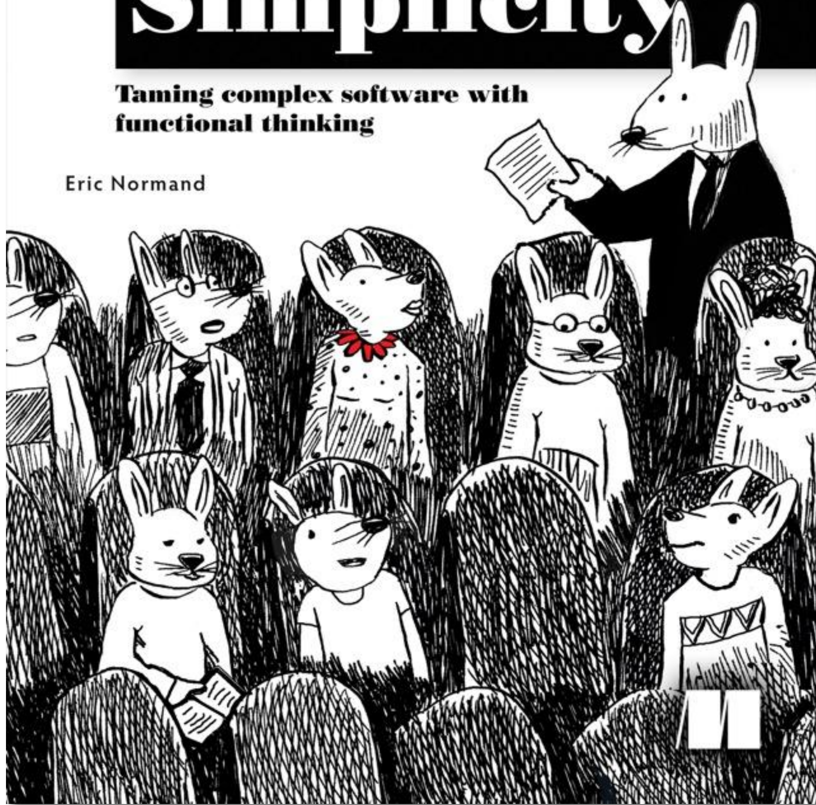
A Theory of Functional Programming

April 2020

Eric Normand



lispcast.com/gs

TSSIMPLICITY (50% off)

What is functional programming?

Functional programming is a set of skills

Why functional programming?

Distributed systems

3 Levels of Functional Programming

1. Distinguishing Actions, Calculations, Data

2. First-class abstractions

3. Building composable models

Part 1: Distinguishing Actions, Calculations, Data

- What are actions, calculations, and data?
- How the spreading rule makes actions pernicious
- Recognizing implicit inputs and outputs to functions
- Immutability

stars indicate you need to be careful

```
{"firstname": "Eric",
 "lastname": "Normand"}
```

information about a person

```
* sendEmail(to, from, subject, body)
```

be careful with this one, it sends an email

```
sum(numbers)
```

a handy function for adding up some numbers

```
* saveUserDB(user)
```

once you save it to the db, other parts of the system can see it

```
string_length(str)
```

if you pass it the same string twice, it returns the same length twice

```
* getCurrentTime()
```

each time you call it, you get a different time

```
[1, 10, 2, 45, 3, 98]
```

just a list of numbers

actions depend on
when they are called

Actions

*sendEmail(to, from, subject, body)

*saveUserDB(user)

*getCurrentTime()

everything else does
not depend on when
it is called

{"firstname": "Eric",
"lastname": "Normand"}

sum(numbers)

string_length(str)

[1, 10, 2, 45, 3, 98]

actions depend on when they are called

Actions

```
sendEmail(to, from, subject, body)
saveUserDB(user)
getCurrentTime()
```

---

calculations are computations from inputs to outputs

Calculations

```
sum(numbers)
string_length(str)
```

---

data is recorded facts about events

Data

```
[1, 10, 2, 45, 3, 98]
{"firstname": "Eric",
 "lastname": "Normand"}
```

Actions

- [x] Actions
- [ ] Calculations
- [ ] Data

the process of doing something, typically to achieve an aim

- Typically: *effects* or *side-effects*
- Depend on *when you run them* or *how many times you run them*

Calculations

- ✅ Actions
- ✅ Calculations
- ☐ Data

computation from inputs to outputs

- Typically: *pure functions*
- Eternal — outside of time
- Referentially transparent

Data

✅Actions
✅Calculations
✅Data

facts about events used as a basis for reasoning, discussion, or calculation

- Inert
- Serializable
- Requiring interpretation

```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main(affiliates) {
  affiliatePayout(affiliates);
}
```

we highlight the "one action" Jenna is talking about

Jenna on dev team

```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don’t send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main(affiliates) {
  affiliatePayout(affiliates);
}
```

```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main(affiliates) {
  affiliatePayout(affiliates);
}
```

the whole function is an action because it calls an action

we highlight the line where we call figurePayout(), a known action

```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main(affiliates) {
  affiliatePayout(affiliates);
}
```

highlight the whole function since it calls an action

called here

```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main(affiliates) {
  affiliatePayout(affiliates);
}
```

```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout() {
  var affiliates = fetchAffiliates();
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main() {
  affiliatePayout();
}
```

we're writing to a global variable, those are outputs

```
function calc_cart_total() {
  shopping_cart_total = 0;
  for(var i = 0; i < shopping_cart.length; i++) {
    var item = shopping_cart[i];
    shopping_cart_total += item.price;
  }
  set_cart_total_dom();
  update_shipping_icons();
  update_tax_dom();
}
```

we are reading from a global variable, which is an input

we're changing the DOM, those are outputs

```
function calc_total() {
  shopping_cart_total = 0;
  for(var i = 0; i < shopping_cart.length; i++) {
    var item = shopping_cart[i];
    shopping_cart_total += item.price;
  }
}
```

output

input

output

Current

```
function calc_cart_total() {
  calc_total();
  set_cart_total_dom();
  update_shipping_icons();
  update_tax_dom();
}

function calc_total() {
  shopping_cart_total = 0;
  for(var i = 0; i < shopping_cart.length; i++) {
    var item = shopping_cart[i];
    shopping_cart_total += item.price;
  }

}
```

Eliminated outputs

```
function calc_cart_total() {
  shopping_cart_total = calc_total();
  set_cart_total_dom();
  update_shipping_icons();
  update_tax_dom();
}

function calc_total() {
  var total = 0;
  for(var i = 0; i < shopping_cart.length; i++) {
    var item = shopping_cart[i];
    total += item.price;
  }
+ return total;                                   +
}
```

use the return value to set the global variable

move the assignment outside to the caller

convert it to a local variable

operate on the local variable

return the local

Copy-on-write

Mutating

```
function drop_first(array) {

  array.shift();

}
```

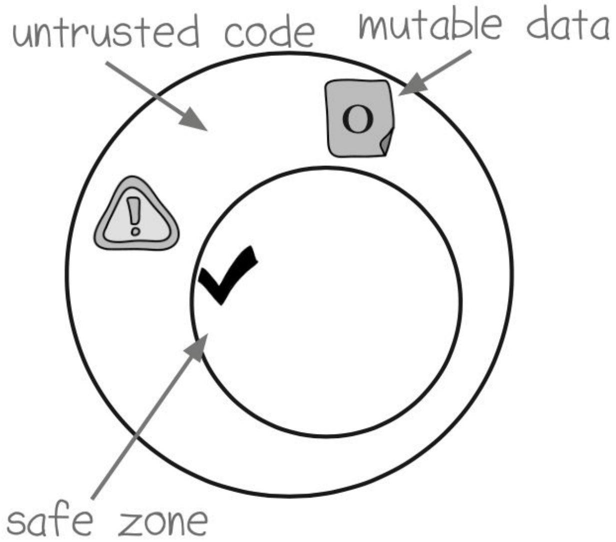
Copy-on-write

```
function drop_first(array) {
+ var array_copy = Array.copy(array);      +
  array_copy.shift();
+ return array_copy;                       +
}
```

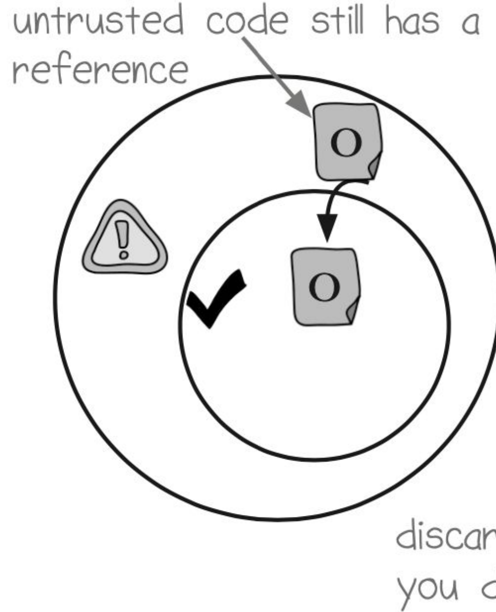
Copy-on-write rules

1. Make a copy
2. Modify the copy
3. Return the copy

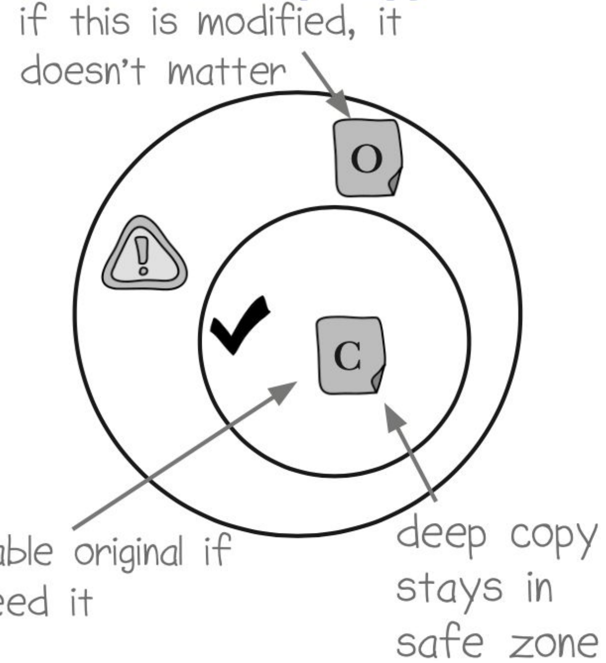
1. Data in untrusted code



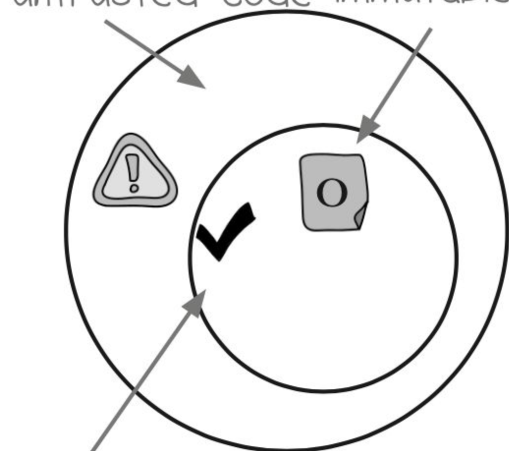
2. Data enters the safe zone



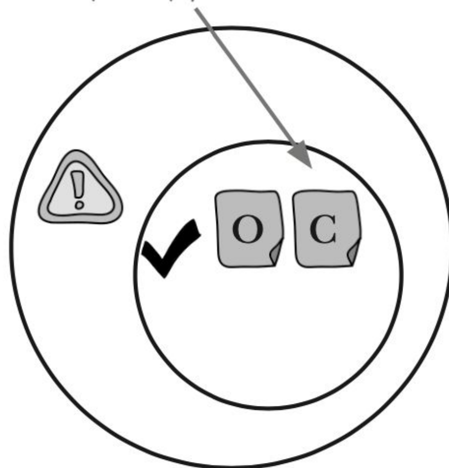
3. Make a deep copy



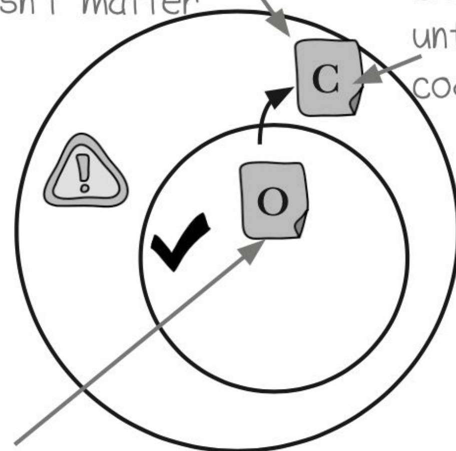
1. Data in safe zone



2. Make a deep copy



3. Make a deep copy



Defensive copying rules

1. Make a **deep copy** as data **leaves** the safe zone
2. Make a **deep copy** as data **enters** the safe zone

Part 2: First-class abstractions

- First-class values help you abstract
- Higher-order iteration (map, filter, reduce) helps clarify your for loops
- Chaining map, filter, and reduce gives you data transformation in steps
- Timelines help you understand how your code might execute
- Higher-order actions help you control your execution

First-class values

```
function setPriceByName(cart, name, price) {
  var item = cart[name];
  var updatedItem = objectSet(item, 'price', price);
  return objectSet(cart, name, updatedItem);
}

function setQuantityByName(cart, name, quantity)
function setDiscountByName(cart, name, discount)
```

First-class values

```
function setPriceByName(cart, name, price) {
  var item = cart[name];
  var updatedItem = objectSet(item, 'price', price);
  return objectSet(cart, name, updatedItem);
}

function setQuantityByName(cart, name, quantity)
function setDiscountByName(cart, name, discount)
```

First-class values

```
function setFieldByName(cart, name, field, value) {
  var item = cart[name];
  var updatedItem = objectSet(item, field, value);
  return objectSet(cart, name, updatedItem);
}
```

Replace body with callback

```
for(var i = 0; i < foods.length; i++) {
  var food = foods[i];
  cook(food);
  eat(food);
}

for(var i = 0; i < dishes.length; i++) {
  var dish = dishes[i];
  wash(dish);
  dry(dish);
  putAway(dish);
}
```

Replace body with callback

```
for(var i = 0; i < foods.length; i++) {   ← before
  var food = foods[i];
  cook(food);                              ← body
  eat(food);
}                                          ← after

for(var i = 0; i < dishes.length; i++) {  ← before
  var dish = dishes[i];
  wash(dish);                              ← body
  dry(dish);
  putAway(dish);
}                                          ← after
```

Replace body with callback

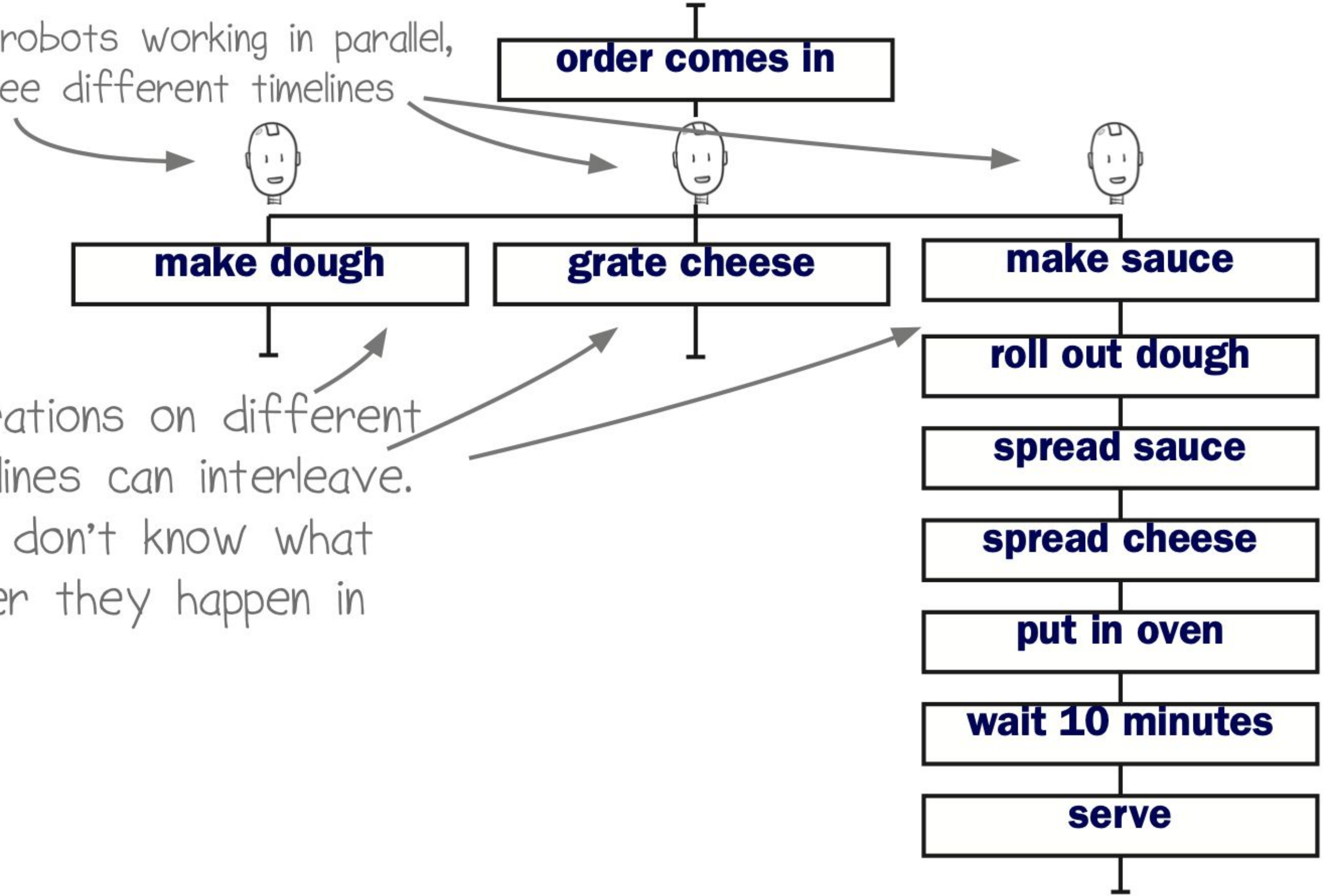
```
function forEach(array, f) {
  for(var i = 0; i < array; i++) {
    f(array[i]);
  }
}

forEach(foods, function(food) {
  cook(food);
  eat(food);
});
forEach(dishes, function(dish) {
  wash(dish);
  dry(dish);
  putAway(dish);
});
```

Chaining map, filter, reduce

```
// Average order of a good customer

var sum = 0;

var count = 0;

for(var i = 0; i < customers.length; i++) {
  var customer = customers[i];
  if(customer.purchases.length > 5) {
    for(var p = 0; p < customer.purchases.length; p++) {
      var purchase = customer.purchases[p];
      sum += purchase.total;
      count += 1;
    }
  }
}

var average = sum / count;
```

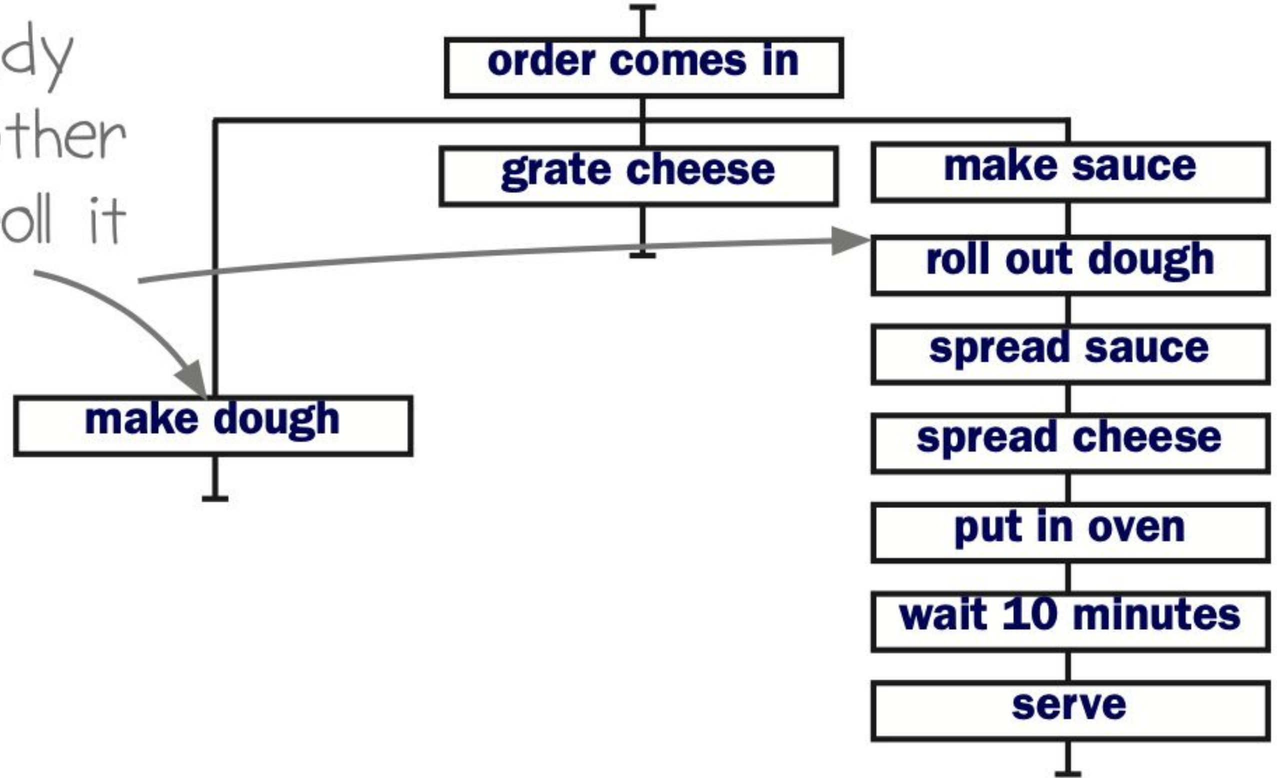
Chaining map, filter, reduce

```
// Average order of a good customer

var goodCustomers = filter(customers, function(customer) {
  return customer.purchases.length > 5;
});

var purchases = flatMap(goodCustomers, function(customer) {
  return customer.purchases;
});

var purchasePrices = map(purchases, function(purchase) {
  return purchase.price;
});

var sum = reduce(purchasePrices, 0, function(a, b) { return a+b; });

var average = sum / purchasePrices.length;
```

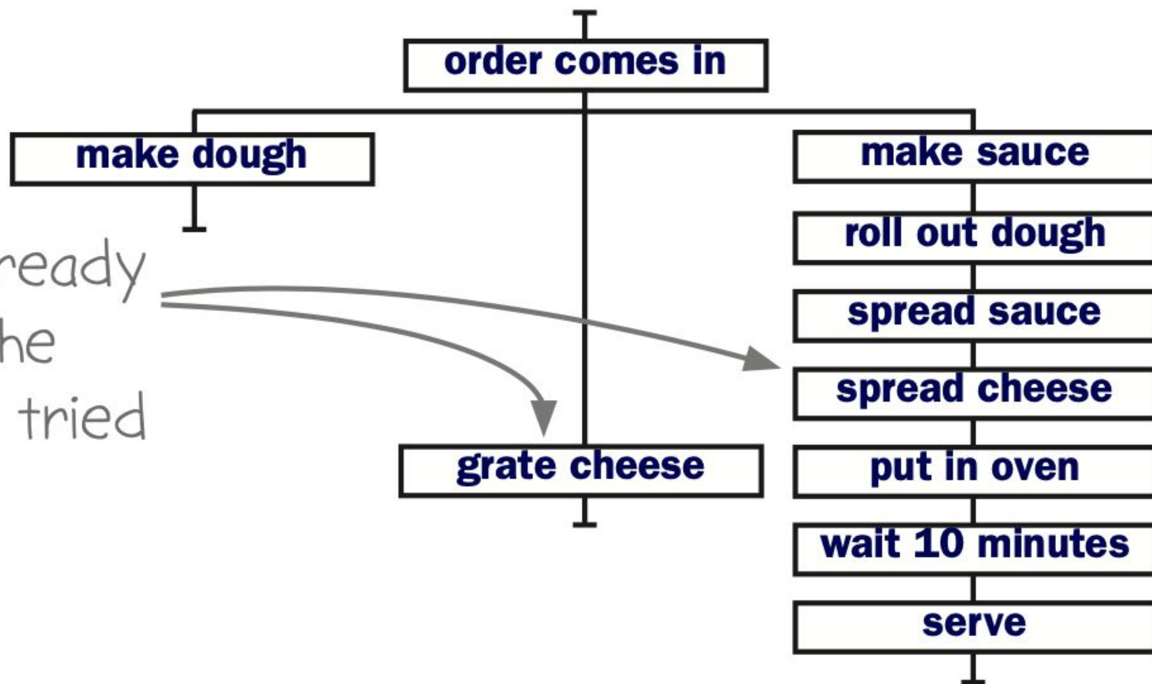

Making a cheese pizza
three robots working in parallel, so three different timelines
order comes in
make dough
grate cheese
make sauce
roll out dough
spread sauce
spread cheese
put in oven
wait 10 minutes
serve
operations on different timelines can interleave. you don't know what order they happen in

dough takes longer

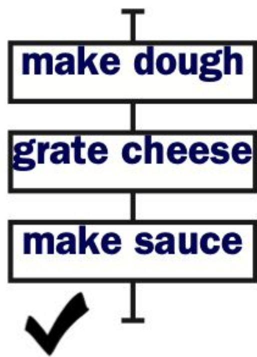
dough is not ready until after the other robot tried to roll it out

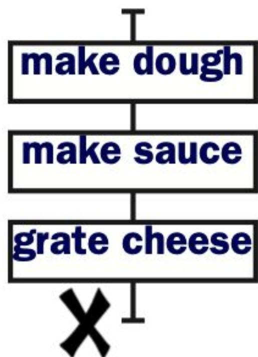


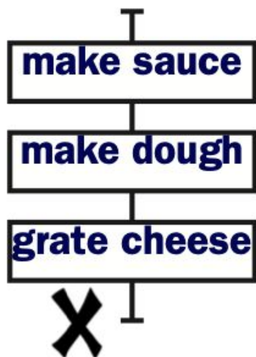
cheese takes longer

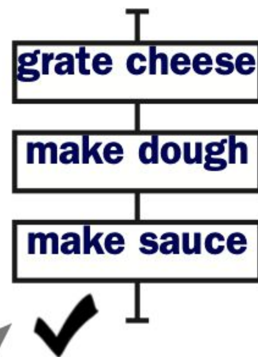


cheese not ready until after the other robot tried to spread it

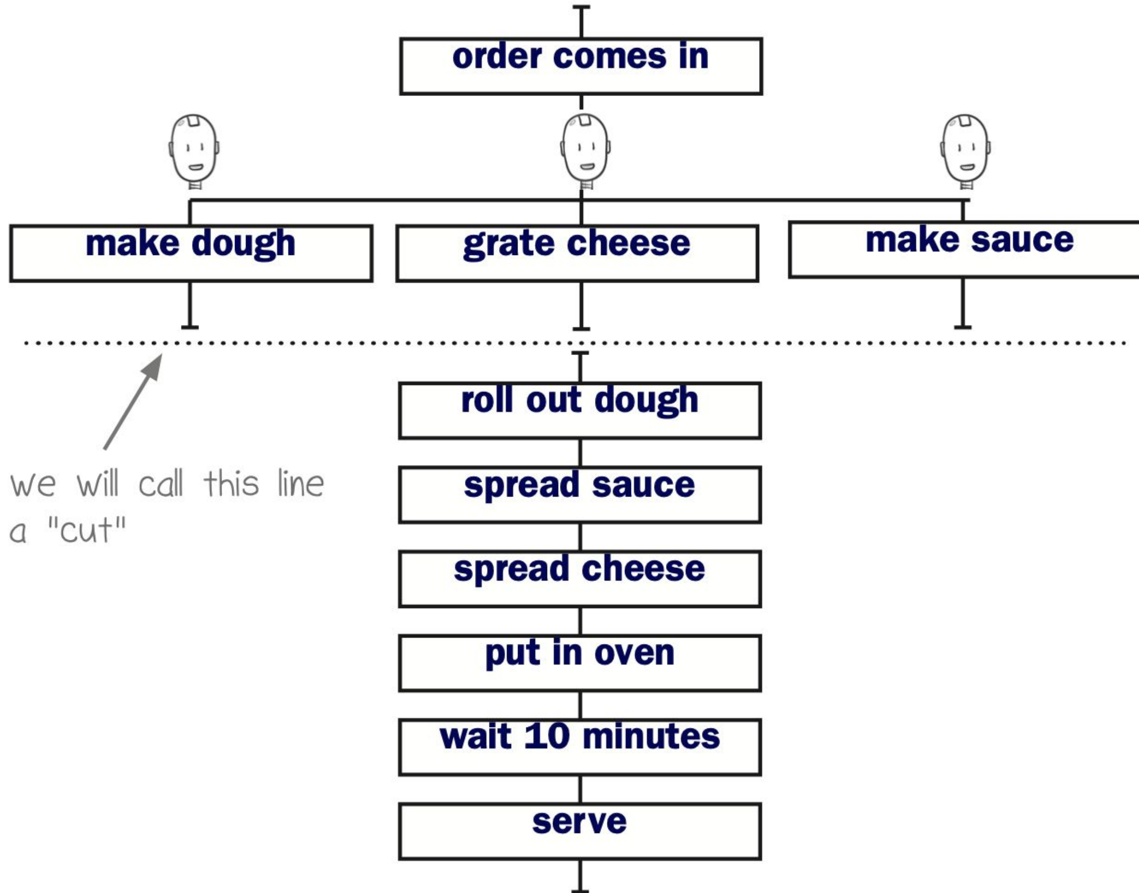








it only works when "make sauce" is last


3-robot setup with coordination
order comes in
make dough
grate cheese
make sauce
roll out dough
spread sauce
spread cheese
put in oven
wait 10 minutes
serve
we will call this line
a "cut"

Part 3: Building composable models

Modeling facts with data

The closure property lets us create infinitely complex expressions

Data can be interpreted in many way

A word about data modeling

Data modeling is next-level

You can get by without it

Just add complexity

Sum types, product types, combinatorial types

```
{
 "name": "pepperoni pizza",
 "ingredients": {
   "flour"      : 2,
   "tomatoes" : 5,
   "cheese"    : 1,
   "pepperoni": 1
 },
 "preparation": {
   "dough"     : {"prepare": "pizza-dough recipe"},
   "sauce"     : {"prepare": "tomato-sauce recipe"},
   "cheese"    : {"action" : "grate"},
   "pepperoni": {"action" : "slice"}
 },
 "assembly": [
   {"operation": "rollOut",
    "argument" : "dough"},
   {"operation": "spread",
    "argument" : "sauce"},
   {"operation": "spread",
    "argument" : "cheese"},
   {"operation": "spread",
    "argument" : "pepperoni"},
   {"operation": "bake",
    "argument" : "10 min"}
 ]
}
```

we can use objects because order of ingredients does not matter

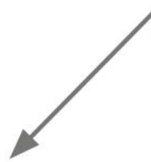
we use an array to capture the order of steps

these "operations" refer to function names

```
Pepperoni Pizza

Ingredients
2 cups flour, 5 tomatoes, 1 cup cheese, 1 link of pepperoni

Preparation
Prepare pizza dough (see pizza dough card)
Grate cheese
Prepare tomato sauce (see tomato sauce card)
Slice pepperoni

Assembly
Roll out dough
Spread sauce
Spread cheese
Spread pepperoni
Bake for 10 minutes
```

a suitable representation of the data on this card as JSON. notice they have similar structure

the operations take ingredient lists as arguments



```
ingredientListPlus({“pepperoni” : 1, “cheese” : 2}, {“cheese” : 2, “flour”   : 3})
```

```
{“pepperoni” : 1, “cheese” : 4, “flour” : 3}
```

the operations return ingredient lists as return values

```
ingredientsListPlus()
ingredientsListMinus()
ingredientsListTimes()
ingredientsListDivide()
ingredientsListSplit()
```

these five operations all share the closure property

Modeling a Starbucks coffee

Lots of options (tall, venti, grande, decaf, dark, medium, light, blonde, soy, espresso shot, almond, vanilla, etc, etc)

How to represent all of these that allows

- Calculate price
- Produce the coffee
- Track popularity
- Send it to a central location

Modeling a Starbucks coffee

```
{
  "size" : "grande",
  "brew" : "decaf",
  "additions" : [
    "soy",
    "almond"
  ]
}
```

Modeling a Starbucks coffee

```
{
 "size" : "grande",
 "brew" : "decaf",
 "additions" : {
   "soy": 1,
   "almond": 2
 }
}
```

Modeling a coffee editing process

Several different editing operations possible

Represent the intentions of the user to allow for

- Undo/redo
- Representation in previous coffee model

Modeling a coffee editing process

```
[
  ["set size", "venti"],
  ["set size", "grande"],
  ["add", "mocha"],
  ["set brew", "dark"],
  …
]
```

Where to find me

Blog: lispcast.com

Podcast: lispcast.com/podcast

Clojure & functional programming video courses: purelyfunctional.tv

Clojure Newsletter: purelyfunctional.tv/newsletter

Twitter: @ericnormand