



LAMBDA LOUNGE

Pixie

Christopher Mark Gore

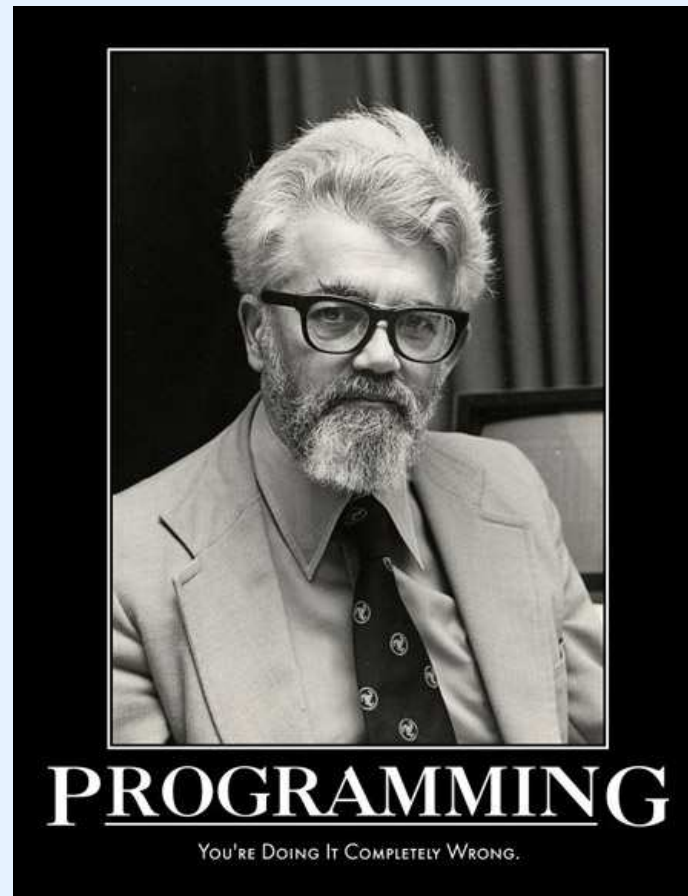
[cgore.com](http://cgore.com)

Thursday, December 3, AD 2015

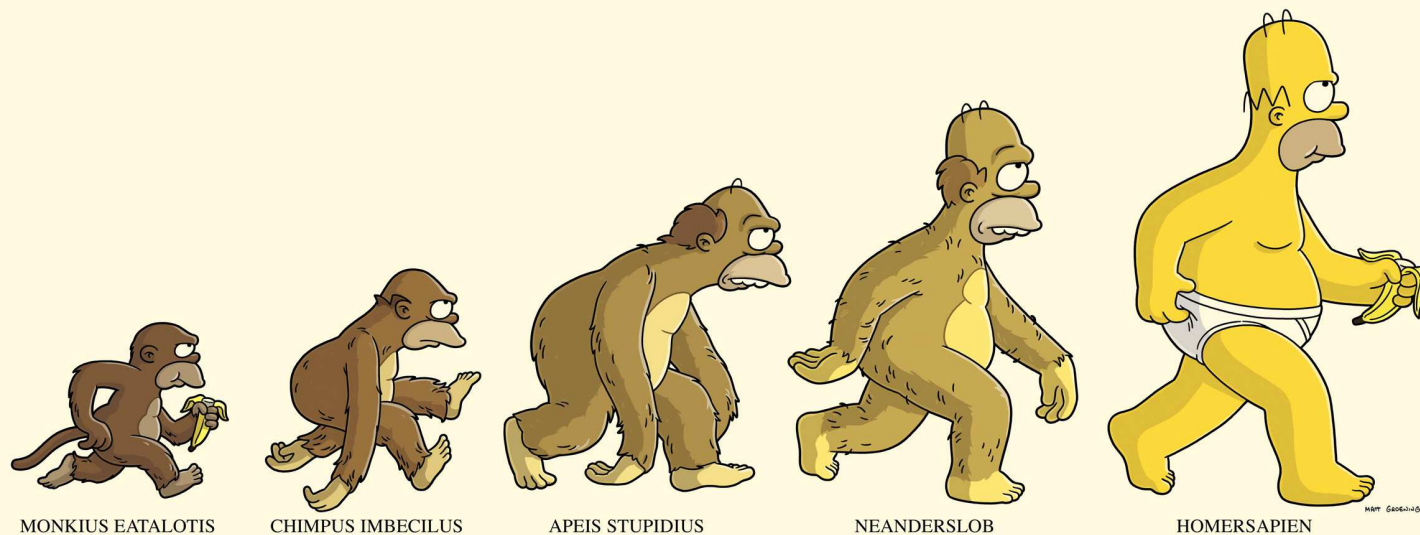
**We write Clojure at The Climate Corporation,  
and we're hiring! Come work with us!**



Some people actually program in languages  
other than Lisp.



I started using Common Lisp in 2004 for evolutionary computation as my M.S. thesis, and quickly learned to love Lisp.



HOMERSAPIEN

## I even think markup languages in web forums should be full-fledged lisps.

```
1 Welcome to the future of crapflooding!
2
3 \defun{\crapflood [\n]
4   \dotimes{\n
5     \b{Netcraft \blink{confirms} it;}
6     the JVM is naked and petrified!
7     \br
8   }
9 }
10 \crapflood{1000}
11
12 \it{Wasn't that fun?}
```

And then I got a real job doing embedded C for  
an avionics firm up in Milwaukee.



Around 2009 I started messing around with Ruby a lot, and it's actually pretty nice for a not-quite-Lisp.



But for the last two years, I've been doing  
Clojure as my main gig, and that's been pretty  
awesome.

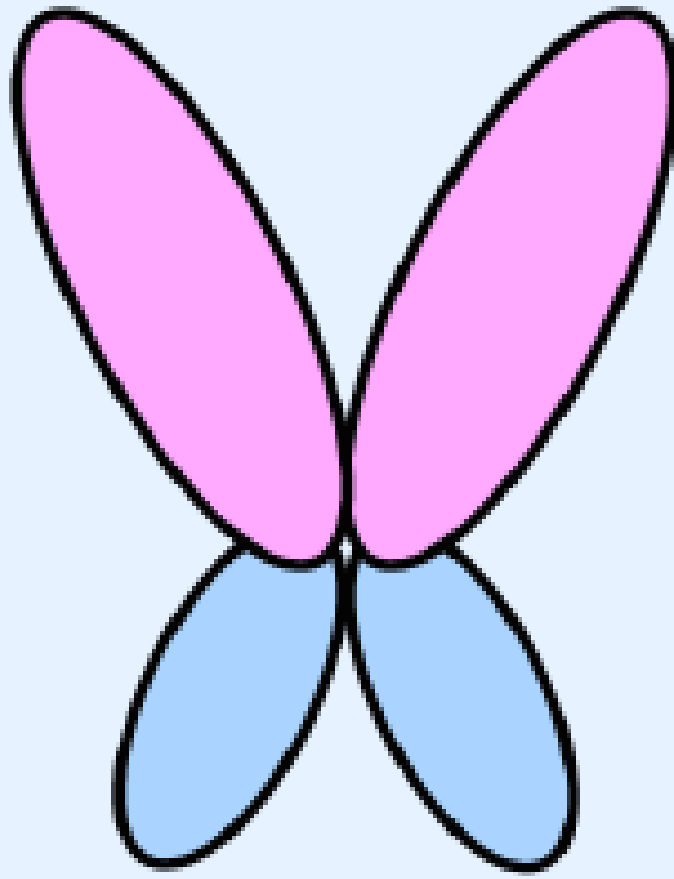




There's just one problem I really have with  
Clojure ...



Pixie is very early in development, inspired by Clojure (but not a port/fork/clone), and doesn't run on top of the JVM.



Pixie sits on top of RPython, a weird variant of Python for creating programming languages, originally created for PyPy.



## Let's make a Pixie!

```
1 $ git clone git@github.com:pixie-lang/pixie.git
2 $ cd pixie
3 $ make build_with_jit # This takes a while ...
4 $ ./pixie-vm # REPL = goodness
```

Building Pixie takes a while, but at least it's pretty to watch it go.

[illegible]

**Pixie has a decent startup time, so it's usable for tasks run from shell scripts.**

```
1 $ time ./hello-world.pxi
2 Hello, world!
3
4 real    0m0.120s
5 user    0m0.110s
6 sys     0m0.008s
```

**Ruby for comparison:**



```
1 $ time ./hello-world.rb
2 Hello, world!
3
4 real    0m0.033s
5 user    0m0.026s
6 sys     0m0.006s
```

## How long does this take with Clojure?

```
1 $ time lein exec hello-world.clj
2 Hello, world!
3
4 real      0m4.823s
5 user      0m17.321s
6 sys       0m0.599s
```

The JVM takes forever to start.

## There's a pixie-mode for Emacs.

- On Github: <https://github.com/johnwalker/pixie-mode>
- M-x package-install  pixie-mode 
- Make sure to have a build of Pixie on your path, I put mine at /opt/pixie.
- Add that to PATH environment variable in Emacs.
- Add that to exec-path in Emacs too.
- Then just C-c C-z to launch a Pixie REPL from a Pixie code file.



## There's already a lot of cool stuff there.

```
1 $ ./pixie-vm
2 user => "Hello, □Pixie!"
3 "Hello, □Pixie!"
4 user => (println "Hello, □Pixie!")
5 Hello, Pixie!
6 nil
7 user => (+ 1 2 3)
8 6
9 user => (defn foo [x] (+ x 4077))
10 <inst pixie.stdlib.Var>
11 user => (foo 12)
12 4089
```

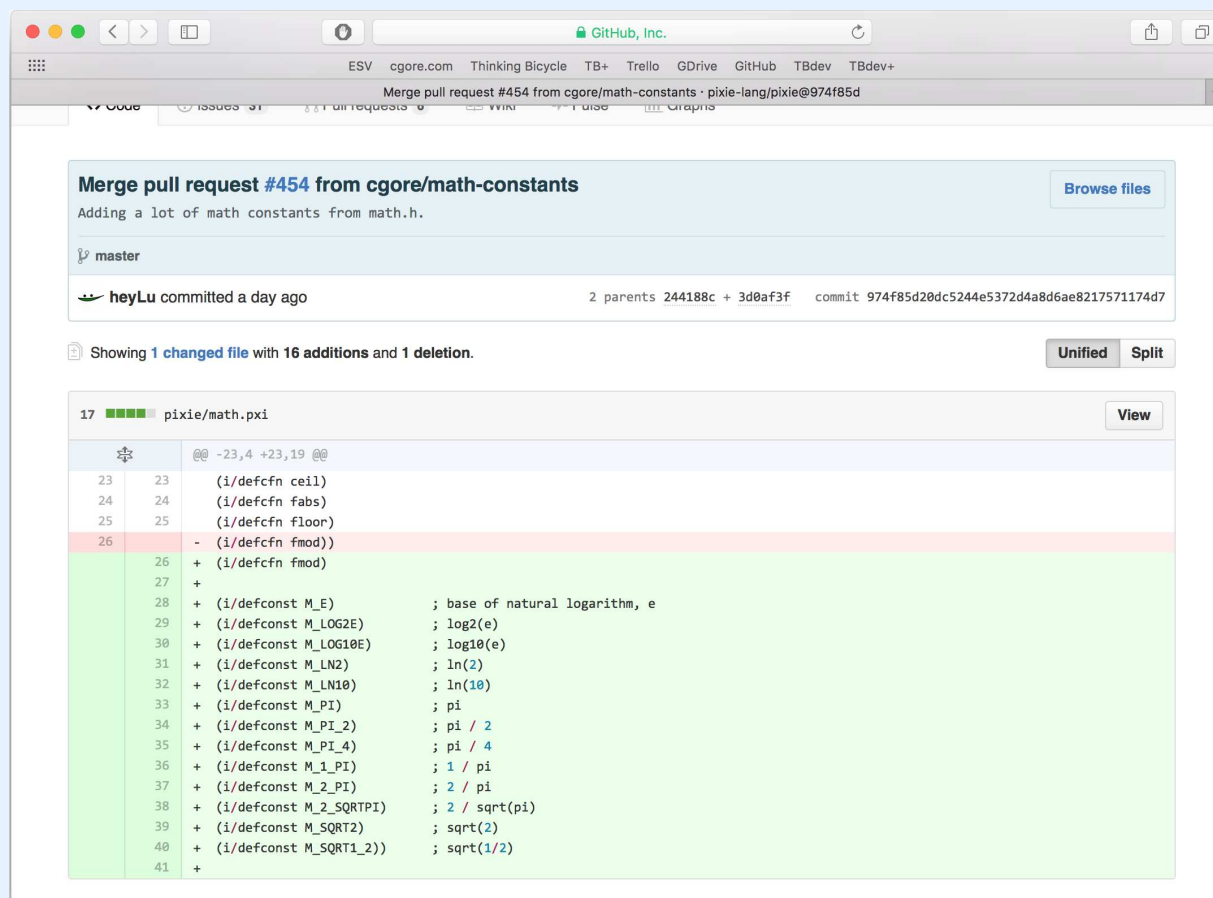
## Namespaces work in manner just like in Clojure.

```
1 user => (ns foo (:require [pixie.math :as math]))
2 nil
3 foo => (math/sin 1.2)
4 0.932039
5 foo => (math/sin 0.0)
6 0.000000
7 foo => (math/sin 3.14159)
8 0.000003
9 foo => (math/sin (/ 3.14159 2))
10 1.000000
```

## Lots of basic stuff isn't quite there yet though.

```
1 foo => math/PI
2 ERROR:
3   in pixie function repl_fn
4
5   in pixie/repl.pxi at 27:24
6           (let [x (eval form)]
7               ^
8   in internal function eval
9
10  in <unknown> at 5:1
11  math/PI
12  ^
13  RuntimeException: :pixie.stdlib/AssertionException
14  Var PI is undefined
```

But it's open source and they are quite open to pull requests, so we can add what we want!



Merge pull request #454 from cgore/math-constants

Adding a lot of math constants from math.h.

master

heyLu committed a day ago 2 parents 244188c + 3d0af3f commit 974f85d20dc5244e5372d4a8d6ae8217571174d7

Showing 1 changed file with 16 additions and 1 deletion. Unified Split

17 pixie/math.pxi View

```
@@ -23,4 +23,19 @@
23      (i/defcfn ceil)
24      (i/defcfn fabs)
25      (i/defcfn floor)
26      - (i/defcfn fmod))
26      + (i/defcfn fmod)
27      +
28      + (i/defconst M_E)           ; base of natural logarithm, e
29      + (i/defconst M_LOG2E)      ; log2(e)
30      + (i/defconst M_LOG10E)     ; log10(e)
31      + (i/defconst M_LN2)        ; ln(2)
32      + (i/defconst M_LN10)       ; ln(10)
33      + (i/defconst M_PI)         ; pi
34      + (i/defconst M_PI_2)       ; pi / 2
35      + (i/defconst M_PI_4)       ; pi / 4
36      + (i/defconst M_1_PI)       ; 1 / pi
37      + (i/defconst M_2_PI)       ; 2 / pi
38      + (i/defconst M_2_SQRTPI)   ; 2 / sqrt(pi)
39      + (i/defconst M_SQRT2)      ; sqrt(2)
40      + (i/defconst M_SQRT1_2)    ; sqrt(1/2)
41      +
```

## Numerics work similar to Clojure.

```
1 user => (+ 1 2)
2 3
3 user => (+ 1 2.0)
4 3.000000
5 user => (/ 1 2)
6 1/2
7 user => (/ 1 2.0)
8 0.500000
9 user => (/ 12)
10 1/12
```

## Strings work just like in Clojure.

```
1 user => "foo"
2 "foo"
3 user => (ns foo (:require [pixie.string :as s]))
4 nil
5 foo => (str "foo" "bar")
6 "foobar"
7 foo => (count "foo")
8 3
9 foo => (s/upper-case "why_should_we_shout?")
10 "WHY_SHOULD_WE_SHOUT?"
```

## Vectors work just like in Clojure.

```
1 user => (vector)
2 []
3 user => []
4 []
5 user => [1 2 3]
6 [1 2 3]
7 user => [1 2 "three" :four]
8 [1 2 "three" :four]
9 user => (= [1 2 3] [1 2 3])
10 true
11 user => (count [1 2 3])
12 3
13 user => (first [1 2 3])
14 1
15 user => (conj [1 2 3] 4)
16 [1 2 3 4]
```

## Hash maps work just like in Clojure.

```
1 user => {:a 1 :b 2 :c 3}
2 {:a 1, :c 3, :b 2}
3 user => (def m {:a 1 :b 2 :c 3})
4 <inst pixie.stdlib.Var>
5 user => (:a m)
6 1
7 user => (m :a)
8 1
9 user => (merge m {:d 4})
10 {:d 4, :a 1, :c 3, :b 2}
11 user => (keys m)
12 [:a :c :b]
13 user => (vals m)
14 [1 3 2]
```



## Functions work a lot like in Clojure.

```
1 user => (defn f [x] (+ x 2))
2 <inst pixie.stdlib.Var>
3 user => (f 12)
4 14
5 user => (def g (fn [x] (+ x 3)))
6 <inst pixie.stdlib.Var>
7 user => (g 12)
8 15
9 user => ((fn [x] (+ x 4)) 12)
10 16
```

## You have atoms just like in Clojure.

```
1 user => (def a (atom 1))
2 <inst pixie.stdlib.Var>
3 user => a
4 <inst pixie.stdlib.Atom>
5 user => @a
6 1
7 user => (reset! a 7)
8 7
9 user => @a
10 7
```

You can have anonymous function literals, just like in Clojure.

```
1 user => (map #(+ % 2) [1 2 3])  
2 (3 4 5)  
3 user => (map #(+ %1 %2) [1 2 3] [4 5 6])  
4 (5 7 9)
```

All the basic logic and control flow you would expect is available, and typically works exactly like it does in Clojure.

```
1 user => (defn small [x]
2           (if (< x 100)
3               (println x "is_pretty_small.")
4               (println "Wow," x "is_huge!")))
5 <inst pixie.stdlib.Var>
6 user => (small 12)
7 12 is pretty small.
8 nil
9 user => (small 1000)
10 Wow, 1000 is huge!
11 nil
```

## Conclusion

Pixie is a fun little language with a lot of promise. Using it for important production code today is probably asking to get fired, but it's already an okay choice for some simple scripting tasks. Help out and it'll be ready for prime time sometime soon!

*Questions?*