Lecture 5: Higher-Order Functions

Brian Hou
June 27, 2016
Announcements
Announcements

• Homework 2 is due Wednesday 6/29
Announcements

• Homework 2 is due Wednesday 6/29
• Project 1 is due Thursday 6/30
Announcements

• Homework 2 is due Wednesday 6/29
• Project 1 is due Thursday 6/30
  • Earn 1 EC point for completing it by Wednesday 6/29
Announcements

• Homework 2 is due Wednesday 6/29
• Project 1 is due Thursday 6/30
  • Earn 1 EC point for completing it by Wednesday 6/29
• Quiz 2 is on Thursday 6/30 at the beginning of lecture
Announcements

- Homework 2 is due Wednesday 6/29
- Project 1 is due Thursday 6/30
  - Earn 1 EC point for completing it by Wednesday 6/29
- Quiz 2 is on Thursday 6/30 at the beginning of lecture
  - Environment Diagrams and Higher-Order Functions
Announcements

• Homework 2 is due Wednesday 6/29
• Project 1 is due Thursday 6/30
  • Earn 1 EC point for completing it by Wednesday 6/29
• Quiz 2 is on Thursday 6/30 at the beginning of lecture
  • Environment Diagrams and Higher-Order Functions
• Group Tutoring is available! See Piazza for details
Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications
This week (Functions), the goals are:

- To understand the idea of *functional abstraction*
- To study this idea through:
  - higher-order functions
  - recursion
  - orders of growth
Higher-Order Functions
Generalizing Computations
Generalizing Computations

\[
\sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15
\]

\[
\sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225
\]

\[
\sum_{k=1}^{5} \frac{8}{(4k - 3) \cdot (4k - 1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04
\]
Generalizing Computations

\[ \sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15 \]

\[ \sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225 \]

\[ \sum_{k=1}^{5} \frac{8}{(4k - 3) \cdot (4k - 1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04 \]
Generalizing Computations

\[ \sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15 \]

\[ \sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225 \]

\[ \sum_{k=1}^{5} \frac{8}{(4k - 3) \cdot (4k - 1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04 \]
Generalizing Computations

\[ \sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15 \]

\[ \sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225 \]

\[ \sum_{k=1}^{5} \frac{8}{(4k - 3) \cdot (4k - 1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04 \]
Generalizing Computations (demo)

\[ \sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15 \]

\[ \sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225 \]

\[ \sum_{k=1}^{5} \frac{8}{(4k-3) \cdot (4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04 \]
def sum_naturals(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + k, k + 1
    return total

def sum_cubes(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + pow(k, 3), k + 1
    return total
def sum_naturals(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + k, k + 1
    return total

def sum_cubes(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + pow(k, 3), k + 1
    return total
def sum_naturals(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + k, k + 1
    return total

def sum_cubes(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + pow(k, 3), k + 1
    return total
def sum_naturals(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + k, k + 1
    return total

def sum_cubes(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + pow(k, 3), k + 1
    return total
cube = \texttt{lambda }k:\ \texttt{pow}(k,\ 3)

def summation(n, \text{term}):
    """Sum the first N terms of a sequence."

    >>> summation(5, cube)
    225
    ""
    total, k = 0, 1
    \textbf{while} k <= n:
        total, k = total + \text{term}(k), k + 1
    \textbf{return} total
Summation Example

cube = lambda k: pow(k, 3)

```python
def summation(n, term):
    """Sum the first N terms of a sequence."

    >>> summation(5, cube)
    225
    """

    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Function of a single argument (not called "term")
Summation Example

cube = lambda k: pow(k, 3)

def summation(n, term):
    """Sum the first N terms of a sequence."

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
Summation Example

cube = lambda k: pow(k, 3)

```python
def summation(n, term):
    """Sum the first N terms of a sequence.""

    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225
"""
```

Function of a single argument (not called "term")

A parameter that will be bound to a function

The function bound to term gets called here
Summation Example

cube = lambda k: pow(k, 3)

```python
def summation(n, term):
    # Sum the first N terms of a sequence.
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

```python
>>> summation(5, cube)
225

"""The cube function is passed as an argument value
The function bound to term gets called here"
```
Locally Defined Functions
Locally Defined Functions (demo)
Locally Defined Functions

- Functions defined within other function bodies are bound to names in a *local* frame
def make_adder(n):
    """Return a function that takes one argument K and returns K + N."
    adder = lambda k: k + n
    return adder

>>> add_three = make_adder(3)
>>> add_three(4)
7
"""

def adder(k):
    return k + n

return adder
def make_adder(n):
    """Return a function that takes one argument K and returns K + N.
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder

Locally Defined Functions

- Functions defined within other function bodies are bound to names in a local frame

A function that returns a function
def make_adder(n):
    """Return a function that takes one argument K and returns K + N."
    
    def adder(k):
        return k + n
    
    return adder

Locally Defined Functions

- Functions defined within other function bodies are bound to names in a local frame

A function that returns a function

The name add_three is bound to a function
Locally Defined Functions (demo)

- Functions defined within other function bodies are bound to names in a local frame

```python
def make_adder(n):
    
    """Return a function that takes one argument K and returns K + N."

    def adder(k):
        return k + n
    
    return adder

make_adder(3)(4)  # 7

"""
```

A function that returns a function

The name add_three is bound to a function

A def statement within another def statement
```
def make_adder(n):
    """Return a function that takes one argument K and returns K + N.
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

Locally Defined Functions

- Functions defined within other function bodies are bound to names in a *local* frame.

A function that returns a function

The name add_three is bound to a function

A def statement within another def statement

Can refer to names in the enclosing function
def make_adder(n):
    """Return a function that takes one argument K
    and returns K + N."
    adder = lambda k: k + n
    return adder

>>> add_three = make_adder(3)
>>> add_three(4)
7

"""

def adder(k):
    return k + n

return adder

Locally Defined Functions

- Functions defined within other function bodies are bound to names in a local frame.

A function that returns a function

The name add_three is bound to a function

A def statement within another def statement

Can refer to names in the enclosing function
Higher-Order Functions
Higher-Order Functions

**Functions are first-class:** Functions can be manipulated as values in our programming language
Higher-Order Functions

**Functions are first-class:** Functions can be manipulated as values in our programming language

**Higher-order function:**
Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language

Higher-order function:
1. A function that takes a function as an argument value or
Higher-Order Functions

**Functions are first-class:** Functions can be manipulated as values in our programming language.

**Higher-order function:**
1. A function that takes a function as an argument value or
2. A function that returns a function as a return value
Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language

Higher-order function:
1. A function that takes a function as an argument value or
2. A function that returns a function as a return value

Higher-order functions:
Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

Higher-order function:
1. A function that takes a function as an argument value or
2. A function that returns a function as a return value

Higher-order functions:
• Express general methods of computation
Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language

Higher-order function:
1. A function that takes a function as an argument value or
2. A function that returns a function as a return value

Higher-order functions:
- Express general methods of computation
- Remove repetition from programs
Higher-Order Functions

**Functions are first-class:** Functions can be manipulated as values in our programming language

**Higher-order function:**
1. A function that takes a function as an argument value or
2. A function that returns a function as a return value

Higher-order functions:
- Express general methods of computation
- Remove repetition from programs
- Separate concerns among functions
Break!
Environments (Round 2)
Nested Definitions
Nested Definitions (demo)
1. def make_adder(n):
   2.     def adder(k):
   3.         return k + n
   4.     return adder
   5.
   6. add_three = make_adder(3)
   7. add_three(4)
Nested Definitions

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add_three = make_adder(3)
add_three(4)
```
Every user-defined function has a parent frame.
Nested Definitions

- Every user-defined function has a parent frame
- The parent of a function is the frame in which it was defined
Nested Definitions

- Every user-defined function has a parent frame.
- The parent of a function is the frame in which it was defined.
- Every local frame has a parent frame.

def make_adder:
    def adder(k):
        return k + n
    return adder

add_three = make_adder(3)
add_three(4)
Every user-defined function has a parent frame

The parent of a function is the frame in which it was defined

Every local frame has a parent frame

The parent of a frame is the parent of the function called
Environment Diagram Rules (version 2)
Environment Diagram Rules (version 2)

Rules for `def` Statements:

1. Create a function with signature `<name>`(<parameters>) and `parent [parent=<label>]` (parent is the current frame)

   ```python
   f1: make_adder
   func adder(k) [parent=f1]
   ```

2. Set the body of that function to be everything indented after the first line

3. Bind `<name>` to that function in the current frame
Environment Diagram Rules (version 2)

Rules for **def** Statements:

1. Create a function with signature `<name> (<parameters>) and parent [parent=<label>]` (parent is the current frame)

   ```
   f1: make_adder
   func adder(k) [parent=f1]
   ```

2. Set the body of that function to be everything indented after the first line

3. Bind `<name>` to that function in the current frame

Rules for calling user-defined functions:

1. Create a new environment frame

2. **Copy the parent of the function to the local frame:**
   ```
   [parent=<label>]
   ```

3. Bind the function's parameters to its arguments in that frame

4. Execute the body of the function in the new environment
Function Composition
Environment Diagram

```python
def square(x):
    return x * x

def make_adder(n):
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```
```python
def square(x):
    return x * x

def make_adder(n):
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```
def square(x):
    return x * x

def make_adder(n):
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```python
def square(x):
    return x * x

def make_adder(n):
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```
Return value of make_adder is an argument to compose1.
Return value of make_adder is an argument to compose1

Environment Diagram

```python
def square(x):
    return x * x

def make_adder(n):
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```
Return value of make_adder is an argument to compose1.
def square(x):
    return x * x

def make_adder(n):
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

return_value_of_make_adder_is_an_argument_to_compose1

Environment Diagram
Return value of make_adder is an argument to compose1
Return value of make_adder is an argument to compose1
Return value of make_adder is an argument to compose1.
Application: Currying
Application: Currying

- add is a two-argument function that returns the sum of the two arguments
Application: Currying

• add is a two-argument function that returns the sum of the two arguments

• make_adder is a one-argument function that returns a one-argument function that returns the sum of the two arguments
Application: Currying

- add is a two-argument function that returns the sum of the two arguments
- make_adder is a one-argument function that returns a one-argument function that returns the sum of the two arguments
- Currying allows us to represent functions with multiple variables as chains of functions with single variables
Application: Currying

- add is a two-argument function that returns the sum of the two arguments
- make_adder is a one-argument function that returns a one-argument function that returns the sum of the two arguments
- Currying allows us to represent functions with multiple variables as chains of functions with single variables

```
(lambda x, y: x * y + 1)(3, 4)
```
Application: Currying

• add is a two-argument function that returns the sum of the two arguments

• make_adder is a one-argument function that returns a one-argument function that returns the sum of the two arguments

• Currying allows us to represent functions with multiple variables as chains of functions with single variables

\[
\text{(lambda } x, y: x \ast y + 1)(3, 4)
\]

\[
\text{(lambda } x: \text{lambda } y: x \ast y + 1)(3)(4)
\]
Application: Currying

• add is a two-argument function that returns the sum of the two arguments

• make_adder is a one-argument function that returns a one-argument function that returns the sum of the two arguments

• Currying allows us to represent functions with multiple variables as chains of functions with single variables

• It is named after mathematician and logician Haskell Curry (who rediscovered it after Moses Schönfinkel)

\[
\text{add}(x, y) = x + y
\]

\[
\text{make_adder}(x)(y) = x + y
\]

\[
\lambda x, y : x + y + 1(3, 4)
\]

\[
\lambda x : \lambda y : x + y + 1(3)(4)
\]
Application: Currying

• add is a two-argument function that returns the sum of the two arguments

• make_adder is a one-argument function that returns a one-argument function that returns the sum of the two arguments

• Currying allows us to represent functions with multiple variables as chains of functions with single variables

• It is named after mathematician and logician Haskell Brooks Curry (who rediscovered it after Moses Schönfinkel)

\[
\begin{align*}
(\text{lambda } x, y: x \cdot y + 1)(3, 4) \\
(\text{lambda } x: (\text{lambda } y: x \cdot y + 1))(3)(4)
\end{align*}
\]