Just How Functional is Raku?

Here's a Random Pick from Rosetta code: 100 doors

Let's see how it looks in Elm (a famous functional language)

[this is the unoptimized variation]

100 doors

There are 100 doors in a row that are all initially closed.
You make 100 passes by the doors.
The first time through, visit every door and *toggle* the door (if the door is closed, open it; if it is open, close it).
The second time, only visit every 2nd door (door #2, #4, #6, ...), and toggle it.
The third time, visit every 3rd door (door #3, #6, #9, ...), etc, until you only visit the 100th door.

Task
Answer the question: what state are the doors in after the last pass? Which are open, which are closed?

#viz.
https://rosettacode.org/wiki/100_doors

```
import List exposing (indexedMap, foldl, repeat, range)
import Html exposing (text)
import Debug exposing (toString)

type Door = Open | Closed

toggle d = if d == Open then Closed else Open

toggleEvery : Int -> List Door -> List Door
toggleEvery k doors = indexedMap
  (\i door -> if modBy k (i+1) == 0 then toggle door else door)
  doors

n = 100

main =
  text (toString (foldl toggleEvery (repeat n Closed) (range 1 n)))
```

Elm

functional

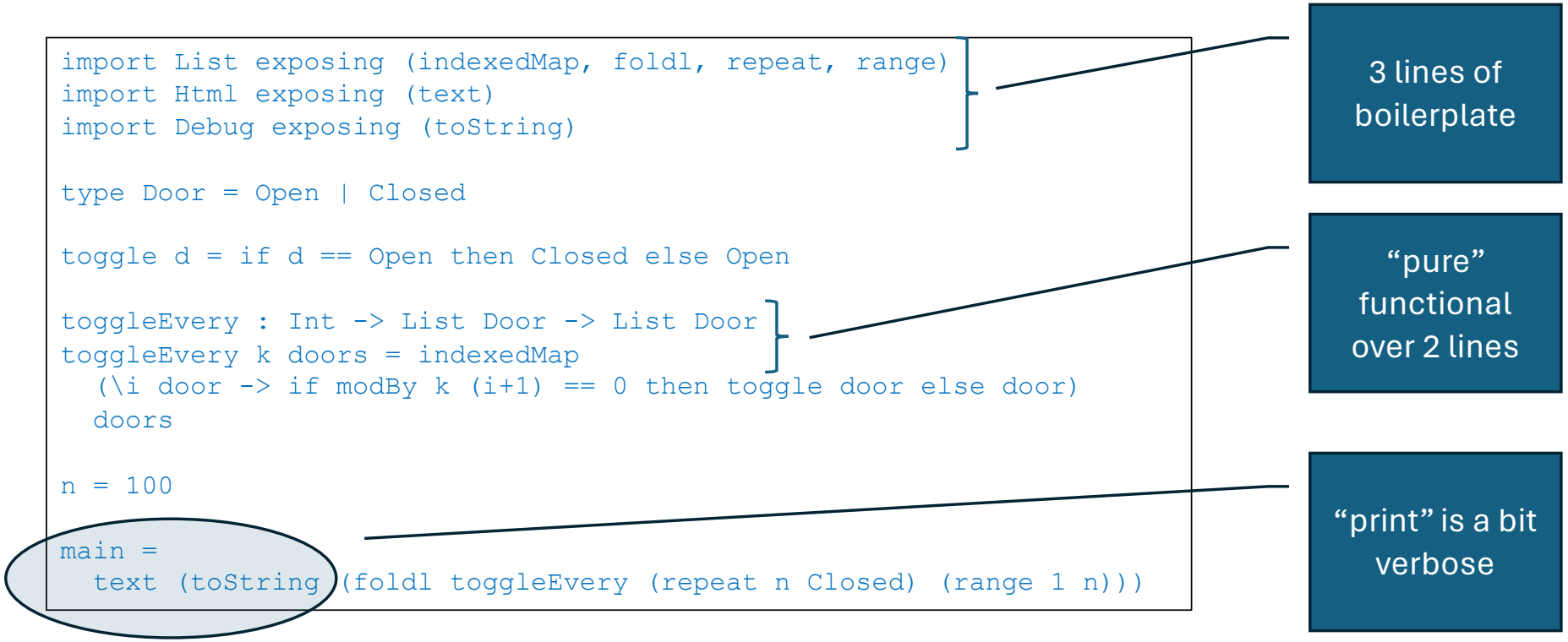
```
enum Door <Closed Open>;

sub toggle(\d) { if d {Closed} else {Open} };

sub toggleEvery( Door @doors, Int \k --> Array[Door]() ) {
    @doors.map: -> \door {if ++$ %% k {toggle door} else {door}};
}

my \n = 100;
my Door @doors = Closed xx n;

say reduce &toggleEvery, @doors, |(1..n);
```

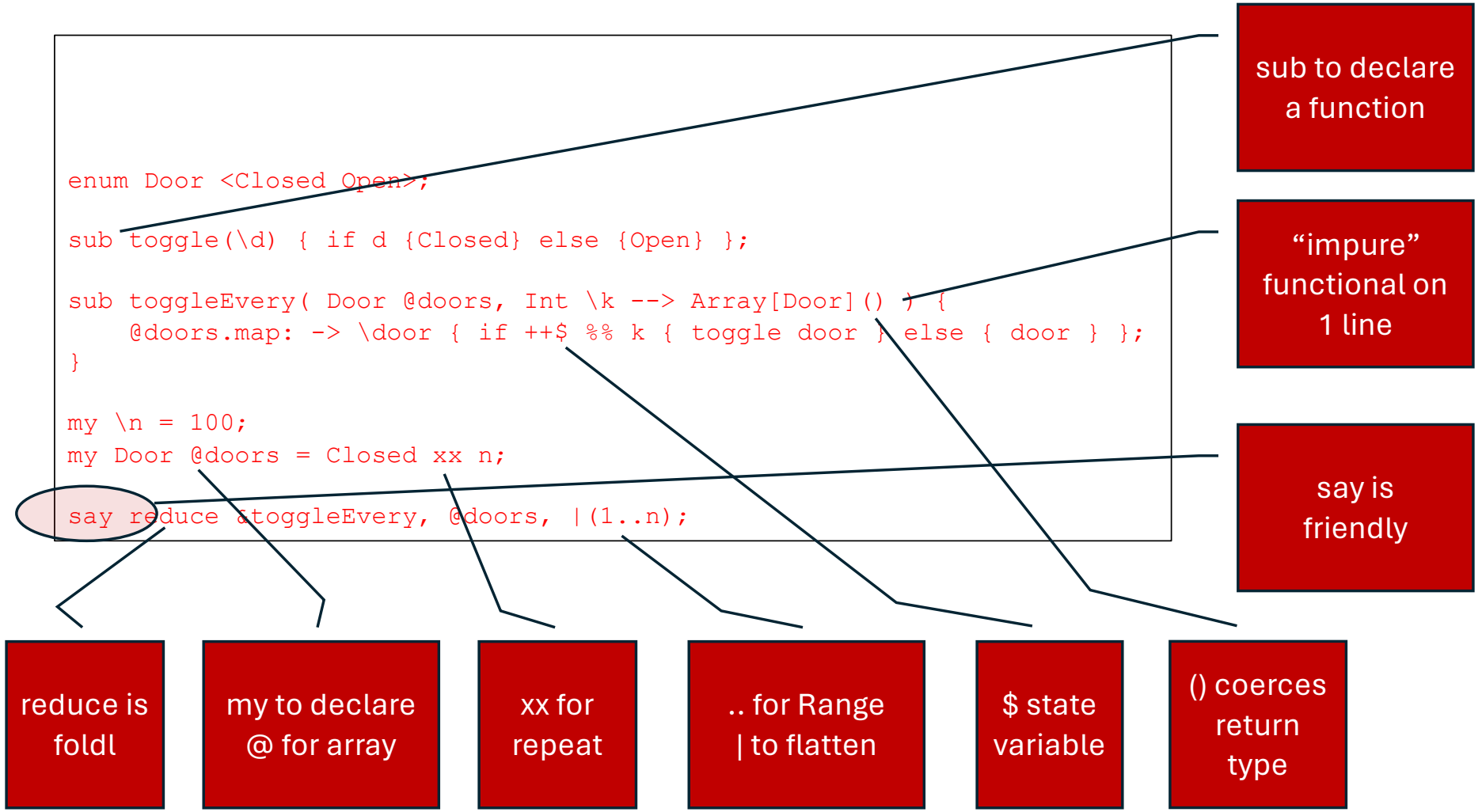
Raku

functional

```
import List exposing (indexedMap, foldl, repeat, range)
import Html exposing (text)
import Debug exposing (toString)

type Door =<Closed | Open>;

sub toggle(\d)={ if d == Open then{Closed} else {Open} };

sub toggleEvery(:Door @doors, Int \k--> List[Door]() ) {
    toggleEvery k doors = indexedMap
  (\idx door -> if (idx+1) %% k { toggle door } else { door } );
} doors

my \n = 100;
my Door @doors = Closed xx n;
main =
  text (toString (foldl toggleEvery, @doors, (1..n)))
```

The nice surprise is how well Raku can ape Elm. As the direct descendant of perl (home of .map, .grep and so on) plus types Raku can be home to all those who want their code functional.


import List exposing (indexedMap, foldl, repeat, range)
import Html exposing (text)
import Debug exposing (toString)
type Door = Open | Closed
toggle d = if d == Open then Closed else Open
toggleEvery : Int -> List Door -> List Door
toggleEvery k doors = indexedMap
(\i door -> if modBy k (i+1) == 0 then toggle door else door)
doors
n = 100
main =
text (toString (foldl toggleEvery (repeat n Closed) (range 1 n)))
3 lines of boilerplate
“pure” functional over 2 lines
“print” is a bit verbose


enum Door <Closed Open>;
sub toggle(\d) { if d {Closed} else {Open} };
sub toggleEvery(Door @doors, Int \k --> Array[Door]()) {
@doors.map: -> \door { if ++$ %% k { toggle door } else { door } };
}
my \n = 100;
my Door @doors = Closed xx n;
say reduce &toggleEvery, @doors, |(1..n);
sub to declare a function
“impure” functional on 1 line
say is friendly
reduce is foldl
my to declare @ for array
xx for repeat
.. for Range | to flatten
$ state variable
() coerces return type

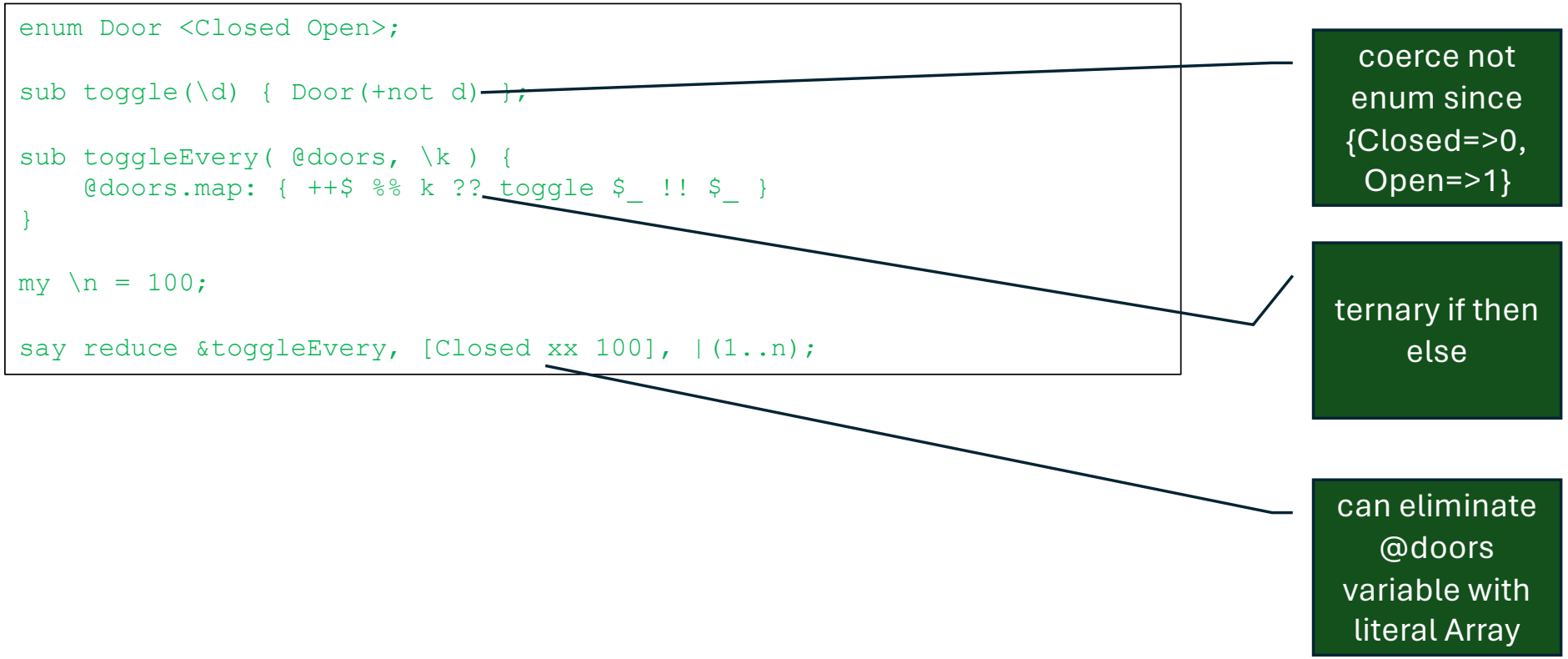
```
enum Door <Closed Open>;

sub toggle(\d) { Door(+not d) };

sub toggleEvery( @doors, \k ) {
    @doors.map: { ++$ %% k ?? toggle $_ !! $_ }
}

my \n = 100;


say reduce &toggleEvery, [Closed xx 100], |(1..n);
```

Raku

functional && untyped

Or, you may prefer untyped Raku, to whip something up

```
enum Door <Closed Open>;

sub toggle(\d) { if d ~~ Closed {Open} else {Closed} };

sub toggleEvery( @doors, \k ) { [1..k] ==> map { @doors[$_ - 1] = toggle @doors[$_ - 1] } };

my \n = 100;
my Door @doors = Closed xx n;

toggleEvery @doors, 1..n;

say @doors.kv.grep( { $^v ~~ Open } )».key »+» 1;
```


enum Door <Closed Open>;
sub toggle(\d) { Door(+not d) };
sub toggleEvery(@doors, \k) {
@doors.map: { ++$ %% k ?? toggle $_ !! $_ }
}
my \n = 100;
say reduce &toggleEvery, [Closed xx 100], |(1..n);
coerce not enum since {Closed=>0, Open=>1}
ternary if then else
can eliminate @doors variable with literal Array

```
my \n = 100;
my @doors = False xx n;

(.=not for @doors[0, $_ ... n]) for 1..n;

print <Closed Open>[ @doors[$_] ] for 1..n;
```

Raku

set to eleven

Or, you may prefer Raku with all the toys to maximize -Ofun

```
my \n = 100;
my @doors = False xx n;

(.=not for @doors[0, $_ ... n]) for 1..n;

print <Closed Open>[ @doors[$_] ] for 1..n;
```

.= to apply operation in place

with Bools, toggle becomes not

... is the sequence operator

Bools have value 0,1 can be indices

Raku

set to eleven

Or, you may prefer Raku with all the toys to maximize -Ofun

Just How (Fun)ctional is Raku?

viz. https://docs.raku.org/language/haskell-to-p6
code. https://gist.github.com/librasteve/36ca4a2876618ea426d30aa80667e923