# Functional Programming 2: First-Class Functions

CS 350

---

Dr. Joseph Eremondi

Last updated: July 17, 2024

# Higher Order Functions

# Functions on Functions

- Functions let us be abstract over the data they work on

# Functions on Functions

- Functions let us be abstract over the data they work on
- But why can’t we be abstract over what they do to that data?

# Functions on Functions

- Functions let us be abstract over the data they work on
- But why can’t we be abstract over what they do to that data?
    - We can!

# Functions on Functions

- Functions let us be abstract over the data they work on
- But why can't we be abstract over what they do to that data?
  - We can!
- A **higher order function** is a function that takes functions as an argument, or returns functions as a value.

# Functions on Functions

- Functions let us be abstract over the data they work on
- But why can't we be abstract over what they do to that data?
  - We can!
- A **higher order function** is a function that takes functions as an argument, or returns functions as a value.
- Examples:

# Functions on Functions

- Functions let us be abstract over the data they work on
- But why can't we be abstract over what they do to that data?
  - We can!
- A **higher order function** is a function that takes functions as an argument, or returns functions as a value.
- Examples:
  - Callbacks

# Functions on Functions

- Functions let us be abstract over the data they work on
- But why can't we be abstract over what they do to that data?
  - We can!
- A **higher order function** is a function that takes functions as an argument, or returns functions as a value.
- Examples:
  - Callbacks
    - Give a GUI element the function to run when clicked

# Functions on Functions

- Functions let us be abstract over the data they work on
- But why can't we be abstract over what they do to that data?
  - We can!
- A **higher order function** is a function that takes functions as an argument, or returns functions as a value.
- Examples:
  - Callbacks
    - Give a GUI element the function to run when clicked
  - Threads

# Functions on Functions

- Functions let us be abstract over the data they work on
- But why can't we be abstract over what they do to that data?
  - We can!
- A **higher order function** is a function that takes functions as an argument, or returns functions as a value.
- Examples:
  - Callbacks
    - Give a GUI element the function to run when clicked
  - Threads
    - Give the function for each thread to compute

# Function types

- Type `(T1 T2 ... Tn -> S)`

# Function types

- Type `(T1 T2 ... Tn -> S)`
  - The type of functions that:

# Function types

- Type `(T1 T2 ... Tn -> S)`
  - The type of functions that:
    - take *n* arguments

# Function types

- Type `(T1 T2 ... Tn -> S)`
  - The type of functions that:
    - take *n* arguments
    - each with type $T_i$ respectively

# Function types

- Type `(T1 T2 ... Tn -> S)`
  - The type of functions that:
    - take *n* arguments
    - each with type $T_i$ respectively
    - produces a result of type S

# Function types

- Type `(T1 T2 ... Tn -> S)`
  - The type of functions that:
    - take *n* arguments
    - each with type $T_i$ respectively
    - produces a result of type S
- Functions can be defined, where their arguments *are function types!*

# First Class Functions

- We say a language has *first class functions* if functions are treated like any other expression/value in a language

# First Class Functions

- We say a language has *first class functions* if functions are treated like any other expression/value in a language
  - We can construct them at any point, not just at the top level

# First Class Functions

- We say a language has *first class functions* if functions are treated like any other expression/value in a language
  - We can construct them at any point, not just at the top level
  - We can give them as arguments to functions

# First Class Functions

- We say a language has *first class functions* if functions are treated like any other expression/value in a language
  - We can construct them at any point, not just at the top level
  - We can give them as arguments to functions
  - We can return them as results of functions

# Example: Repeatedly apply a function

# Example: Repeatedly apply a function

```
(define (applyNTimes [f : (Number -> Number)]
                     [x : Number]
                     [nTimes : Number]) : Number
  (if (<= nTimes 0)
      x
      (applyNTimes f (f x) (- nTimes 1))))
```

- Takes 3 arguments

# Example: Repeatedly apply a function

```
(define (applyNTimes [f : (Number -> Number)]
                     [x : Number]
                     [nTimes : Number]) : Number
  (if (<= nTimes 0)
      x
      (applyNTimes f (f x) (- nTimes 1))))
```

- Takes 3 arguments
  - A function from Number to Number

# Example: Repeatedly apply a function

```
(define (applyNTimes [f : (Number -> Number)]
                     [x : Number]
                     [nTimes : Number]) : Number
  (if (<= nTimes 0)
      x
      (applyNTimes f (f x) (- nTimes 1))))
```

- Takes 3 arguments
  - A function from Number to Number
  - A number

# Example: Repeatedly apply a function

```
(define (applyNTimes [f : (Number -> Number)]
                     [x : Number]
                     [nTimes : Number]) : Number
  (if (<= nTimes 0)
      x
      (applyNTimes f (f x) (- nTimes 1))))
```

- Takes 3 arguments
  - A function from Number to Number
  - A number
  - A number

# Example: Repeatedly apply a function

```
(define (applyNTimes [f : (Number -> Number)]
                     [x : Number]
                     [nTimes : Number]) : Number
  (if (<= nTimes 0)
      x
      (applyNTimes f (f x) (- nTimes 1))))
```

- Takes 3 arguments
  - A function from Number to Number
  - A number
  - A number
- Returns a number

# Example: Repeatedly apply a function

```
(define (applyNTimes [f : (Number -> Number)]
                     [x : Number]
                     [nTimes : Number]) : Number
  (if (<= nTimes 0)
      x
      (applyNTimes f (f x) (- nTimes 1))))
```

- Takes 3 arguments
  - A function from Number to Number
  - A number
  - A number
- Returns a number
- In the body:

# Example: Repeatedly apply a function

```
(define (applyNTimes [f : (Number -> Number)]
                     [x : Number]
                     [nTimes : Number]) : Number
  (if (<= nTimes 0)
      x
      (applyNTimes f (f x) (- nTimes 1))))
```

- Takes 3 arguments
  - A function from Number to Number
  - A number
  - A number
- Returns a number
- In the body:
  - Calls the parameter `f` as a function on `x`

# ctd

# ctd

```
(define (timesTen x) (* 10 x))

(applyNTimes add1 3 5)
(applyNTimes timesTen 3 5)
```

# ctd

```
(define (timesTen x) (* 10 x))

(applyNTimes add1 3 5)
(applyNTimes timesTen 3 5)
```

```
8
300000
```

- Takes whatever function we pass in, applies it to 3, 5 times

# ctd

```
(define (timesTen x) (* 10 x))

(applyNTimes add1 3 5)
(applyNTimes timesTen 3 5)
```

```
8
300000
```

- Takes whatever function we pass in, applies it to 3, 5 times
  - `(f (f (f (f (f 3)))))`

# Example: apply an operation to each number in a list

# Example: apply an operation to each number in a list

```
(define (mapNum [f : (Number -> Number)]
                [xs : (Listof Number)]) : (Listof Number)
  (type-case (Listof Number) xs
             [empty empty]
             [(cons x rest)
                (cons (f x) (mapNum f rest))]))
```

- Takes a function from Numbers to Numbers, and a list of numbers

# Example: apply an operation to each number in a list

```
(define (mapNum [f : (Number -> Number)]
                [xs : (Listof Number)]) : (Listof Number)
  (type-case (Listof Number) xs
             [empty empty]
             [(cons x rest)
                (cons (f x) (mapNum f rest))]))
```

- Takes a function from Numbers to Numbers, and a list of numbers
- Applies `f` to each element of the list

# Example: apply an operation to each number in a list

```
(define (mapNum [f : (Number -> Number)]
                [xs : (Listof Number)]) : (Listof Number)
  (type-case (Listof Number) xs
             [empty empty]
             [(cons x rest)
                (cons (f x) (mapNum f rest))]))
```

- Takes a function from Numbers to Numbers, and a list of numbers
- Applies `f` to each element of the list
  - Apply `f` to everything in the empty list

# Example: apply an operation to each number in a list

```
(define (mapNum [f : (Number -> Number)]
                [xs : (Listof Number)]) : (Listof Number)
  (type-case (Listof Number) xs
             [empty empty]
             [(cons x rest)
                (cons (f x) (mapNum f rest))]))
```

- Takes a function from Numbers to Numbers, and a list of numbers
- Applies `f` to each element of the list
  - Apply `f` to everything in the empty list
    - Produces empty list

# Example: apply an operation to each number in a list

```
(define (mapNum [f : (Number -> Number)]
                [xs : (Listof Number)]) : (Listof Number)
  (type-case (Listof Number) xs
             [empty empty]
             [(cons x rest)
                (cons (f x) (mapNum f rest))]))
```

- Takes a function from Numbers to Numbers, and a list of numbers
- Applies `f` to each element of the list
  - Apply `f` to everything in the empty list
    - Produces empty list
  - Apply `f` to each in `(cons x rest)`

# Example: apply an operation to each number in a list

```
(define (mapNum [f : (Number -> Number)]
                [xs : (Listof Number)]) : (Listof Number)
  (type-case (Listof Number) xs
             [empty empty]
             [(cons x rest)
                (cons (f x) (mapNum f rest))]))
```

- Takes a function from Numbers to Numbers, and a list of numbers
- Applies `f` to each element of the list
  - Apply `f` to everything in the empty list
    - Produces empty list
  - Apply `f` to each in `(cons x rest)`
    - Apply `f` to `x`, recursively apply `f` to everything in `rest`

# Example: apply an operation to each number in a list

```
(define (mapNum [f : (Number -> Number)]
                [xs : (Listof Number)]) : (Listof Number)
  (type-case (Listof Number) xs
             [empty empty]
             [(cons x rest)
                (cons (f x) (mapNum f rest))]))
```

- Takes a function from Numbers to Numbers, and a list of numbers
- Applies `f` to each element of the list
  - Apply `f` to everything in the empty list
    - Produces empty list
  - Apply `f` to each in `(cons x rest)`
    - Apply `f` to `x`, recursively apply `f` to everything in `rest`
    - Combine the results with `cons`

# ctd

# ctd

```
(define (timesTen x) (* 10 x))
(mapNum add1 '(1 2 3 4))
(mapNum timesTen '(1 2 3 4))
```

# ctd

```
(define (timesTen x) (* 10 x))
(mapNum add1 '(1 2 3 4))
(mapNum timesTen '(1 2 3 4))
```

```
'(2 3 4 5)
'(10 20 30 40)
```

# Creating Anonymous Functions

- So far, have only given functions that we defined with `define` as arguments to other functions

# Creating Anonymous Functions

- So far, have only given functions that we defined with `define` as arguments to other functions
- What if we want to make a small little function that we use only once?

# Creating Anonymous Functions

- So far, have only given functions that we defined with `define` as arguments to other functions
- What if we want to make a small little function that we use only once?
- What if we want to make a function dynamically?

# Lambda

```
(lambda (x) body)
```

- Creates a function with argument `x` that returns `body`

# Lambda

```
(lambda (x) body)
```

- Creates a function with argument `x` that returns `body`
- `x` may occur in body

# Lambda

```
(lambda (x) body)
```

- Creates a function with argument `x` that returns `body`
- `x` may occur in body
- Is an expression, not a declaration

# Lambda

```
(lambda (x) body)
```

- Creates a function with argument `x` that returns `body`
- `x` may occur in body
- Is an expression, not a declaration
    - Can occur anywhere else

# Lambda variations

# Lambda variations

```
;; Type annotation
(lambda ([x : Number]) : Number
  (+ x 1))

;; Multiple arguments
(lambda (x y) (+ x (+ x y)))

;;Multiple type annotations
(lambda ([x : Number]
   [y : Number]) (+ x (+ x y)))

;; Unicode Greek lambda
;; In Dr. Racket: either cmd-\ or ctrl-\ depending on os
(λ (x) (+ x x))
```

# Example

# Example

```
(define (timesTen x) (* 10 x))
(mapNum timesTen '(1 2 3 4))
(mapNum (lambda (x) (* x 10)) '(1 2 3 4))
```

# Example

```
(define (timesTen x) (* 10 x))
(mapNum timesTen '(1 2 3 4))
(mapNum (lambda (x) (* x 10)) '(1 2 3 4))
```

```
'(10 20 30 40)
'(10 20 30 40)
```

# Define as sugar

- `(define (f x) body)`

# Define as sugar

- `(define (f x) body)`
- Same as `(define f (lambda (x) body))`

# Define as sugar

- `(define (f x) body)`
- Same as `(define f (lambda (x) body))`
- Defining functions is *syntactic sugar* for lambda in Plait

# Lambda in a context

- Don’t have to use lambda at the top level

# Lambda in a context

- Don’t have to use lambda at the top level
- Can refer to other variables in the body of the lambda

# Lambda in a context

- Don’t have to use lambda at the top level
- Can refer to other variables in the body of the lambda

# Lambda in a context

- Don’t have to use lambda at the top level
- Can refer to other variables in the body of the lambda

```
(define (addNToEach [numToAdd : Number]
                    [xs : (Listof Number)]) : (Listof Number)
  (mapNum (lambda(x) (+ x numToAdd)) xs))
(addNToEach 3 '(1 2 3 4))
```

# Lambda in a context

- Don't have to use lambda at the top level
- Can refer to other variables in the body of the lambda

```
(define (addNToEach [numToAdd : Number]
                    [xs : (Listof Number)]) : (Listof Number)
  (mapNum (lambda(x) (+ x numToAdd)) xs))
(addNToEach 3 '(1 2 3 4))
```

```
'(4 5 6 7)
```

- The lambda **captures** the variable numToAdd

# Lambda in a context

- Don’t have to use lambda at the top level
- Can refer to other variables in the body of the lambda

```
(define (addNToEach [numToAdd : Number]
                    [xs : (Listof Number)]) : (Listof Number)
  (mapNum (lambda(x) (+ x numToAdd)) xs))
(addNToEach 3 '(1 2 3 4))
```

```
'(4 5 6 7)
```

- The lambda **captures** the variable numToAdd
- Dynamically creates the function that adds its argument to whatever numToAdd is

# Functions as return values

- We can also make functions that produce other functions as a result

# Functions as return values

- We can also make functions that produce other functions as a result
  - Just use `lambda` in the body of the function

# Functions as return values

- We can also make functions that produce other functions as a result
  - Just use `lambda` in the body of the function

# Functions as return values

- We can also make functions that produce other functions as a result
  - Just use `lambda` in the body of the function

```
(define (makeAdderWith n) : (Number -> Number)
  (lambda (x) (+ n x)))
(makeAdderWith 3)
(mapNum (makeAdderWith 3))
```

# Combinators

- Functions that take and return functions

# Combinators

- Functions that take and return functions
- Can “lift” operations to whole functions

# Combinators

- Functions that take and return functions
- Can “lift” operations to whole functions
- Create a new function by specifying how it should behave on each input

# Combinators

- Functions that take and return functions
- Can “lift” operations to whole functions
- Create a new function by specifying how it should behave on each input
    - Lambda lets us refer to this variable

# Combinators

- Functions that take and return functions
- Can “lift” operations to whole functions
- Create a new function by specifying how it should behave on each input
    - Lambda lets us refer to this variable

# Combinators

- Functions that take and return functions
- Can “lift” operations to whole functions
- Create a new function by specifying how it should behave on each input
  - Lambda lets us refer to this variable

```
(define (+fun [f : (Number -> Number)]
              [g : (Number -> Number)]) : (Number -> Number)
  (lambda (x) (+ (f x) (g x))))
;; e.g. Make the function that computes
```

# Example: Beyond Numbers

# Example: Beyond Numbers

```
(define (liftOption [f : (Number -> Number)])
  : ((Optionof Number) -> (Optionof Number))
  (lambda ([optionN : (Optionof Number)])
    (type-case (Optionof Number) optionN
      [(none) (none)]
      [(some x) (some (f x))]
      )))
(define optionPlusOne (liftOption add1))
(optionPlusOne (some 3))
(optionPlusOne (none))
```

# Example: Beyond Numbers

```
(define (liftOption [f : (Number -> Number)])
  : ((Optionof Number) -> (Optionof Number))
  (lambda ([optionN : (Optionof Number)])
    (type-case (Optionof Number) optionN
      [(none) (none)]
      [(some x) (some (f x))]
      )))
(define optionPlusOne (liftOption add1))
(optionPlusOne (some 3))
(optionPlusOne (none))
```

```
(some 4)
(none)
```