Functional Programming

An Introduction

Principles

- First Class and Higher Order Functions
- Pure functions
- Recursion
- Non-strict (Lazy) Evaluation
- Type Systems
- Data Structures

First Class Functions

- Functions are values
- Can be stored as values
- Can call a stored function easily

Higher Order Functions

- Takes other functions as arguments
- Returns functions as values
- Can construct functions to return

Pure Functions

- No side effects (memory or I/O)
- Referentially transparent
- Can be reordered
- Can be memoized
- Can be parallelized

Recursion

- Loops are performed with recursion
- Describes the task, rather than the mechanism
- Optimization, to avoid deep recursion causing errors

Lazy Evaluation

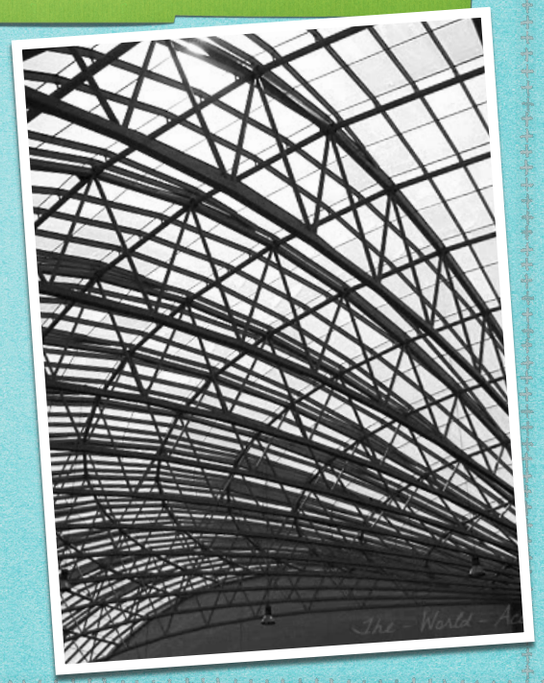
- Values are only calculated if required
- Parts of structures may be invalid
- Structures may be infinite

Type Systems

- Strong typing rejects programs that are not valid
- Type research tries to make all buggy programs invalid
- Not all Functional languages support type systems

Data Structures

- Structures cannot be modified
- Additions or removals create new structures
- Enables pure functions
- Optimized by sharing structures



Pros and Cons

Pros and Cons

- Con: Need to understand a problem much better to solve with Functional Programming
- Pro: Need to understand a problem much better to solve with Functional Programming

Cons

- Efficiency in memory
- Immutable structures are more complex and use more memory
  (but more efficient than most expect)

Cons

- Coming from Imperative Programming is hard
- Requires thinking about solutions very differently
  But with enough practice it becomes natural.

Cons

- Coming from Imperative Programming is hard
- Requires thinking about solutions very differently
  But with enough practice it becomes natural.
  Imperative programming is "Icky"

Cons

- Less Common
- Fewer libraries

Cons

- Less Common
- Fewer libraries

Some systems build on existing infrastructure, allowing for external libraries

Pros

- Less bug prone
- Programs can be “proven” correct

More Pros

- Less code writing
- Less complexity
- Naturally thread safe
- Automatic optimizations
- Refactoring can be much less work

Best Pro

Functional Programming is fun! 🎉🎈

Code!

Functions as Values: JavaScript

```
function add(x, y) {
  return x + y
}
a = add
```

Set “a” to a function called “add”
- Can apply “a” to 2 and 3
- Alternative syntax: set “b” to an anonymous function
- Can apply “b” the same way

Functions as Values: JavaScript

```
function add(x, y) {
  return x + y
}
a = add
a(2,3)
```

Set “a” to a function called “add”
- Can apply “a” to 2 and 3
- Alternative syntax: set “b” to an anonymous function
- Can apply “b” the same way

Functions as Values: JavaScript

```
function add(x, y) {
  return x + y
}
a = add
a(2,3)
⇒ 5
```

Set “a” to a function called “add”
- Can apply “a” to 2 and 3
- Alternative syntax: set “b” to an anonymous function
- Can apply “b” the same way

Functions as Values: JavaScript

```
function add(x, y) {
  return x + y
}
a = add
a(2,3)
⇒ 5

b = function(x, y) { return x + y }
b(2,3)
⇒ 5
```

Set “a” to a function called “add”
- Can apply “a” to 2 and 3
- Alternative syntax: set “b” to an anonymous function
- Can apply “b” the same way

Functions as Values: Python

```
def add(x, y):
  return x + y

a = add
```

Set “a” to a function called “add”
- Can apply “a” to 2 and 3
- Alternative syntax: set “b” to an anonymous function, also called a “lambda”
- Can apply “b” the same way

Functions as Values: Python

```
def add(x, y):
  return x + y

a = add
a(2,3)
```

Set “a” to a function called “add”
- Can apply “a” to 2 and 3
- Alternative syntax: set “b” to an anonymous function, also called a “lambda”
- Can apply “b” the same way

Functions as Values: Python

```
def add(x, y):
  return x + y

a = add
a(2,3)
⇒ 5
```

Set “a” to a function called “add”
- Can apply “a” to 2 and 3
- Alternative syntax: set “b” to an anonymous function, also called a “lambda”
- Can apply “b” the same way

Functions as Values: Python

```
def add(x, y):
  return x + y

a = add
a(2,3)
⇒ 5

b = lambda x,y: x + y
```

Set “a” to a function called “add”
- Can apply “a” to 2 and 3
- Alternative syntax: set “b” to an anonymous function, also called a “lambda”
- Can apply “b” the same way

Functions as Values: Python

```
def add(x, y):
  return x + y

a = add
a(2,3)
⇒ 5

b = lambda x,y: x + y
b(2,3)
⇒ 5
```

Set “a” to a function called “add”
- Can apply “a” to 2 and 3
- Alternative syntax: set “b” to an anonymous function, also called a “lambda”
- Can apply “b” the same way

Functions as Values: Ruby

```
def add(x, y)
    x + y
end

a = add
```

- Setting a variable to a function in Ruby leads to an error.
- However, lambdas do exist and are values.
- Unfortunately, they are called differently to functions.
- So Ruby does NOT have first class functions. There are some facilities though

Functions as Values: Ruby

```
def add(x, y)
    x + y
end

a = add ✗
```

- Setting a variable to a function in Ruby leads to an error.
- However, lambdas do exist and are values.
- Unfortunately, they are called differently to functions.
- So Ruby does NOT have first class functions. There are some facilities though

Functions as Values: Ruby

```
def add(x, y)
    x + y
end

a = add ✗
a = lambda {|x,y| add(x,y)}
```

- Setting a variable to a function in Ruby leads to an error.
- However, lambdas do exist and are values.
- Unfortunately, they are called differently to functions.
- So Ruby does NOT have first class functions. There are some facilities though

Functions as Values: Ruby

```
def add(x, y)
    x + y
end

a = add ✗
a = lambda {|x,y| add(x,y)}

a(2,3)
```

- Setting a variable to a function in Ruby leads to an error.
- However, lambdas do exist and are values.
- Unfortunately, they are called differently to functions.
- So Ruby does NOT have first class functions. There are some facilities though

Functions as Values: Ruby

```
def add(x, y)
    x + y
end

a = add ✗
a = lambda {|x,y| add(x,y)}

a(2,3) ✗
```

- Setting a variable to a function in Ruby leads to an error.
- However, lambdas do exist and are values.
- Unfortunately, they are called differently to functions.
- So Ruby does NOT have first class functions. There are some facilities though

Functions as Values: Ruby

```
def add(x, y)
    x + y
end

a = add ✗
a = lambda {|x,y| add(x,y)}

a(2,3) ✗

a.call(2,3)
⇒ 5
```

- Setting a variable to a function in Ruby leads to an error.
- However, lambdas do exist and are values.
- Unfortunately, they are called differently to functions.
- So Ruby does NOT have first class functions. There are some facilities though

Functions ~~as Values~~: Ruby

```
def add(x, y)
    x + y
end

a = add ✗
a = lambda {|x,y| add(x,y)}

a(2,3) ✗

a.call(2,3)
⇒ 5
```

- Setting a variable to a function in Ruby leads to an error.
- However, lambdas do exist and are values.
- Unfortunately, they are called differently to functions.
- So Ruby does NOT have first class functions. There are some facilities though

Process an Array: JavaScript

```
function double(x) { return 2 * x }
data = [1, 2, 3, 4, 5]


for (i=0; i < data.length; i++) {
  data[i] = double(data[i])
}
```

A common idiom in JS is to update an array.
But if this happens in a function then this violates the principle of functions modifying values.

- So want to create a new array, and populate that with the updated values, leaving the original alone
-

Process an Array: JavaScript

```
function double(x) { return 2 * x }
data = [1, 2, 3, 4, 5]


newData = []
for (i=0; i < data.length; i++) {
  newData[i] = double(data[i])
}
```

A common idiom in JS is to update an array.
But if this happens in a function then this violates the principle of functions modifying values.

- So want to create a new array, and populate that with the updated values, leaving the original alone
-

Process an Array: JavaScript

```
function double(x) { return 2 * x }
data = [1, 2, 3, 4, 5]


newData = []
for (i=0; i < data.length; i++) {
  newData[i] = double(data[i])
}


newData ⇒ [2, 4, 6, 8, 10]
```

A common idiom in JS is to update an array.
But if this happens in a function then this violates the principle of functions modifying values.

- So want to create a new array, and populate that with the updated values, leaving the original alone
-

Process an Array: Python

```
def double(x):
  return 2 * x

data = [1, 2, 3, 4, 5]
```

We want to do the same thing in Python, where we create a new array containing updated results, and return that

Process an Array: Python

```
def double(x):
  return 2 * x

data = [1, 2, 3, 4, 5]

newData = []
for d in data:
  newData.append(double(d))
```

We want to do the same thing in Python, where we create a new array containing updated results, and return that

Process an Array: Python

```
def double(x):
  return 2 * x

data = [1, 2, 3, 4, 5]

newData = []
for d in data:
  newData.append(double(d))


newData ⇒ [2, 4, 6, 8, 10]
```

We want to do the same thing in Python, where we create a new array containing updated results, and return that

Process an Array: JavaScript

```
function double(x) { return 2 * x }
data = [1, 2, 3, 4, 5]


newData = []
for (i=0; i < data.length; i++) {
  newData[i] = double(data[i])
}
```

The structure of this code is always the same.
- The only thing that ever changes is the data to be processed, and the operation to do the processing.
- So this code can be wrapped up as a function. The name for this operation is “map”.
- Applying this gives the result you’d expect. Note how the first parameter is the function being passed as a value

Process an Array: JavaScript

```
function double(x) { return 2 * x }
data = [1, 2, 3, 4, 5]


newData = []
for (i=0; i < data.length; i++) {
  newData[i] = double(data[i])
}
```

The structure of this code is always the same.
- The only thing that ever changes is the data to be processed, and the operation to do the processing.
- So this code can be wrapped up as a function. The name for this operation is “map”.
- Applying this gives the result you’d expect. Note how the first parameter is the function being passed as a value

Process an Array: JavaScript

```
function double(x) { return 2 * x }
data = [1, 2, 3, 4, 5]

function map(fn,array) {
   newData = []
   for (i=0; i < array.length; i++) {
     newData[i] = fn(array[i])
   }
  return newData
}
```

The structure of this code is always the same.
- The only thing that ever changes is the data to be processed, and the operation to do the processing.
- So this code can be wrapped up as a function. The name for this operation is “map”.
- Applying this gives the result you’d expect. Note how the first parameter is the function being passed as a value

Process an Array: JavaScript

```
function double(x) { return 2 * x }
data = [1, 2, 3, 4, 5]

function map(fn,array) {
   newData = []
   for (i=0; i < array.length; i++) {
     newData[i] = fn(array[i])
   }
  return newData
}

map(double,data)
⇒ [2, 4, 6, 8, 10]
```

The structure of this code is always the same.

- The only thing that ever changes is the data to be processed, and the operation to do the processing.
- So this code can be wrapped up as a function. The name for this operation is “map”.
- Applying this gives the result you’d expect. Note how the first parameter is the function being passed as a value

Process an Array: JavaScript

```
function double(x) { return 2 * x }
data = [1, 2, 3, 4, 5]
```

But if we back up for a moment…
- map is built in to JS as a method on arrays

Process an Array: JavaScript

```
function double(x) { return 2 * x }
data = [1, 2, 3, 4, 5]
```

Built in!

```
data.map(double)
⇒ [2, 4, 6, 8, 10]
```

But if we back up for a moment…
- map is built in to JS as a method on arrays

Process an Array: Python (2.x)

```
def double(x):
  return 2 * x

data = [1, 2, 3, 4, 5]
```

- Python also has a map function built in
- Python 3 prefers the use of loop comprehensions

Process an Array: Python (2.x)

```
def double(x):
  return 2 * x

data = [1, 2, 3, 4, 5]
```

Built in!

```
map(double,data)
⇒ [2, 4, 6, 8, 10]
```

- Python also has a map function built in
- Python 3 prefers the use of loop comprehensions

Process an Array: Python (2.x)

```
def double(x):
  return 2 * x

data = [1, 2, 3, 4, 5]
```

Built in!

```
map(double,data)
⇒ [2, 4, 6, 8, 10]

[double(i) for i in data]
⇒ [2, 4, 6, 8, 10]
```

- Python also has a map function built in
- Python 3 prefers the use of loop comprehensions

Process an Array: Ruby

```
def double(x)
  2 * x
end
data = [1, 2, 3, 4, 5]
```

- Ruby's built in map requires a code block, which you can use to call your function
- but code blocks can be represented as a λ (which looks like an anonymous function), and passed using the "&" idiom

Process an Array: Ruby

```
def double(x)
  2 * x
end
data = [1, 2, 3, 4, 5]
```

Built in!

```
data.map {|d| double(d)}
⇒ [2, 4, 6, 8, 10]
```

- Ruby's built in map requires a code block, which you can use to call your function
- but code blocks can be represented as a λ (which looks like an anonymous function), and passed using the "&" idiom

Process an Array: Ruby

```
def double(x)
  2 * x
end
data = [1, 2, 3, 4, 5]
```

Built in!

```
l_double = lambda {|x| 2*x}
data.map &l_double
⇒ [2, 4, 6, 8, 10]
```

- Ruby's built in map requires a code block, which you can use to call your function
- but code blocks can be represented as a λ (which looks like an anonymous function), and passed using the "&" idiom

Functions Making Functions: JavaScript

```
function make_adder(x) {
  return function (y) { return x + y }
}
inc = make_adder(1)
```

Important element of FP is being able to construct function, and pass a constructed value around. This constructs a function that returns a specified value

- constructed value can be used normally
- constructed value can also be passed to functions that use functions

If you use JS you're probably familiar with this sort of thing!

Functions Making Functions: JavaScript

```
function make_adder(x) {
  return function (y) { return x + y }
}
inc = make_adder(1)

inc(2)
⇒ 3
```

Important element of FP is being able to construct function, and pass a constructed value around. This constructs a function that returns a specified value

- constructed value can be used normally
- constructed value can also be passed to functions that use functions

If you use JS you're probably familiar with this sort of thing!

Functions Making Functions: JavaScript

```
function make_adder(x) {
  return function (y) { return x + y }
}
inc = make_adder(1)

inc(2)
⇒ 3

data.map(inc)
⇒ [2, 3, 4, 5, 6]
```

Important element of FP is being able to construct function, and pass a constructed value around. This constructs a function that returns a specified value

- constructed value can be used normally
- constructed value can also be passed to functions that use functions

If you use JS you're probably familiar with this sort of thing!

Functions Making Functions: Python

```
def make_adder(x):
  return lambda y: x + y

inc = make_adder(1)
```

Python is pretty much the same

- constructed value can be used normally
- constructed value can also be passed to functions that use functions

Functions Making Functions: Python

```
def make_adder(x):
  return lambda y: x + y

inc = make_adder(1)

inc(2)
⇒ 3
```

Python is pretty much the same

- constructed value can be used normally
- constructed value can also be passed to functions that use functions

Functions Making Functions: Python

```
def make_adder(x):
  return lambda y: x + y

inc = make_adder(1)

inc(2)
⇒ 3

map(inc,data)
⇒ [2, 3, 4, 5, 6]
```

Python is pretty much the same

- constructed value can be used normally
- constructed value can also be passed to functions that use functions

Functions Making Functions: Ruby

```
def make_adder(x)
  lambda {|y| x + y}
end
inc = make_adder(1)
```

And so long as we're constructing and passing around lambdas then Ruby is mostly the same

- constructed value can be used with a call
- constructed value can also be used with the "&" idiom as a code block

Functions Making Functions: Ruby

```
def make_adder(x)
  lambda {|y| x + y}
end
inc = make_adder(1)

inc.call(2)
```

And so long as we're constructing and passing around lambdas then Ruby is mostly the same

- constructed value can be used with a call
- constructed value can also be used with the "&" idiom as a code block

Functions Making Functions: Ruby

```
def make_adder(x)
  lambda {|y| x + y}
end
inc = make_adder(1)

inc.call(2)
⇒ 3
```

And so long as we're constructing and passing around lambdas then Ruby is mostly the same

- constructed value can be used with a call
- constructed value can also be used with the "&" idiom as a code block

Functions Making Functions: Ruby

```
def make_adder(x)
  lambda {|y| x + y}
end
inc = make_adder(1)

inc.call(2)
⇒ 3

data.map &inc
```

And so long as we're constructing and passing around lambdas then Ruby is mostly the same

- constructed value can be used with a call
- constructed value can also be used with the "&" idiom as a code block

Functions Making Functions: Ruby

```
def make_adder(x)
  lambda {|y| x + y}
end
inc = make_adder(1)

inc.call(2)
⇒ 3

data.map &inc
⇒ [2, 3, 4, 5, 6]
```

And so long as we're constructing and passing around lambdas then Ruby is mostly the same

- constructed value can be used with a call
- constructed value can also be used with the "&" idiom as a code block

Filter an Array: JavaScript

```
function even(x) { return x % 2 == 0 }
data = [1, 2, 3, 4, 5]
```

Another important operation is filtering a collection. This is based on a function that returns a boolean value to indicate if an element should be kept in the collection.

Filter an Array: JavaScript

```
function even(x) { return x % 2 == 0 }
data = [1, 2, 3, 4, 5]



data.filter(even)
⇒ [2, 4]
```

Another important operation is filtering a collection. This is based on a function that returns a boolean value to indicate if an element should be kept in the collection.

Filter an Array: Python (2.x)

```
def even(x):
  return x % 2 == 0

data = [1, 2, 3, 4, 5]
```

Filters are also available in Python, using similarly constructed predicates

- (filter application)
- Python's list comprehensions also handle filtering
- This syntax is based on for/in/if the other elements are up to the context

Filter an Array: Python (2.x)

```
def even(x):
  return x % 2 == 0

data = [1, 2, 3, 4, 5]

filter(even,data)
⇒ [2, 4]
```

Filters are also available in Python, using similarly constructed predicates

- (filter application)
- Python's list comprehensions also handle filtering
- This syntax is based on for/in/if the other elements are up to the context

Filter an Array: Python (2.x)

```
def even(x):
  return x % 2 == 0

data = [1, 2, 3, 4, 5]

filter(even,data)
⇒ [2, 4]

[i for i in data if even(i)]
⇒ [2, 4]
```

Filters are also available in Python, using similarly constructed predicates

- (filter application)
- Python's list comprehensions also handle filtering
- This syntax is based on for/in/if the other elements are up to the context

Filter an Array: Python (2.x)

```
def even(x):
  return x % 2 == 0

data = [1, 2, 3, 4, 5]

filter(even,data)
⇒ [2, 4]

[i for i in data if even(i)]
⇒ [2, 4]
```

Filters are also available in Python, using similarly constructed predicates

- (filter application)
- Python's list comprehensions also handle filtering
- This syntax is based on for/in/if the other elements are up to the context

Filter an Array: Ruby

```
data = [1, 2, 3, 4, 5]
```

The even? predicate is already available. Rather than wrap it in a λ to pass it, it makes more sense to use it in a code block.

- rather than filter, Ruby has an equivalent called “select”
- There is a variant with an exclamation point. This is anathema to FP, as it changes the array

Filter an Array: Ruby

```
data = [1, 2, 3, 4, 5]

data.select {|d| d.even? }
⇒ [2, 4]
```

The even? predicate is already available. Rather than wrap it in a λ to pass it, it makes more sense to use it in a code block.

- rather than filter, Ruby has an equivalent called “select”
- There is a variant with an exclamation point. This is anathema to FP, as it changes the array

Filter an Array: Ruby

```
data = [1, 2, 3, 4, 5]

data.select {|d| d.even? }
⇒ [2, 4]
```

Don't use `select!`

```
data.select! {|d| d.even? }
⇒ [2, 4]
data
⇒ [2, 4]
```

The even? predicate is already available. Rather than wrap it in a λ to pass it, it makes more sense to use it in a code block.

- rather than filter, Ruby has an equivalent called “select”
- There is a variant with an exclamation point. This is anathema to FP, as it changes the array

Filter an Array: Ruby

```
data = [1, 2, 3, 4, 5]

data.select {|d| !d.even? }
⇒ [1, 3, 5]
```

Sometimes you want to select the opposite items. Which can be done just by negating the test.
- Like many languages, Ruby includes an operation that selects the opposite to the predicate. JS and Python don’t include this, but it’s not that big a deal to achieve the same operation

Filter an Array: Ruby

```
data = [1, 2, 3, 4, 5]

data.select {|d| !d.even? }
⇒ [1, 3, 5]

data.reject {|d| d.even? }
⇒ [1, 3, 5]
```

Sometimes you want to select the opposite items. Which can be done just by negating the test.
- Like many languages, Ruby includes an operation that selects the opposite to the predicate. JS and Python don’t include this, but it’s not that big a deal to achieve the same operation

Reduce an Array: JavaScript

```
function add(x,y) { return x + y }
data = [1, 2, 3, 4, 5]
```

Reducing is about processing collections into a single result. It builds up a total using an operation that combines the current total with the next value in a collection

- reducing with the add function leads to a total
- if the first element is a string, then the add function will act as string concatenation

Reduce an Array: JavaScript

```
function add(x,y) { return x + y }
data = [1, 2, 3, 4, 5]

data.reduce(add)
⇒ 15
```

Reducing is about processing collections into a single result. It builds up a total using an operation that combines the current total with the next value in a collection

- reducing with the add function leads to a total
- if the first element is a string, then the add function will act as string concatenation

Reduce an Array: JavaScript

```
function add(x,y) { return x + y }
data = [1, 2, 3, 4, 5]

data.reduce(add)
⇒ 15

data = [‘’, 1, 2, 3, 4, 5]
```

Reducing is about processing collections into a single result. It builds up a total using an operation that combines the current total with the next value in a collection

- reducing with the add function leads to a total
- if the first element is a string, then the add function will act as string concatenation

Reduce an Array: JavaScript

```
function add(x,y) { return x + y }
data = [1, 2, 3, 4, 5]

data.reduce(add)
⇒ 15

data = [‘’, 1, 2, 3, 4, 5]

data.reduce(add)
⇒ ‘12345’
```

Reducing is about processing collections into a single result. It builds up a total using an operation that combines the current total with the next value in a collection

- reducing with the add function leads to a total
- if the first element is a string, then the add function will act as string concatenation

Reduce an Array: Python (2.x)

```
def add(x,y):
  return x + y

data = [1, 2, 3, 4, 5]
```

Same goes for python.

- reducing numbers adds them all
- string concatenation with Python needs correct typing. We can also add a string to the start of the data… although this DOES modify the data
- reducing will then do string concatenation

Reduce an Array: Python (2.x)

```
def add(x,y):
  return x + y

data = [1, 2, 3, 4, 5]
reduce(add, data)
⇒ 15
```

Same goes for python.

- reducing numbers adds them all
- string concatenation with Python needs correct typing. We can also add a string to the start of the data… although this DOES modify the data
- reducing will then do string concatenation

Reduce an Array: Python (2.x)

```
def add(x,y):
  return x + y

data = [1, 2, 3, 4, 5]
reduce(add, data)
⇒ 15
def adds(s,y):
  return s + str(y)

data.insert(0, ‘’)
```

Same goes for python.

- reducing numbers adds them all
- string concatenation with Python needs correct typing. We can also add a string to the start of the data… although this DOES modify the data
- reducing will then do string concatenation

Reduce an Array: Python (2.x)

```
def add(x,y):
  return x + y

data = [1, 2, 3, 4, 5]
reduce(add, data)
⇒ 15
def adds(s,y):
  return s + str(y)

data.insert(0, ‘’)
reduce(adds, data)
⇒ ‘12345’
```

Same goes for python.

- reducing numbers adds them all
- string concatenation with Python needs correct typing. We can also add a string to the start of the data… although this DOES modify the data
- reducing will then do string concatenation

Reduce an Array: Ruby

```
add = lambda {|x, y| x + y}

data = [1, 2, 3, 4, 5]
```

The same process can be done with Ruby

- again, a code block can be passed as a λ
- Similarly, we can use a string concatenation function, and destructively update the data
- This creates the output

Reduce an Array: Ruby

```
add = lambda {|x, y| x + y}

data = [1, 2, 3, 4, 5]
data.reduce &add
⇒ 15
```

The same process can be done with Ruby

- again, a code block can be passed as a λ
- Similarly, we can use a string concatenation function, and destructively update the data
- This creates the output

Reduce an Array: Ruby

```
add = lambda {|x, y| x + y}

data = [1, 2, 3, 4, 5]
data.reduce &add
⇒ 15

adds = lambda {|s, y| s + y.to_s}

data.insert(0, ‘’)
```

The same process can be done with Ruby

- again, a code block can be passed as a λ
- Similarly, we can use a string concatenation function, and destructively update the data
- This creates the output

Reduce an Array: Ruby

```
add = lambda {|x, y| x + y}

data = [1, 2, 3, 4, 5]
data.reduce &add
⇒ 15

adds = lambda {|s, y| s + y.to_s}

data.insert(0, ‘’)

data.reduce &adds
⇒ ‘12345’
```

The same process can be done with Ruby

- again, a code block can be passed as a λ
- Similarly, we can use a string concatenation function, and destructively update the data
- This creates the output

Reduce an Array: Ruby

```
add = lambda {|x, y| x + y}

data = [1, 2, 3, 4, 5]
data.reduce &add
⇒ 15

data.reduce("") {|s, y| s + y.to_s}
⇒ '12345'
```

The same process can be done with Ruby

- again, a code block can be passed as a λ
- Similarly, we can use a string concatenation function, and destructively update the data
- This creates the output

Processing a Properties file

```
# Set to default
redis.host=localhost
redis.port=6379

# change this to your local queue name
redis.queue-name=example-event-queue
redis.timeout-ms=1000
```

Read the file

JavaScript

```
var fs = require('fs');
data = fs.readFileSync(‘redis.properties’)
         .toString().split("\n");
```

Python

```
data = open(‘redis.properties', ‘r’).readlines()
```

Ruby

```
data = File.readlines(‘redis.properties’)
```

Properties file: JavaScript

```
function not_blank(l) {
  return !/^ *($|#)/.test(l)
}
```

Operations that we're going to use:
test on the lines that are blank. This includes comment lines

- operation to split the line into a property and it's value
- operation to insert a value at the head of an array. We saw how this was needed for reduce
- operation to reduce with. In this case we want to take a property/value pair and add it to a map. Note that the new map is returned, since this is the value that will be passed in to the new add_map

Properties file: JavaScript

```
function not_blank(l) {
  return !/^ *($|#)/.test(l)
}

function prop_val(l) {return l.split('=')}
```

Operations that we're going to use:
test on the lines that are blank. This includes comment lines

- operation to split the line into a property and it's value
- operation to insert a value at the head of an array. We saw how this was needed for reduce
- operation to reduce with. In this case we want to take a property/value pair and add it to a map. Note that the new map is returned, since this is the value that will be passed in to the new add_map

Properties file: JavaScript

```
function not_blank(l) {
  return !/^ *($|#)/.test(l)
}

function prop_val(l) {return l.split('=')}

function prepend(a,v) {
  a.unshift(v)
  return a
}
```

Operations that we're going to use:
test on the lines that are blank. This includes comment lines

- operation to split the line into a property and it's value
- operation to insert a value at the head of an array. We saw how this was needed for reduce
- operation to reduce with. In this case we want to take a property/value pair and add it to a map. Note that the new map is returned, since this is the value that will be passed in to the new add_map

Properties file: JavaScript

```
function not_blank(l) {
  return !/^ *($|#)/.test(l)
}

function prop_val(l) {return l.split('=')}

function prepend(a,v) {
  a.unshift(v)
  return a
}

function add_map(m,p) {
  m[p[0]] = p[1]
  return m
}
```

Operations that we're going to use:
test on the lines that are blank. This includes comment lines

- operation to split the line into a property and it's value
- operation to insert a value at the head of an array. We saw how this was needed for reduce
- operation to reduce with. In this case we want to take a property/value pair and add it to a map. Note that the new map is returned, since this is the value that will be passed in to the new add_map

Properties file: JavaScript

```
data




# Set to default
redis.host=localhost
redis.port=6379

# change this to your local queue name
redis.queue-name=example-event-queue
redis.timeout-ms=1000
```

Properties file: JavaScript

```
data.filter(not_blank)
```

Properties file: JavaScript

```
data.filter(not_blank)
```

```
redis.host=localhost
redis.port=6379
redis.queue-name=example-event-queue
redis.timeout-ms=1000
```

Properties file: JavaScript

```
data.filter(not_blank).map(prop_val)
```

Properties file: JavaScript

```
data.filter(not_blank).map(prop_val)
```

```
[[‘redis.host’, ‘localhost’],
 [‘redis.port’, ‘6379’],
 [‘redis.queue-name’, ‘example-event-queue’],
 [‘redis.timeout-ms’, ‘1000’]]
```

Properties file: JavaScript

```
prepend(
  data.filter(not_blank).map(prop_val),
  {})
```

Properties file: JavaScript

```
prepend(
  data.filter(not_blank).map(prop_val),
  {})
```

```
[{},
 ['redis.host', 'localhost'],
 ['redis.port', '6379'],
 ['redis.queue-name', 'example-event-queue'],
 ['redis.timeout-ms', '1000']]
```

Properties file: JavaScript

```
prepend(
  data.filter(not_blank).map(prop_val),
  {}).reduce(add_map)
```

Properties file: JavaScript

```
prepend(
  data.filter(not_blank).map(prop_val),
  {}).reduce(add_map)
```

```
{ 'redis.host': 'localhost',
  'redis.port': '6379',
  'redis.queue-name': 'example-event-queue',
  'redis.timeout-ms': '1000' }
```

Properties file: JavaScript Without Named Functions

```
d = data.filter(
      function(s){return !/^ *($|#)/.test(s)}
    ).map(
      function(s){ return s.split('=')}
    )

d.unshift({})

d.reduce(function(m,p) {
  m[p[0]] = p[1]
  return m
})
```

Properties file: Python

```
import re
def not_blank(l):
  return not(re.match("^ *($|#)", l))
```

Operations that we're going to use:
test on the lines that are blank. This includes comment lines. Regular expressions need to be imported

- operation to split the line into a property and it's value
- operation to insert a value at the head of an array. We saw how this was needed for reduce
- operation to reduce with. In this case we want to take a property/value pair and add it to a map. Note that the new map is returned, since this is the value that will be passed in to the new add_map

Essentially identical to the Javascript version

Properties file: Python

```
import re
def not_blank(l):
  return not(re.match("^ *($|#)", l))

def prop_val(l):
  return l.split('=')
```

Operations that we're going to use:
test on the lines that are blank. This includes comment lines. Regular expressions need to be imported

- operation to split the line into a property and it's value
- operation to insert a value at the head of an array. We saw how this was needed for reduce
- operation to reduce with. In this case we want to take a property/value pair and add it to a map. Note that the new map is returned, since this is the value that will be passed in to the new add_map

Essentially identical to the Javascript version

Properties file: Python

```
import re
def not_blank(l):
  return not(re.match("^ *($|#)", l))

def prop_val(l):
  return l.split('=')

def prepend(a,v):
  a.insert(0,v)
  return a
```

Operations that we're going to use:
test on the lines that are blank. This includes comment lines. Regular expressions need to be imported

- operation to split the line into a property and it's value
- operation to insert a value at the head of an array. We saw how this was needed for reduce
- operation to reduce with. In this case we want to take a property/value pair and add it to a map. Note that the new map is returned, since this is the value that will be passed in to the new add_map

Essentially identical to the Javascript version

Properties file: Python

```
import re
def not_blank(l):
  return not(re.match("^ *($|#)", l))

def prop_val(l):
  return l.split('=')

def prepend(a,v):
  a.insert(0,v)
  return a

def add_map(m,p):
  m[p[0]] = p[1]
  return m
```

Operations that we're going to use:
test on the lines that are blank. This includes comment lines. Regular expressions need to be imported

- operation to split the line into a property and it's value
- operation to insert a value at the head of an array. We saw how this was needed for reduce
- operation to reduce with. In this case we want to take a property/value pair and add it to a map. Note that the new map is returned, since this is the value that will be passed in to the new add_map

Essentially identical to the Javascript version

Properties file: Python

lines

```
# Set to default
redis.host=localhost
redis.port=6379

# change this to your local queue name
redis.queue-name=example-event-queue
redis.timeout-ms=1000
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python

```
filter(not_blank, lines)
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python

```
filter(not_blank, lines)
```

```
redis.host=localhost
redis.port=6379
redis.queue-name=example-event-queue
redis.timeout-ms=1000
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python

```
map(prop_val,
    filter(not_blank, lines))
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python

```
map(prop_val,
    filter(not_blank, lines))
```

```
[['redis.host', 'localhost'],
 ['redis.port', '6379'],
 ['redis.queue-name', 'example-event-queue'],
 ['redis.timeout-ms', '1000']]
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python

```
prepend(
  map(prop_val,
      filter(not_blank, lines)),
  {})
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python

```
prepend(
  map(prop_val,
      filter(not_blank, lines)),
  {})

[{},
 ['redis.host', 'localhost'],
 ['redis.port', '6379'],
 ['redis.queue-name', 'example-event-queue'],
 ['redis.timeout-ms', '1000']]
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python

```
reduce(add_map,
  prepend(
    map(prop_val,
        filter(not_blank, lines)),
    {}))
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python

```
reduce(add_map,
  prepend(
    map(prop_val,
        filter(not_blank, lines)),
    {}))

{ 'redis.host': 'localhost',
  'redis.port': '6379',
  'redis.queue-name': 'example-event-queue',
  'redis.timeout-ms': '1000' }
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python

```
{k: v for k, v in
    map(prop_val,
        filter(not_blank, lines))}


{ 'redis.host': 'localhost',
  'redis.port': '6379',
  'redis.queue-name': 'example-event-queue',
  'redis.timeout-ms': '1000' }
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python

```
{k: v for k, v in
  [prop_val(l) for l in lines
                if not_blank(l)]}


{ 'redis.host': 'localhost',
  'redis.port': '6379',
  'redis.queue-name': 'example-event-queue',
  'redis.timeout-ms': '1000' }
```

data
- remove blank
- result
- map to pairs
- result
- add the map at the start to reduce into (this is a hack)
- result
- reduce
- result
- instead of reduce, can use a dict comprehension from the pairs
- and the pairs can be created using a list comprehension

Properties file: Python Without Named Functions

```
{k: v for k, v in
  [l.split('=')
    for l in lines
    if not(re.match("^ *($|#)", l))]}

{ 'redis.host': 'localhost',
  'redis.port': '6379',
  'redis.queue-name': 'example-event-queue',
  'redis.timeout-ms': '1000' }
```

Properties file: Ruby

```
blank = lambda {|l| l =~ /^ *($|#)/}
```

Ruby isn’t really friendly with naming some of the operations we want, so we will just create a λ to test the lines
- and another to split the lines into property/value

Properties file: Ruby

```
blank = lambda {|l| l =~ /^ *($|#)/}

prop_val = lambda {|l| l.split('=')}
```

Ruby isn't really friendly with naming some of the operations we want, so we will just create a λ to test the lines
- and another to split the lines into property/value

Properties file: Ruby

```
lines



# Set to default
redis.host=localhost
redis.port=6379

# change this to your local queue name
redis.queue-name=example-event-queue
redis.timeout-ms=1000
```

initial lines
- filter out the blank lines
- result
- map to property/values
- result
- reduce into a map
- result
- but Ruby has a built in function that already does the reduction step

Properties file: Ruby

```
lines.reject &blank
```

initial lines
- filter out the blank lines
- result
- map to property/values
- result
- reduce into a map
- result
- but Ruby has a built in function that already does the reduction step

Properties file: Ruby

```
lines.reject &blank
```

```
redis.host=localhost
redis.port=6379
redis.queue-name=example-event-queue
redis.timeout-ms=1000
```

initial lines
- filter out the blank lines
- result
- map to property/values
- result
- reduce into a map
- result
- but Ruby has a built in function that already does the reduction step

Properties file: Ruby

```
(lines.reject &blank).map &prop_val
```

initial lines
- filter out the blank lines
- result
- map to property/values
- result
- reduce into a map
- result
- but Ruby has a built in function that already does the reduction step

Properties file: Ruby

```
(lines.reject &blank).map &prop_val
```

```
[[‘redis.host’, ‘localhost’],
 [‘redis.port’, ‘6379’],
 [‘redis.queue-name’, ‘example-event-queue’],
 [‘redis.timeout-ms’, ‘1000’]]
```

initial lines
- filter out the blank lines
- result
- map to property/values
- result
- reduce into a map
- result
- but Ruby has a built in function that already does the reduction step

Properties file: Ruby

```
((lines.reject &blank).map &prop_val)
  .reduce({}) {|m,p| m.update(p[0]=>p[1])}
```

initial lines
- filter out the blank lines
- result
- map to property/values
- result
- reduce into a map
- result
- but Ruby has a built in function that already does the reduction step

Properties file: Ruby

```
((lines.reject &blank).map &prop_val)
  .reduce({}) {|m,p| m.update(p[0]=>p[1])}
```

```
{ 'redis.host': 'localhost',
  'redis.port': '6379',
  'redis.queue-name': 'example-event-queue',
  'redis.timeout-ms': '1000' }
```

initial lines
- filter out the blank lines
- result
- map to property/values
- result
- reduce into a map
- result
- but Ruby has a built in function that already does the reduction step

Properties file: Ruby

```
((lines.reject &blank).map &prop_val)
  .to_h
```

```
{ 'redis.host': 'localhost',
  'redis.port': '6379',
  'redis.queue-name': 'example-event-queue',
  'redis.timeout-ms': '1000' }
```

initial lines
- filter out the blank lines
- result
- map to property/values
- result
- reduce into a map
- result
- but Ruby has a built in function that already does the reduction step

Properties file: Ruby Without Named Functions

```
((lines.reject {|l| l =~ /^ *($|#)/})
  .map {|l| l.split('=')})
  .to_h
```

Other Features

- Returning multiple values with tuples
- Lazy results
- Reordering operations

There are occasions when a block of code updates several values at once. Since functions can’t modify data, functions need to be able to return multiple items. Ruby and Python make this easy. Javascript requires arrays, and data extracted manually

Because order doesn’t matter, instead of returning data structures, functional languages can return code that will calculate that structure on access. The access looks no different. Allows invalid values that are never looked at, and infinite structures.

Operations can be reordered transparently, usually for efficiency.

- For instance, mapping 3 times, means iterating through a list 3 times, which can be expensive
- instead, by using lazy evaluation, the system effectively performs function composition for each of the arguments

Other Features

- Returning multiple values with tuples
- Lazy results
- Reordering operations

```
data.map(f1).map(f2).map(f3)
```

There are occasions when a block of code updates several values at once. Since functions can't modify data, functions need to be able to return multiple items. Ruby and Python make this easy. Javascript requires arrays, and data extracted manually

Because order doesn't matter, instead of returning data structures, functional languages can return code that will calculate that structure on access. The access looks no different. Allows invalid values that are never looked at, and infinite structures.

Operations can be reordered transparently, usually for efficiency.

- For instance, mapping 3 times, means iterating through a list 3 times, which can be expensive
- instead, by using lazy evaluation, the system effectively performs function composition for each of the arguments

Other Features

- Returning multiple values with tuples
- Lazy results
- Reordering operations

```
data.map(f1).map(f2).map(f3)
```

⇓

```
data.map(compose(f1, f2, f3))
```

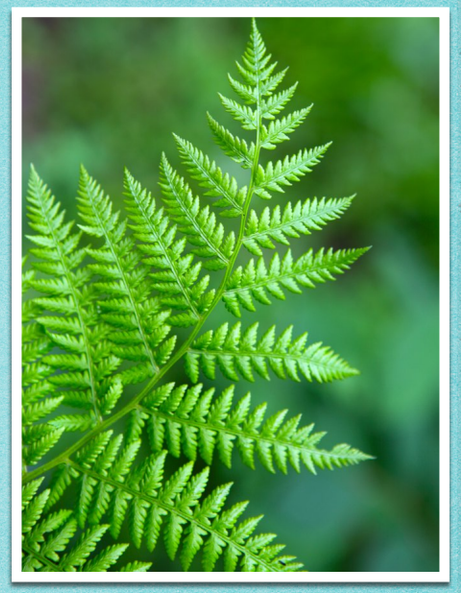
There are occasions when a block of code updates several values at once. Since functions can't modify data, functions need to be able to return multiple items. Ruby and Python make this easy. Javascript requires arrays, and data extracted manually

Because order doesn't matter, instead of returning data structures, functional languages can return code that will calculate that structure on access. The access looks no different. Allows invalid values that are never looked at, and infinite structures.

Operations can be reordered transparently, usually for efficiency.

- For instance, mapping 3 times, means iterating through a list 3 times, which can be expensive
- instead, by using lazy evaluation, the system effectively performs function composition for each of the arguments

Paula Gearon
@quoll



https://www.lambdaladies.com/resources/