Functional vs. Imperative Programming

A short story

Steven Libby [1]

[1]Department of Computer Science
Portland State University

March 16, 2022

Imperative History

Remember: computers are intricately organized sand.

Imperative History

Remember: computers are intricately organized sand.

The goal of PL is to help us organize our thoughts.

Imperative History

Remember: computers are intricately organized sand.

The goal of PL is to help us organize our thoughts.

- First we programmed with tubes

Imperative History

Remember: computers are intricately organized sand.

The goal of PL is to help us organize our thoughts.

- First we programmed with tubes
- Then we moved on to punch cards
  - no more thinking about electronics

Imperative History

Remember: computers are intricately organized sand.

The goal of PL is to help us organize our thoughts.

- First we programmed with tubes
- Then we moved on to punch cards
  - no more thinking about electronics
- Then we got assembly language
  - we can now put instructions together ourselves

Imperative History

Remember: computers are intricately organized sand.

The goal of PL is to help us organize our thoughts.

- First we programmed with tubes
- Then we moved on to punch cards
    - no more thinking about electronics
- Then we got assembly language
    - we can now put instructions together ourselves
- Finally we made FORTRAN
    - machine independent
    - we can write expressions
    - we can call functions

Imperative History

Remember: computers are intricately organized sand.

The goal of PL is to help us organize our thoughts.

- First we programmed with tubes
- Then we moved on to punch cards
  - no more thinking about electronics
- Then we got assembly language
  - we can now put instructions together ourselves
- Finally we made FORTRAN
  - machine independent
  - we can write expressions
  - we can call functions
- This paved the way for structured programming.

Imperative Programming

It's hard to overstate how pervasive structured programming is.

Imperative Programming

It's hard to overstate how pervasive structured programming is.

The core ideas:

- control flows from one statement to the next.

Imperative Programming

It's hard to overstate how pervasive structured programming is.

The core ideas:

- control flows from one statement to the next.
- branching control flow with `if` and `while`

Imperative Programming

It's hard to overstate how pervasive structured programming is.

The core ideas:

- control flows from one statement to the next.
- branching control flow with `if` and `while`
- intermediate values are stored in variables

Imperative Programming

It's hard to overstate how pervasive structured programming is.

The core ideas:

- control flows from one statement to the next.
- branching control flow with `if` and `while`
- intermediate values are stored in variables
- pass control with a function call

Imperative Programming

It's hard to overstate how pervasive structured programming is.

The core ideas:

- control flows from one statement to the next.
- branching control flow with `if` and `while`
- intermediate values are stored in variables
- pass control with a function call

There's no reason we have to program this way,
but I'll bet you still would, even in assembly.

Imperative Programming

It's hard to overstate how pervasive structured programming is.

The core ideas:

- control flows from one statement to the next.
- branching control flow with `if` and `while`
- intermediate values are stored in variables
- pass control with a function call

There's no reason we have to program this way,
but I'll bet you still would, even in assembly.

So, why?

Imperative Programming

The point is,
limiting what we *can* do with programs
makes us better programmers.

Imperative Programming

The point is,
limiting what we *can* do with programs
makes us better programmers.

Just because you can, doesn't mean you should.

Imperative Programming

The point is,
limiting what we *can* do with programs
makes us better programmers.

Just because you can, doesn't mean you should.

I want you to remember that.
Because for the rest of this lecture, I'm taking away side effects.

Imperative Programming

The point is,
limiting what we *can* do with programs
makes us better programmers.

Just because you can, doesn't mean you should.

I want you to remember that.
Because for the rest of this lecture, I'm taking away side effects.

That means:

- No reassigning variables

Imperative Programming

The point is,
limiting what we *can* do with programs
makes us better programmers.

Just because you can, doesn't mean you should.

I want you to remember that.
Because for the rest of this lecture, I'm taking away side effects.

That means:

- No reassigning variables
- No writing to memory

Imperative Programming

The point is,
limiting what we *can* do with programs
makes us better programmers.

Just because you can, doesn't mean you should.

I want you to remember that.
Because for the rest of this lecture, I'm taking away side effects.

That means:

- No reassigning variables
- No writing to memory

Once You've made an object, you can't change it.

Imperative Programming

The point is,
limiting what we *can* do with programs
makes us better programmers.

Just because you can, doesn't mean you should.

I want you to remember that.
Because for the rest of this lecture, I'm taking away side effects.

That means:

- No reassigning variables
- No writing to memory

Once You've made an object, you can't change it.
Have fun!

Functional Programming

Ok, so why do this?

Functional Programming

Ok, so why do this?

Variables are kind of the core of programming.
How do I even program without them?

Functional Programming

Ok, so why do this?

Variables are kind of the core of programming.
How do I even program without them?

The answer is that we're not just giving up variables.
We're shifting our focus.

Functional Programming

Ok, so why do this?

Variables are kind of the core of programming.
How do I even program without them?

The answer is that we're not just giving up variables.
We're shifting our focus.

Arguably one of the most important developments with imperative programming was the introduction of functions.

Functional Programming

Ok, so why do this?

Variables are kind of the core of programming.
How do I even program without them?

The answer is that we're not just giving up variables.
We're shifting our focus.

Arguably one of the most important developments with imperative programming was the introduction of functions.

In some form or another functions are the basis for all of our modular programming techniques.

Functional Programming

Instead of focusing on what we lose with variables,
I want to focus on what we gain with functions.

Functional Programming

Instead of focusing on what we lose with variables,
I want to focus on what we gain with functions.

So, lets look at the functional language Haskell.

Functional Programming

Instead of focusing on what we lose with variables,
I want to focus on what we gain with functions.

So, lets look at the functional language Haskell.

Example

```
square :: Int -> Int
square x = x * x

isPalindrome :: String -> Bool
isPalindrome str = reverse str == str
```

Functional Syntax

Example

```
square :: Int -> Int
square x = x * x

isPalindrome :: String -> Bool
isPalindrome str = reverse str == str
```

Things to note:

- Functions are defined with equations.

Functional Syntax

Example

```
square :: Int -> Int
square x = x * x

isPalindrome :: String -> Bool
isPalindrome str = reverse str == str
```

Things to note:

- Functions are defined with equations.
- The type of a function is math style, $f : \mathbb{R} \to \mathbb{R}$.

Functional Syntax

Example

```
square :: Int -> Int
square x = x * x

isPalindrome :: String -> Bool
isPalindrome str = reverse str == str
```

Things to note:

- Functions are defined with equations.
- The type of a function is math style, $f : \mathbb{R} \to \mathbb{R}$.
- Function calls are juxtaposition $f\ x$.
    - instead of parenthesis $f(x)$

Functional Syntax

Example

```
square :: Int -> Int
square x = x * x

isPalindrome :: String -> Bool
isPalindrome str = reverse str == str
```

Things to note:

- Functions are defined with equations.
- The type of a function is math style, $f : \mathbb{R} \to \mathbb{R}$.
- Function calls are juxtaposition $f\ x$.
  - instead of parenthesis $f(x)$
- There are no statements

Functional Syntax

Great, what about flow of control?

Functional Syntax

Great, what about flow of control?

We can use guards and recursion.

Functional Syntax

Great, what about flow of control?

We can use guards and recursion.

Let's count the digits in a number.

Functional Syntax

Great, what about flow of control?

We can use guards and recursion.

Let's count the digits in a number.
If n is less than 10, then it has 1 digit.

Functional Syntax

Great, what about flow of control?

We can use guards and recursion.

Let's count the digits in a number.
If n is less than 10, then it has 1 digit.
If n is greater than 10, it has 1 digit, plus the digits of $n/10$.

Functional Syntax

Great, what about flow of control?

We can use guards and recursion.

Let's count the digits in a number.
If n is less than 10, then it has 1 digit.
If n is greater than 10, it has 1 digit, plus the digits of $n/10$.

Example

```
ilog :: Int -> Int
ilog n
 | n < 10     = 1
 | otherwise = 1 + ilog (div n 10)
```

Higher Order Functions

Ok, so we have weird syntax.
What makes this *functional* programming?

Ok, so we have weird syntax.
What makes this *functional* programming?

There are two features at the heart of functional programming:

Higher Order Functions

Ok, so we have weird syntax.
What makes this *functional* programming?

There are two features at the heart of functional programming:

- We can pass functions as arguments.

Higher Order Functions

Ok, so we have weird syntax.
What makes this *functional* programming?

There are two features at the heart of functional programming:

- We can pass functions as arguments.
- We can return function from other functions.

Higher Order Functions

Ok, so we have weird syntax.
What makes this *functional* programming?

There are two features at the heart of functional programming:

- We can pass functions as arguments.
- We can return function from other functions.

This is really it.

Higher Order Functions

Ok, so we have weird syntax.
What makes this *functional* programming?

There are two features at the heart of functional programming:

- We can pass functions as arguments.
- We can return function from other functions.

This is really it.

This isn't even unique to function programming.
Python, C++, Java, Javascript, and C# all have these features.

Higher Order Functions

A function that can take and return functions is a *higher order function*.

A function that can take and return functions is a *higher order function*.

This isn't new, you've all used these.

$$\frac{d}{dx}(f) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}$$

Higher Order Functions

A function that can take and return functions is a *higher order function*.

This isn't new, you've all used these.

$$\frac{d}{dx}(f) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}$$

$\frac{d}{dx}$ is a function that takes f and returns the derivative.

Higher Order functions

There's one very important "glue" function in haskell

Example

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

This is function composition.

Higher Order functions

There's one very important "glue" function in haskell

Example

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

This is function composition.

It's supposed to resemble $f \circ g$ from math.
$f \circ g(x) = f(g(x))$

Lists

So, how do we use the ideas from functional programming?

Lists

So, how do we use the ideas from functional programming?

Let's look at a slightly more complicated example involving more complex data.

Lists

So, how do we use the ideas from functional programming?

Let's look at a slightly more complicated example involving more complex data.

A list in Haskell is a sequence of elements

Example

```
numbers :: [Int]
numbers = [1,2,3,4,5]

strings :: [String]
strings = ["The", "quick", "brown", "fox"]

booleans :: [Bool]
booleans = [True, False, False, True]
```

Lists

There are a lot of functions on lists, but let's look at a few helpful ones.

Example

```
reverse :: [a] -> [a]

reverse [1,2,3,4,5] == [5,4,3,2,1]
reverse ["The", "quick", "brown", "fox"]
     == ["fox", "brown", "quick", "The"]
```

Lists

There are a lot of functions on lists, but let's look at a few helpful ones.

Example

```
groupBy :: Int -> [a] -> [[a]]

groupBy 2 [1,2,3,4,5,6] == [[1,2], [3,4], [5,6]]
groupBy 3 [1,2,3,4,5,6] == [[1,2,3], [4,5,6]]
```

Lists

There are a lot of functions on lists, but let's look at a few helpful ones.

Example

```
intercalate :: [a] -> a -> [a]

intercalate 0 [1,2,3,4,5,6] == [1,0,2,0,3,0,4,0,5,0]
intercalate " " ["add","spaces","to","words"]
             == ["add"," ","spaces"," ","to"," ","words"]
```

Lists

There are a lot of functions on lists, but let's look at a few helpful ones.

Example

```
concat :: [[a]] -> [a]

concat [[1,0,0],[0,1,0],[0,0,1]] == [1,0,0,0,1,0,0,0,1]
concat ["join ", "these ", "strings"]
    == "join these strings"
```

Adding Commas

So, why functional programming?

Let's solve a problem.
I have an integer `n`.
I want to convert it to a string, and add commas every 3 digits.

Example

input: `3141598`
output: `"3,141,598`

input: `1189998819991197253`
input: `"1,189,998,819,991,197,253"`

Adding Commas in C?

Here's my algorithm:

Adding Commas in C?

Here's my algorithm:

- read the number into a string

Adding Commas in C?

Here's my algorithm:

- read the number into a string
- get the size of the string

Adding Commas in C?

Here's my algorithm:

- read the number into a string
- get the size of the string
- calculate the size of the string with commas

Adding Commas in C?

Here's my algorithm:

- read the number into a string
- get the size of the string
- calculate the size of the string with commas
- allocate memory

Adding Commas in C?

Here's my algorithm:

- read the number into a string
- get the size of the string
- calculate the size of the string with commas
- allocate memory
- loop backwards over the string

Adding Commas in C?

Here's my algorithm:

- read the number into a string
- get the size of the string
- calculate the size of the string with commas
- allocate memory
- loop backwards over the string
    - move the character from the number string to the comma string

Adding Commas in C?

Here's my algorithm:

- read the number into a string
- get the size of the string
- calculate the size of the string with commas
- allocate memory
- loop backwards over the string
  - move the character from the number string to the comma string
  - if this is the 3^{rd} character, then add a comma

Adding Commas in C?

Here's my algorithm:

- read the number into a string
- get the size of the string
- calculate the size of the string with commas
- allocate memory
- loop backwards over the string
  - move the character from the number string to the comma string
  - if this is the 3^{rd} character, then add a comma
- return the string

Adding Commas in C

```
char* addCommas(int n)
{
    char numStr[50];
    sprintf(numStr, "%d", n);
    int numLen = strlen(numStr);

    int bufLen = numLen + numLen/3 - (numLen%3 == 0);
    char* commaStr = (char*)malloc((bufLen+1)*sizeof(char));
    ...
}
```

Adding Commas in C

```
char* addCommas(int n)
{
    ...
    int bufi = bufLen - 1;
    int addComma = 0;
    for(int i = numLen-1; i >= 0; i--)
    {
        commaStr[bufi--] = numStr[i];
        addComma = (addComma + 1) % 3;
        if(i > 0 && addComma == 0)
        {
            commaStr[bufi--] = ',';
        }
    }
    return commaStr;
}
```

Adding commas in Haskell

Ok, let's look at a functional approach.

Here's my algorithm:

Adding commas in Haskell

Ok, let's look at a functional approach.

Here's my algorithm:

- read the number into a string

Adding commas in Haskell

Ok, let's look at a functional approach.

Here's my algorithm:

- read the number into a string
- reverse the list

Adding commas in Haskell

Ok, let's look at a functional approach.

Here's my algorithm:

- read the number into a string
- reverse the list
- group the numbers into chunks of 3

Adding commas in Haskell

Ok, let's look at a functional approach.

Here's my algorithm:

- read the number into a string
- reverse the list
- group the numbers into chunks of 3
- put commas inbetween the groups of 3

Adding commas in Haskell

Ok, let's look at a functional approach.

Here's my algorithm:

- read the number into a string
- reverse the list
- group the numbers into chunks of 3
- put commas inbetween the groups of 3
- concatenate the numbers

Adding commas in Haskell

Ok, let's look at a functional approach.

Here's my algorithm:

- read the number into a string
- reverse the list
- group the numbers into chunks of 3
- put commas inbetween the groups of 3
- concatenate the numbers
- reverse the list again

Adding commas in Haskell

```
addCommas :: Int -> String
addCommas = reverse
          . concat
          . intersperse ","
          . groupBy 3
          . reverse
          . show
```

Summary: Why do we care?

So, why do we care about functional programming?

Summary: Why do we care?

So, why do we care about functional programming?

- when it's this easy to write and compose functions, it's much easier to split large projects up.

Summary: Why do we care?

So, why do we care about functional programming?

- when it's this easy to write and compose functions, it's much easier to split large projects up.
- If there's no side effects, then I never have to worry about someone else modifying global state.

Summary: Why do we care?

So, why do we care about functional programming?

- when it's this easy to write and compose functions, it's much easier to split large projects up.
- If there's no side effects, then I never have to worry about someone else modifying global state.
- many of these functions can be run in parallel.

Summary: Why do we care?

So, why do we care about functional programming?

- when it's this easy to write and compose functions, it's much easier to split large projects up.
- If there's no side effects, then I never have to worry about someone else modifying global state.
- many of these functions can be run in parallel.
- It gives you a new way of looking at problems.

Summary: Why do we care?

So, why do we care about functional programming?

- when it's this easy to write and compose functions, it's much easier to split large projects up.
- If there's no side effects, then I never have to worry about someone else modifying global state.
- many of these functions can be run in parallel.
- It gives you a new way of looking at problems.

You may not use functional languages often, but learning them will change how you approach programming.