Concatenative Programming

Zhuyang Wang

March 31, 2018

Outline

Imperative

```
data = read("data.txt")
data_sorted = sort(data)
data_sorted_unique = uniq(data_sorted)

x = double(42)
s = toString(x)
r = reverse(s)
```

One-liner

```
uniq(sort(read("data.txt")))

reverse(toString(double(42)))
```

Pipeline

```
cat data.txt | sort | uniq
```

```
42 |> double |> toString |> reverse
```

Function composition in math

$$f(g(x)) = (f \circ g)(x)$$

$$(f \circ g) \circ h = f \circ (g \circ h)$$

Javascript

```
function o(f, g) {
    return function(x) {
        return f(g(x));
    }
}

o(reverse, o(toString, double))
```

Haskell

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)

reverse . toString . double
```

Concatenative

```
"data.txt" cat sort uniq

25 double toString reverse
```

Stack

```
4 2 1 + * 3 -
```

```
Item  Stack
4:    4
2:    4 2
1:    4 2 1
+:    4 3
*:    12
3:    12 3
-:    9
```

What about if/else

```
2 3 +
1 4 *
5 6 <
if
```

No way to express if/else!

Need some notation to group codes.

Quotation

```
2 3 [ + 5 * ]          ==> 2 3 [ + 5 * ]

2 3 [ + 5 * ] apply ==> 2 3 + 5 *
                       ==> 25
```

```
          [A] zap  ==
          [A] i    == A
          [A] unit == [[A]]
          [A] rep  == A A
          [A] dup  == [A] [A]
     [B] [A] k    == A
     [B] [A] nip  == [A]
     [B] [A] dip  == A [B]
     [B] [A] cat  == [B A]
     [B] [A] swap == [A] [B]
     [B] [A] sip  == [B] A [B]
     [B] [A] w    == [B] [B] A
[C] [B] [A] b    == [[C] B] A
[C] [B] [A] c    == [B] [C] A
[C] [B] [A] s    == [[C] B] [C] A
```

if/else

```
[ 2 3 + ]
[ 1 4 * ]
5 6 <
if
==>
5
```

Actually we need more combinators to implement if/else.

See The Theory of Concatenative Combinators for details.

Higher-order

```
{ "hello" "world" } [ reverse ] map
==>
{ "olleh" "dlrow" }

{ 2 3 4 } [ 3 * ] map
==>
{ 6 9 12 }

{ "1" "2" "3" } [ string>number ] map
==>
{ 1 2 3 }
```

```
{ 1 2 3 4 5 } [ 4 < ] filter
==>
{ 1 2 3 }

{ "1" "22" "333" } [ length 2 < ] filter
==>
{ "1" }
```

```
{ 1 2 3 4 } 0 [ + ] reduce
==>
(((0+1)+2)+3)+4

{ 2 3 4 5 } 1 [ * ] reduce
==>
(((1*2)*3)*4)*5
```

Pointless

```
{ 2 3 4 } isPrime? filter          ==> { 2 3 }
{ 2 3 } length                     ==> 2
{ 2 3 4 } isPrime? filter length ==> 2

Define countWhere = filter length
```

Javascript

```
function filter(predicate, sequence) { ... }
function length(sequence) { ... }

function countWhere(predicate, sequence) {
  return length(filter(predicate, sequence));
}
```

Haskell

```
countWhere predicate sequence =
    length (filter predicate sequence)

countWhere = ((.).(.)) length filter
```

Example

```
"http://factorcode.org" http-get nip string>xml
"a" deep-tags-named
[ "href" attr ] map
[ print ] each
```

Utils

```
a dup              ==> a a
a drop             ==>
a b swap           ==> b a
a b c rot          ==> b c a
a b c -rot         ==> c a b
x [ f ] [ g ] bi ==> x f x g
```

Arithmetic

Define $f(x, y, z) = (x^2 + y^2) - |y|$

```
square = dup *
f = drop [ square ] [ abs ] bi rot square rot + swap -
```

```
x y z drop                    ==> x y
x y [ square ] [ abs ] bi     ==> x (y*y) |y|
x (y*y) |y| rot               ==> (y*y) |y| x
(y*y) |y| x square rot        ==> |y| (x*x) (y*y)
|y| (x*x) (y*y) +             ==> |y| (x*x+y*y)
|y| (x*x+y*y) swap -          ==> (x*x+y*y-|y|)
```

Lexically Scoped Variables

Factor:

```
:: f ( x y z -- n )
  x square y square + y abs -
```

PostScript:

```
/f {
  /z exch def
  /y exch def
  /x exch def
  x square y square + y abs -
} def
```

Factor: A Dynamic Stack-based Programming Language:

> *We have found lexical variables useful only in rare cases where there is no obvious solution to a problem in terms of dataflow combinators and the stack. Out of 38,088 word and method definitions in the source code of Factor and its development environment at the time of this writing, 310 were defined with named parameters. Despite their low rate of use, we consider lexical variables, and in particular lexically-scoped closures, a useful extension of the concatenative paradigm.*

Factor

- Concatenative
- Stack Effect Checker
- Lexically Scoped Variables
- Module System
- Object System
- Macro and Metaprogramming
- Coroutines/Multiprocess
- Optimized Compiler ~~(faster than CPython/V8)~~
- FFI
- IDE!

Example

```
"~/concat-lang/demo.txt" utf8 file-lines
[ "/" split1-last nip string>number prime? ] filter
[ http-get nip json>
  [ "userId" of number>string ]
  [ "title" of ]
  bi append
] map
```

More concatenative languages

- Forth
- Joy
- PostScript
- Factor
- Cat
- Kitten
- XY

Reference

- Why Concatenative Programming Matters (Discussion)
- The Theory of Concatenative Combinators
- Concatenative Wiki
- Writing a Concatenative Programming Language
- Factor tutorial