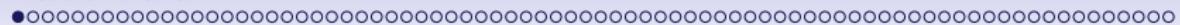




Abstraction

November 27, 2019



[https://en.wikipedia.org/
wiki/Abstract_nonsense](https://en.wikipedia.org/wiki/Abstract_nonsense)

Software Engineers and Patterns



Software Engineers and Patterns



- Software engineers love architecture principles.

Software Engineers and Patterns



- Software engineers love architecture principles.
 - Who has heard of *design patterns* and the gang of four book?

Software Engineers and Patterns



- Software engineers love architecture principles.
 - Who has heard of *design patterns* and the gang of four book?
 - The authors were actually inspired by a book that came up with a pattern language in architecture

Software Engineers and Patterns



- Software engineers love architecture principles.
 - Who has heard of *design patterns* and the gang of four book?
 - The authors were actually inspired by a book that came up with a pattern language in architecture
 - Such patterns were meant to provide guidelines across buildings in multiple style *abstracting* away concrete details of individual buildings.

Software Engineers and Patterns



- Software engineers love architecture principles.
 - Who has heard of *design patterns* and the gang of four book?
 - The authors were actually inspired by a book that came up with a pattern language in architecture
 - Such patterns were meant to provide guidelines across buildings in multiple style *abstracting* away concrete details of individual buildings.
 - Software design patterns are thus all about providing *abstractions*.



Theories of Abstraction

Let's talk about abstraction.



Theories of Abstraction

Let's talk about abstraction.



Theories of Abstraction

Let's talk about abstraction.



- Abstraction in visual art is about avoiding concrete subjects. It attempts to convey something (often an emotion) without appealing to sentiment.



Theories of Abstraction

Let's talk about abstraction.



- Abstraction in visual art is about avoiding concrete subjects. It attempts to convey something (often an emotion) without appealing to sentiment.
 - Oddly enough there's a bit of a stigma against it, despite the fact that music without lyrics has far less stigma.



Theories of Abstraction

Let's talk about abstraction.



- Abstraction in visual art is about avoiding concrete subjects. It attempts to convey something (often an emotion) without appealing to sentiment.
 - Oddly enough there's a bit of a stigma against it, despite the fact that music without lyrics has far less stigma.
 - Abstraction in mathematics and computer science is about *generalization*. Take away the concrete details of certain objects and see how they are similar.



Why Abstract?

This can lead us to *classify* different objects into related groups.

Why Abstract?

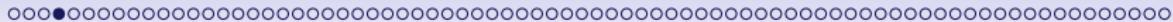
This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.

Why Abstract?

This can lead us to *classify* different objects into related groups.

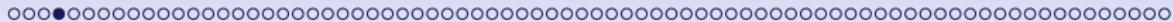
- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
 - This is important to computing because the point of programs are to construct some form of answer.



Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
 - This is important to computing because the point of programs are to construct some form of answer.
 - Although categorical abstractions are powerful, I will not discuss them much (and most of the really powerful abstractions go over my head).



Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
 - This is important to computing because the point of programs are to construct some form of answer.
 - Although categorical abstractions are powerful, I will not discuss them much (and most of the really powerful abstractions go over my head).
 - But we will study abstraction from a less mathematical viewpoint.



Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
 - This is important to computing because the point of programs are to construct some form of answer.
 - Although categorical abstractions are powerful, I will not discuss them much (and most of the really powerful abstractions go over my head).
 - But we will study abstraction from a less mathematical viewpoint.
 - I will show instances of *almost identical* code and how a programming language feature allows the two pieces of code to be generalized.

Buddy Functions

Let's consider the following functions in Java:

```
public boolean hasSmith(List<String> names) {  
    for (String name : names) {  
        if (name == "Smith")  
            return true;  
    }  
    return false;  
}  
  
public boolean hasBob(List<String> names) {  
    for (String name : names) {  
        if (name == "Bob")  
            return true;  
    }  
    return false;  
}
```

Buddy Functions

Let's consider the following functions in Java:

```
public boolean hasSmith(List<String> names) {  
    for (String name : names) {  
        if (name == "Smith")  
            return true;  
    }  
    return false;  
}  
  
public boolean hasBob(List<String> names) {  
    for (String name : names) {  
        if (name == "Bob")  
            return true;  
    }  
    return false;  
}
```

These look pretty similar, right?

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean hasName(String searchName, List<String> names) {  
    for (String name : names) {  
        if (name == searchName)  
            return true;  
    }  
    return false;  
}
```

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean hasName(String searchName, List<String> names) {  
    for (String name : names) {  
        if (name == searchName)  
            return true;  
    }  
    return false;  
}
```

- Why only design a function that searches for names?

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean hasName(String searchName, List<String> names) {  
    for (String name : names) {  
        if (name == searchName)  
            return true;  
    }  
    return false;  
}
```

- Why only design a function that searches for names?
- Why not search for arbitrary items so long as they are comparable?

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean hasName(String searchName, List<String> names) {  
    for (String name : names) {  
        if (name == searchName)  
            return true;  
    }  
    return false;  
}
```

- Why only design a function that searches for names?
- Why not search for arbitrary items so long as they are comparable?
- Then searching for names becomes an instance of a more general problem that is solved.

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>>
    boolean hasItem(T searchItem, List<T> items) {
        for (T item : items) {
            if (item.equals(searchItem))
                return true;
        }
        return false;
    }
```

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>>
    boolean hasItem(T searchItem, List<T> items) {
        for (T item : items) {
            if (item.equals(searchItem))
                return true;
        }
        return false;
    }
```

- Alright, now we can search for arbitrary comparable items!

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>>
    boolean hasItem(T searchItem, List<T> items) {
        for (T item : items) {
            if (item.equals(searchItem))
                return true;
        }
        return false;
    }
```

- Alright, now we can search for arbitrary comparable items!
- This is nice and general!

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>>
    boolean hasItem(T searchItem, List<T> items) {
        for (T item : items) {
            if (item.equals(searchItem))
                return true;
        }
        return false;
    }
```

- Alright, now we can search for arbitrary comparable items!
- This is nice and general!
- Can we generalize any more?

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>>
    boolean hasItem(T searchItem, List<T> items) {
        for (T item : items) {
            if (item.equals(searchItem))
                return true;
        }
        return false;
    }
```

- Alright, now we can search for arbitrary comparable items!
- This is nice and general!
- Can we generalize any more?
- We actually have 2 more abstractions that we can apply!

Abstract Nonsense

oo

Two Abstractions??



I'm Sorry



Clear Lectures

Lectures with memes

Back on Track

Ok, let's get down to business.

Back on Track

Ok, let's get down to business.



Back on Track

Ok, let's get down to business.



- We can actually consider searching for a specific item via equality as the process of seeing if an arbitrary predicate returns true for an item in a list.

Back on Track

Ok, let's get down to business.



- We can actually consider searching for a specific item via equality as the process of seeing if an arbitrary predicate returns true for an item in a list.
- So if we wanted to check if a string equaled smith we could write:

```
Predicate<String> isSmith = str -> str == "Smith";
```

Back on Track

Ok, let's get down to business.



- We can actually consider searching for a specific item via equality as the process of seeing if an arbitrary predicate returns true for an item in a list.
- So if we wanted to check if a string equaled smith we could write:

```
Predicate<String> isSmith = str -> str == "Smith";
```

- I could then pass this as the first argument to a function TestItems that I will now define.

Last Abstraction

Here is that function now:

```
public <T extends Comparable<T>>
    boolean testItems(Predicate<T> pred, List<T> items) {
        for (T item : items) {
            if (pred.test(searchItem))
                return true;
        }
        return false;
    }
```

Last Abstraction

Here is that function now:

```
public <T extends Comparable<T>>
    boolean testItems(Predicate<T> pred, List<T> items) {
        for (T item : items) {
            if (pred.test(searchItem))
                return true;
        }
        return false;
    }
```

- We can provide one last abstraction that applies predicates to every item in *any iterable collection*.

Last Abstraction

Here is that function now:

```
public <T extends Comparable<T>>
    boolean testItems(Predicate<T> pred, List<T> items) {
        for (T item : items) {
            if (pred.test(searchItem))
                return true;
        }
        return false;
    }
```

- We can provide one last abstraction that applies predicates to every item in *any iterable collection*.
- For example, if I define iterators over dictionaries or trees we should still be able to apply a predicate to them.

Last Abstraction

Here is that function now:

```
public <T extends Comparable<T>>
    boolean testItems(Predicate<T> pred, List<T> items) {
        for (T item : items) {
            if (pred.test(searchItem))
                return true;
        }
        return false;
    }
```

- We can provide one last abstraction that applies predicates to every item in *any iterable collection*.
- For example, if I define iterators over dictionaries or trees we should still be able to apply a predicate to them.

```
public <T extends Comparable<T>>
    boolean testItems(Predicate<T> pred, Iterable<T> items) {
        for (T item : items) {
            if (pred.test(item))
                return true;
        }
        return false;
```



Predicates as Arguments

Our third abstraction was a bit strange, right?

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.
- This is a function with no name, and it is useful if you only need to use a function in one place in your program.

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.
- This is a function with no name, and it is useful if you only need to use a function in one place in your program.
- The second thing is that we had a parameter that received a function as input.

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.
- This is a function with no name, and it is useful if you only need to use a function in one place in your program.
- The second thing is that we had a parameter that received a function as input.
- It then applied this function to every element in the list and observed whether it returned `true` for that element.

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.
- This is a function with no name, and it is useful if you only need to use a function in one place in your program.
- The second thing is that we had a parameter that received a function as input.
- It then applied this function to every element in the list and observed whether it returned `true` for that element.
- This is a very powerful concept where we can determine if perhaps all elements in a collection satisfy a property or even one element. Or we can collect all individuals in a collection that satisfy some property.

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.
- This is a function with no name, and it is useful if you only need to use a function in one place in your program.
- The second thing is that we had a parameter that received a function as input.
- It then applied this function to every element in the list and observed whether it returned `true` for that element.
- This is a very powerful concept where we can determine if perhaps all elements in a collection satisfy a property or even one element. Or we can collect all individuals in a collection that satisfy some property.
- For example, let's consider writing a program that only admits people that are 18 and older.

Guarding Entry

Let's consider that we have a list of people whose ages are represented by Natural numbers and that we want to only collect the people over 18. Assume a person has an age field that we can project.

Guarding Entry

Let's consider that we have a list of people whose ages are represented by Natural numbers and that we want to only collect the people over 18. Assume a person has an age field that we can project.

- How do we start writing such a function?

Guarding Entry

Let's consider that we have a list of people whose ages are represented by Natural numbers and that we want to only collect the people over 18. Assume a person has an age field that we can project.

- How do we start writing such a function?
- Via recursion over a list of course.

Guarding Entry

Let's consider that we have a list of people whose ages are represented by Natural numbers and that we want to only collect the people over 18. Assume a person has an age field that we can project.

- How do we start writing such a function?
- Via recursion over a list of course.
- I'll skip to providing a skeleton:

Guarding Entry

Let's consider that we have a list of people whose ages are represented by Natural numbers and that we want to only collect the people over 18. Assume a person has an age field that we can project.

- How do we start writing such a function?
- Via recursion over a list of course.
- I'll skip to providing a skeleton:

```
;; List<Person> -> List<Person>
;; Only allows patrons over 18 to enter the bar
(define (bar-entry patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons)
      (... (first patrons)) ... (bar-entry (rest patrons)))]))
```



Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparison.

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparison.

- Assume that (`first` patrons) returns
(person "Sara" 26).

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparison.

- Assume that (`first` patrons) returns
(person "Sara" 26).
- How do I get the age out?

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparison.

- Assume that (`first` `patrons`) returns
(`person "Sara" 26`).
- How do I get the age out?
- (`person-age` (`first` `patrons`))

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparison.

- Assume that (`first` `patrons`) returns
(`person "Sara" 26`).
- How do I get the age out?
- (`person-age` (`first` `patrons`))
- How do I check whether the age is less than 18?

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparison.

- Assume that (`first` `patrons`) returns
(`person "Sara" 26`).
- How do I get the age out?
- (`person-age` (`first` `patrons`))
- How do I check whether the age is less than 18?
- (`< (person-age` (`first` `patrons`)) `18`)

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparison.

- Assume that (`first` `patrons`) returns (`person "Sara"` 26).
- How do I get the age out?
- (`person-age` (`first` `patrons`))
- How do I check whether the age is less than 18?
- (`< (person-age (first patrons)) 18`)
- Thus, we should get (`< 26 18`) since we are getting Sara as our person, and this should return false.

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparison.

- Assume that (`first` `patrons`) returns (`person "Sara" 26`).
- How do I get the age out?
- (`person-age` (`first` `patrons`))
- How do I check whether the age is less than 18?
- (`< (person-age (first patrons)) 18`)
- Thus, we should get (`< 26 18`) since we are getting Sara as our person, and this should return false.
- Since this returns false, we should throw Sara into the list we're building. How do we do that?

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparison.

- Assume that (`first` `patrons`) returns (`person "Sara" 26`).
- How do I get the age out?
- (`person-age` (`first` `patrons`))
- How do I check whether the age is less than 18?
- (`<` (`person-age` (`first` `patrons`)) `18`)
- Thus, we should get (`< 26 18`) since we are getting Sara as our person, and this should return false.
- Since this returns false, we should throw Sara into the list we're building. How do we do that?
- With (`cons` (`first` `patrons`) ...)

Only Collecting *Some* Items

But what if (`first` patrons) returns (person "Dustin" 16)?

Only Collecting *Some* Items

But what if (`first` patrons) returns (person "Dustin" 16)?

- Then, we had

(< (person-age (person "Dustin" 16)) 18)

Only Collecting *Some* Items

But what if (`first` patrons) returns (person "Dustin" 16)?

- Then, we had
`(< (person-age (person "Dustin" 16)) 18)`
- which evaluates to `(< 16 18)` and then return true.

Only Collecting *Some* Items

But what if (`first` patrons) returns (person "Dustin" 16)?

- Then, we had
`(< (person-age (person "Dustin" 16)) 18)`
- which evaluates to `(< 16 18)` and then return true.
- This means that we must not add Dustin to our patron list.
So we shouldn't use a cons operation.

Only Collecting *Some* Items

But what if (`first` patrons) returns (person "Dustin" 16)?

- Then, we had
`(< (person-age (person "Dustin" 16)) 18)`
- which evaluates to `(< 16 18)` and then return true.
- This means that we must not add Dustin to our patron list.
So we shouldn't use a cons operation.
- There are two ways around this. One is to locally add a complicated if expression. The second is to design a new function that only does a cons operation if the first element of the list has an age greater than 18 and otherwise returns the sublist that already had filtered out underage people.

Only Collecting *Some* Items

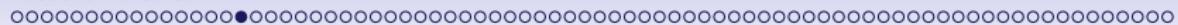
But what if (`first` patrons) returns (person "Dustin" 16)?

- Then, we had
`(< (person-age (person "Dustin" 16)) 18)`
- which evaluates to `(< 16 18)` and then return true.
- This means that we must not add Dustin to our patron list.
So we shouldn't use a cons operation.
- There are two ways around this. One is to locally add a complicated if expression. The second is to design a new function that only does a cons operation if the first element of the list has an age greater than 18 and otherwise returns the sublist that already had filtered out underage people.
- Let's consider designing this second function, called `cons-over-18`

Only Collecting *Some* Items

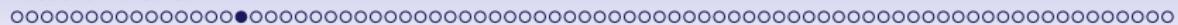
But what if (`first` patrons) returns (person "Dustin" 16)?

- Then, we had
`(< (person-age (person "Dustin" 16)) 18)`
- which evaluates to `(< 16 18)` and then return true.
- This means that we must not add Dustin to our patron list.
So we shouldn't use a cons operation.
- There are two ways around this. One is to locally add a complicated if expression. The second is to design a new function that only does a cons operation if the first element of the list has an age greater than 18 and otherwise returns the sublist that already had filtered out underage people.
- Let's consider designing this second function, called `cons-over-18`
- Does this function need to do recursion itself?



Designing a Helper Function

Thankfully, our helper function does not need to do recursion!



Designing a Helper Function

Thankfully, our helper function does not need to do recursion!

- Why?

Designing a Helper Function

Thankfully, our helper function does not need to do recursion!

- Why?
- We assume that the recursion for bar-entry already filters out people from sublists.

Designing a Helper Function

Thankfully, our helper function does not need to do recursion!

- Why?
- We assume that the recursion for bar-entry already filters out people from sublists.
- So, our helper function simply takes in a person and a list and only conses the person onto the list if they are over 18.

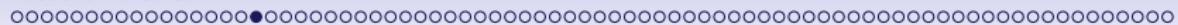
Designing a Helper Function

Thankfully, our helper function does not need to do recursion!

- Why?
- We assume that the recursion for bar-entry already filters out people from sublists.
- So, our helper function simply takes in a person and a list and only conses the person onto the list if they are over 18.

```
; ; List<person> -> List<person>
; ; Add a person to the patron list as
; ; long as they are over 18.
(define (cons-over-18 possible-patron patrons)
  (if (>= (person-age possible-patron) 18)
      (cons possible-patron patrons)
      patrons))
```





Finishing Our Bouncer Program

Alright, with our helper combinator we can structure our recursive function in the usual way, instead of having to add an additional condition in the body of the function.

Finishing Our Bouncer Program

Alright, with our helper combinator we can structure our recursive function in the usual way, instead of having to add an additional condition in the body of the function.

```
;; List<Person> -> List<Person>
;; Only allows patrons over 18 to enter the bar
(define (bar-entry patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons)
      (cons-over-18 (first patrons) (bar-entry (rest patrons))))]))
```



Finishing Our Bouncer Program

Alright, with our helper combinator we can structure our recursive function in the usual way, instead of having to add an additional condition in the body of the function.

```
;; List<Person> -> List<Person>
;; Only allows patrons over 18 to enter the bar
(define (bar-entry patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons)
      (cons-over-18 (first patrons) (bar-entry (rest patrons))))]))
```



- In general, if it seems like the list you're building depends on the structure of the values in the list, then you need some kind of cons operation that checks properties on the head of the list.

Finishing Our Bouncer Program

Alright, with our helper combinator we can structure our recursive function in the usual way, instead of having to add an additional condition in the body of the function.

```
;; List<Person> -> List<Person>
;; Only allows patrons over 18 to enter the bar
(define (bar-entry patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons)
      (cons-over-18 (first patrons) (bar-entry (rest patrons)))])))
```

-
- In general, if it seems like the list you're building depends on the structure of the values in the list, then you need some kind of cons operation that checks properties on the head of the list.
- But what happens if the condition we are checking needs to change?



Adjusting Our Bouncer Program

Concretely, let's say that that our bar was serving alcohol to patrons under 21 (this is illegal but common...) and a new Sheriff comes in and is stricter on enforcing alcohol laws.

Adjusting Our Bouncer Program

Concretely, let's say that that our bar was serving alcohol to patrons under 21 (this is illegal but common...) and a new Sheriff comes in and is stricter on enforcing alcohol laws.

- Now our bar can't afford to give drinks to patrons under 21 years old. We decide that the easiest solution is to only allow 21 and over patrons, to prevent underage people from getting in and sneakily get drinks.

Adjusting Our Bouncer Program

Concretely, let's say that that our bar was serving alcohol to patrons under 21 (this is illegal but common...) and a new Sheriff comes in and is stricter on enforcing alcohol laws.

- Now our bar can't afford to give drinks to patrons under 21 years old. We decide that the easiest solution is to only allow 21 and over patrons, to prevent underage people from getting in and sneakily get drinks.
- To model this behavior, our program must change `cons-over-18` to something named `cons-over-21` that changes its check to make sure patrons are at least 21.

Adjusting Our Bouncer Program

Concretely, let's say that our bar was serving alcohol to patrons under 21 (this is illegal but common...) and a new Sheriff comes in and is stricter on enforcing alcohol laws.

- Now our bar can't afford to give drinks to patrons under 21 years old. We decide that the easiest solution is to only allow 21 and over patrons, to prevent underage people from getting in and sneakily get drinks.
- To model this behavior, our program must change `cons-over-18` to something named `cons-over-21` that changes its check to make sure patrons are at least 21.
- But let's say we still need to model another bar that allows people over 18, but charges them covers. Then we will need four functions for our program.

Repetitive Bar Program



How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.



How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

- Redundant code should cause developers an itch.

How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

- Redundant code should cause developers an itch.
- On the one hand, there are more places for the program to go wrong.

How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

- Redundant code should cause developers an itch.
- On the one hand, there are more places for the program to go wrong.
- On the other, most developers get bored of re-reading similar code.

How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

- Redundant code should cause developers an itch.
- On the one hand, there are more places for the program to go wrong.
- On the other, most developers get bored of re-reading similar code.
- To minimize failure locations and keep interest, we should *abstract* out patterns in the code.

How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

- Redundant code should cause developers an itch.
- On the one hand, there are more places for the program to go wrong.
- On the other, most developers get bored of re-reading similar code.
- To minimize failure locations and keep interest, we should *abstract* out patterns in the code.
- So, how do we get started?

Adding Parameters

A rule of thumb when abstracting code is that either you need another parameter or you need another function.

Adding Parameters

A rule of thumb when abstracting code is that either you need another parameter or you need another function.

- Let's consider adding another parameter. We can add a boolean parameter that is a flag indicating whether to check if patrons are over 18 if the flag is false and over 21 if it is true.

Adding Parameters

A rule of thumb when abstracting code is that either you need another parameter or you need another function.

- Let's consider adding another parameter. We can add a boolean parameter that is a flag indicating whether to check if patrons are over 18 if the flag is false and over 21 if it is true.

Adding Parameters

A rule of thumb when abstracting code is that either you need another parameter or you need another function.

- Let's consider adding another parameter. We can add a boolean parameter that is a flag indicating whether to check if patrons are over 18 if the flag is false and over 21 if it is true.
- But wouldn't it be more elegant to just have a numeric parameter that is used as the threshold for our checks?

Adding Parameters

A rule of thumb when abstracting code is that either you need another parameter or you need another function.

- Let's consider adding another parameter. We can add a boolean parameter that is a flag indicating whether to check if patrons are over 18 if the flag is false and over 21 if it is true.
- But wouldn't it be more elegant to just have a numeric parameter that is used as the threshold for our checks?

```
(define (cons-over age possible-patron patrons)
  (if (>= (person-age possible-patron) age)
      (cons possible-patron patrons)
      patrons))

(define (bar-entry age patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons) (cons-over age
                                (first patrons)
                                (bar-entry (rest patrons)))])))
```





What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change cons-over to add an additional check.

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons_over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.
- But in general, as we add restrictions, we have to keep modifying `cons-over`.

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.
- But in general, as we add restrictions, we have to keep modifying `cons-over`.
- Is there any way we can abstract over these changes?

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.
- But in general, as we add restrictions, we have to keep modifying `cons-over`.
- Is there any way we can abstract over these changes?
- Just like the abstraction I provided at the beginning of this material in Java, we can add a predicate as a parameter!

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.
- But in general, as we add restrictions, we have to keep modifying `cons-over`.
- Is there any way we can abstract over these changes?
- Just like the abstraction I provided at the beginning of this material in Java, we can add a predicate as a parameter!
- We then only cons items that meet this predicate!

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.
- But in general, as we add restrictions, we have to keep modifying `cons-over`.
- Is there any way we can abstract over these changes?
- Just like the abstraction I provided at the beginning of this material in Java, we can add a predicate as a parameter!
- We then only cons items that meet this predicate!
- This is the power of having first class functions in your language!

Filtering Functions

You must be thinking “show me the code!” by now, so here it is:

```
(define (over-18? patron) (>= (person-age patron) 18))
(define (over-21? patron) (>= (person-age patron) 21))

(define (cons-over-p pred possible-patron patrons)
  (if (pred possible-patron)
      (cons possible-patron patrons)
      patrons))

(define (bar-entry pred patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons) (cons-over-p
                      pred
                      (first patrons)
                      (bar-entry (rest patrons)))])))
```

Accidental Derivations

Accidental Derivations

- We can now have our separate bars filter out by different ages with: (bar-entry over-18? PATRONS) and (bar-entry over-21? PATRONS)

Accidental Derivations

- We can now have our separate bars filter out by different ages with: (bar-entry over-18? PATRONS) and (bar-entry over-21? PATRONS)
- It turns out however that our functions have become general to the point of being badly named.

Accidental Derivations

- We can now have our separate bars filter out by different ages with: (bar-entry over-18? PATRONS) and (bar-entry over-21? PATRONS)
- It turns out however that our functions have become general to the point of being badly named.
- For example, I can call
`(bar-entry? string? '(1 "foo" (point 1 2) "bar"))`
and the function call returns '`("foo" "bar")`

Accidental Derivations

- We can now have our separate bars filter out by different ages with: (bar-entry over-18? PATRONS) and (bar-entry over-21? PATRONS)
- It turns out however that our functions have become general to the point of being badly named.
- For example, I can call
`(bar-entry? string? '(1 "foo" (point 1 2) "bar"))`
and the function call returns `'("foo" "bar")`
- It turns out our bouncer function ended up being able to get all of the lists out of a heterogeneous list.

Accidental Derivations

- We can now have our separate bars filter out by different ages with: (bar-entry over-18? PATRONS) and (bar-entry over-21? PATRONS)
- It turns out however that our functions have become general to the point of being badly named.
- For example, I can call
`(bar-entry? string? '(1 "foo" (point 1 2) "bar"))`
and the function call returns '`("foo" "bar")`
- It turns out our bouncer function ended up being able to get all of the lists out of a heterogeneous list.
- It turns out that we have derived a famous function and called it bar-entry!

Accidental Derivations

- We can now have our separate bars filter out by different ages with: (bar-entry over-18? PATRONS) and (bar-entry over-21? PATRONS)
- It turns out however that our functions have become general to the point of being badly named.
- For example, I can call
`(bar-entry? string? '(1 "foo" (point 1 2) "bar"))`
and the function call returns `'("foo" "bar")`
- It turns out our bouncer function ended up being able to get all of the lists out of a heterogeneous list.
- It turns out that we have derived a famous function and called it bar-entry!
- The name of this function is **filter**



Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!



Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call (`filter even? '(1 2 3 4 5 6)`)

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`
- The result is `'(2 4 6)`

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`
- The result is `'(2 4 6)`
- We can call:

```
(filter
  (lambda (s) (> (string-length s) 1))
  '("Smith" "Jenkins" "Brown" "Z"))
```

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`
- The result is `'(2 4 6)`
- We can call:

```
(filter
  (lambda (s) (> (string-length s) 1))
  '("Smith" "Jenkins" "Brown" "Z"))
```

- And this returns `'("Smith" "Jenkins" "Brown")`

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`
- The result is `'(2 4 6)`
- We can call:

```
(filter
  (lambda (s) (> (string-length s) 1))
  '("Smith" "Jenkins" "Brown" "Z"))
```

- And this returns `'("Smith" "Jenkins" "Brown")`
- You might be wondering what `(lambda (s) (> (string-length s) 1))` is.

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`
- The result is `'(2 4 6)`
- We can call:

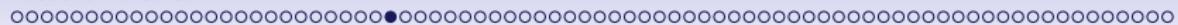
```
(filter
  (lambda (s) (> (string-length s) 1))
  '("Smith" "Jenkins" "Brown" "Z"))
```

- And this returns `'("Smith" "Jenkins" "Brown")`
- You might be wondering what `(lambda (s) (> (string-length s) 1))` is.
- It is an anonymous function, just like the one I previously showed in Java.



Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.



Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`

Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`
- `(lambda (x) (add1 x))` is an anonymous function with one parameter `x` that increments the number.



Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`
- `(lambda (x) (add1 x))` is an anonymous function with one parameter `x` that increments the number.
- We can also take functions as arguments to functions.

Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`
- `(lambda (x) (add1 x))` is an anonymous function with one parameter `x` that increments the number.
- We can also take functions as arguments to functions.
- `(filter (lambda (x) (< x 5)) '(1 2 4 8 16))` provides an anonymous function (that is a predicate) as an argument to `filter`



Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`
- `(lambda (x) (add1 x))` is an anonymous function with one parameter `x` that increments the number.
- We can also take functions as arguments to functions.
- `(filter (lambda (x) (< x 5)) '(1 2 4 8 16))` provides an anonymous function (that is a predicate) as an argument to `filter`
- We can also write functions that return functions.

Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

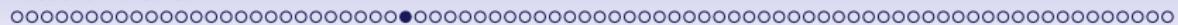
- In Racket, we introduce an anonymous function with `lambda`
- `(lambda (x) (add1 x))` is an anonymous function with one parameter `x` that increments the number.
- We can also take functions as arguments to functions.
- `(filter (lambda (x) (< x 5)) '(1 2 4 8 16))` provides an anonymous function (that is a predicate) as an argument to `filter`
- We can also write functions that return functions.
- Here is a small example:

```
(define (add x)
  (lambda (y)
    (+ x y)))
```



Returning Functions

What do you think the following code is doing?



Returning Functions

What do you think the following code is doing?

```
(define add-two (add 2))  
(define add-three (add 3))
```

-

Returning Functions

What do you think the following code is doing?

```
(define add-two (add 2))  
(define add-three (add 3))
```

-
- Well, let's substitute 2 for x and then look at the body of the original function:

```
(lambda (y)  
  (+ 2 y))
```

Returning Functions

What do you think the following code is doing?

```
(define add-two (add 2))  
(define add-three (add 3))
```

-
- Well, let's substitute 2 for x and then look at the body of the original function:

```
(lambda (y)  
  (+ 2 y))
```

- And when we apply 3 in defining add-three we substitute 3 and have:

```
(lambda (y)  
  (+ 3 y))
```

Returning Functions

What do you think the following code is doing?

```
(define add-two (add 2))  
(define add-three (add 3))
```

-
- Well, let's substitute 2 for x and then look at the body of the original function:

```
(lambda (y)  
  (+ 2 y))
```
- And when we apply 3 in defining add-three we substitute 3 and have:

```
(lambda (y)  
  (+ 3 y))
```
- So, in the first case we are returning an anonymous function that adds 2 to y, and in the latter case we are returning an anonymous function that adds 3 to y.

Returning Functions

What do you think the following code is doing?

```
(define add-two (add 2))  
(define add-three (add 3))
```

- Well, let's substitute 2 for x and then look at the body of the original function:

```
(lambda (y)  
  (+ 2 y))
```

- And when we apply 3 in defining add-three we substitute 3 and have:

```
(lambda (y)  
  (+ 3 y))
```

- So, in the first case we are returning an anonymous function that adds 2 to y, and in the latter case we are returning an anonymous function that adds 3 to y.
- So, in a sense we are deanonymizing these two functions by using define with add-two and add-three.

Previously on Functional Programming with Racket

Last time, we started by trying to make a bar tender program that was easy to modify to changing requirements.



Previously on Functional Programming with Racket

Last time, we started by trying to make a bar tender program that was easy to modify to changing requirements.



And we ended up with something different:



Famous Higher Order Functions

I want to motivate filter some more, but first we'll go over some famous higher order functions that you may have seen from math courses. Namely derivation, integration, and composition.

Famous Higher Order Functions

I want to motivate filter some more, but first we'll go over some famous higher order functions that you may have seen from math courses. Namely derivation, integration, and composition.

- Let's first think about derivation:

Famous Higher Order Functions

I want to motivate filter some more, but first we'll go over some famous higher order functions that you may have seen from math courses. Namely derivation, integration, and composition.

- Let's first think about derivation:
- What is $\frac{d}{dx} x^2$?

Famous Higher Order Functions

I want to motivate filter some more, but first we'll go over some famous higher order functions that you may have seen from math courses. Namely derivation, integration, and composition.

- Let's first think about derivation:
- What is $\frac{d}{dx} x^2$?
- It is another function written in terms of x : $2x$.

Famous Higher Order Functions

I want to motivate filter some more, but first we'll go over some famous higher order functions that you may have seen from math courses. Namely derivation, integration, and composition.

- Let's first think about derivation:
- What is $\frac{d}{dx} x^2$?
- It is another function written in terms of x : $2x$.
- Similarly, we know that the integration of $2x$ is $x^2 + C$ for some constant C .

Famous Higher Order Functions

I want to motivate filter some more, but first we'll go over some famous higher order functions that you may have seen from math courses. Namely derivation, integration, and composition.

- Let's first think about derivation:
- What is $\frac{d}{dx} x^2$?
- It is another function written in terms of x : $2x$.
- Similarly, we know that the integration of $2x$ is $x^2 + C$ for some constant C .
- In general, integration is uncomputable, so we'll focus on programming an approximate derivative by approximating the limit rule with a “small enough” change.

Famous Higher Order Functions

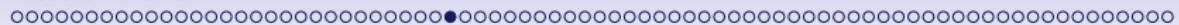
I want to motivate filter some more, but first we'll go over some famous higher order functions that you may have seen from math courses. Namely derivation, integration, and composition.

- Let's first think about derivation:
- What is $\frac{d}{dx} x^2$?
- It is another function written in terms of x : $2x$.
- Similarly, we know that the integration of $2x$ is $x^2 + C$ for some constant C .
- In general, integration is uncomputable, so we'll focus on programming an approximate derivative by approximating the limit rule with a “small enough” change.

```
(define delta-h 1/100000000000000)
```

```
(define (deriv f)
  (lambda (x)
    (exact->inexact
      (/ (- (f (+ x delta-h)) (f x))
          delta-h))))
```





Famous Higher Order Functions

We can then use our derivative calculating function to create new functions:

Famous Higher Order Functions

We can then use our derivative calculating function to create new functions:

- (`define double (deriv sqr)`)

Famous Higher Order Functions

We can then use our derivative calculating function to create new functions:

- (`define double (deriv sqr)`)
- I have taken the derivative of the square function and defined a new function `double`, to be the result.

Famous Higher Order Functions

We can then use our derivative calculating function to create new functions:

- (`define double (deriv sqr)`)
- I have taken the derivative of the square function and defined a new function `double`, to be the result.
- I can now call `(double 3)` to get 6.000000000000001

Famous Higher Order Functions

We can then use our derivative calculating function to create new functions:

- (`define double (deriv sqr)`)
- I have taken the derivative of the square function and defined a new function `double`, to be the result.
- I can now call `(double 3)` to get `6.000000000000001`
- If I now need to calculate numerical derivatives of functions repeatedly in some program, I can use this `deriv` function.

Famous Higher Order Functions

We can then use our derivative calculating function to create new functions:

- (`define double (deriv sqr)`)
- I have taken the derivative of the square function and defined a new function `double`, to be the result.
- I can now call `(double 3)` to get `6.000000000000001`
- If I now need to calculate numerical derivatives of functions repeatedly in some program, I can use this `deriv` function.
- For example, I can take `(deriv double)` and then get a constant function that always returns (approximately) `2`.

Famous Higher Order Functions

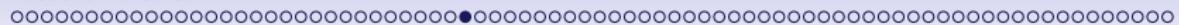
We can then use our derivative calculating function to create new functions:

- (`define double (deriv sqr)`)
- I have taken the derivative of the square function and defined a new function `double`, to be the result.
- I can now call `(double 3)` to get `6.000000000000001`
- If I now need to calculate numerical derivatives of functions repeatedly in some program, I can use this `deriv` function.
- For example, I can take `(deriv double)` and then get a constant function that always returns (approximately) `2`.
- In general, the idea of treating functions as data that can be manipulated is interesting and comprises entire areas of math and computer science.

Famous Higher Order Functions

We can then use our derivative calculating function to create new functions:

- (`define double (deriv sqr)`)
- I have taken the derivative of the square function and defined a new function `double`, to be the result.
- I can now call `(double 3)` to get 6.000000000000001
- If I now need to calculate numerical derivatives of functions repeatedly in some program, I can use this `deriv` function.
- For example, I can take `(deriv double)` and then get a constant function that always returns (approximately) 2 .
- In general, the idea of treating functions as data that can be manipulated is interesting and comprises entire areas of math and computer science.
- In particular, the idea of modern AI is to learn functions from large sets of data or live input sources.



Programs are Composition!

In our math education, we are taught about composition rather abstractly.



Programs are Composition!

In our math education, we are taught about composition rather abstractly.

- But in this class we learn that programs can be represented as compositions of pure functions, and we designed (small but) real programs in this fashion.

Programs are Composition!

In our math education, we are taught about composition rather abstractly.

- But in this class we learn that programs can be represented as compositions of pure functions, and we designed (small but) real programs in this fashion.
- But the composition function, \circ , is a function of type:
 $(A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$ that takes two functions as input and returns a new function as output.

Programs are Composition!

In our math education, we are taught about composition rather abstractly.

- But in this class we learn that programs can be represented as compositions of pure functions, and we designed (small but) real programs in this fashion.
- But the composition function, \circ , is a function of type:
 $(A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$ that takes two functions as input and returns a new function as output.

Programs are Composition!

In our math education, we are taught about composition rather abstractly.

- But in this class we learn that programs can be represented as compositions of pure functions, and we designed (small but) real programs in this fashion.
- But the composition function, \circ , is a function of type:
 $(A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$ that takes two functions as input and returns a new function as output.
- For example, $y^2 \circ 2x$, returns a function that first doubles a number and then squares it: $(2x)^2$.

Programs are Composition!

In our math education, we are taught about composition rather abstractly.

- But in this class we learn that programs can be represented as compositions of pure functions, and we designed (small but) real programs in this fashion.
- But the composition function, \circ , is a function of type:
 $(A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$ that takes two functions as input and returns a new function as output.
- For example, $y^2 \circ 2x$, returns a function that first doubles a number and then squares it: $(2x)^2$.
- To define this in racket, we need a function that takes in two arguments as input and returns an anonymous function as output.

Programs are Composition!

In our math education, we are taught about composition rather abstractly.

- But in this class we learn that programs can be represented as compositions of pure functions, and we designed (small but) real programs in this fashion.
- But the composition function, \circ , is a function of type:
 $(A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$ that takes two functions as input and returns a new function as output.
- For example, $y^2 \circ 2x$, returns a function that first doubles a number and then squares it: $(2x)^2$.
- To define this in racket, we need a function that takes in two arguments as input and returns an anonymous function as output.

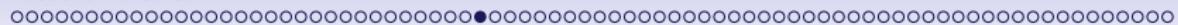
```
(define (compose f g)
  (lambda (x) (f (g x))))
```





Composing Functions

We can compose simple functions like inverses:



Composing Functions

We can compose simple functions like inverses:

- Here's an unnecessarily complicated definition of the identity function: (`define id (compose sqrt sqr)`)

Composing Functions

We can compose simple functions like inverses:

- Here's an unnecessarily complicated definition of the identity function: (`define id (compose sqrt sqr)`)
- But in general, we can replace certain programming patterns with composition.

Composing Functions

We can compose simple functions like inverses:

- Here's an unnecessarily complicated definition of the identity function: (`define id (compose sqrt sqr)`)
- But in general, we can replace certain programming patterns with composition.
- For example, we can replace the many bindings in programs like:

```
(define old-tank (aim-tank game-state))
(define old-loc (tank-loc old-tank))
(define new-loc (add1 old-loc))
(define new-tank (tank-set-loc old-tank new-loc)
(aim (aim-ufo game-state) new-tank)
```

Composing Functions

We can compose simple functions like inverses:

- Here's an unnecessarily complicated definition of the identity function: (`define id (compose sqrt sqr)`)
- But in general, we can replace certain programming patterns with composition.
- For example, we can replace the many bindings in programs like:

```
(define old-tank (aim-tank game-state))
(define old-loc (tank-loc old-tank))
(define new-loc (add1 old-loc))
(define new-tank (tank-set-loc old-tank new-loc)
(aim (aim-ufo game-state) new-tank))
```

- With a (nearly) *pointfree* program that uses `compose`:

Composing Functions

We can compose simple functions like inverses:

- Here's an unnecessarily complicated definition of the identity function: (`define id (compose sqrt sqr)`)
- But in general, we can replace certain programming patterns with composition.
- For example, we can replace the many bindings in programs like:

```
(define old-tank (aim-tank game-state))
(define old-loc (tank-loc old-tank))
(define new-loc (add1 old-loc))
(define new-tank (tank-set-loc old-tank new-loc)
(aim (aim-ufo game-state) new-tank))
```

- With a (nearly) *pointfree* program that uses compose:

```
(aim (aim-ufo game-state)
  ((compose
    (lambda (t l) tank-set-loc) add1 tank-loc aim-tank) game-state)
  (tank-vel (aim-tank game-state))))
```



Back To Abstraction

Note that the previous program was ugly, and used a version of `compose` that took in an arbitrary amount of arguments. Despite the ugliness, it's interesting that we can represent a series of sequential computations that involve assigning to local bindings, to a version that uses pointfree functions and avoids adding variable names.

Back To Abstraction

Note that the previous program was ugly, and used a version of compose that took in an arbitrary amount of arguments. Despite the ugliness, it's interesting that we can represent a series of sequential computations that involve assigning to local bindings, to a version that uses pointfree functions and avoids adding variable names.

- However, this lecture is about abstractions, and I need to illustrate why being able to define functions like the pointfree one above is useful to abstracting away unnecessary details from programs.

Back To Abstraction

Note that the previous program was ugly, and used a version of compose that took in an arbitrary amount of arguments. Despite the ugliness, it's interesting that we can represent a series of sequential computations that involve assigning to local bindings, to a version that uses pointfree functions and avoids adding variable names.

- However, this lecture is about abstractions, and I need to illustrate why being able to define functions like the pointfree one above is useful to abstracting away unnecessary details from programs.
- In general, a specific form of composition is useful for sequencing computations in a *purely* functional programming language like Haskell. But this discussion is too advanced and abstract for now.



Infimums and Supremums

I need to motivate the need for higher order functions some more.

Infimums and Supremums

I need to motivate the need for higher order functions some more.

- Let's consider writing two programs. One finds the minimum element in a list of totally orderable items, and the other finds the maximum element in a list of totally orderable items.

Infimums and Supremums

I need to motivate the need for higher order functions some more.

- Let's consider writing two programs. One finds the minimum element in a list of totally orderable items, and the other finds the maximum element in a list of totally orderable items.
- These are sometimes called computing the infimum and supremum of sets, respectively (or the meet and join of a set).

Infimums and Supremums

I need to motivate the need for higher order functions some more.

- Let's consider writing two programs. One finds the minimum element in a list of totally orderable items, and the other finds the maximum element in a list of totally orderable items.
- These are sometimes called computing the infimum and supremum of sets, respectively (or the meet and join of a set).
- Needing to do this comes up in programming a lot.

Infimums and Supremums

I need to motivate the need for higher order functions some more.

- Let's consider writing two programs. One finds the minimum element in a list of totally orderable items, and the other finds the maximum element in a list of totally orderable items.
- These are sometimes called computing the infimum and supremum of sets, respectively (or the meet and join of a set).
- Needing to do this comes up in programming a lot.
- *Especially* in research, because...

Infimums and Supremums

I need to motivate the need for higher order functions some more.

- Let's consider writing two programs. One finds the minimum element in a list of totally orderable items, and the other finds the maximum element in a list of totally orderable items.
- These are sometimes called computing the infimum and supremum of sets, respectively (or the meet and join of a set).
- Needing to do this comes up in programming a lot.
- *Especially* in research, because...





Infimums

So, let's start with trying to find the infimum of list of numbers.



Infimums

So, let's start with trying to find the infimum of list of numbers.

- Let's think about '(4 2 5 9 3 23 12)



Infimums

So, let's start with trying to find the infimum of list of numbers.

- Let's think about '(4 2 5 9 3 23 12)
- As usual, if we need to think about doing computation over a whole list, we need recursion.

Infimums

So, let's start with trying to find the infimum of list of numbers.

- Let's think about '(4 2 5 9 3 23 12)
- As usual, if we need to think about doing computation over a whole list, we need recursion.
- Thinking about recursion by starting with the first element of the list is difficult, so let's process the list in reverse order.

Infimums

So, let's start with trying to find the infimum of list of numbers.

- Let's think about '(4 2 5 9 3 23 12)
- As usual, if we need to think about doing computation over a whole list, we need recursion.
- Thinking about recursion by starting with the first element of the list is difficult, so let's process the list in reverse order.
- For these programs, we need to assume we have non-empty lists as input. I.E. our base case is a list of length 1.

Infimums

So, let's start with trying to find the infimum of list of numbers.

- Let's think about '(4 2 5 9 3 23 12)
- As usual, if we need to think about doing computation over a whole list, we need recursion.
- Thinking about recursion by starting with the first element of the list is difficult, so let's process the list in reverse order.
- For these programs, we need to assume we have non-empty lists as input. I.E. our base case is a list of length 1.
- In this case, the infimum of the one element list is simply the element in the list.

Infimums

So, let's start with trying to find the infimum of list of numbers.

- Let's think about `'(4 2 5 9 3 23 12)`
- As usual, if we need to think about doing computation over a whole list, we need recursion.
- Thinking about recursion by starting with the first element of the list is difficult, so let's process the list in reverse order.
- For these programs, we need to assume we have non-empty lists as input. I.E. our base case is a list of length 1.
- In this case, the infimum of the one element list is simply the element in the list.
- So, `(inf '(12))` returns 12.

Infimums

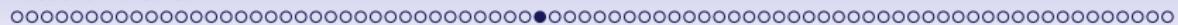
So, let's start with trying to find the infimum of list of numbers.

- Let's think about `'(4 2 5 9 3 23 12)`
- As usual, if we need to think about doing computation over a whole list, we need recursion.
- Thinking about recursion by starting with the first element of the list is difficult, so let's process the list in reverse order.
- For these programs, we need to assume we have non-empty lists as input. I.E. our base case is a list of length 1.
- In this case, the infimum of the one element list is simply the element in the list.
- So, `(inf '(12))` returns 12.
- What should `(inf '(23 12))` return?



Infimums

We need to compare 23 and 12, right?



Infimums

We need to compare 23 and 12, right?

- Since 23 is *not less* than 12, then $(\inf' (23 \ 12))$ is 12.

Infimums

We need to compare 23 and 12, right?

- Since 23 is *not less* than 12, then $(\inf \ ' (23 \ 12))$ is 12.
- Then, when we consider when 3 is the head of the list and $(\inf \ ' (23 \ 12))$ is 12.

Infimums

We need to compare 23 and 12, right?

- Since 23 is *not less* than 12, then $(\inf '(23 \ 12))$ is 12.
- Then, when we consider when 3 is the head of the list and $(\inf '(23 \ 12))$ is 12.
- We see that 3 is less than 12, and so the infimum of this sublist $(\inf '(3 \ 23 \ 12))$ is 3.

Infimums

We need to compare 23 and 12, right?

- Since 23 is *not less* than 12, then $(\inf '(23 \ 12))$ is 12.
- Then, when we consider when 3 is the head of the list and $(\inf '(23 \ 12))$ is 12.
- We see that 3 is less than 12, and so the infimum of this sublist $(\inf '(3 \ 23 \ 12))$ is 3.
- We can continue on with this process until we find that 2 is the infimum of our original list.

Infimums

We need to compare 23 and 12, right?

- Since 23 is *not less* than 12, then $(\inf '(23 \ 12))$ is 12.
- Then, when we consider when 3 is the head of the list and $(\inf '(23 \ 12))$ is 12.
- We see that 3 is less than 12, and so the infimum of this sublist $(\inf '(3 \ 23 \ 12))$ is 3.
- We can continue on with this process until we find that 2 is the infimum of our original list.
- So, how do we go about coding such an algorithm?

Infimums

We need to compare 23 and 12, right?

- Since 23 is *not less* than 12, then $(\inf \ '(23 \ 12))$ is 12.
- Then, when we consider when 3 is the head of the list and $(\inf \ '(23 \ 12))$ is 12.
- We see that 3 is less than 12, and so the infimum of this sublist $(\inf \ '(3 \ 23 \ 12))$ is 3.
- We can continue on with this process until we find that 2 is the infimum of our original list.
- So, how do we go about coding such an algorithm?
- We start with our recursive skeleton as usual.

Infimums

The skeleton is straightforward, except now we check lists of length 1 as our base case:

Infimums

The skeleton is straightforward, except now we check lists of length 1 as our base case:

```
; Nelson -> Number
; determines the smallest
; number on l
(define (inf l)
  (cond
    [(empty? (rest l)) ...]
    [else
      ... (first l) ... (inf (rest l))]))
```



Infimums

The skeleton is straightforward, except now we check lists of length 1 as our base case:

```
; Nelson -> Number
; determines the smallest
; number on l
(define (inf l)
  (cond
    [(empty? (rest l)) ...]
    [else
      ... (first l) ... (inf (rest l))]))
```

-
- Notice that like our bar-entry function, we update our answer *conditionally*.

Infimums

The skeleton is straightforward, except now we check lists of length 1 as our base case:

```
; Nelson -> Number
; determines the smallest
; number on l
(define (inf l)
  (cond
    [(empty? (rest l)) ...]
    [else
      ... (first l) ... (inf (rest l))]))
```

-
- Notice that like our bar-entry function, we update our answer *conditionally*.
- Basically, the code for putting together our recursive case is:

Infimums

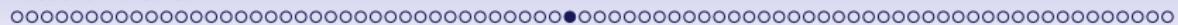
The skeleton is straightforward, except now we check lists of length 1 as our base case:

```
; Nelson -> Number
; determines the smallest
; number on l
(define (inf l)
  (cond
    [(empty? (rest l)) ...]
    [else
      ... (first l) ... (inf (rest l))]))
```

-
- Notice that like our bar-entry function, we update our answer *conditionally*.
- Basically, the code for putting together our recursive case is:

```
(if (< (first l)
        (inf (rest l)))
    (first l)
    (inf (rest l)))
```

-



Infimums

Like earlier, I think the code looks better if we would design a helper function instead of using a local if, but we will keep this for now. What would such helper function do, and what should it be called?



Infimums

Like earlier, I think the code looks better if we would design a helper function instead of using a local if, but we will keep this for now. What would such helper function do, and what should it be called?

- It would be called `min` of course.

Infimums

Like earlier, I think the code looks better if we would design a helper function instead of using a local if, but we will keep this for now. What would such helper function do, and what should it be called?

- It would be called `min` of course.
- Here is our final version:

```
; Nelson -> Number
; determines the smallest
; number on l
(define (inf l)
  (cond
    [(empty? (rest l))
     (first l)]
    [else
     (if (< (first l)
              (inf (rest l)))
         (first l)
         (inf (rest l))))]))
```



Defining Supremum

If I now want to define computing the supremum, how would I go about doing it?

Defining Supremum

If I now want to define computing the supremum, how would I go about doing it?

- Well, instead of getting the minimum element, we get the maximum—so we just change a < to a >:

Defining Supremum

If I now want to define computing the supremum, how would I go about doing it?

- Well, instead of getting the minimum element, we get the maximum—so we just change a < to a >:

```
; Nelson -> Number
; determines the largest
; number on l
(define (sup l)
  (cond
    [(empty? (rest l))
     (first l)]
    [else
     (if (> (first l)
              (sup (rest l)))
         (first l)
         (sup (rest l))))]))
```





Abstracting The Two

So, our code for the two functions differs by one thing, which comparison operator we want to use on the collection.



Abstracting The Two

So, our code for the two functions differs by one thing, which comparison operator we want to use on the collection.

- So, how should we abstract this out?

Abstracting The Two

So, our code for the two functions differs by one thing, which comparison operator we want to use on the collection.

- So, how should we abstract this out?
- As usual, let's take a predicate as an argument. But this time, it's a binary predicate!

Abstracting The Two

So, our code for the two functions differs by one thing, which comparison operator we want to use on the collection.

- So, how should we abstract this out?
- As usual, let's take a predicate as an argument. But this time, it's a binary predicate!

```
; Nelson -> Number
; determines the "largest"
; number on l according to p
(define (extremum l p)
  (cond
    [(empty? (rest l))
     (first l)]
    [else
     (if (p (first l)
            (extremum (rest l) p))
         (first l)
         (extremum (rest l) p))]))
```





Can We Abstract Further?

Can we abstract futher?



Can We Abstract Further?

Can we abstract futher?

- This idea of repeatedly doing a binary operation over a collection of items seems similar to another program we defined, right? Which one?



Can We Abstract Further?

Can we abstract futher?

- This idea of repeatedly doing a binary operation over a collection of items seems similar to another program we defined, right? Which one?
- It was similar to our functions that summed a list of numbers!

Can We Abstract Further?

Can we abstract futher?

- This idea of repeatedly doing a binary operation over a collection of items seems similar to another program we defined, right? Which one?
- It was similar to our functions that summed a list of numbers!
- In that, we repeatedly applied `+` on the numbers in the list.

Can We Abstract Further?

Can we abstract futher?

- This idea of repeatedly doing a binary operation over a collection of items seems similar to another program we defined, right? Which one?
- It was similar to our functions that summed a list of numbers!
- In that, we repeatedly applied `+` on the numbers in the list.
- These kind of functions are called *reducing functions*, *folds*, or *catamorphisms* (I prefer saying folds).

Can We Abstract Further?

Can we abstract futher?

- This idea of repeatedly doing a binary operation over a collection of items seems similar to another program we defined, right? Which one?
- It was similar to our functions that summed a list of numbers!
- In that, we repeatedly applied `+` on the numbers in the list.
- These kind of functions are called *reducing functions*, *folds*, or *catamorphisms* (I prefer saying folds).
- But before covering folds, I'd prefer to go over another class of higher order function.



Squaring and Summing

To motivate our new type of higher order function, let's consider programming the following program. Given a list of numbers, sum the squares of the odd numbers in the list. Let's assume we have a function `sum` to do the totalling.



Squaring and Summing

To motivate our new type of higher order function, let's consider programming the following program. Given a list of numbers, sum the squares of the odd numbers in the list. Let's assume we have a function `sum` to do the totalling.

- Where should we start our program?

Squaring and Summing

To motivate our new type of higher order function, let's consider programming the following program. Given a list of numbers, sum the squares of the odd numbers in the list. Let's assume we have a function `sum` to do the totalling.

- Where should we start our program?
- Well it sounds like we will need recursion to square each odd number.

Squaring and Summing

To motivate our new type of higher order function, let's consider programming the following program. Given a list of numbers, sum the squares of the odd numbers in the list. Let's assume we have a function `sum` to do the totalling.

- Where should we start our program?
- Well it sounds like we will need recursion to square each odd number.

```
; ; List<number> -> number
; ; Sum the odd numbers' squares in some list of numbers
(define (sum-odd-squares num-list)
  ...
  (cond
    [(empty? num-list) '()]
    [(cons? num-list)
     (sqr ... (first num-list))
     ... (sum-odd-squares (rest num-list) ...) ) ...]))
```





Breaking it Down



©2015 Chari Pere www.playinggrownup.com

Breaking it Down



©2015 Chari Pere www.playinggrownup.com

- The complexity of the skeleton really means that we need to define two functions.

Breaking it Down



- The complexity of the skeleton really means that we need to define two functions.
 1. The top level function which feeds the odd numbers to a function that squares each, which is then fed to sum.

Breaking it Down



©2015 Chari Pere www.playinggrownup.com

- The complexity of the skeleton really means that we need to define two functions.
 1. The top level function which feeds the odd numbers to a function that squares each, which is then fed to sum.
 2. And the function that does the squaring of each number.

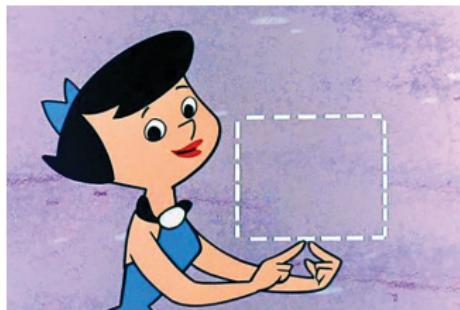
Breaking it Down



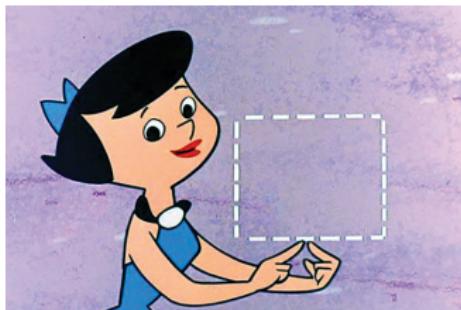
©2015 Chari Pere www.playinggrownup.com

- The complexity of the skeleton really means that we need to define two functions.
 1. The top level function which feeds the odd numbers to a function that squares each, which is then fed to sum.
 2. And the function that does the squaring of each number.
- Here's the final version of the first function:
`(define (sum-odd-squares num-list)
 ((compose sum square-each) (filter odd? num-list)))`

Square Up



Square Up



- The skeleton for our function to square each number is as follows:

```
; ; List<Number> -> List<Number>
; ; Square Each number in a list
(define (square-each num-list)
  (cond
    [(empty? num-list) ...]
    [(cons? num-list)
     ... sqr
     ... (first num-list)
     ... (square-each (rest num-list)) ...]]))
```



Fill Out Our Skeleton

Let's fill out our skeleton now.



Fill Out Our Skeleton

Let's fill out our skeleton now.

- `(empty? num-list) ...]`



Fill Out Our Skeleton

Let's fill out our skeleton now.

- `[(empty? num-list) ...]`
- We are producing a list of numbers as the output of our function. What should we replace the above ... with?

Fill Out Our Skeleton

Let's fill out our skeleton now.

- `[(empty? num-list) ...]`
- We are producing a list of numbers as the output of our function. What should we replace the above ... with?
- Our good friend `'()`

Fill Out Our Skeleton

Let's fill out our skeleton now.

- [`(empty? num-list) ...`]
- We are producing a list of numbers as the output of our function. What should we replace the above ... with?
- Our good friend `'()`
- Let's start to fill in the first ... in the else clause



Fill Out Our Skeleton

Let's fill out our skeleton now.

- `[(empty? num-list) ...]`
- We are producing a list of numbers as the output of our function. What should we replace the above ... with?
- Our good friend `'()`
- Let's start to fill in the first ... in the else clause
- `... sqr ... (first num-list) ...`

Fill Out Our Skeleton

Let's fill out our skeleton now.

- `[(empty? num-list) ...]`
- We are producing a list of numbers as the output of our function. What should we replace the above ... with?
- Our good friend `'()`
- Let's start to fill in the first ... in the else clause
- `... sqr ... (first num-list) ...`
- What have we been using to build lists recursively?

Fill Out Our Skeleton

Let's fill out our skeleton now.

- `[(empty? num-list) ...]`
- We are producing a list of numbers as the output of our function. What should we replace the above ... with?
- Our good friend `'()`
- Let's start to fill in the first ... in the else clause
- `... sqr ... (first num-list) ...`
- What have we been using to build lists recursively?
- Why our good friend `cons`, of course! Here he is:

Fill Out Our Skeleton

Let's fill out our skeleton now.

- `[(empty? num-list) ...]`
- We are producing a list of numbers as the output of our function. What should we replace the above `...` with?
- Our good friend `'()`
- Let's start to fill in the first `...` in the else clause
- `... sqr ... (first num-list) ...`
- What have we been using to build lists recursively?
- Why our good friend `cons`, of course! Here he is:





Cons and Recursion, a Pair Made in Heaven

We can fill out part of our skeleton now

Cons and Recursion, a Pair Made in Heaven

We can fill out part of our skeleton now

```
(cons ... sqr  
      ... (first num-list)  
      (square-each (rest num-list)))
```



Cons and Recursion, a Pair Made in Heaven

We can fill out part of our skeleton now

```
(cons ... sqr  
      ... (first num-list)  
      (square-each (rest num-list)))
```

-
- How do we fill out the part that needs to use `sqr`?

Cons and Recursion, a Pair Made in Heaven

We can fill out part of our skeleton now

```
(cons ... sqr  
      ... (first num-list)  
      (square-each (rest num-list)))
```

-
- How do we fill out the part that needs to use `sqr`?
- Well we need to square each number in the list, and we can only get the first item out at a time.

Cons and Recursion, a Pair Made in Heaven

We can fill out part of our skeleton now

```
(cons ... sqr  
      ... (first num-list)  
            (square-each (rest num-list)))
```

-
- How do we fill out the part that needs to use `sqr`?
- Well we need to square each number in the list, and we can only get the first item out at a time.
- And we assume that the recursive call already squared all of the numbers in the rest of the list.

Cons and Recursion, a Pair Made in Heaven

We can fill out part of our skeleton now

```
(cons ... sqr  
      ... (first num-list)  
            (square-each (rest num-list)))
```

-
- How do we fill out the part that needs to use `sqr`?
- Well we need to square each number in the list, and we can only get the first item out at a time.
- And we assume that the recursive call already squared all of the numbers in the rest of the list.
- With this insight, we can finish our function.



Funky Square Dance

Now we've finished our funky square dance (Je t'aime Phoenix)!

Funky Square Dance

Now we've finished our funky square dance (Je t'aime Phoenix)!

```
;; List<Number> -> List<Number>
;; Square Each number in a list
(define (square-each num-list)
  (cond
    [(empty? num-list) '()]
    [(cons? num-list)
     (cons
       (sqr (first num-list))
       (square-each (rest num-list))))]))
```



Funky Square Dance

Now we've finished our funky square dance (Je t'aime Phoenix)!

```
;; List<Number> -> List<Number>
;; Square Each number in a list
(define (square-each num-list)
  (cond
    [(empty? num-list) '()]
    [(cons? num-list)
     (cons
       (sqr (first num-list))
       (square-each (rest num-list))))]))
```

-
- This looks pretty nice!

Funky Square Dance

Now we've finished our funky square dance (Je t'aime Phoenix)!

```
;; List<Number> -> List<Number>
;; Square Each number in a list
(define (square-each num-list)
  (cond
    [(empty? num-list) '()]
    [(cons? num-list)
     (cons
       (sqr (first num-list))
       (square-each (rest num-list))))]))
```

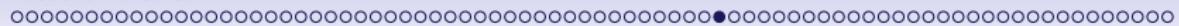
- This looks pretty nice!
- But what if for some reason we misheard the requirements and we were supposed to double the even numbers along with squaring the odd numbers.

Funky Square Dance

Now we've finished our funky square dance (Je t'aime Phoenix)!

```
;; List<Number> -> List<Number>
;; Square Each number in a list
(define (square-each num-list)
  (cond
    [(empty? num-list) '()]
    [(cons? num-list)
     (cons
       (sqr (first num-list))
       (square-each (rest num-list))))]))
```

- This looks pretty nice!
- But what if for some reason we misheard the requirements and we were supposed to double the even numbers along with squaring the odd numbers.
- So, how do we modify our program for this new requirement?



Changing the Program

Well I can filter out the even numbers and double each of them. Then I can add that to the result of our previously defined function.

Changing the Program

Well I can filter out the even numbers and double each of them. Then I can add that to the result of our previously defined function.

- Now our program looks something like the following:

```
(define (sum-odd-squares num-list)
  ((compose sum square-each) (filter odd? num-list)))

(define (sum-even-doubles num-list)
  ((compose sum double-each) (filter even? num-list)))

(define (even-odd-sum num-list)
  (+ (sum-odd-squares num-list) (sum-even-doubles num-list)))
```

Changing the Program

Well I can filter out the even numbers and double each of them. Then I can add that to the result of our previously defined function.

- Now our program looks something like the following:

```
(define (sum-odd-squares num-list)
  ((compose sum square-each) (filter odd? num-list)))

(define (sum-even-doubles num-list)
  ((compose sum double-each) (filter even? num-list)))

(define (even-odd-sum num-list)
  (+ (sum-odd-squares num-list) (sum-even-doubles num-list)))
```

- Hopefully you realized that we needed to then make another function that took the result of our two summing functions and added them.

Changing the Program

Well I can filter out the even numbers and double each of them. Then I can add that to the result of our previously defined function.

- Now our program looks something like the following:

```
(define (sum-odd-squares num-list)
  ((compose sum square-each) (filter odd? num-list)))

(define (sum-even-doubles num-list)
  ((compose sum double-each) (filter even? num-list)))

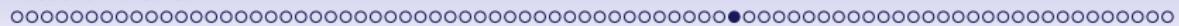
(define (even-odd-sum num-list)
  (+ (sum-odd-squares num-list) (sum-even-doubles num-list)))
```

- Hopefully you realized that we needed to then make another function that took the result of our two summing functions and added them.
- Notice however, that our two summing functions are almost identical.



Doubling Numbers

Having two nearly identical functions smells like an opportunity for abstraction to me.



Doubling Numbers

Having two nearly identical functions smells like an opportunity for abstraction to me.

- But let's focus on writing our function to double numbers first.



Doubling Numbers

Having two nearly identical functions smells like an opportunity for abstraction to me.

- But let's focus on writing our function to double numbers first.
- This seems pretty similar to our function for squaring numbers, right?

Doubling Numbers

Having two nearly identical functions smells like an opportunity for abstraction to me.

- But let's focus on writing our function to double numbers first.
- This seems pretty similar to our function for squaring numbers, right?
- It should look something like this:

```
; ; List<Number> -> List<Number>
; ; Double each number in a list
(define (double-each num-list)
  (cond
    [(empty? num-list) '()]
    [(cons? num-list)
     (cons
       (double (first num-list))
       (double-each (rest num-list))))]))
```

Abstract Nonsense

oo●oooooooooooooooooooo

Oops We Did It Again



Twin Functions Again

```
;; List<Number> -> List<Number>
;; Square Each number in a list
(define (square-each num-list)
  (cond
    [(empty? num-list) '()]
    [(cons? num-list)
     (cons
       (sqr (first num-list))
       (square-each (rest num-list))))]

;; List<Number> -> List<Number>
;; Double each number in a list
(define (double-each num-list)
  (cond
    [(empty? num-list) '()]
    [(cons? num-list)
     (cons
       (double (first num-list))
       (double-each (rest num-list))))])
```

Spoopy Code

The Abstraction Principle: avoid spoopy code that looks like the following:





So Let's Abstract This

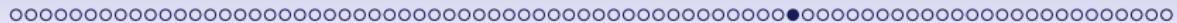
Ok, so we want to abstract this code.



So Let's Abstract This

Ok, so we want to abstract this code.

- So, how do we start?



So Let's Abstract This

Ok, so we want to abstract this code.

- So, how do we start?
- First, we notice that the only difference between the code is the use of `sqr` vs `double`.



So Let's Abstract This

Ok, so we want to abstract this code.

- So, how do we start?
- First, we notice that the only difference between the code is the use of `sqr` vs `double`.
- So, we should be able to replace them with some kind of parameter...

So Let's Abstract This

Ok, so we want to abstract this code.

- So, how do we start?
- First, we notice that the only difference between the code is the use of `sqr` vs `double`.
- So, we should be able to replace them with some kind of parameter...
- Specifically, a higher-order one!

So Let's Abstract This

Ok, so we want to abstract this code.

- So, how do we start?
- First, we notice that the only difference between the code is the use of `sqr` vs `double`.
- So, we should be able to replace them with some kind of parameter...
- Specifically, a higher-order one!

```
; ; List<Number> -> List<Number>
; ; Double each number in a list
(define (do-each op num-list)
  (cond
    [(empty? num-list) '()]
    [(cons? num-list)
     (cons
       (op (first num-list))
       (do-each op (rest num-list))))]))
```





Using do-each

So, let's see how we can use our function:



Using do-each

So, let's see how we can use our function:

- What should (do-each `sqr` '(1 3 5)) do?



Using do-each

So, let's see how we can use our function:

- What should (do-each `sqr` '(1 3 5)) do?
- Well it will do the following: '((`sqr` 1) (`sqr` 3) (`sqr` 5)).

Using do-each

So, let's see how we can use our function:

- What should (do-each `sqr` '(1 3 5)) do?
- Well it will do the following: '((`sqr` 1) (`sqr` 3) (`sqr` 5)).
- This results in: '(1 9 25)

Using do-each

So, let's see how we can use our function:

- What should (do-each `sqr` '(1 3 5)) do?
- Well it will do the following: '((`sqr` 1) (`sqr` 3) (`sqr` 5)).
- This results in: '(1 9 25)
- What should (do-each `double` '(2 4 6)) do?

Using do-each

So, let's see how we can use our function:

- What should (do-each `sqr` '(1 3 5)) do?
- Well it will do the following: '((`sqr` 1) (`sqr` 3) (`sqr` 5)).
- This results in: '(1 9 25)
- What should (do-each `double` '(2 4 6)) do?
- Well it will do the following:
'((`double` 2) (`double` 4) (`double` 6)).

Using do-each

So, let's see how we can use our function:

- What should (do-each `sqr` '(1 3 5)) do?
- Well it will do the following: '((`sqr` 1) (`sqr` 3) (`sqr` 5)).
- This results in: '(1 9 25)
- What should (do-each `double` '(2 4 6)) do?
- Well it will do the following:
'((`double` 2) (`double` 4) (`double` 6)).
- This results in: '(4 8 12)

Using do-each

So, let's see how we can use our function:

- What should (do-each `sqr` '(1 3 5)) do?
- Well it will do the following: '((`sqr` 1) (`sqr` 3) (`sqr` 5)).
- This results in: '(1 9 25)
- What should (do-each `double` '(2 4 6)) do?
- Well it will do the following:
'((`double` 2) (`double` 4) (`double` 6)).
- This results in: '(4 8 12)
- So, do-each just applies a numeric operation to each number in a list of numbers.



Rewriting our Program



Rewriting our Program

- It turns out we can call do-each on more than just lists of numbers, with functions that return numbers.

Rewriting our Program

- It turns out we can call do-each on more than just lists of numbers, with functions that return numbers.
- For example, we can do:

```
(do-each (lambda (s) (substring s 1)) '("abc" "foo" "ba
```

Rewriting our Program

- It turns out we can call do-each on more than just lists of numbers, with functions that return numbers.
- For example, we can do:
`(do-each (lambda (s) (substring s 1)) '("abc" "foo" "ba`
- This returns '`("bc" "oo" "bar")`

Rewriting our Program

- It turns out we can call do-each on more than just lists of numbers, with functions that return numbers.
- For example, we can do:
`(do-each (lambda (s) (substring s 1)) '("abc" "foo" "ba`
- This returns '`("bc" "oo" "bar")`
- So, the name of the function isn't that bad, but we have rediscovered a more general function called `map` that is a function of type $(A \rightarrow B) \times List < A > \rightarrow List < B >$.

Rewriting our Program

- It turns out we can call do-each on more than just lists of numbers, with functions that return numbers.
- For example, we can do:
`(do-each (lambda (s) (substring s 1)) '("abc" "foo" "ba`
- This returns '`("bc" "oo" "bar")`
- So, the name of the function isn't that bad, but we have rediscovered a more general function called `map` that is a function of type $(A \rightarrow B) \times List < A > \rightarrow List < B >$.
- That is, `map` takes in a function that turns elements of type A into elements of type B , a list containing some amount of A elements, and returns a list containing the same amount of B elements.

Rewriting our Program

- It turns out we can call do-each on more than just lists of numbers, with functions that return numbers.
- For example, we can do:
`(do-each (lambda (s) (substring s 1)) '("abc" "foo" "ba`
- This returns '`("bc" "oo" "bar")`
- So, the name of the function isn't that bad, but we have rediscovered a more general function called `map` that is a function of type $(A \rightarrow B) \times List < A > \rightarrow List < B >$.
- That is, map takes in a function that turns elements of type A into elements of type B , a list containing some amount of A elements, and returns a list containing the same amount of B elements.
- In the case of `(map sqr '(1 2 3))`, A and B are both numbers.

Rewriting our Program

- It turns out we can call do-each on more than just lists of numbers, with functions that return numbers.
- For example, we can do:
`(do-each (lambda (s) (substring s 1)) '("abc" "foo" "ba`
- This returns '`("bc" "oo" "bar")`
- So, the name of the function isn't that bad, but we have rediscovered a more general function called `map` that is a function of type $(A \rightarrow B) \times List < A > \rightarrow List < B >$.
- That is, map takes in a function that turns elements of type A into elements of type B , a list containing some amount of A elements, and returns a list containing the same amount of B elements.
- In the case of `(map sqr '(1 2 3))`, A and B are both numbers.
- In the case of `(map number->string '(1 2 3))` B becomes String.



Why is it called Map?

Why is function called map?



Why is it called Map?

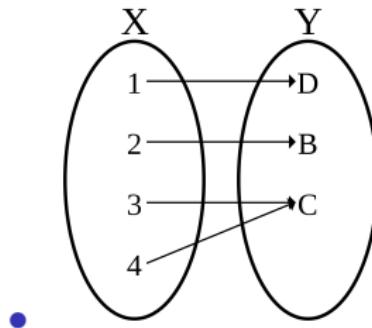
Why is is function called map?

- Because we are navigating between sets!

Why is it called Map?

Why is function called map?

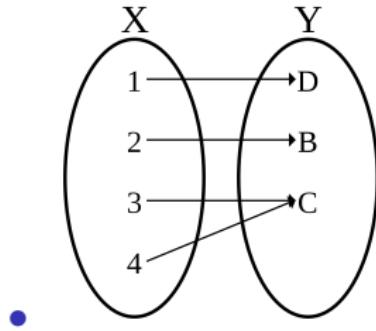
- Because we are navigating between sets!



Why is it called Map?

Why is is function called map?

- Because we are navigating between sets!

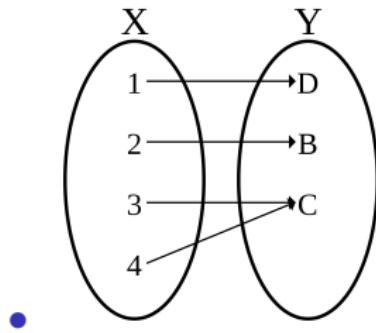


- Our first argument is a function which *maps* elements of type A to elements of type B.

Why is it called Map?

Why is is function called map?

- Because we are navigating between sets!

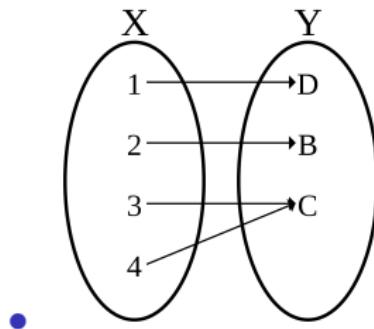


- Our first argument is a function which *maps* elements of type *A* to elements of type *B*.
- I.E. given a starting point in *A* we arrive at some end point in *B*.

Why is it called Map?

Why is this function called map?

- Because we are navigating between sets!



- Our first argument is a function which *maps* elements of type *A* to elements of type *B*.
- I.E. given a starting point in *A* we arrive at some end point in *B*.
- Then, if we are given some subset of *A* as starting points *B* we have subset of destinations that we arrive at in *B*.



Navigating the World of Programs



Navigating the World of Programs



- Now we've got our navigational **map** and can sail the seven seas of cheese...errr I mean programs!

Navigating the World of Programs



- Now we've got our navigational **map** and can sail the seven seas of cheese...errr I mean programs!
- Let's sail in the Bermuda Triangle of examples.

Navigating the World of Programs



- Now we've got our navigational **map** and can sail the seven seas of cheese...errr I mean programs!
- Let's sail in the Bermuda Triangle of examples.

```
(define list-of-functions
  (map
    (lambda (g) (compose sqr g))
    (list double sqrt log)))
```



Scared Emoji



Functions...Functions Everywhere





Back to Code

Back to Code

- We are using map to compose each function in a list of functions with `sqr`.

Back to Code

- We are using map to compose each function in a list of functions with `sqr`.
- `'((compose sqr double) (compose sqr sqrt) (compose sqr log))`

Back to Code

- We are using map to compose each function in a list of functions with `sqr`.
- `'((compose sqr double) (compose sqr sqrt) (compose sqr log))`
- So, what can I do with such a list?

Back to Code

- We are using map to compose each function in a list of functions with `sqr`.
- `'((compose sqr double) (compose sqr sqrt) (compose sqr log))`
- So, what can I do with such a list?
- I can get the first element out of the list and apply it to a number.

Back to Code

- We are using map to compose each function in a list of functions with `sqr`.
- `'((compose sqr double) (compose sqr sqrt) (compose sqr log))`
- So, what can I do with such a list?
- I can get the first element out of the list and apply it to a number.
- `((first list-of-functions) 2)`

Back to Code

- We are using map to compose each function in a list of functions with `sqr`.
- `'((compose sqr double) (compose sqr sqrt) (compose sqr log))`
- So, what can I do with such a list?
- I can get the first element out of the list and apply it to a number.
- `((first list-of-functions) 2)`
- Along those lines, I can map the first element to a list of numbers: `(map (first list-of-functions) '(1 2 3))`

Back to Code

- We are using map to compose each function in a list of functions with `sqr`.
- `'((compose sqr double) (compose sqr sqrt) (compose sqr log))`
- So, what can I do with such a list?
- I can get the first element out of the list and apply it to a number.
- `((first list-of-functions) 2)`
- Along those lines, I can map the first element to a list of numbers: `(map (first list-of-functions) '(1 2 3))`
- This returns `'(4 16 36)`



Spoopy Time

We can get even spoopier with `map`.



Spoopy Time

We can get even spoopier with `map`.

```
(define (apply-2 f) (f 2))  
(map apply-2 lof)
```



Spoopy Time

We can get even spoopier with `map`.

```
(define (apply-2 f) (f 2))  
(map apply-2 lof)
```

-
- This returns:
`'(16 2.0000000000000004 0.4804530139182014)`

Spoopy Time

We can get even spoopier with `map`.

```
(define (apply-2 f) (f 2))  
(map apply-2 lof)
```

-
- This returns:
`'(16 2.0000000000000004 0.4804530139182014)`



De-Spoopifying

Ok, so let's clear up what's happening in this example.

De-Spoopifying

Ok, so let's clear up what's happening in this example.

- First, as usual we are just applying our function apply-2 to each element in our list

De-Spoopifying

Ok, so let's clear up what's happening in this example.

- First, as usual we are just applying our function `apply-2` to each element in our list
- `'((apply-2 (compose sqr double)) (apply-2 (compose sqr sqrt)) ...)`

De-Spoopifying

Ok, so let's clear up what's happening in this example.

- First, as usual we are just applying our function `apply-2` to each element in our list
- `'((apply-2 (compose sqr double)) (apply-2 (compose sqr sqrt)) ...)`
- So, what is the value of `(apply-2 (compose sqr double))`?

De-Spoopifying

Ok, so let's clear up what's happening in this example.

- First, as usual we are just applying our function `apply-2` to each element in our list
- `'((apply-2 (compose sqr double)) (apply-2 (compose sqr sqrt)) ...)`
- So, what is the value of `(apply-2 (compose sqr double))`?
- Well, we substitute `(compose sqr double)` in for the body of `apply-2`

De-Spoopifying

Ok, so let's clear up what's happening in this example.

- First, as usual we are just applying our function `apply-2` to each element in our list
- `'((apply-2 (compose sqr double)) (apply-2 (compose sqr sqrt)) ...)`
- So, what is the value of `(apply-2 (compose sqr double))`?
- Well, we substitute `(compose sqr double)` in for the body of `apply-2`
- This becomes: `((compose sqr double) 2)`

De-Spoopifying

Ok, so let's clear up what's happening in this example.

- First, as usual we are just applying our function `apply-2` to each element in our list
- `'((apply-2 (compose sqr double)) (apply-2 (compose sqr sqrt)) ...)`
- So, what is the value of `(apply-2 (compose sqr double))`?
- Well, we substitute `(compose sqr double)` in for the body of `apply-2`
- This becomes: `((compose sqr double) 2)`
- This then doubles 2, producing 4, and finally squares 4-producing 16.

Algebra of Programming

If we return back to our program for squaring odd numbers, you saw that we first filtered odd numbers and then squared them.



Algebra of Programming

If we return back to our program for squaring odd numbers, you saw that we first filtered odd numbers and then squared them.

- But could we first square all numbers and then filter the odd ones after?

Algebra of Programming

If we return back to our program for squaring odd numbers, you saw that we first filtered odd numbers and then squared them.

- But could we first square all numbers and then filter the odd ones after?
- (`map sqr (filter odd? '(1 2 3 4 5 6))`) produces
`'(1 9 25)`

Algebra of Programming

If we return back to our program for squaring odd numbers, you saw that we first filtered odd numbers and then squared them.

- But could we first square all numbers and then filter the odd ones after?
- (`map sqr (filter odd? '(1 2 3 4 5 6))`) produces
`'(1 9 25)`
- (`filter odd? (map sqr '(1 2 3 4 5 6))`) produces
`'(1 9 25)`

Algebra of Programming

If we return back to our program for squaring odd numbers, you saw that we first filtered odd numbers and then squared them.

- But could we first square all numbers and then filter the odd ones after?
- (`map sqr (filter odd? '(1 2 3 4 5 6))`) produces
`'(1 9 25)`
- (`filter odd? (map sqr '(1 2 3 4 5 6))`) produces
`'(1 9 25)`
- So, indeed these programs are equivalent.

Algebra of Programming

If we return back to our program for squaring odd numbers, you saw that we first filtered odd numbers and then squared them.

- But could we first square all numbers and then filter the odd ones after?
- (`map sqr (filter odd? '(1 2 3 4 5 6))`) produces
`'(1 9 25)`
- (`filter odd? (map sqr '(1 2 3 4 5 6))`) produces
`'(1 9 25)`
- So, indeed these programs are equivalent.
- But can we commute `filter` and `map` in general?

Algebra of Programming

If we return back to our program for squaring odd numbers, you saw that we first filtered odd numbers and then squared them.

- But could we first square all numbers and then filter the odd ones after?
- (`map sqr (filter odd? '(1 2 3 4 5 6))`) produces
`'(1 9 25)`
- (`filter odd? (map sqr '(1 2 3 4 5 6))`) produces
`'(1 9 25)`
- So, indeed these programs are equivalent.
- But can we commute `filter` and `map` in general?
- No, this equivalence was only valid because of the properties of squaring on even and odd numbers

Algebra of Programming

If we return back to our program for squaring odd numbers, you saw that we first filtered odd numbers and then squared them.

- But could we first square all numbers and then filter the odd ones after?
- `(map sqr (filter odd? '(1 2 3 4 5 6)))` produces
`'(1 9 25)`
- `(filter odd? (map sqr '(1 2 3 4 5 6)))` produces
`'(1 9 25)`
- So, indeed these programs are equivalent.
- But can we commute `filter` and `map` in general?
- No, this equivalence was only valid because of the properties of squaring on even and odd numbers
- `(map sqr (filter odd? '(1 2 3 4 5 6))) ≠ (filter odd? (map double '(1 2 3 4 5 6)))`



Properties of Map and Filter

Are there any nice mathematical properties of `map` and `filter`?

Properties of Map and Filter

Are there any nice mathematical properties of `map` and `filter`?

- There are plenty of observations we can make about them.

Properties of Map and Filter

Are there any nice mathematical properties of `map` and `filter`?

- There are plenty of observations we can make about them.
- One is that if given an isomorphism f as input, then we know that `(map f 1st)` is also an isomorphism.

Properties of Map and Filter

Are there any nice mathematical properties of `map` and `filter`?

- There are plenty of observations we can make about them.
- One is that if given an isomorphism f as input, then we know that `(map f lst)` is also an isomorphism.
- That is, if we compute the inverse of f , finv , then:

$$\begin{aligned} (\text{map } \text{finv } (\text{map } f \text{ lst})) &= (\text{map } (\text{compose } \text{finv } f) \text{ lst}) \\ &= \text{lst} \\ &= (\text{map } (\text{compose } f \text{ finv}) \text{ lst}) \\ &= (\text{map } f \text{ (map } \text{finv } \text{ lst})) \end{aligned}$$

Properties of Map and Filter

Are there any nice mathematical properties of `map` and `filter`?

- There are plenty of observations we can make about them.
- One is that if given an isomorphism f as input, then we know that (`map f lst`) is also an isomorphism.
- That is, if we compute the inverse of f , finv , then:

$$\begin{aligned}(\text{map } \text{finv } (\text{map } f \text{ lst})) &= (\text{map } (\text{compose } \text{finv } f) \text{ lst}) \\&= \text{lst} \\&= (\text{map } (\text{compose } f \text{ finv}) \text{ lst}) \\&= (\text{map } f \text{ (map } \text{finv } \text{ lst}))\end{aligned}$$

- Filter also has some nice properties, but because it's a function that returns lists of smaller size, equivalent programs using filter usually piggyback on properties of other functions.

Reducing Functions

This isn't a theory course though, so let's continue to *discover* another useful function. What do these functions have in common?

```
(define (sum lst)
```

```
  (cond
```

```
    [(empty? lst) 0]
```

```
    [else (+ (first lst) (sum (rest lst))))]))
```

```
(define (prod lst)
```

```
  (cond
```

```
    [(empty? lst) 1]
```

```
    [else (* (first lst) (prod (rest lst))))]))
```

```
(define (str-append* lst)
```

```
  (cond
```

```
    [(empty? lst) ""]
```

```
    [else (string-append
```

```
      (first lst)
```

```
      (str-append* (rest lst))))]))
```



Folds

The functions on the previous slide had the same recursive structure where they applied a binary operation over a list, combining a running total with the head of the list.

Folds

The functions on the previous slide had the same recursive structure where they applied a binary operation over a list, combining a running total with the head of the list.

- Basically, we summed the head of the list onto the recursive call that totalled the rest of the list, starting with our identity element being returned for when the list is empty.

Folds

The functions on the previous slide had the same recursive structure where they applied a binary operation over a list, combining a running total with the head of the list.

- Basically, we summed the head of the list onto the recursive call that totalled the rest of the list, starting with our identity element being returned for when the list is empty.
- For example, totalling the list '(1 2 3), we can start by assuming that when the recursion hit '()' in lst (our parameter), then 0 is returned.

Folds

The functions on the previous slide had the same recursive structure where they applied a binary operation over a list, combining a running total with the head of the list.

- Basically, we summed the head of the list onto the recursive call that totalled the rest of the list, starting with our identity element being returned for when the list is empty.
- For example, totalling the list '(1 2 3), we can start by assuming that when the recursion hit '()' in lst (our parameter), then 0 is returned.
- Then our operation becomes $3+0$

Folds

The functions on the previous slide had the same recursive structure where they applied a binary operation over a list, combining a running total with the head of the list.

- Basically, we summed the head of the list onto the recursive call that totalled the rest of the list, starting with our identity element being returned for when the list is empty.
- For example, totalling the list '(1 2 3), we can start by assuming that when the recursion hit '()' in lst (our parameter), then 0 is returned.
- Then our operation becomes $3+0$
- Then $2 + 3$ (the running sum is 3)

Folds

The functions on the previous slide had the same recursive structure where they applied a binary operation over a list, combining a running total with the head of the list.

- Basically, we summed the head of the list onto the recursive call that totalled the rest of the list, starting with our identity element being returned for when the list is empty.
- For example, totalling the list '(1 2 3), we can start by assuming that when the recursion hit '()' in lst (our parameter), then 0 is returned.
- Then our operation becomes $3+0$
- Then $2 + 3$ (the running sum is 3)
- Then $1 + 5$ (the running sum is 5)

Folds

The functions on the previous slide had the same recursive structure where they applied a binary operation over a list, combining a running total with the head of the list.

- Basically, we summed the head of the list onto the recursive call that totalled the rest of the list, starting with our identity element being returned for when the list is empty.
- For example, totalling the list '(1 2 3), we can start by assuming that when the recursion hit '()' in lst (our parameter), then 0 is returned.
- Then our operation becomes $3+0$
- Then $2 + 3$ (the running sum is 3)
- Then $1 + 5$ (the running sum is 5)
- This is essentially using Σ in calculus courses



Folds

We can also think about products of lists. This is sometimes denoted with Π

Folds

We can also think about products of lists. This is sometimes denoted with Π

- For example, the product of the list '(1 2 3) with the identity element 1.

Folds

We can also think about products of lists. This is sometimes denoted with Π

- For example, the product of the list '(1 2 3) with the identity element 1.
- When the recursion hit '() in 1st (our parameter), then 1 is returned.

Folds

We can also think about products of lists. This is sometimes denoted with Π

- For example, the product of the list '(1 2 3) with the identity element 1.
- When the recursion hit '() in lst (our parameter), then 1 is returned.
- We take $3 * 1$

Folds

We can also think about products of lists. This is sometimes denoted with Π

- For example, the product of the list '(1 2 3) with the identity element 1.
- When the recursion hit '() in lst (our parameter), then 1 is returned.
- We take $3 * 1$
- The running total becomes 3. And then we do $2 * 3$

Folds

We can also think about products of lists. This is sometimes denoted with Π

- For example, the product of the list '(1 2 3) with the identity element 1.
- When the recursion hit '()' in lst (our parameter), then 1 is returned.
- We take $3 * 1$
- The running total becomes 3. And then we do $2 * 3$
- The running total becomes 6, and then we do $1 * 6$ and our final result is 6

Folds

And of course taking the string-append of many strings in a list.
We will denote string-append as a binary operation with \oplus

Folds

And of course taking the string-append of many strings in a list.
We will denote string-append as a binary operation with \oplus

- Let's consider taking the multi string-append of

`'("Brian Wecht" " Commander Meouch" " Matt Watson")`

Folds

And of course taking the string-append of many strings in a list.
We will denote string-append as a binary operation with \oplus

- Let's consider taking the multi string-append of
`'("Brian Wecht" " Commander Meouch" " Matt Watson")`
- We start with the empty list being in 1st and returning ""

Folds

And of course taking the string-append of many strings in a list.
We will denote string-append as a binary operation with \oplus

- Let's consider taking the multi string-append of
`'("Brian Wecht" " Commander Meouch" " Matt Watson")`
- We start with the empty list being in 1st and returning ""
- Then we do "Matt Watson" \oplus ""

Folds

And of course taking the string-append of many strings in a list.
We will denote string-append as a binary operation with \oplus

- Let's consider taking the multi string-append of
`'("Brian Wecht" " Commander Meouch" " Matt Watson")`
- We start with the empty list being in 1st and returning ""
- Then we do "Matt Watson" \oplus ""
- Our running sum is "Matt Watson"

Folds

And of course taking the string-append of many strings in a list.
We will denote string-append as a binary operation with \oplus

- Let's consider taking the multi string-append of
`'("Brian Wecht" " Commander Meouch" " Matt Watson")`
- We start with the empty list being in 1st and returning ""
- Then we do " Matt Watson" \oplus ""
- Our running sum is " Matt Watson"
- Then we do " Commander Meouch" \oplus " Matt Watson"

Folds

And of course taking the string-append of many strings in a list.
We will denote string-append as a binary operation with \oplus

- Let's consider taking the multi string-append of
`'("Brian Wecht" " Commander Meouch" " Matt Watson")`
- We start with the empty list being in 1st and returning ""
- Then we do " Matt Watson" \oplus ""
- Our running sum is " Matt Watson"
- Then we do " Commander Meouch" \oplus " Matt Watson"
- Our running sum is then " Commander Meouch Matt Watson"

Folds

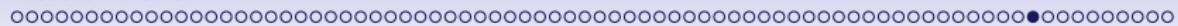
And of course taking the string-append of many strings in a list.
We will denote string-append as a binary operation with \oplus

- Let's consider taking the multi string-append of
`'("Brian Wecht" " Commander Meouch" " Matt Watson")`
- We start with the empty list being in 1st and returning ""
- Then we do " Matt Watson" \oplus ""
- Our running sum is " Matt Watson"
- Then we do " Commander Meouch" \oplus " Matt Watson"
- Our running sum is then " Commander Meouch Matt Watson"
- We finally do " Brian Wecht" \oplus " Commander Meouch Matt Watson"

Folds

And of course taking the string-append of many strings in a list.
We will denote string-append as a binary operation with \oplus

- Let's consider taking the multi string-append of
`'("Brian Wecht" " Commander Meouch" " Matt Watson")`
- We start with the empty list being in 1st and returning ""
- Then we do " Matt Watson" \oplus ""
- Our running sum is " Matt Watson"
- Then we do " Commander Meouch" \oplus " Matt Watson"
- Our running sum is then " Commander Meouch Matt Watson"
- We finally do " Brian Wecht" \oplus " Commander Meouch Matt Watson"
- Our result is " Brian Wecht Commander Meouch Matt Watson"



Defining Fold

The idea of taking a repeated binary operation over a list is known as *reduce* (or aggregate) in many languages and `foldl` in Racket (there is also `foldr` (more in a bit)).

Defining Fold

The idea of taking a repeated binary operation over a list is known as *reduce* (or aggregate) in many languages and `foldl` in Racket (there is also `foldr` (more in a bit)).

- So, with each of these functions we defined previously, we were taking a binary operation and applying it to the first element of the list and the recursive call to these functions, where our running total is called acc.

Defining Fold

The idea of taking a repeated binary operation over a list is known as *reduce* (or aggregate) in many languages and `foldl` in Racket (there is also `foldr` (more in a bit)).

- So, with each of these functions we defined previously, we were taking a binary operation and applying it to the first element of the list and the recursive call to these functions, where our running total is called acc.
- Thus, if our binary operation is named `o`, then we are doing `(o (first lst) acc)`.

Defining Fold

The idea of taking a repeated binary operation over a list is known as *reduce* (or aggregate) in many languages and `foldl` in Racket (there is also `foldr` (more in a bit)).

- So, with each of these functions we defined previously, we were taking a binary operation and applying it to the first element of the list and the recursive call to these functions, where our running total is called acc.
- Thus, if our binary operation is named `o`, then we are doing `(o (first lst) acc)`.
- Then, if this is assigned to the accumulator `acc` at each recursive call, then our code looks like:

Defining Fold

The idea of taking a repeated binary operation over a list is known as *reduce* (or aggregate) in many languages and `foldl` in Racket (there is also `foldr` (more in a bit)).

- So, with each of these functions we defined previously, we were taking a binary operation and applying it to the first element of the list and the recursive call to these functions, where our running total is called acc.
- Thus, if our binary operation is named `o`, then we are doing `(o (first lst) acc)`.
- Then, if this is assigned to the accumulator `acc` at each recursive call, then our code looks like:
`(foldl o (o (first lst) acc) (rest lst))`
-

Defining Fold

The rest of the function is just returning acc as our base case for our recursion.

Defining Fold

The rest of the function is just returning acc as our base case for our recursion.

```
(define (foldl o acc lst)
  (cond
    [(empty? lst) acc]
    [else (foldl o (o (first lst) acc) (rest lst))]))
```



Defining Fold

The rest of the function is just returning acc as our base case for our recursion.

```
(define (foldl o acc lst)
  (cond
    [(empty? lst) acc]
    [else (foldl o (o (first lst) acc) (rest lst))]))
```

-
- If we compared the structure of our recursion in foldl to our previously defined recursive functions, you might notice a difference.

Defining Fold

The rest of the function is just returning acc as our base case for our recursion.

```
(define (foldl o acc lst)
  (cond
    [(empty? lst) acc]
    [else (foldl o (o (first lst) acc) (rest lst))]))
```

-
- If we compared the structure of our recursion in foldl to our previously defined recursive functions, you might notice a difference.
- First, consider:

```
(foldl o (o (first lst) acc) (rest lst))
```

Defining Fold

The rest of the function is just returning acc as our base case for our recursion.

```
(define (foldl o acc lst)
  (cond
    [(empty? lst) acc]
    [else (foldl o (o (first lst) acc) (rest lst))]))
```

-
- If we compared the structure of our recursion in foldl to our previously defined recursive functions, you might notice a difference.
- First, consider:
`(foldl o (o (first lst) acc) (rest lst))`
- Now let's consider our recursive call in our sum function.

Defining Fold

The rest of the function is just returning acc as our base case for our recursion.

```
(define (foldl o acc lst)
  (cond
    [(empty? lst) acc]
    [else (foldl o (o (first lst) acc) (rest lst))]))
```

-
- If we compared the structure of our recursion in foldl to our previously defined recursive functions, you might notice a difference.
- First, consider:
`(foldl o (o (first lst) acc) (rest lst))`
- Now let's consider our recursive call in our sum function.
- It was: `(+ (first lst) (sum (rest lst)))`

Defining Fold

The rest of the function is just returning acc as our base case for our recursion.

```
(define (foldl o acc lst)
  (cond
    [(empty? lst) acc]
    [else (foldl o (o (first lst) acc) (rest lst))]))
```

-
- If we compared the structure of our recursion in foldl to our previously defined recursive functions, you might notice a difference.
- First, consider:
`(foldl o (o (first lst) acc) (rest lst))`
- Now let's consider our recursive call in our sum function.
- It was: `(+ (first lst) (sum (rest lst)))`
- So, in `foldl`, we immediately made the recursive call, while in the `sum`, the recursive call was an argument to our binary operator `(+)`

Foldl vs Foldr

So, it turns out that if we can foldl on a list like '(1 2 3), our running total is created by processing the elements of the list in order. I.E. 1, then 2, then 3.

Foldl vs Foldr

So, it turns out that if we can foldl on a list like '(1 2 3), our running total is created by processing the elements of the list in order. I.E. 1, then 2, then 3.

- When going over the example for sum, our recursion processed 3+0, then 2+3,

Foldl vs Foldr

So, it turns out that if we can foldl on a list like '(1 2 3), our running total is created by processing the elements of the list in order. I.E. 1, then 2, then 3.

- When going over the example for sum, our recursion processed $3+0$, then $2+3$,
- So, there is a mismatch between the recursion in `foldl` and our examples.

Foldl vs Foldr

So, it turns out that if we can foldl on a list like '(1 2 3), our running total is created by processing the elements of the list in order. I.E. 1, then 2, then 3.

- When going over the example for sum, our recursion processed 3+0, then 2+3,
- So, there is a mismatch between the recursion in `foldl` and our examples.

```
(define (foldr o acc lst)
  (cond
    [(empty? lst) acc]
    [else
      (o (first lst) (foldr o acc (rest lst))))]))
```



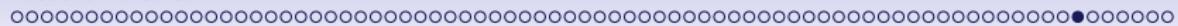
Foldl vs Foldr

So, it turns out that if we can foldl on a list like '(1 2 3), our running total is created by processing the elements of the list in order. I.E. 1, then 2, then 3.

- When going over the example for sum, our recursion processed 3+0, then 2+3,
- So, there is a mismatch between the recursion in `foldl` and our examples.

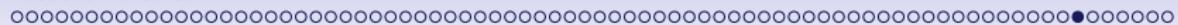
```
(define (foldr o acc lst)
  (cond
    [(empty? lst) acc]
    [else
      (o (first lst) (foldr o acc (rest lst))))]))
```

-
- foldr came to the rescue by applying the function in the same manner.



Foldl vs Foldr

Which of foldl and foldr you want to use is dependent on whether the order the elements are process matters.



Foldl vs Foldr

Which of foldl and foldr you want to use is dependent on whether the order the elements are process matters.

- If the operation is commutative, use foldl.



Foldl vs Foldr

Which of foldl and foldr you want to use is dependent on whether the order the elements are process matters.

- If the operation is commutative, use foldl.
- For example (`foldl + 0 '(1 2 3)`) is the same as (`foldr + 0 '(1 2 3)`) because `+` is a commutative operator.

Foldl vs Foldr

Which of foldl and foldr you want to use is dependent on whether the order the elements are process matters.

- If the operation is commutative, use foldl.
- For example (`foldl + 0 '(1 2 3)`) is the same as (`foldr + 0 '(1 2 3)`) because + is a commutative operator.
- Because foldl is tail recursive it will generally be faster.

Foldl vs Foldr

Which of foldl and foldr you want to use is dependent on whether the order the elements are process matters.

- If the operation is commutative, use foldl.
- For example (`foldl + 0 '(1 2 3)`) is the same as (`foldr + 0 '(1 2 3)`) because + is a commutative operator.
- Because foldl is tail recursive it will generally be faster.
- But be careful. For example, (`foldl cons '() '(1 2)`) returns:

Foldl vs Foldr

Which of foldl and foldr you want to use is dependent on whether the order the elements are process matters.

- If the operation is commutative, use foldl.
- For example (`foldl + 0 '(1 2 3)`) is the same as (`foldr + 0 '(1 2 3)`) because + is a commutative operator.
- Because foldl is tail recursive it will generally be faster.
- But be careful. For example, (`foldl cons '() '(1 2)`) returns:
- `'(2 1)`

Foldl vs Foldr

Which of foldl and foldr you want to use is dependent on whether the order the elements are process matters.

- If the operation is commutative, use foldl.
- For example (`foldl + 0 '(1 2 3)`) is the same as (`foldr + 0 '(1 2 3)`) because + is a commutative operator.
- Because foldl is tail recursive it will generally be faster.
- But be careful. For example, (`foldl cons '() '(1 2)`) returns:
 - `'(2 1)`
 - (`foldr cons '() '(1 2)`)

Foldl vs Foldr

Which of foldl and foldr you want to use is dependent on whether the order the elements are process matters.

- If the operation is commutative, use foldl.
- For example (`foldl + 0 '(1 2 3)`) is the same as (`foldr + 0 '(1 2 3)`) because + is a commutative operator.
- Because foldl is tail recursive it will generally be faster.
- But be careful. For example, (`foldl cons '() '(1 2)`) returns:
 - `'(2 1)`
 - (`foldr cons '() '(1 2)`)
 - This returns `'(1 2)`

Defining Other Functions as Folds

We can define many existing functions as folds.

Defining Other Functions as Folds

We can define many existing functions as folds.

- (`define (sum l) (foldl + 0 l))`

Defining Other Functions as Folds

We can define many existing functions as folds.

- `(define (sum l) (foldl + 0 l))`
- `(define (prod l) (foldl * 1 l))`

Defining Other Functions as Folds

We can define many existing functions as folds.

- `(define (sum l) (foldl + 0 l))`
- `(define (prod l) (foldl * 1 l))`
- `(define (string-append* l) (foldr string-append "" l))`

Defining Other Functions as Folds

We can define many existing functions as folds.

- `(define (sum l) (foldl + 0 l))`
- `(define (prod l) (foldl * 1 l))`
- `(define (string-append* l) (foldr string-append "" l))`
- `(define (reverse l) (foldl cons '() l))`

Defining Other Functions as Folds

We can define many existing functions as folds.

- `(define (sum l) (foldl + 0 l))`
- `(define (prod l) (foldl * 1 l))`
- `(define (string-append* l) (foldr string-append "" l))`
- `(define (reverse l) (foldl cons '() l))`
- `(define (id-list l) (foldr cons '() l))`

Defining Other Functions as Folds

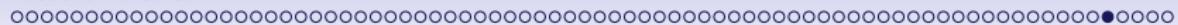
We can define many existing functions as folds.

- `(define (sum l) (foldl + 0 l))`
- `(define (prod l) (foldl * 1 l))`
- `(define (string-append* l) (foldr string-append "" l))`
- `(define (reverse l) (foldl cons '() l))`
- `(define (id-list l) (foldr cons '() l))`
- `(define (map f l)`
 `(foldr (lambda (e acc) (cons (f e) acc)) '() l))`
-

Defining Other Functions as Folds

We can define many existing functions as folds.

- `(define (sum l) (foldl + 0 l))`
- `(define (prod l) (foldl * 1 l))`
- `(define (string-append* l) (foldr string-append "" l))`
- `(define (reverse l) (foldl cons '() l))`
- `(define (id-list l) (foldr cons '() l))`
- `(define (map f l)
 (foldr (lambda (e acc) (cons (f e) acc)) '() l))`
- `(define (filter p l)
 (foldr
 (lambda (e acc)
 (if (p e) (cons e acc) acc))
 '()
 l))`
-



Folds are Powerful

Notice that the last definition for filter was more complex than the rest because the binary operation `cons` was only applied *conditionally*

Folds are Powerful

Notice that the last definition for filter was more complex than the rest because the binary operation `cons` was only applied *conditionally*

- We could've defined such a lambda as a named function:

```
(define (cons-pred p e lst)
  (if (p e) (cons e lst) lst))
```

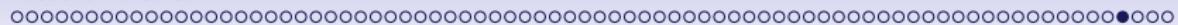
Folds are Powerful

Notice that the last definition for filter was more complex than the rest because the binary operation `cons` was only applied *conditionally*

- We could've defined such a lambda as a named function:
`(define (cons-pred p e lst)
 (if (p e) (cons e lst) lst))`



Download from
Dreamstime.com



Folds are Powerful

The point I'm trying to make here, is that we can define tons of useful functions with folds, although the complexity grows with the complexity of the function we are trying to define.



Folds are Powerful

The point I'm trying to make here, is that we can define tons of useful functions with folds, although the complexity grows with the complexity of the function we are trying to define.

- For example, I can consider defining `foldl` in terms of `foldr`.

Folds are Powerful

The point I'm trying to make here, is that we can define tons of useful functions with folds, although the complexity grows with the complexity of the function we are trying to define.

- For example, I can consider defining `foldl` in terms of `foldr`.
- I will refrain from doing this, because it is complex and would take me too long to explain it in even a half decent way. But you can look it up.

Abstracting an Existing Program

Let's recall our wage calculation program from earlier in the course.

Abstracting an Existing Program

Let's recall our wage calculation program from earlier in the course.

- Do you think we will have abstractions we can apply to the functions we designed?

Abstracting an Existing Program

Let's recall our wage calculation program from earlier in the course.

- Do you think we will have abstractions we can apply to the functions we designed?

```
; ; List<Employee\_data> -> List<Wage\_data>
; ; Calculates the wage for each employee
(define (calculate-wages emps)
  (cond
    [(empty? emps) '()]
    [(cons? emps)
      (cons (calculate-wage (first emps))
            (calculate-wages (rest emps))))]))
```

-

Abstracting an Existing Program

Let's recall our wage calculation program from earlier in the course.

- Do you think we will have abstractions we can apply to the functions we designed?

```
;; List<Employee\_data> -> List<Wage\_data>
;; Calculates the wage for each employee
(define (calculate-wages emps)
  (cond
    [(empty? emps) '()]
    [(cons? emps)
     (cons (calculate-wage (first emps))
           (calculate-wages (rest emps))))]))
```

-
- What abstraction can we apply to the above function?

Abstracting an Existing Program

Let's recall our wage calculation program from earlier in the course.

- Do you think we will have abstractions we can apply to the functions we designed?

```
;; List<Employee\_data> -> List<Wage\_data>
;; Calculates the wage for each employee
(define (calculate-wages emps)
  (cond
    [(empty? emps) '()]
    [(cons? emps)
      (cons (calculate-wage (first emps))
            (calculate-wages (rest emps))))]))
```

-
- What abstraction can we apply to the above function?

```
(define (calculate-wages lst)
  (map calculate-wage lst))
```

Abstracting an Existing Program

Ok, great we could use map to get rid of manually written and error-prone recursion!

Abstracting an Existing Program

Ok, great we could use map to get rid of manually written and error-prone recursion!

```
; ; List<Wage_data> -> Number
; ; Sums all of the wage data to see how much the company paid
(define (total-wages wages)
  (cond
    [(empty? wages) 0]
    [(cons? wages)
      (+ (wage\_data-pay (first wages))
         (total-wages (rest wages))))]))
```



Abstracting an Existing Program

Ok, great we could use map to get rid of manually written and error-prone recursion!

```
; ; List<Wage_data> -> Number
; ; Sums all of the wage data to see how much the company paid
(define (total-wages wages)
  (cond
    [(empty? wages) 0]
    [(cons? wages)
      (+ (wage\_data-pay (first wages))
         (total-wages (rest wages))))]))
```

- We are summing a bunch of wages. Any abstraction here?

Abstracting an Existing Program

Ok, great we could use map to get rid of manually written and error-prone recursion!

```
; ; List<Wage_data> -> Number
; ; Sums all of the wage data to see how much the company paid
(define (total-wages wages)
  (cond
    [(empty? wages) 0]
    [(cons? wages)
      (+ (wage\_data-pay (first wages))
         (total-wages (rest wages))))]))
```

- We are summing a bunch of wages. Any abstraction here?
- Two ways, map then fold or just fold.

Abstracting an Existing Program

Here's the first way:

```
(define (total-wages wages)
  (foldl + 0 (map wage\_data-pay wages)))
```



Abstracting an Existing Program

Here's the first way:

```
(define (total-wages wages)
  (foldl + 0 (map wage\_data-pay wages)))
```

-
- This looks pretty natural although we could clean it up even more

Abstracting an Existing Program

Here's the first way:

```
(define (total-wages wages)
  (foldl + 0 (map wage\_data-pay wages)))
```

-
- This looks pretty natural although we could clean it up even more

```
(define (total-wages wages)
  (foldl
    (lambda (e acc) (+ (wage\_data-pay e) acc))
    0
    wages))
```

-

Abstracting an Existing Program

Here's the first way:

```
(define (total-wages wages)
  (foldl + 0 (map wage\_data-pay wages)))
```

-
- This looks pretty natural although we could clean it up even more

```
(define (total-wages wages)
  (foldl
    (lambda (e acc) (+ (wage\_data-pay e) acc))
    0
    wages))
```

-
- This way looks less natural, though we could clean it up by turning the lambda into a helper function.