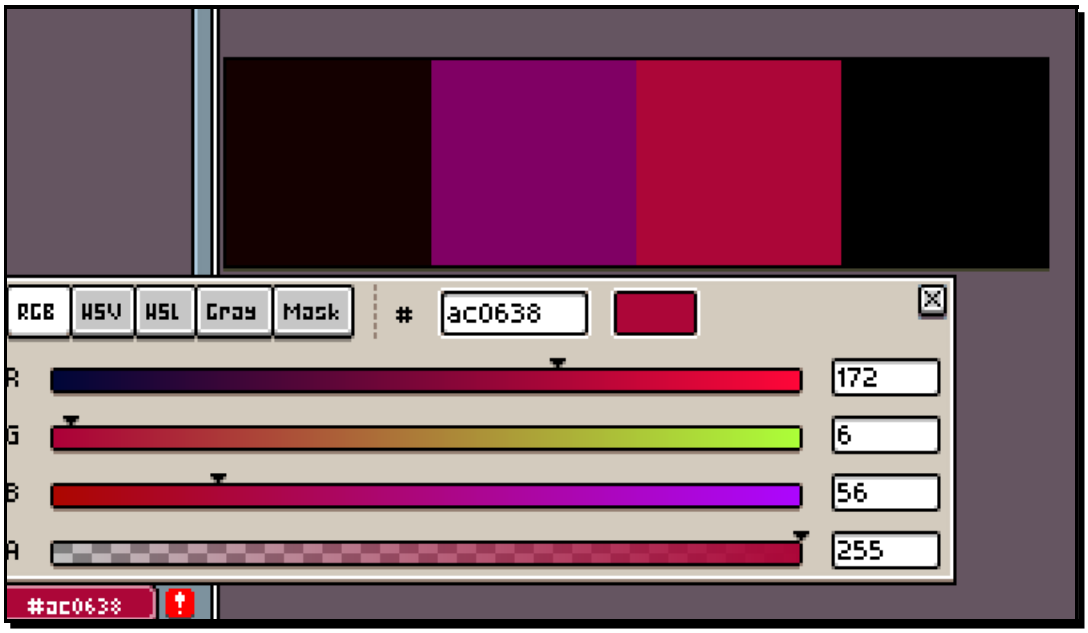
Here's a list of all the mathematical operators available.

BOOLEAN_OPERATIONS = [:<, :>, :<=, :>=, :==, :!=]
ARITHMETIC_OPERATIONS = [:+, :-, :*, :/, :**, :&, :|, :^, :%]

Using this instruction we can easily make a program to add 100 to an input number.

The program itself only needs 4 instructions. Start, Insert(100), AR(I(0) + I(0) -> O(0)), and End.

When running the program, you need to include an input number into the Engine before starting or it will always display 100, since I always deafults to 0 when there is no input.



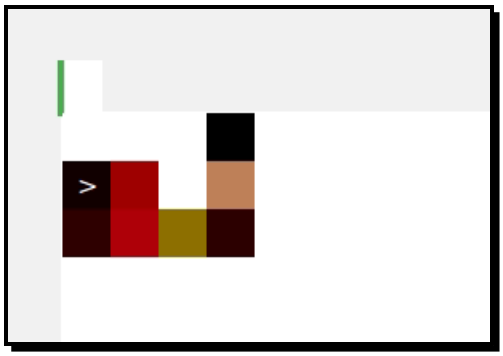
Conditional

Conditional allows pistons to make decisions on where they will go, based on a mathematical expression. You choose two directions, a true direction, and a false direction, and if the mathematical expression evaluates to 0 it goes the false direction, otherwise the true direction.

Boolean operators always produce either a 0 (false), or a 1 (true).

Conditional lets us create decisions and loops that will be the basic building blocks of our programs.

To show off what the Conditional can do, we are going to make a "count to" program which will take a number, and count up to that number.



In this example, the value of MB is always 1. We use that to increment MA each loop, and use the beige instruction (a Conditional) to determine if we have hit our max number yet. If not, we output the number, output a space, and start over again. Interesting note, Start instructions operate as Direction instructions when executed by a Piston. This allows us to restart programs easily if necessary.

Call/Return

These two instructions work in tandem to allow a Piston to return to a previous state, and choose what it takes along with it. Call and Return can be some of the most powerful instructions if used right.

Call takes a couple arguments, the first being an action, which is either :none, :push, :none_run, :push_run. This determines behavior of the Piston after running the Call, if it should push it's current frame to stack, or not, or if it should step once after the Call. For example, the none option does not push a frame to the call stack, where push does.

Call also takes a signed X and Y argument.

Call moves the Piston X and Y spaces away from the Call instructions. This is all relative spacing, there is no absolute values for Call.

When a Return instruction is read by a Piston, it checks to see if there is a frame on it's call stack. If there is a frame, the Return instruction chooses what values to copy back to the piston, for example, you can choose to restore the X position but not the Y, the direction, you can choose to keep MA the same, or restore it from the frame, that sort of thing. If there is no frame on the call stack, Return does nothing.

Return has a lot of arguments, a full list from the color_helper dev module:

Return Instruction
Returns a frame from the call stack.
0bCCCC00000000PPABSIIMMXYD
C = Control Code (Instruction) [4 bits]
P = Action bits [:pop, :peek, :pop_push, :peek_push]
A = Copy MA?
B = Copy MB?
S = Copy S?
I = Copy I action? [:keep, :restore, :clear]
M = Copy memory action? [:keep, :restore, :clear]
X = Jump back to X?
Y = Jump back to Y?
D = Change the direction?

First, the action specifies whether or not a frame should peeked, or popped off the call stack and/or if the current frame of the Piston should be added back to the call stack.

Next, should we copy MA, MB, S, X, Y, etc?

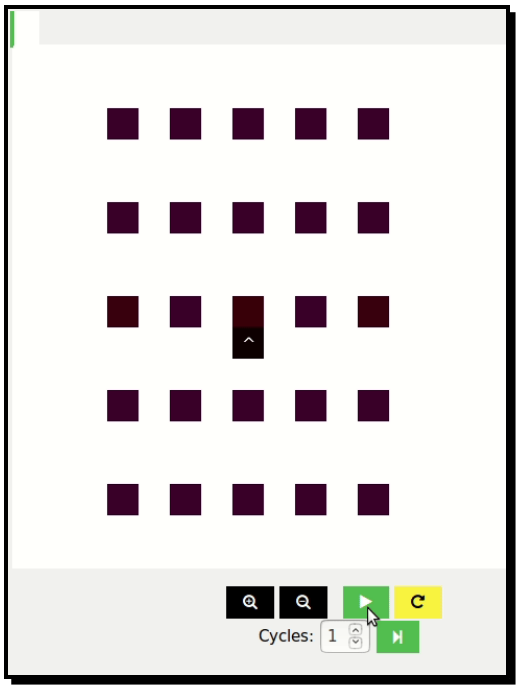
Lastly, I and Memory (since they are collections) both have special operations. Whether or not we should keep them, restore the frame's version, or clear altogether. When using :clear, even if there is nothing on the call stack, that item will still be cleared.

Again Return is super powerful, with it, we can set up all sorts of interesting interactions with the program code.

Fork

Fork is used to make exact duplicates of Pistons, facing in different (or the same) directions. One Fork instruction can make up to 4 new pistons. The original piston always follows direction #1, and each piston created afterwards executes after the last one created. Imagine two pistons one the same space that hit a fork instruction. Let's call them P1 and P2, after their priority numbers. P1 creates a new Piston, sandwiched between it and P2 in the execution order, while P2's created clone will be below it in the execution order.

Here is an example program I wrote to test the limits of Fork (like how many times can you fork before the program crashes).



I also use Fork in my Ackermann implementation, since it's perfect for the job of expansion!

InstructionMeta

InstructionMeta is an instruction that houses four other functions, Get, Set, Resize, and Property.

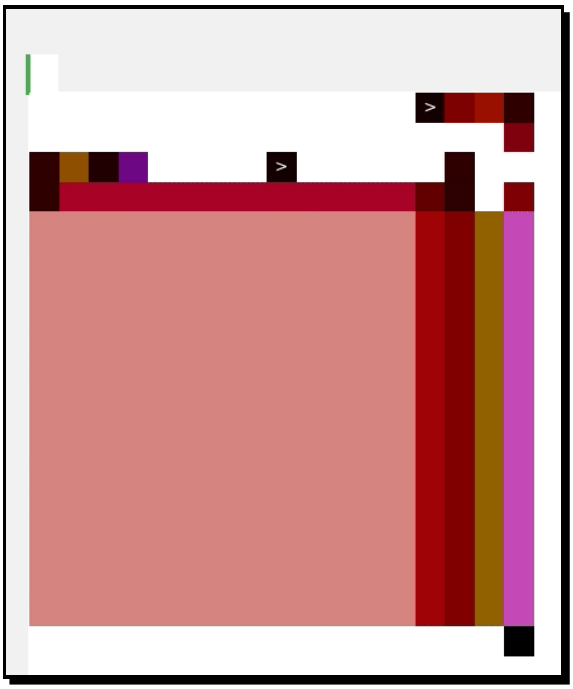
Get allows a Piston to get the value of a color located at the X and Y values specified by two registers. Puts the control code, then the control value on the I stack.

Set allows a Piston to set the value of a color located at the X and Y values specified by two registers, to the value located on the I Stack. Since the I stack can only hold values up to 0xFFFFF, the I stack is read twice and the values bitshifted and combined to for the color. For example, if 0xBBBBB then 0xA were on the I stack, the color would be 0xABBBBBB.

Resize allows a piston to resize the instructions width and height.

Property allows a Piston to get the width and height of the instruction set.

Using these instructions, we can modify the instructions on the board! Here is an example program, a painter bot.



The program above uses two pistons. One goes in a loop and counts up from 0 to the width of the "drawing canvas". The other piston reads the IMetaSet instructions, and executes them, changing the current square they are on, then moving to another. This gives the effect of a bot painting the ground behind it.

TAGGED SKILLS:

Crystal Esoteric Programming Generative Art Raylib

+--+
© 2026 Ian Rash - Built with Crystal & Made into a PDF
+--+