

Programming and Programming Languages

Alan Kay

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

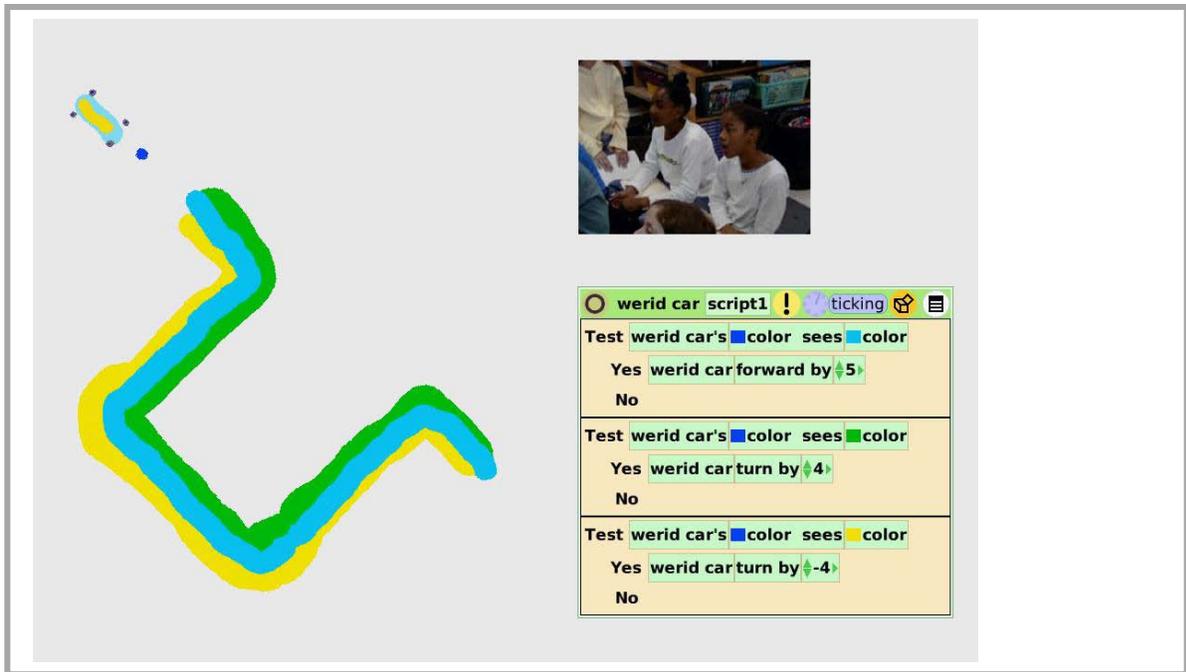
VPRI Research Note RN-2010-001



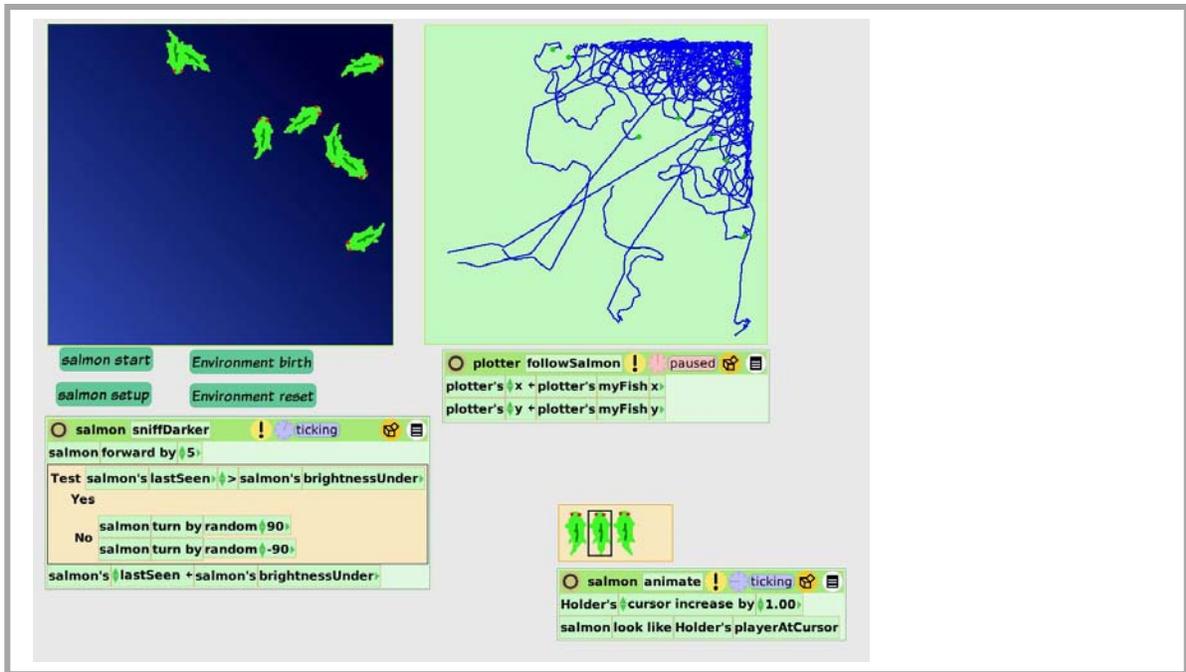
Many thousands of programming languages have been designed and implemented in many styles. Too many for 45 minutes! So let's look at a few styles starting with the Etoys language and environment made for 11 year olds. See [Etoys, Children and Learning](#) and [Etoys Authoring and Media](#) for some of the things I showed in the live demo.

The screenshot shows the Etoys programming environment. At the top left, a graph displays a blue curve rising from 0% to 100% on a y-axis labeled 25%, 50%, 75%, and 100%. Below the graph is a black square representing a village, filled with numerous small red dots. A text box below the square reads "KedamaWorld's villagers = 1000". To the right of the square are two code blocks. The first block is titled "villager move" and contains the following code: "villager forward by 1", "villager turn by random(10) - random(10)". The second block is titled "villager do" and contains a conditional code block: "Test villager's color sees color", "Yes villager's color - color", "No". At the bottom of the interface are several buttons: "Village setup", "Village go", "Village clearit", "KedamaWorld stopit", and "person back".

We can easily make thousands (millions) of objects and program them all at once. This shows that contagion can be delayed by changing the time constant (by having more or less villagers) but not eliminated. All the villagers eventually get sick and die. Being able to program “massively parallel” and easily is important. Be critical of languages that can't do this easily.



Feedback and correction is a powerful idea. These two 11 year old girls came up with this nice program to always keep their programmable car on the road. Most systems use this idea whether biological or human made.

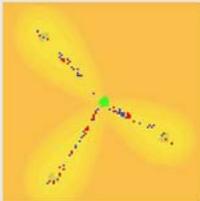


This idea can be used to follow gradients. The salmon are “looking for dark”. The basic scheme is “if things are OK, keep going, otherwise do something random and keep going”. All life uses this, and so does evolution. And so does the Ethernet, and with a few more elements, so does the Internet.

"Particles and Fields"

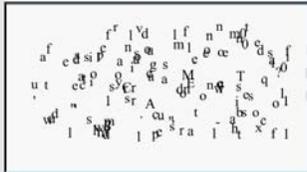


Ant World setup
foodCreator setupFood



7 Rules for Layout = ~25 LOC

<pre>wordrap layout When client contents isEmpty not When Always Do client contents first box pivotBecomes (client shape leftARY: 0) * mouseUp() + inset rule tell client contents first successor to "place". Do rule tell later: rule to: "showDelection".</pre>	<pre>wordrap place When I am() When Always Do return me. Do grid := my predecessor box. my box pivot: grid right + grid pivotOffset * @ grid pivot y. Do return rule tell my successor to "place". Do rule tell my box to "backToStart". Do rule tell my successor to "place".</pre>
<pre>wordrap placeAfterReturns When my predecessor box shape notNil and: [my predecessor box shape isNotLine] When Always Do return false.</pre>	<pre>Do "start of the next line" my pivotYIncreasedBy: my height. my pivotLeft: (client shape leftARY: my pivotPosition) + inset x. return true.</pre>
<pre>wordrap isClipped When my shape isNotSpace When Always Do return false. Do return my right + inset x < (client shape rightARY: my pivotPosition)</pre>	
<pre>wordrap backToStart When Always When self isStartOfLine: letterToMove index When Always Do letterToMove := self startOfWord: me. Do letterToMove := me. Do letterToMove pivotYIncreasedBy: letterToMove height. Do letterToMove pivotLeft: (client shape leftARY: letterToMove pivotPosition) + inset x. Do rule tell: letterToMove successor to: "place".</pre>	
<pre>wordrap startOfWord When my predecessor isNil Do return me. When my predecessor box shape isNotSpace When Always Do return me. Do return rule startOfWord: my predecessor box.</pre>	
<pre>wordrap isStartOfLine When Always Do return (client contents at: me) box left - inset x == (client shape leftARY: (client head: me) pivotPosition) "left margin"</pre>	

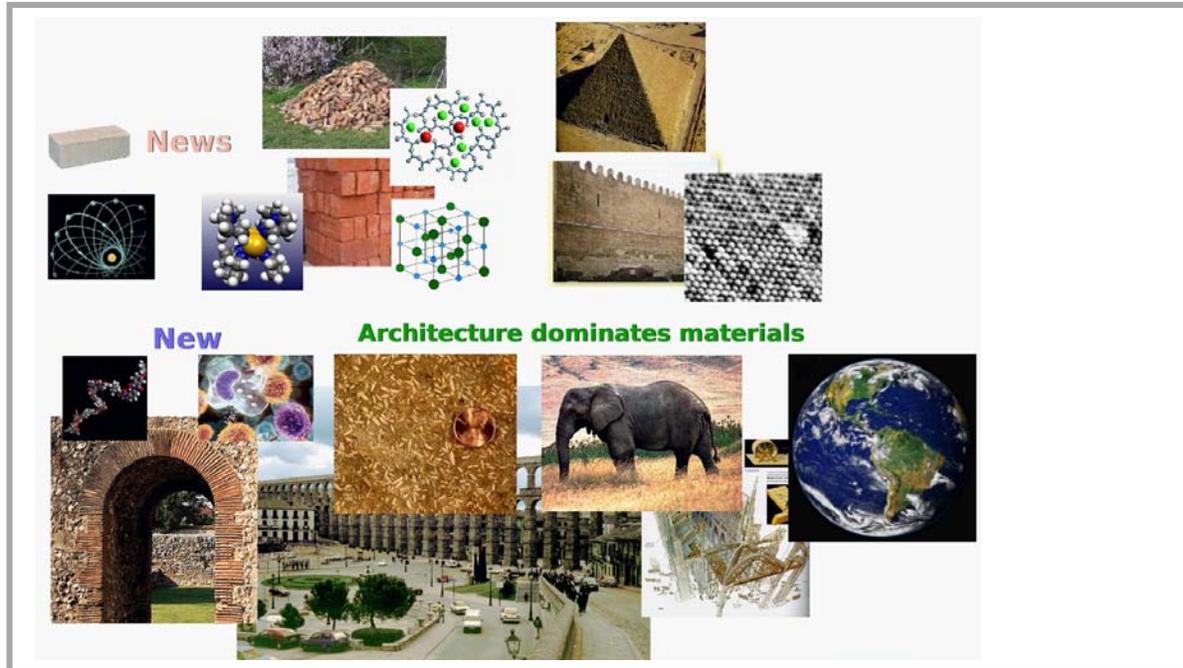


Ants are made efficient by food-carrying ants laying down a scent trail that other ants can follow upstream to quickly find the food and then go downstream to quickly find the nest. This is "loose coupling" between objects. The same idea can be used to program a text editor in just 35 lines of code. Point of view is worth 80 IQ points!

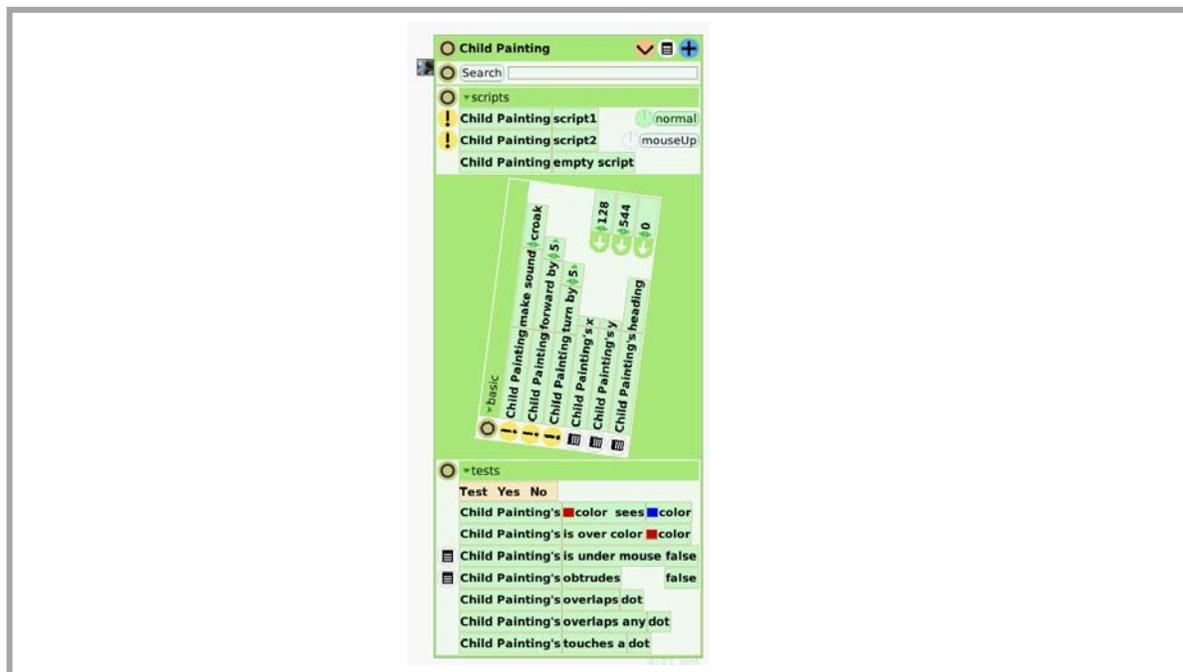
From Gears To ... ?



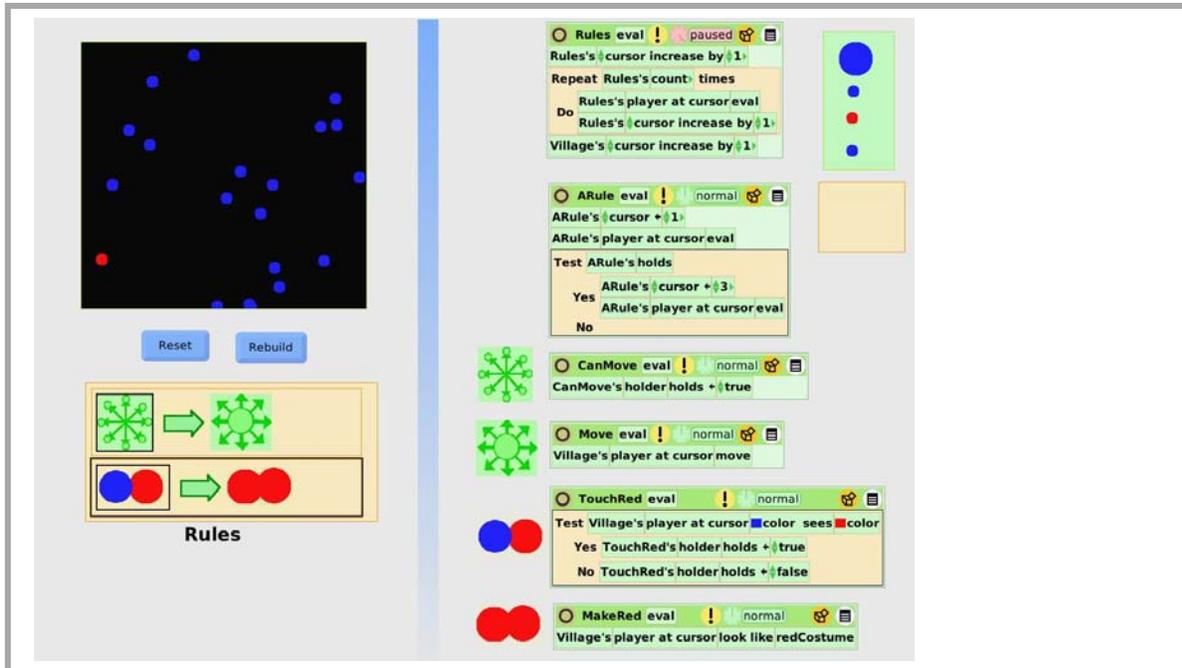
The future of programming is not like most programming today. But will be more like making biological systems. The Internet is one of the few systems organized this way. It is the most scalable and robust human artifact on the Earth.



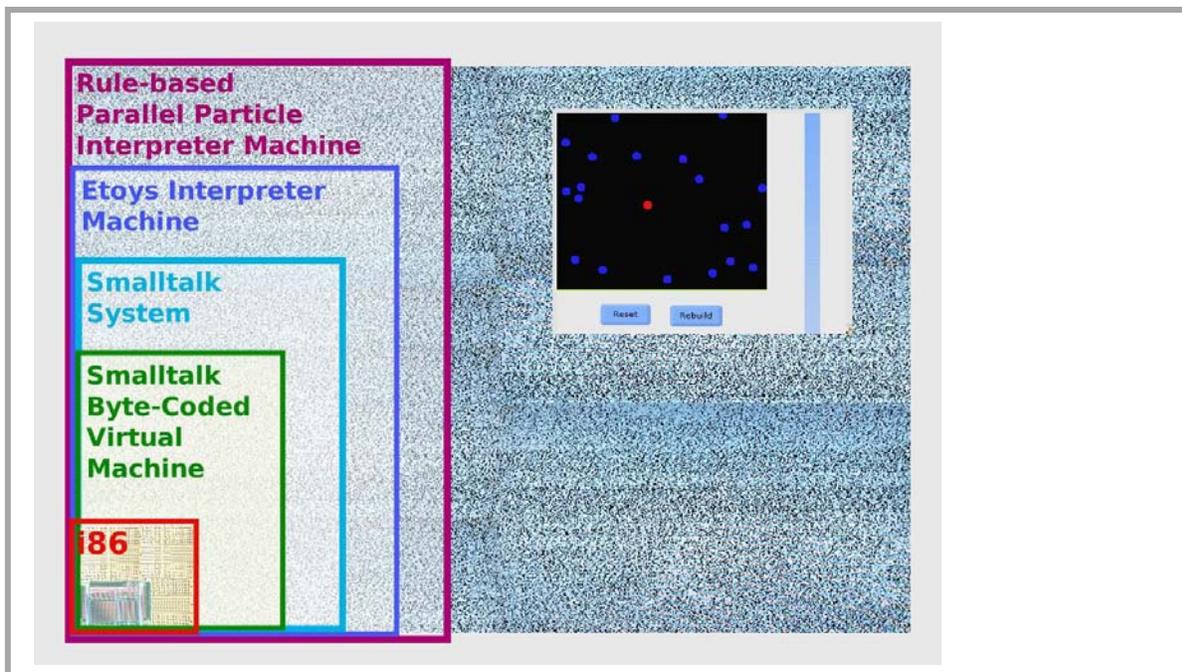
Building blocks can be organized simply – with results that are like the building block – walls and pyramids are like piles and stacks. Many molecules are like atoms. Or, completely new much more powerful structures can be built that are not like the parts – arches and life – this is what we need to try to do when we design systems and programming languages!



The children's programming environment is made from the very same simple materials as the toy cars they draw. And all is dynamic and "alive". All interactive development environments for programming could be like this. If you are working with one that isn't then you are working with a language and environment that have serious flaws!



It is easy to make a simple interpreter, even in the children's system. Here is an example of a rule-based language for controlling the particles doing the epidemic example. All the scripts needed are on the right.



Here is the situation of the previous "slide". The blue "noise" are the bits in the main memory of the computer. This presentation is actually given in the children's system, where the new interpreter we just made is built on the Etoys system which is built from Smalltalk, which runs on a virtual machine made for Smalltalk that ultimately runs on i86 machine code which is interpreted by microcode running in the CPU with simple logic.

Virginia Pham CS1:

... how can something you type be decoded into machine language, and how does the computer know what to do?

A good question from a CS1 student!

A typical CPU can do a few things pretty quickly

```
fetch, fetch[i], store, store[i]
+, -, *, /
=, <
and, or
shift
jump, if ... jump, jump-linked
```

ans := a + b * 42

```
lw r1, a
lw r2, b
lw r3, tc42
mul r2, r3, r2
add r1, r2, r1
sw r1, ans
```

1963: Val Schorre at UCLA

```
statement = destVar " := " expression
expression = product {"+" product}
product = primary {"*" primary}
primary = number
         | sourceVar
         | "(" exp ")"

destVar = identifier
sourceVar = identifier

identifier = letter { letter | digit }
number = digit { digit }

letter = "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"
        |"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z"

digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

```
def product
  if primary
  then
    repeat
      if peekSymbol("*")
      then nextSymbol;
       if primary then continue;
       else return false
    end repeat
  else return false
```

4:24 pm

Val Schorre at UCLA in 1963 noticed that the mathematical expression of a higher level programming language could itself be turned into a programming language. The circled line of metalanguage is equivalent to the procedure on the right.

A typical CPU can do a few things pretty quickly

fetch, fetch[i], store, store[i]
+, -, *, /
=, <
and, or
shift
jump, if ... jump, jump-linked

ans := a + b * 42

```

10:101110001110101000010001100
11:lw r1, a
12:
13:
14:

```

1963: Val Schorre at UCLA

```

statement :: destVar:a := expr  → out("sw r1, " · a)
expr      :: product {"+" product → out("add r" · s-1 · ", r" · s · ", r" · s-1); s := s-1}
product   :: primary {"**" primary → out("mul r" · s-1 · ", r" · s · ", r" · s-1); s := s-1}
primary   :: number
           | sourceVar
           | "(" exp ")"

destVar   :: identifier
sourceVar :: identifier:a      → s := s+1; out("lw r" · s · ", " · a)

identifier :: letter:a { (letter | digit):b → a := a · b }
number    :: digit:a { digit:b → a := a · b } → a := "tc" · a

letter    :: "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"
           |"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z"

digit     :: "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

```

This allows “grammar programming” by adding the stuff in red and writing an interpreter for it. This can be run as a program (starting in the upper left corner of the code) via mostly doing function calls until some concrete symbol in the grammar can be compared with the input. Here “a” has been matched, and the output statement has just put out the first line of the machine code

Math Wins! (especially MetaMath)

Dan Amelang **Rendering**

Mom

$$\sigma(P, Q) = (Q_y - P_y)(x + 1) - \frac{Q_x + P_x}{2}$$

$$\gamma(P) = \min(x + 1, \max(x, P_x)), \min(y + 1, \max(y, P_y))$$

$$\omega(P) = \frac{1}{m}(\gamma(P)_y - P_y) + P_x, m(\gamma(P)_x - P_x) + P_y$$

$$\text{coverage}(\overline{AB}) = \sigma(\gamma(A), \gamma(\omega(A))) + \sigma(\gamma(\omega(A)), \gamma(\omega(B))) + \sigma(\gamma(\omega(B)), \gamma(B))$$

$$\min(|\sum \text{coverage}(\overline{AB}_i)|, 1)$$

"The Formula"

Nile: "Ken Iverson meets Christopher Strachey"

```

Color < [x, r, g, b]
Point < [x, y]
Matrix < [a, b, c, d, e, f]
Bezier < [a, b, c, Point, w, h]
EdgeContribution < [p, Point, w, h]

Sampler :: Point → Color
Composer :: [Color, Color]
Canvas :: [start: Point]

m = Matrix < p : Point > P
[ka * x + b, kd * x + p, y +
 kb * x + b, kb * x + p]

CompositedSampler (s : Sampler, c : Canvas) : EdgeContribution =>
  Interleave (s, c)

MultiColor (c : Color) : Sampler
  p >> [c.a, c.a * c.r,
        c.a * c.g, c.a * c.b]

CompositedSampler : Composer
  [a, b]
  => a * b * (1 - a.a)

FillBetweenEdges (start : Point) : EdgeContribution => Real
  s = start.x
  local = 0
  run = 0
  [ (s, y), w, h]
  [m] => run = h
  a = x - s
  if a < 0
    local = local + w + h
  else
    local = run + w + h
    => | local | < 1
    min - 1 > | run | < 1
  => | local | < 1

CreatesamplePoints (start : Point) : Real → Point
  x = start.x
  y = start.y
  [ c
  x' = x + 1
  => [x, y]

Render (s : Sampler, c : Canvas) : EdgeContribution =>
  [ p, w, h]
  => FillBetweenEdges (p) -
  Interleave (CreatesamplePoints (p + 0.5) - s, (-) -
  c (p + 0.5))

Render (s : Sampler, c : Canvas) : EdgeContribution =>
  Render (s, c)
  r => Bezier
  er => Bezier
  e |
  s bin |
  s bin |

- c)
[ < 0, 1
| = 0, 1
|) (max (nearmax)? abbc)
b = c, c]
tribution
= [ c ]

w = p.x + 1 - (c.x - a.x)
h = c.y - a.y
=> [p, w, h]

else
  abbc = (a - b) - (b - c)
  min = [ abbc ]
  nearmax = [ abbc - min | < 0, 1
nearmax = [ abbc - max | < 0, 1
n = min (nearmin)? (max (nearmax)? abbc)
=> [a, a - b, b] => [b, b - c, c]

```

This seems like a lot (the current commercial examples and operating systems require 10s of millions to 100s of millions of lines of code). We ask: "How complex is this really?" Could active mathematics be invented to represent the semantic essence of "personal computing from the end-user down to the metal?" Could we produce runnable code that is many orders of magnitude smaller? A few "100s of Maxwell's Equations" T-shirts vs. 10s of millions of pages of code?

"The Formula" in Nile (~81 LOC)

A real example is drawn from a project to go from millions of lines of code to hundreds by inventing new languages. Here’s a new math for rendering in 40 lines, and the math in a programmable language called Nile in 80 lines. The language automatically deals with streams of points and pixels with mathematical transformations of coordinates and signal processing.

Take Aways About Programming Languages

**As with mathematics,
there are many powerful programming styles**

So:

**Learn 3-10 *different* kinds of
programming languages before drawing
conclusions!**

And ...

**As with mathematics,
there are many not yet invented powerful styles**

So:

**Learn how to *design and make new*
programming languages!**

You must ask questions and “be critical via knowledge” -- that is, learn a lot and try to gain perspective on what people are trying to get you to learn. A lot of it might not be a good idea any more (and some of it might never have been a good idea!)