The Principles of Functional Programming

Panicz Maciej Godek

godek.maciek@gmail.com

https://github.com/panicz/writings/tree/master/talks/datamass

datamass.io summit, 29.09.2017

The goals of the talk

- explain what Functional Programming is
- expose some common confusion
- debunk some widespread myths
- show the value and applicability of FP
- and fallacies that arise when **not using FP**

The goals of the talk

- explain what Functional Programming is
- expose some common confusion
- debunk some widespread myths
- show the value and applicability of FP
- and fallacies that arise when **not using FP**

The goals of the talk

- explain what Functional Programming is
- expose some common confusion
- debunk some widespread myths
- show the value and applicability of FP
- and fallacies that arise when **not using FP**

The goals of the talk

- explain what Functional Programming is
- expose some common confusion
- debunk some widespread myths
- show the value and applicability of FP
- and fallacies that arise when **not using FP**

The goals of the talk

- explain what Functional Programming is
- expose some common confusion
- debunk some widespread myths
- show the value and applicability of FP
- and fallacies that arise when **not using FP**

The goals of the talk

- explain what Functional Programming is
- expose some common confusion
- debunk some widespread myths
- show the value and applicability of FP
- and fallacies that arise when **not using FP**

What is functional programming?

Myth #1: Functional programming isn't well defined.

> ***functional programming*** *– programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data*

`https://en.wikipedia.org/wiki/Functional_programming`

What is functional programming?

Myth #1: Functional programming isn't well defined.

> ***functional programming*** *– programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data*

`https://en.wikipedia.org/wiki/Functional_programming`

What is functional programming?

Myth #1: Functional programming isn't well defined.

> ***functional programming*** *– programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data*

`https://en.wikipedia.org/wiki/Functional_programming`

Function vs. procedure

procedure *– a sequence of instructions that show how to achieve some result, such as to prepare or make something*

https://en.wikipedia.org/wiki/Procedure

```
int main(void) {
  printf("Hello world!\n");
  return 0;
}
```

Function vs. procedure

procedure *– a sequence of instructions that show how to achieve some result, such as to prepare or make something*

`https://en.wikipedia.org/wiki/Procedure`

```
int main(void) {
  printf("Hello world!\n");
  return 0;
}
```

Function vs. procedure

procedure *– a sequence of instructions that show how to achieve some result, such as to prepare or make something*

`https://en.wikipedia.org/wiki/Procedure`

```
int main(void) {
  printf("Hello world!\n");
  return 0;
}
```

Procedure – real life example

Function (in mathematical sense)

function *– a relation that associates an input to a single output according to some rule*

https://en.wikipedia.org/wiki/Function

```
int square(int x) {
  return x * x;
}
```

(a procedure can implement a function)

Function (in mathematical sense)

function *– a relation that associates an input to a single output according to some rule*

https://en.wikipedia.org/wiki/Function

```
int square(int x) {
  return x * x;
}
```

(a procedure can implement a function)

Function (in mathematical sense)

function *– a relation that associates an input to a single output according to some rule*

https://en.wikipedia.org/wiki/Function

```
int square(int x) {
  return x * x;
}
```

(a procedure can implement a function)

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Functions and procedures in standard C library

functions

```
tolower
isdigit
strlen
strcmp
sqrt
+ - * / < == >
```

procedures

```
printf
scanf
memcpy
clock
rand
++ -- = += *=
```

Lambdas

Myth #2: functional programming is about using *lambdas* or *closures* or *higher-order functions/procedures*

Lambda, Λ, λ *– is the 11th letter of the Greek alphabet*

https://en.wikipedia.org/wiki/Lambda

```
function make_counter() {
  var counter = 0;
  return function() {
    return ++counter;
  };
}
```

Lambdas

Myth #2: functional programming is about using *lambdas* or *closures* or *higher-order functions/procedures*

Lambda, *Λ, λ – is the 11th letter of the Greek alphabet*

https://en.wikipedia.org/wiki/Lambda

```
function make_counter() {
  var counter = 0;
  return function() {
    return ++counter;
  };
}
```

Lambdas

Myth #2: functional programming is about using *lambdas* or *closures* or *higher-order functions/procedures*

Lambda, Λ, λ – *is the 11th letter of the Greek alphabet*

https://en.wikipedia.org/wiki/Lambda

```
function make_counter() {
  var counter = 0;
  return function() {
    return ++counter;
  };
}
```

Lambdas

Myth #2: functional programming is about using *lambdas* or *closures* or *higher-order functions/procedures*

Lambda, Λ, λ *– is the 11th letter of the Greek alphabet*

https://en.wikipedia.org/wiki/Lambda

```
function make_counter() {
  var counter = 0;
  return function() {
    return ++counter;
  };
}
```

Parts of speech

Myth #3: objects are like nouns and functions are like verbs

```
insertions::  a -> [a] -> [[a]]
insertions x []     = [[x]]
insertions x (h:t) = [(x:h:t)] ++ entwined
  where entwined = (map (h:) (insertions x t))

e.g. insertions 0 [1,2,3] ===>
  [[0,1,2,3],[1,0,2,3],[1,2,0,3],[1,2,3,0]]
```

other examples: `powerset`, `permutations`, `sorted`

Parts of speech

Myth #3: objects are like nouns and functions are like verbs

```
insertions::  a -> [a] -> [[a]]
insertions x []     = [[x]]
insertions x (h:t) = [(x:h:t)] ++ entwined
  where entwined = (map (h:) (insertions x t))

e.g. insertions 0 [1,2,3] ===>
  [[0,1,2,3],[1,0,2,3],[1,2,0,3],[1,2,3,0]]
```

other examples: `powerset`, `permutations`, `sorted`

Parts of speech

Myth #3: objects are like nouns and functions are like verbs

```
insertions::  a -> [a] -> [[a]]
insertions x []     = [[x]]
insertions x (h:t) = [(x:h:t)] ++ entwined
  where entwined = (map (h:) (insertions x t))
```

```
e.g. insertions 0 [1,2,3] ===>
  [[0,1,2,3],[1,0,2,3],[1,2,0,3],[1,2,3,0]]
```

other examples: `powerset`, `permutations`, `sorted`

Parts of speech

Myth #3: objects are like nouns and functions are like verbs

```
insertions::  a -> [a] -> [[a]]
insertions x []     = [[x]]
insertions x (h:t) = [(x:h:t)] ++ entwined
  where entwined = (map (h:) (insertions x t))

e.g. insertions 0 [1,2,3] ===>
  [[0,1,2,3],[1,0,2,3],[1,2,0,3],[1,2,3,0]]
```

other examples: `powerset`, `permutations`, `sorted`

Parts of speech

Myth #3: objects are like nouns and functions are like verbs

```
insertions::  a -> [a] -> [[a]]
insertions x []     = [[x]]
insertions x (h:t) = [(x:h:t)] ++ entwined
  where entwined = (map (h:) (insertions x t))
```

```
e.g. insertions 0 [1,2,3] ===>
  [[0,1,2,3],[1,0,2,3],[1,2,0,3],[1,2,3,0]]
```

other examples: `powerset`, `permutations`, `sorted`

Imperative functional programming

Myth #4: (pure) functions cannot be implemented in non-functional (imperative) way

```
int factorial(int n) {
  int result = 1;
  while (n > 0)
    result *= n--;
  return result;
}
```

```
int factorial(int n) {
  if (n < 1) return 1;
  else return
    n*factorial(n-1);
}
```

Imperative functional programming

Myth #4: (pure) functions cannot be implemented in non-functional (imperative) way

```
int factorial(int n) {
  int result = 1;
  while (n > 0)
    result *= n--;
  return result;
}
```

```
int factorial(int n) {
  if (n < 1) return 1;
  else return
    n*factorial(n-1);
}
```

Imperative functional programming

Myth #4: (pure) functions cannot be implemented in non-functional (imperative) way

```
int factorial(int n) {
  int result = 1;
  while (n > 0)
    result *= n--;
  return result;
}
```

```
int factorial(int n) {
  if (n < 1) return 1;
  else return
    n*factorial(n-1);
}
```

Imperative functional programming

Myth #4: (pure) functions cannot be implemented in non-functional (imperative) way

```
int factorial(int n) {
  int result = 1;
  while (n > 0)
    result *= n--;
  return result;
}
```

```
int factorial(int n) {
  if (n < 1) return 1;
  else return
    n*factorial(n-1);
}
```

Why functional programming?

Myth #5: you need to understand Category Theory to use functional programming

Imperative programs explain how to **do** things (perform *actions*), while functional programs **mean** things (refer to *objects*).

Why functional programming?

Myth #5: you need to understand Category Theory to use functional programming

Imperative programs explain how to **do** things (perform *actions*), while functional programs **mean** things (refer to *objects*).

Why functional programming?

Myth #5: you need to understand Category Theory to use functional programming

Imperative programs explain how to **do** things (perform *actions*), while functional programs **mean** things (refer to *objects*).

Sum of squares of initial 7 prime numbers (procedural)

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

After its execution, the `sum` variable contains the desired value

Sum of squares of initial 7 prime numbers (procedural)

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

After its execution, the `sum` variable contains the desired value

Sum of squares of initial 7 prime numbers (procedural)

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

After its execution, the `sum` variable contains the desired value

Sum of squares of initial 7 prime numbers (procedural)

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

After its execution, the `sum` variable contains the desired value

Sum of squares of initial 7 prime numbers (procedural)

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

After its execution, the `sum` variable contains the desired value

Sum of squares of initial 7 prime numbers (procedural)

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

After its execution, the `sum` variable contains the desired value

Sum of squares of initial 7 prime numbers (procedural)

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

After its execution, the `sum` variable contains the desired value

Sum of squares of initial 7 prime numbers (procedural)

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

After its execution, the `sum` variable contains the desired value

Sum of squares of initial 7 prime numbers (procedural)

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

After its execution, the `sum` variable contains the desired value

Sum of squares of initial 7 prime numbers (procedural)

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

After its execution, the `sum` variable contains the desired value

Sum of squares of initial 7 prime numbers (functional)

```
(sum (map square
          (initial 7 (only prime? numbers))))
```

(no need to tell where to look for the result)

Sum of squares of initial 7 prime numbers (functional)

```
(sum (map square
          (initial 7 (only prime? numbers))))
```

(no need to tell where to look for the result)

Sum of squares of initial 7 prime numbers (functional)

```
(sum (map square
          (initial 7 (only prime? numbers))))
```

(no need to tell where to look for the result)

Sum of squares of initial 7 prime numbers (definitions)

```
(define (numbers-from n)
  `(,n . ,(numbers-from (+ n 1))))

(define numbers (numbers-from 0))

(define (only qualifying? elements)
  (match elements
    ('()
     '())
    (`(,first . ,rest)
     (if (qualifying? first)
         `(,first . ,(only qualifying? rest))
         (only qualifying? rest)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (numbers-from n)
  `(,n . ,(numbers-from (+ n 1))))

(define numbers (numbers-from 0))

(define (only qualifying? elements)
  (match elements
    ('()
     '())
    (`(,first . ,rest)
     (if (qualifying? first)
         `(,first . ,(only qualifying? rest))
         (only qualifying? rest)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (numbers-from n)
  `(,n . ,(numbers-from (+ n 1))))

(define numbers (numbers-from 0))

(define (only qualifying? elements)
  (match elements
    ('()
     '())
    (`(,first . ,rest)
     (if (qualifying? first)
         `(,first . ,(only qualifying? rest))
         (only qualifying? rest)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (numbers-from n)
  `(,n . ,(numbers-from (+ n 1))))

(define numbers (numbers-from 0))

(define (only qualifying? elements)
  (match elements
    ('()
     '())
    (`(,first . ,rest)
     (if (qualifying? first)
         `(,first . ,(only qualifying? rest))
         (only qualifying? rest)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (numbers-from n)
  `(,n . ,(numbers-from (+ n 1))))

(define numbers (numbers-from 0))

(define (only qualifying? elements)
  (match elements
    ('()
     '())
    (`(,first . ,rest)
     (if (qualifying? first)
         `(,first . ,(only qualifying? rest))
         (only qualifying? rest)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (numbers-from n)
  `(,n . ,(numbers-from (+ n 1))))

(define numbers (numbers-from 0))

(define (only qualifying? elements)
  (match elements
    ('()
     '())
    (`(,first . ,rest)
     (if (qualifying? first)
         `(,first . ,(only qualifying? rest))
         (only qualifying? rest)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (numbers-from n)
  `(,n . ,(numbers-from (+ n 1))))

(define numbers (numbers-from 0))

(define (only qualifying? elements)
  (match elements
    ('()
     '())
    (`(,first . ,rest)
     (if (qualifying? first)
         `(,first . ,(only qualifying? rest))
         (only qualifying? rest)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (numbers-from n)
  `(,n . ,(numbers-from (+ n 1))))

(define numbers (numbers-from 0))

(define (only qualifying? elements)
  (match elements
    ('()
     '())
    (`(,first . ,rest)
     (if (qualifying? first)
         `(,first . ,(only qualifying? rest))
       (only qualifying? rest)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (numbers-from n)
  `(,n . ,(numbers-from (+ n 1))))

(define numbers (numbers-from 0))

(define (only qualifying? elements)
  (match elements
    ('()
     '())
    (`(,first . ,rest)
     (if (qualifying? first)
         `(,first . ,(only qualifying? rest))
       (only qualifying? rest)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (initial n elements)
  (if (= n 0)
      '()
    (let ((`(,first . ,rest) elements))
      `(,first . ,(initial (- n 1) rest)))))

(define (sum numbers)
  (match numbers
    ('()
     0)
    (`(,number . ,other-numbers)
     (+ number (sum other-numbers)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (initial n elements)
  (if (= n 0)
      '()
      (let ((`(,first . ,rest) elements))
        `(,first . ,(initial (- n 1) rest)))))

(define (sum numbers)
  (match numbers
    ('()
     0)
    (`(,number . ,other-numbers)
     (+ number (sum other-numbers)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (initial n elements)
  (if (= n 0)
      '()
    (let ((`(,first . ,rest) elements))
      `(,first . ,(initial (- n 1) rest)))))

(define (sum numbers)
  (match numbers
    ('()
     0)
    (`(,number . ,other-numbers)
     (+ number (sum other-numbers)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (initial n elements)
  (if (= n 0)
      '()
    (let ((`(,first . ,rest) elements))
       `(,first . ,(initial (- n 1) rest)))))

(define (sum numbers)
  (match numbers
    ('()
     0)
    (`(,number . ,other-numbers)
     (+ number (sum other-numbers)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (initial n elements)
  (if (= n 0)
      '()
    (let ((`(,first . ,rest) elements))
       `(,first . ,(initial (- n 1) rest)))))

(define (sum numbers)
  (match numbers
    ('()
     0)
    (`(,number . ,other-numbers)
     (+ number (sum other-numbers)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (initial n elements)
  (if (= n 0)
      '()
    (let ((`(,first . ,rest) elements))
       `(,first . ,(initial (- n 1) rest)))))

(define (sum numbers)
  (match numbers
    ('()
     0)
    (`(,number . ,other-numbers)
     (+ number (sum other-numbers)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (initial n elements)
  (if (= n 0)
      '()
    (let ((`(,first . ,rest) elements))
       `(,first . ,(initial (- n 1) rest)))))

(define (sum numbers)
  (match numbers
    ('()
     0)
    (`(,number . ,other-numbers)
     (+ number (sum other-numbers)))))
```

Sum of squares of initial 7 prime numbers (definitions)

```
(define (initial n elements)
  (if (= n 0)
      '()
    (let ((`(,first . ,rest) elements))
       `(,first . ,(initial (- n 1) rest)))))

(define (sum numbers)
  (match numbers
    ('()
     0)
    (`(,number . ,other-numbers)
     (+ number (sum other-numbers)))))
```

The substitution model of computation

```
(sum
  (map square
     (initial 7
        (only prime? numbers))))
```

The substitution model of computation

```
(sum
  (map square
    (initial 7
      (only prime? '(0 1 2 3 4 5 6 ...)))))
```

The substitution model of computation

```
(sum
  (map square
    (initial 7
      (only prime? '(0 1 2 3 4 5 6 ...)))))
```

The substitution model of computation

```
(sum
  (map square
     (initial 7
        '(2 3 5 7 11 13 19 23 29 31 37 ...))))
```

The substitution model of computation

```
(sum
  (map square
     (initial 7
        '(2 3 5 7 11 13 19 23 29 31 37 ...))))
```

The substitution model of computation

```
(sum
   (map square
      '(2 3 5 7 11 13 19)))
```

The substitution model of computation

```
(sum
   (map square
      '(2 3 5 7 11 13 19)))
```

The substitution model of computation

```
(sum
  '(4 9 25 49 121 169 361))
```

The substitution model of computation

```
(sum
  '(4 9 25 49 121 169 361))
```

Binding vs. assignment

Myth #6: binding and assignment are the same thing

- **definition** – creates a new binding in the current scope

```
(define variable value)
```

- **assignment** – changes the value bound by a variable in the current scope

```
(set! variable new-value)
```

- **binding** – creates a new scope, possibly shadowing some existing binding

```
(let ((variable some-value))
  ;; any outer binding of `variable` is shadowed here
  ...)
;; outer context - 'variable' has its original value
```

Binding vs. assignment

Myth #6: binding and assignment are the same thing

- **definition** – creates a new binding in the current scope

```
(define variable value)
```

- **assignment** – changes the value bound by a variable in the current scope

```
(set! variable new-value)
```

- **binding** – creates a new scope, possibly shadowing some existing binding

```
(let ((variable some-value))
  ;; any outer binding of `variable` is shadowed here
  ...)
;; outer context - 'variable' has its original value
```

Binding vs. assignment

Myth #6: binding and assignment are the same thing

- **definition** – creates a new binding in the current scope

```
(define variable value)
```

- **assignment** – changes the value bound by a variable in the current scope

```
(set! variable new-value)
```

- **binding** – creates a new scope, possibly shadowing some existing binding

```
(let ((variable some-value))
  ;; any outer binding of `variable` is shadowed here
  ...)
;; outer context - 'variable' has its original value
```

Binding vs. assignment

Myth #6: binding and assignment are the same thing

- **definition** – creates a new binding in the current scope

```
(define variable value)
```

- **assignment** – changes the value bound by a variable in the current scope

```
(set! variable new-value)
```

- **binding** – creates a new scope, possibly shadowing some existing binding

```
(let ((variable some-value))
  ;; any outer binding of `variable` is shadowed here
  ...)
;; outer context - 'variable' has its original value
```

Binding vs. assignment

Myth #6: binding and assignment are the same thing

- **definition** – creates a new binding in the current scope

```
(define variable value)
```

- **assignment** – changes the value bound by a variable in the current scope

```
(set! variable new-value)
```

- **binding** – creates a new scope, possibly shadowing some existing binding

```
(let ((variable some-value))
  ;; any outer binding of `variable` is shadowed here
  ...)
;; outer context - 'variable' has its original value
```

Binding vs. assignment

Myth #6: binding and assignment are the same thing

- **definition** – creates a new binding in the current scope
  `(define variable value)`
- **assignment** – changes the value bound by a variable in the current scope
  `(set! variable new-value)`
- **binding** – creates a new scope, possibly shadowing some existing binding

```
(let ((variable some-value))
  ;; any outer binding of `variable` is shadowed here
  ...)
;; outer context - 'variable' has its original value
```

Binding vs. assignment

Myth #6: binding and assignment are the same thing

- **definition** – creates a new binding in the current scope
  `(define variable value)`
- **assignment** – changes the value bound by a variable in the current scope
  `(set! variable new-value)`
- **binding** – creates a new scope, possibly shadowing some existing binding

```
(let ((variable some-value))
  ;; any outer binding of `variable` is shadowed here
  ...)
;; outer context - 'variable' has its original value
```

Binding vs. assignment

Myth #6: binding and assignment are the same thing

- **definition** – creates a new binding in the current scope

```
(define variable value)
```

- **assignment** – changes the value bound by a variable in the current scope

```
(set! variable new-value)
```

- **binding** – creates a new scope, possibly shadowing some existing binding

```
(let ((variable some-value))
  ;; any outer binding of `variable` is shadowed here
  ...)
;; outer context - 'variable' has its original value
```

Binding vs. assignment

Myth #6: binding and assignment are the same thing

- **definition** – creates a new binding in the current scope

```
(define variable value)
```

- **assignment** – changes the value bound by a variable in the current scope

```
(set! variable new-value)
```

- **binding** – creates a new scope, possibly shadowing some existing binding

```
(let ((variable some-value))
  ;; any outer binding of `variable` is shadowed here
  ...)
;; outer context - 'variable' has its original value
```

Immutability

How can you change the meaning of a word?

- by explaining the new meaning to everyone
- by modifying the structure of the brains of all the people

What about the previous usages of that word?

Immutability

How can you change the meaning of a word?

- by explaining the new meaning to everyone
- by modifying the structure of the brains of all the people

What about the previous usages of that word?

Immutability

How can you change the meaning of a word?

- by explaining the new meaning to everyone
- by modifying the structure of the brains of all the people

What about the previous usages of that word?

Immutability

How can you change the meaning of a word?

- by explaining the new meaning to everyone
- by modifying the structure of the brains of all the people

What about the previous usages of that word?

Immutability

How can you change the meaning of a word?

- by explaining the new meaning to everyone
- by modifying the structure of the brains of all the people

What about the previous usages of that word?

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Let's see...

```
>>> x = 5
>>> x = x + 3
>>> x
8
```

```
>>> x = 5
>>> x += 3
>>> x
8
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Let's see...

```
>>> x = 5
>>> x = x + 3
>>> x
8
```

```
>>> x = 5
>>> x += 3
>>> x
8
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Let's see...

```
>>> x = 5
>>> x = x + 3
>>> x
8
```

```
>>> x = 5
>>> x += 3
>>> x
8
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Let's see...

```
>>> x = 5

>>> x = x + 3

>>> x
8
```

```
>>> x = 5

>>> x += 3

>>> x
8
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Let's see...

```
>>> x = 5

>>> x = x + 3

>>> x
8
```

```
>>> x = 5

>>> x += 3

>>> x
8
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Let's see...

```
>>> x = 5

>>> x = x + 3

>>> x
8
```

```
>>> x = 5

>>> x += 3

>>> x
8
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Let's see...

```
>>> x = 5

>>> x = x + 3

>>> x
8
```

```
>>> x = 5

>>> x += 3

>>> x
8
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Let's see...

```
>>> x = 5

>>> x = x + 3

>>> x
8
```

```
>>> x = 5

>>> x += 3

>>> x
8
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Let's see...

```
>>> x = 5

>>> x = x + 3

>>> x
8
```

```
>>> x = 5

>>> x += 3

>>> x
8
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Let's see...

```
>>> x = 5

>>> x = x + 3

>>> x
8
```

```
>>> x = 5

>>> x += 3

>>> x
8
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>> x = 5
>>> y = x
>>> x = x + 3
>>> x
8
>>> y
5
```

```
>>> x = 5
>>> y = x
>>> x += 3
>>> x
8
>>> y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>> x = 5

>>> y = x

>>> x = x + 3

>>> x
8

>>> y
5
```

```
>>> x = 5

>>> y = x

>>> x += 3

>>> x
8

>>> y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>> x = 5

>>> y = x

>>> x = x + 3

>>> x
8

>>> y
5
```

```
>>> x = 5

>>> y = x

>>> x += 3

>>> x
8

>>> y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>> x = 5

>>> y = x

>>> x = x + 3

>>> x
8

>>> y
5
```

```
>>> x = 5

>>> y = x

>>> x += 3

>>> x
8

>>> y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>>  x = 5

>>>  y = x

>>>  x = x + 3

>>>  x
8

>>>  y
5
```

```
>>>  x = 5

>>>  y = x

>>>  x += 3

>>>  x
8

>>>  y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>> x = 5

>>> y = x

>>> x = x + 3

>>> x
8

>>> y
5
```

```
>>> x = 5

>>> y = x

>>> x += 3

>>> x
8

>>> y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>> x = 5

>>> y = x

>>> x = x + 3

>>> x
8

>>> y
5
```

```
>>> x = 5

>>> y = x

>>> x += 3

>>> x
8

>>> y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>>  x = 5

>>>  y = x

>>>  x = x + 3

>>>  x
8

>>>  y
5
```

```
>>>  x = 5

>>>  y = x

>>>  x += 3

>>>  x
8

>>>  y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>>  x = 5

>>>  y = x

>>>  x = x + 3

>>>  x
8

>>>  y
5
```

```
>>>  x = 5

>>>  y = x

>>>  x += 3

>>>  x
8

>>>  y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>>  x = 5

>>>  y = x

>>>  x = x + 3

>>>  x
8

>>>  y
5
```

```
>>>  x = 5

>>>  y = x

>>>  x += 3

>>>  x
8

>>>  y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>> x = 5
>>> y = x
>>> x = x + 3
>>> x
8
>>> y
5
```

```
>>> x = 5
>>> y = x
>>> x += 3
>>> x
8
>>> y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>> x = 5
>>> y = x
>>> x = x + 3
>>> x
8
>>> y
5
```

```
>>> x = 5
>>> y = x
>>> x += 3
>>> x
8
>>> y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>> x = 5

>>> y = x

>>> x = x + 3

>>> x
8

>>> y
5
```

```
>>> x = 5

>>> y = x

>>> x += 3

>>> x
8

>>> y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>>  x = 5

>>>  y = x

>>>  x = x + 3

>>>  x
8

>>>  y
5
```

```
>>>  x = 5

>>>  y = x

>>>  x += 3

>>>  x
8

>>>  y
5
```

Problems with mutation

Is `x += y` a shorthand for `x = x + y`?

Interference test

```
>>>  x = 5

>>>  y = x

>>>  x = x + 3

>>>  x
8

>>>  y
5
```

```
>>>  x = 5

>>>  y = x

>>>  x += 3

>>>  x
8

>>>  y
5
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>>  x = [1, 2, 3]
>>>  y = x
>>>  x = x + [4, 5]
>>>  x
[1, 2, 3, 4, 5]
>>>  y
[1, 2, 3]
```

```
>>>  x = [1, 2, 3]
>>>  y = x
>>>  x += [4, 5]
>>>  x
[1, 2, 3, 4, 5]
>>>  y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>>  x = [1, 2, 3]

>>>  y = x

>>>  x = x + [4, 5]

>>>  x

[1, 2, 3, 4, 5]

>>>  y

[1, 2, 3]
```

```
>>>  x = [1, 2, 3]

>>>  y = x

>>>  x += [4, 5]

>>>  x

[1, 2, 3, 4, 5]

>>>  y

[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>>  x = [1, 2, 3]

>>>  y = x

>>>  x = x + [4, 5]

>>>  x

[1, 2, 3, 4, 5]

>>>  y

[1, 2, 3]
```

```
>>>  x = [1, 2, 3]

>>>  y = x

>>>  x += [4, 5]

>>>  x

[1, 2, 3, 4, 5]

>>>  y

[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Spooky action at a distance

Is `x += y` a shorthand for `x = x + y`?

Sanity check

```
>>> x = [1, 2, 3]
>>> y = x
>>> x = x + [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, 3]
>>> y = x
>>> x += [4, 5]
>>> x
[1, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5] # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a':  5}

>>> y = f()

>>> y
{'a':  5} # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a':  5}

>>> y = f()

>>> y
{'a':  5} # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a':  5}

>>> y = f()

>>> y
{'a':  5} # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a':  5}

>>> y = f()

>>> y
{'a':  5} # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a':  5}

>>> y = f()

>>> y
{'a':  5} # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a': 5}

>>> y = f()

>>> y
{'a': 5} # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a':  5}

>>> y = f()

>>> y
{'a':  5} # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a':  5}

>>> y = f()

>>> y
{'a':  5} # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a':  5}

>>> y = f()

>>> y
{'a':  5} # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a':  5}

>>> y = f()

>>> y
{'a':  5} # oops!
```

Problems with mutability

```
>>> def f(x={}):
...         return x

>>> x = f()

>>> x
{}

>>> x['a'] = 5

>>> x
{'a':  5}

>>> y = f()

>>> y
{'a':  5} # oops!
```

Other examples of immutable systems

- functional package managers (NIX, Guix)
- Git
- history (can't be undone)

Other examples of immutable systems

- functional package managers (NIX, Guix)
- Git
- history (can't be undone)

Other examples of immutable systems

- functional package managers (NIX, Guix)
- Git
- history (can't be undone)

Other examples of immutable systems

- functional package managers (NIX, Guix)
- Git
- history (can’t be undone)

No silver bullets

Myth #7: Functional Programming is a silver bullet

No silver bullets

Myth #7: Functional Programming is a silver bullet

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Pros and cons

Advantages:

- code is easier to read and refactor and less prone to errors
- no control flow means more flexible interpretation
- better multicore optimization
- we don't care what the computer will do

Disadvantages:

- may cause performance penalties
- difficult to reason about resource usage
- often leads smug programmers to awkward abstractions
- we don't know what the computer will do

Thanks for your attention!

Questions?

@PaniczGodek