An FP Mini-Series

# Dijkstra Quotes

1. When we take the position that it is not only the programmer's responsibility to produce a correct program but also to demonstrate its correctness in a convincing manner, then the above remarks have a profound influence on the programmer's activity: the object he has to produce must be usefully structured. (1970)
2. The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible. (1970)
3. The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. (1970)
4. it does not suffice to hone your own intellect (that will join you in your grave), you must teach others how to hone theirs. The more you concentrate on these two challenges, the clearer you will see that they are only two sides of the same coin: teaching yourself is discovering what is teachable. (1979)
5. as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them, it has created the problem of using its products. (1972)
6. The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer. (1972)
7. If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with. (1972)
8. The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense. (1975)
9. How do we convince people that in programming simplicity and clarity —in short: what mathematicians call "elegance"— are not a dispensable luxury, but a crucial matter that decides between success and failure? (1980s)
10. Thank goodness we don't have only serious problems, but ridiculous ones as well. (1982)
11. The question of whether Machines Can Think... is about as relevant as the question of whether Submarines Can Swim. (1984)
12. Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better (1984)
13. The question of whether Machines Can Think... is about as relevant as the question of whether Submarines Can Swim. (1984)
14. When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem. (1986)
15. The practice is pervaded by the reassuring illusion that programs are just devices like any others, the only difference admitted being that their manufacture might require a new type of craftsmen, viz. programmers. From there it is only a small step to measuring "programmer productivity" in terms of "number of lines of code produced per month". This is a very costly measuring unit because it encourages the writing of insipid code, but today I am less interested in how foolish a unit it is from even a pure business point of view. My point today is that, if we wish to count lines of code, we should not regard them as "lines produced" but as "lines spent": the current conventional wisdom is so foolish as to book that count on the wrong side of the ledger. (1988)
16. I mean, if 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself "Dijkstra would not have liked this", well, that would be enough immortality for me. (1995)
17. A picture may be worth a thousand words, a formula is worth a thousand pictures. (1990s)
18. Elegance is not a dispensable luxury but a quality that decides between success and failure. (1990s)
19. The required techniques of effective reasoning are pretty formal, but as long as programming is done by people that don't master them, the software crisis will remain with us and will be considered an incurable disease. And you know what incurable diseases do: they invite the quacks and charlatans in, who in this case take the form of Software Engineering gurus (2000).
20. There are very different programming styles. I tend to see them as Mozart versus Beethoven. When Mozart started to write, the composition was finished. He wrote the manuscript and it was 'aus einem Guss' (from one cast). In beautiful handwriting, too. Beethoven was a doubter and a struggler who started writing before he finished the composition and then glued corrections onto the page. In one place he did this nine times. When they peeled them, the last version proved identical to the first one. (2001)
21. In short, I suggest that the programmer should continue to understand what he is doing, that his growing product remains firmly within his intellectual grip. It is my sad experience that this suggestion is repulsive to the average experienced programmer, who clearly derives a major part of his professional excitement from not quite understanding what he is doing. In this streamlined age, one of our most undernourished psychological needs is the craving for Black Magic and apparently the automatic computer can satisfy this need for the professional software engineer, who is secretly enthralled by the gigantic risks he takes in his daring irresponsibility. For his frustrations I have no remedy...... (date unknown)
22. This is generally true: any sizeable piece of program, or even a complete program package, is only a useful tool that can be used in a reliable fashion, provided that the documentation pertinent for the user is much shorter than the program text. If any machine or system requires a very thick manual, its usefulness becomes for that very circumstance subject to doubt! (date unknown)

# About Dijkstra

1. The precious gift that this Turing Award acknowledges is Dijkstra's style: his approach to programming as a high, intellectual challenge; his eloquent insistence and practical demonstration that programs should be composed correct, not just debugged into correctness; and his illuminating perception of problems at the foundations of program design. (M.D. Mcllroy, 1972, about Dijkstra)

0: Introduction

Why

Performance

Readable

“When the bullets start flying”

“Whey 4 goons jump you in a dark alley, it is too late to train”

Comprehensible

Juggling Variables

Easy to Reason about

Intuition

Killer Spaghetti Code

http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences

The Goal is NOT to inform you about FP. That is very easy.

The Goal is to develop an INTUITION for code smells, an how you can be better

“I was blind, but now I see”

Pattern recognition

1 of 4 FP Basics

visual clutter is mental clutter

- Immutable

- Composition

# Statements vs Expressions

Expressions can be reduced

# Higher-Order Functions

# Predicates

Extract your conditionals into its own function

They are simply functions containing conditionals (ie if-then-else)

# How to Sharpen Your Senses

# From For-Loops to FP Loops

##what each means

1. filter() : subset

2. map() : transform

3. reduce() : aggretate

4. forEach() : side effect. use rarely. mostly for console.log()

##Use appropriately

2 Arrow Functions

(An alternative syntax to standard function declaration)

# Standard

let f = function(x) {

return x \* 10

}

f //returns "function(x) {return x \* 10}".

f(3) //applies "function(3) {return 3 \* 10}", reduces to "3 \* 10", reduces to "30".

# FP

let f = x => x \* 10 //returns "x => x \* 10"

f(3) //applies "3 => 3 \* 10", reduces to "3 \* 10", reduces to "30".

# Arrow functions with manual return

let f = x => x \* 10 //returns "x => x \* 10"

f //returns "x => x \* 10"

f(3) //applies "3 => 3 \* 10", reduces to "3 \* 10", reduces to "30".

vs

let f = x => {x \* 10} //returns "x => {x \* 10}"

f //returns "x => {x \* 10}"

f(3) //applies "3 => {3 \* 10}", reduces to "{3 \* 10}", == "{30}", reduces to "undefined".

vs

let f = x => {return x \* 10} //returns "x => {return x \* 10}"

f //returns "x => {return x \* 10}"

f(3) //applies "3 => {return 3 \* 10}", reduces to "{return 3 \* 10}", reduces to "{return 30}", reduces to "30".

# Anonymous Functions (aka Lambdas)

(Arrow functions make it convenient to declare anonymous functions)

3 Partial Application (and on a lesser note, “Currying”)

Shuffling Cards

# Currying simplified

Standard

let f = function(x) {

return x \* 10

}

f //returns "function(x) {return x \* 10}".

f(3) //applies "function(3) {return 3 \* 10}", reduces to "3 \* 10", reduces to "30".

FP

let f = x => x \* 10 //returns "x => x \* 10"

f //returns "x => x \* 10"

f(3) //applies "3 => 3 \* 10", reduces to "3 \* 10", reduces to "30".

# Currying and partial application

Standard

let f = function(x, y) {

return x \* y

}

f //returns "function(x, y) {return x \* y}"

f(2, 3) //applies "function(2, 3) {return 2 \* 3}", reduces to "return 2 \* 3", reduces to "6"

FP

let f = x => y => x \* y //returns "x => y => x \* y"

f //returns "x => y => x \* y"

f(2)(3) //applies "2 => 3 => 2 \* 3", reduces to "2 \* 3", reduces to "6"

let fx = f(2) //returns "y => x \* y", reduces to "y => 2 \* y"

fx //returns "y => x \* y", reduces to "y => 2 \* y"

fx(3) //applies "3 => 2 \* 3", reduces to "2 \* 3", reduces to "6"

let fxy = fx(3) //applies "3 => 2 \* 3", reduces to "2 \* 3", reduces to "6"

fxy //returns "6"

# Abstractions enabled by using curried functions

let multiplyTwoNumbers = x => y => x \* y

let double = multiplyTwoNumbers(2)

let triple = multiplyTwoNumbers(3)

let filter = f => list = list.filter(f)

let isEven = x => x % 2 === 0

let isOdd = x => x % 2 !== 0

let getEvenNumbers = filter(isEven)

let getOddNumbers = filter(isEven)

let myData = [0, 1, 2, 3, 4, 5]

getEvenNumbers(myData) //returns [0, 2, 4]

getOddNumbers(myData) //returns [1, 3, 5]

# Alternative 'standards', from most readable to least readable

FP

(easiet to read)

getEvenNumbers(myData)

Array.filter with named functions

(more to nouns and verbs to juggle than FP, but still ok)

isEven = ...

myData.filter(isEven)

Array.filter with anonymous functions

(you missed the chance for reuseability)

myData.filter(d => d % 2 === 0)

Array.forEach

(lower intent signal. will produce side-effects)

let result = []

myData.forEach(d => {if (d % 2 === 0) result.push(d)})

For loops

(Just. No.)

let result = []

for (let i = 0; i < myData.length; i++) {

if (myData[i] % 2 === 0) {

result.push(myData[i])

}

}

# Practical Partial Application: the Bind() functions

so from last time's composition work,

you now have the ability to compose any number of functions together

Intermission: A Note On Repetition Repetition Repetition

* I'm aware, I could shorten it, but its the basis of pattern matching
* Seeing the patterns will develop your intuition (aka code smell)
* Then when you see sub-optimal code, you will be very, very annoyed.
* I try to use the terms FP paradigm practitioners would use, so When you go online, the terms will be familiar
* "When the bullets start flying"

3. Composition

-----------------------

Chains

# A Note on Naming convention

variables set 1 : x,y,z

variables set 2 : a,b,c

functions : f,g,h

# Important pattern: functions before data. this will support currying

composition

function myFunc(f, g, x){

f(g(x))

}

Or

let compose = f => g => x = f(g(x))

first, we setup reuseable functions we can compose later

let triple = x => x \* 3 //returns "x => x \* 3"

let addOne = x => x + 1 //returns "x => x + 1"

# Simple, 2-step (right-to-left)

let compose = f => g => x => f(g(x)) //returns "f => g => x => f(g(x))"

let sequence = compose(triple, addOne) //returns "x => f(g(x))", //just remember: f=triple, g=addOne

//reduces to "x => f(x + 1)", //g applied to x

//reduces to "x => (x + 1) \* 3" //f applied to (g applied to x)

sequence(2) //applies "2 => (2 + 1) \* 3", reduces to "(3) \* 3", reduces to "9"

sequence(5) //applies "5 => (5 + 1) \* 3", reduces to "(6) \* 3", reduces to "18"

sequence(7) //applies "7 => (7 + 1) \* 3", reduces to "(8) \* 3", reduces to "24"

# Complex - Any number of steps

first, remember Array.reduce() - means aggretation.

let myData = [2, 3] //returns "[2, 3]"

myData.reduce((a, x) => a + x, 0) //reduces to "(0 + 2) + 3", reduces to "(2) + 3", reduces to "5"

N-step (left-to-right)

let pipe = fs => x => fs.reduce((a, f) => f(a), x) //returns "x => fs.reduce((a, f) => f(a), x)"

let pipedF = pipe([triple, addOne]) //returns "x => fs.reduce((a, f) => f(a), x)", reduces to "f(x)"

N-step (right-to-left, more traditional)

let compose = fs => x => fs.reduceRight((a, f) => f(a), x)

let composedF = compose([addOne, triple]) //

# Important! Left-to-Right / Right-to-Left sequence matters!

let sequenceA = compose([triple, addOne])

let sequenceB = composeRight([triple, addOne])

sequenceA(3) //returns 10

sequenceB(3) //returns 12

Intermission: Array.prototype.slice.call()

what's that you say? still unhappy about the [brackets]?

psh... okay I have one more lifehack for you. let's get rid of the [brackets]

compose([addOne, triple, addOne])

Finale: Putting it all together - Refactoring to FP

# Code Smells

* duplicate code
* anonymous functions
* suboptimal parameter order: should be functions first, data last
* mutation/side effects

# how to

* remove all FOR LOOPS
* refactor anonymous functions into named functions

-

# hints at the right direction

* small functions. some are 1 liners
* more readable, semantic, verbose
* pure, no-mutations
* if mutations/states are needed, state is in a traditional "parent" function, calling pure functions

Overture: From FP to PF

Intro to Point-Free programming style

Workflow

1. do the simplest thing
2. get angry
3. keep calm, find the code smells and refactor
4. code all over again

While I do not share the extremes of Mr. Dijkstra’s opinion of COBOL, I know using it is akin to playing with fire.