Pixie

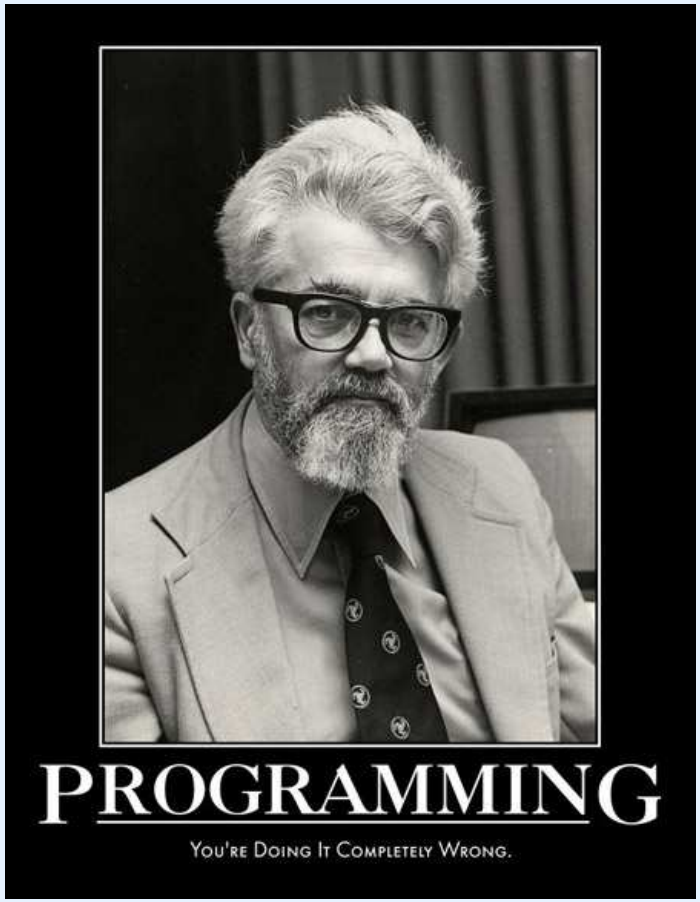
Christopher Mark Gore

cgore.com

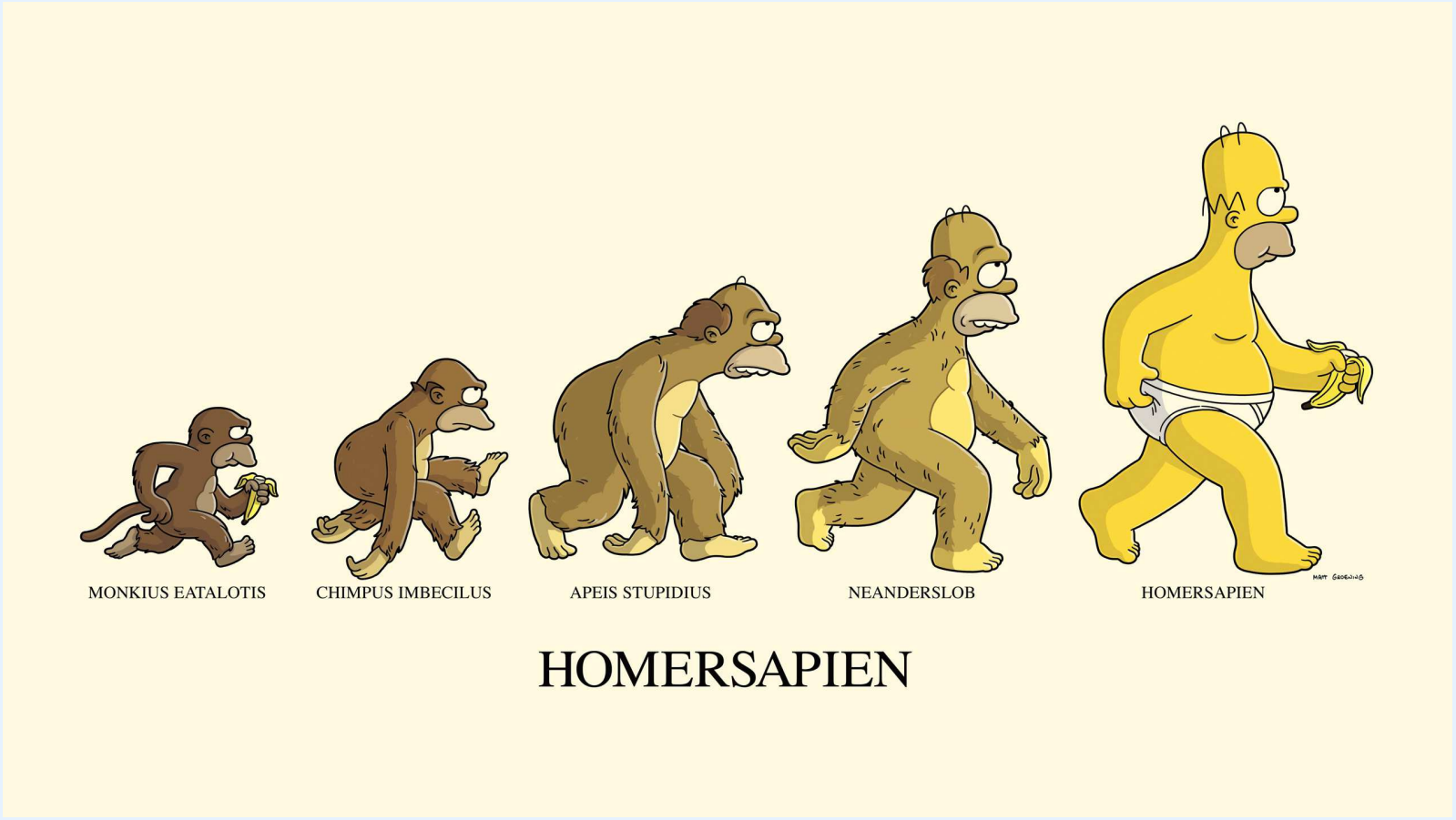
Thursday, December 3, AD 2015

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.



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

```
Welcome to the future of crapflooding!

\defun{\crapflood [\n]
  \dotimes{\n
    \b{Netcraft \blink{confirms} it;}
    the JVM is naked and petrified!
    \br
  }
}
\crapflood{1000}

\it{Wasn't that fun?}
```

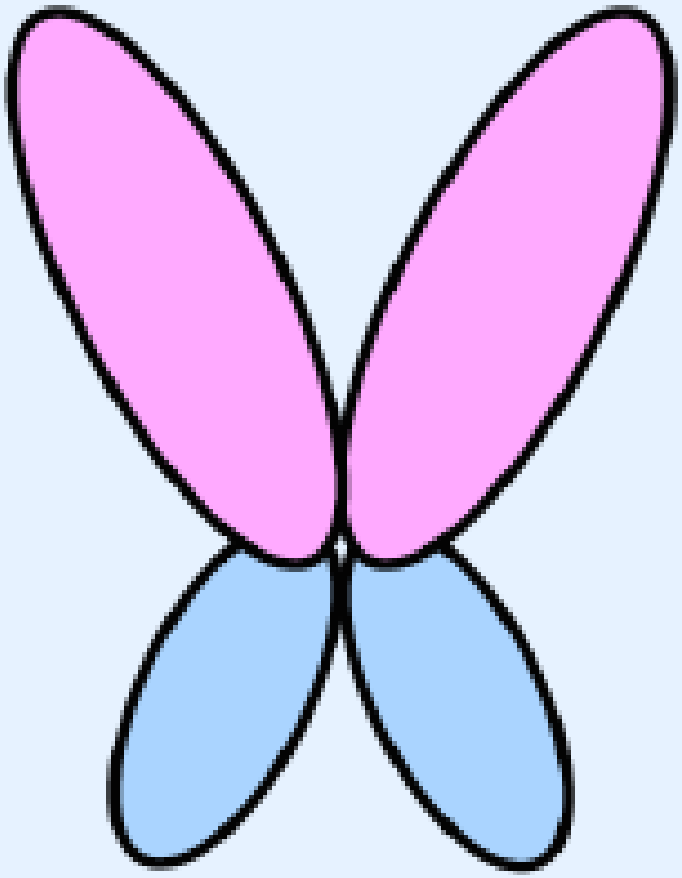
And then I got a real job doing embedded C for an avionics firm up in Milwaukee.

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.

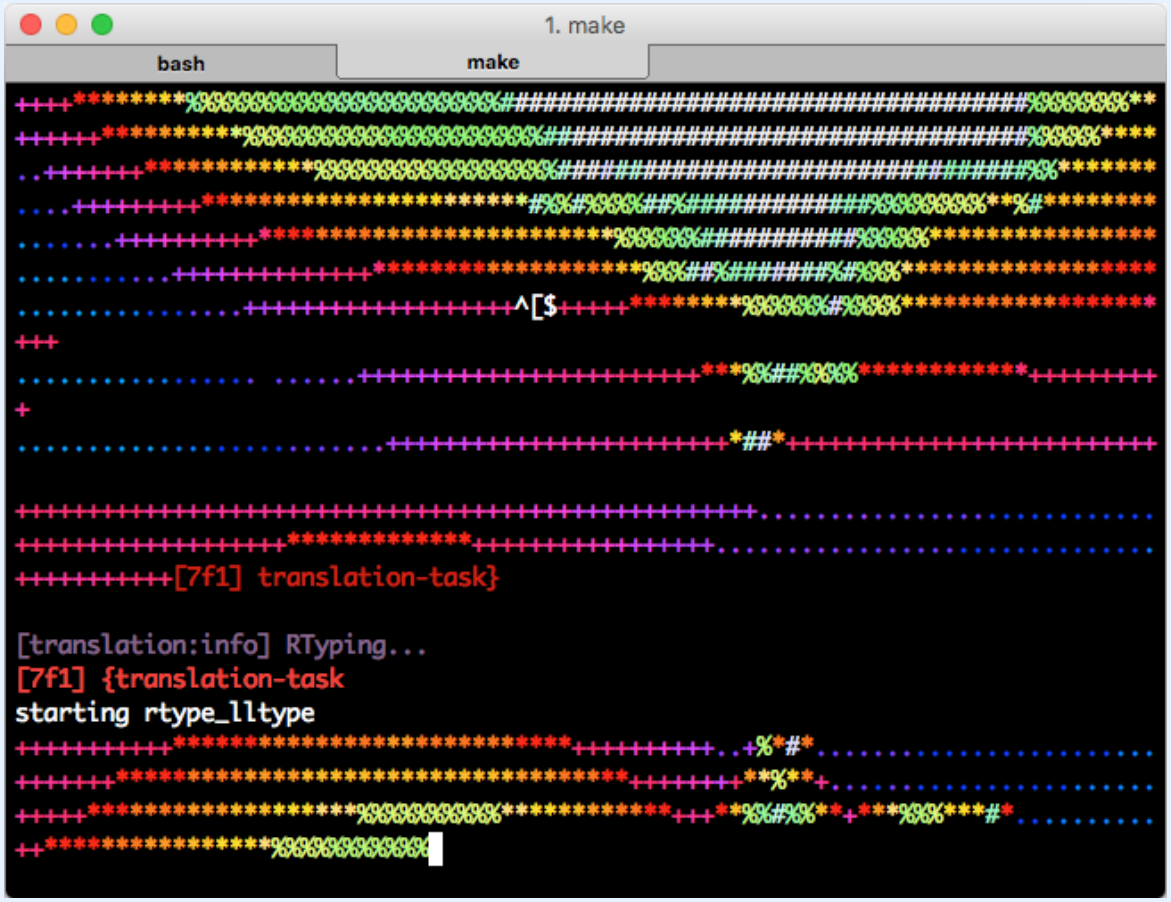


Pixie sits on top of RPython, a weird variant of Python for creating programming languages, originally created for PyPy.

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.



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

```
$ time ./hello-world.pxi
Hello, world!

real    0m0.120s
user    0m0.110s
sys     0m0.008s
```

Ruby for comparison:

```
$ time ./hello-world.rb
Hello, world!

real    0m0.033s
user    0m0.026s
sys     0m0.006s
```

How long does this take with Clojure?

```
$ time lein exec hello-world.clj
Hello, world!

real    0m4.823s
user    0m17.321s
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!" ; Strings!
3  "Hello, Pixie!"
4  user => (println "Hello, Pixie!") ; Printing to the screen!
5  Hello, Pixie!
6  nil
7  user => (+ 1 2 3) ; Math!
8  6
9  user => (defn foo [x] (+ x 4077)) ; Functions!
10 <inst pixie.stdlib.Var>
11 user => (foo 12)
12 4089
```

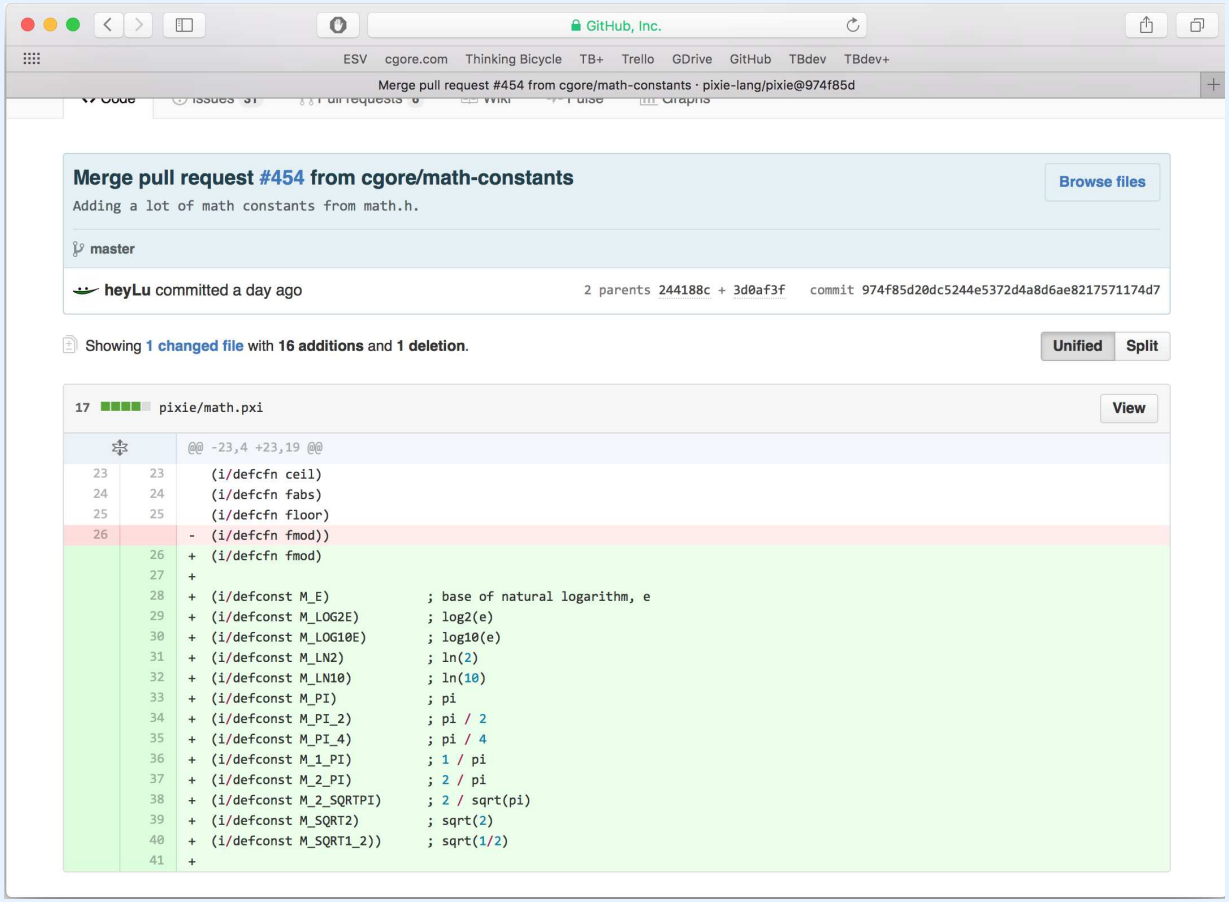
Namespaces work in manner just like in Clojure.

```
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!



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 ; That's right, we can do fractions.
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]
```

Lists work just like in Clojure.

```
1  user => (list)
2  ()
3  user => (1 2 3) ; This isn't right.
4  ERROR: ...
5  user => '(1 2 3) ; This works.
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 (4 1 2 3) ; Clojure does it this way too.
```

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) ; Treating the hash map as the argument.
6  1
7  user => (m :a) ; Treating the hash map as the function.
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]
```

Sets work a lot like in Clojure.

```
1  user => (hash-set)
2  #{}
3  user => #{1 2 3}
4  #{1 3 2}
5  user => (count #{1 2 "three" :four})
6  4
7  user => (conj #{1 2 3} 4)
8  #{1 3 2 4}
9  user => (ns foo (:require [pixie.set :as s]))
10 nil
11 foo => (s/union #{1 2 3} #{4 5 6})
12 #{1 3 5 2 6 4}
13 foo => (s/difference #{1 2 3 4 5} #{3 4})
14 #{1 5 2}
15 foo => (s/intersection #{1 2 3 4 5} #{4 5 6 7 8})
16 #{5 4}
```

Functions work a lot like in Clojure.

```
1  user => (defn f [x] (+ x 2)) ; Define a function.
2  <inst pixie.stdlib.Var>
3  user => (f 12)
4  14
5  user => (def g (fn [x] (+ x 3))) ; Or just def a lambda.
6  <inst pixie.stdlib.Var>
7  user => (g 12)
8  15
9  user => ((fn [x] (+ x 4)) 12) ; Why bother naming things?
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)
2              [1 2 3])
3 (3 4 5)
4 user => (map #(+ %1 %2)
5              [1 2 3]
6              [4 5 6])
7 (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?