Introduction to Functional Programming: Part 1

Formative

Rewrite the function definition *f*(*x*) = *x* + 1 using lambda notation.

[Correct Answer] A.

B.

C.

D.

A. This is correct. When you use lambda notation, you put the Greek letter before the function's parameter and put the function's formula after the right arrow. See “Using Filter, Map, and Fold.”

B. A lambda expression must use the symbol to indicate the function's parameter. See “Using Filter, Map, and Fold.”

C. This is an assignment statement, not a lambda expression. See “Using Filter, Map, and Fold.”

D. A lambda expression uses an arrow to separate the function's parameter from the function's formula. See “Using Filter, Map, and Fold.”

What's the value of the expression (?

[Correct Answer] A. 126

B. 6*x*

C. *f*(*x*) = 6*x*

D. 21

A. If we translate this expression into conventional function notation, we're applying the function to the value 21. And 6 times 21 is 126. See “Using Filter, Map, and Fold.”

B. Evaluating this expression means plugging 21 into . See “Using Filter, Map, and Fold.”

C. The value of this lambda expression is a number, not a function definition. See “Using Filter, Map, and Fold.”

D. If we translate this expression into conventional function notation, we're applying the function to the value 21. But 6 times 21 isn't 21. See “Using Filter, Map, and Fold.”

What's the value of foldFromLeft(minus, 7, [3, –8, 9])?

[Correct Answer] A. 3

B. –13

C. 5

A. Applying minus from left to right, we have . See “Using Filter, Map, and Fold.”

B. Applying minus from left to right, we don't get the value –13. See “Using Filter, Map, and Fold.”

C. Applying minus from left to right, we don't get the value 5. See “Using Filter, Map, and Fold.”

Introduction to Functional Programming: Part 3

Formative

Let , , . What's the value of ?

[Correct Answer] A.

B.

C.

.

. But isn't the same as .

. But isn't the same as

Which of the following is a higher-order function?

[Correct Answer] A. forEach(list, f) acts on the list [l0, l1, ..., ln] and returns [f(l0), f(l1), ..., f(ln)].

B. sum(list) acts on the list [l0, l1, ..., ln] and returns l0 + l1 + ... + ln.

C. addAndSquare(n1, n2) acts on two numbers, n1 and n2, and returns (n1 + n2)2.

D. mapSquare(list) acts on the list [l0, l1, ..., ln] and returns [l02, l12, ..., ln2].

The second argument of the forEach function is a function, f. So forEach is a higher-order function. See “Higher-Order Functions.”

The sum function has only one argument, and that argument isn't a function. See “Higher-Order Functions.”

The addAndSquare function has two arguments, and neither of these arguments is a function. See “Higher-Order Functions.”

At first glance, mapSquare might seem to be a higher-order function because it's the same as map(square, list). But the mapSquare function has only one argument, and that argument is a list, not a function. See “Higher-Order Functions.”

Which of the following is *not* true about function composition?

[Correct Answer] A. For any two functions, and , is the same as .

B. Composition is a higher-order function.

C. For any two functions, and , is a function.

D. sum(map(getAmount, [purchase0, purchase1, ..., purchasen])) is an example of function composition.

Let and . Then and . In this case, . See “Higher-Order Functions.”

The operator takes two functions as its arguments, so is a higher-order function. See “Higher-Order Functions.”

Let and . Then . So is a function. See “Higher-Order Functions.”

In this expression, we apply the map function and then apply the sum function to the result we got when we applied the map function. That's function composition. See “Higher-Order Functions.”

In the video, we describe the filter function as follows:

filter : function, list → list

How can we use this notation to describe the function ?

[Correct Answer] A. f : number → number

B. f : number → number + 1

C. f : function → function

D. f : x → x + 1

The argument of f is a number, and f returns a number as its result. See “Currying.”

When we use this notation, we describe the types of the function's argument(s) and the type of the function's result. We don't describe the formula or rule that the function implements. So + 1 doesn't belong in this kind of notation. See “Currying.”

The argument of f isn't a function, and the result returned by f isn't a function. See “Currying.”

When we use this notation, we describe the types of the function's argument(s). The name x doesn't stand for a type; it stands for a variable. See “Currying.”

In the video, we describe the filter function as follows:

filter : function, list → list

We can obtain a function, f, by partially applying 2 to the first argument in the function . How can we use the notation to describe this new function f?

[Correct Answer] A. f : number → number

B. f : number, number → number

C. f : list → number

D. f : number → number + 2

The new function, f, is defined as follows: . The argument of f is a number, and the result returned by f is a number. See “Currying.”

The new function, f, has one argument, not two arguments. See “Currying.”

The new function's argument is not a list. See “Currying.”

When we use this notation, we describe the types of the function's argument(s) and the type of the function's result. We don't describe the formula or rule that the function implements. So + 2 doesn't belong in this kind of notation. See “Currying.”

What functions do you get when you curry the function ?

[Correct Answer] A. curryF(0) is a function of y that returns -y.

curryF(1) is a function of y that returns 1 - y.

curryF(2) is a function of y that returns 4 - y.

And so on.

B. curryF(0) is a function of x that returns x2.

curryF(1) is a function of x that returns x2 - 1.

curryF(2) is a function of x that returns x2 - 2.

And so on.

C. curryF(x, y) = f(y, x) = y2 - x

D. curryF(0) = f(0, 0) = 02 - 0 = 0  
curryF(1) = f(1, 1) = 12 - 1 = 0  
curryF(2) = f(2, 2) = 22 - 2 = 2

And so on.

When you curry a function of two arguments, you get several single-argument functions. You get each single-argument function by selecting a constant value for the first of the original two arguments. Therefore,

And so on. See “Currying.”

When you curry a two-argument function, you plug a value into the function's first argument. What you get is a function that applies to the original function's second argument. See “Currying.”

Currying isn't the same as flipping. See “Currying.”

*Currying* doesn't mean applying the same value to both arguments. See “Currying.”

In the video, we describe the filter function as follows:

filter : function, list list

How can we use the notation to describe the (function composition) operator?

[Correct Answer] A. : function, function function

B. : function function

C. : number, number number

D. : function, number number

If you have two functions, and , applying to these functions gives you a new function, namely . See “Currying.”

This choice incorrectly asserts that the operator takes only one argument. The operator takes two arguments. For example, in the expression , the first argument is and the second argument is . See “Currying.”

This choice incorrectly asserts that the operator takes two arguments, each of type number. But, in the expression , the arguments and aren't numbers. See “Currying.”

This choice incorrectly asserts that the operator takes two arguments, one of which is a number. But, in the expression , neither of the arguments and are numbers. See “Currying.”

What do you get when you perform partial application twice on foldFromLeft(f, n, list), using sum for f and 0 for n?

[Correct Answer] A. You get a function sum : list number. When you apply this function to a list, you get the sum of the elements in the list.

B. You get a number. It's the sum of the elements in the list.

C. You get a function sumUp : number, list number. When you apply this function to a number and a list, you get the sum of the number plus the elements in the list.

D. You get a function foldFromLeft : function, number, list number.

With one partial application, you get a function sumOf : number, list number. This function adds up its number argument plus all the numbers in its list argument. With the second partial application, the new sum function adds up 0 plus all the numbers in its list. See “Currying.”

Performing two partial applications on a function with three arguments gives you a new function with one argument. It doesn't give you a number. See “Currying.”

This choice describes what you get when you perform partial application only once. See “Currying.”

This choice describes the original foldFromLeft function before you've performed partial application. See “Currying.”

In mathematics, = the product of all numbers from 1 to For example, . Which recursive definition of fact takes a number , and returns as a numeric value?

[Correct Answer] A. fact(1) = 1  
fact(n) = fact(n - 1) \* n

B. fact(1) = 1  
fact(n + 1) = fact(n - 1) \* n

C. fact(1) = 1  
fact(n) = n \* (n - 1)

D. fact(n) = n \* fact(n - 1)

Substituting 2 for n, fact(2) = fact(2-1)\*2 = fact(1)\*2 = 1\*2 = 2.

Substituting 3 for n, fact(3) = fact(3-1)\*3 = fact(2)\*3 = 2\*3 = 6.

Substituting 4 for n, fact(4) = fact(4-1)\*4 = fact(3)\*4 = 6\*4 = 24.

And so on. See “Lists.”

With this definition, fact(3) = fact(2 + 1) = fact(2 - 1) \* 3 = fact(1) \* 3 = 3. But fact(3) is , not . See “Lists.”

With this definition, fact(4) = 4 \* 3 = 12. But fact(4) is , not . See “Lists.”

This code has no base case. So, for example, fact(3) = 3 \* fact(2) = 3 \* 2 \* fact(1) = 3 \* 2 \* 1 \* fact(0) = 3 \* 2 \* 1 \* 0 \* fact(-1), and so on. The calculation never ends. See “Lists.”

The Fibonacci sequence starts with the two numbers 1, 1. Each number thereafter is the sum of the two numbers before it in the sequence. For example, the sequence's third number is 1 + 1 = 2. The sequence's fourth number is 1 + 2 = 3. The sequence's fifth number is 2 + 3 = 5. And so on.

Which recursive definition of fib takes a number, , and returns the th number in the Fibonacci sequence?

[Correct Answer] A. fib(1) = 1  
fib(2) = 1  
fib(n) = fib(n - 2) + fib(n - 1)

B. fib(n) = fib(n - 2) + fib(n - 1)

C. fib(1) = 1  
fib(n) = fib(n - 2) + fib(n - 1)

D. fib(1) = 1  
fib(2) = 1  
fib(n) = fib((n - 2) \* (n - 1))

fib(3) = fib(1) + fib(2) = 1 + 1 = 2  
fib(4) = fib(2) + fib(3) = 1 + 2 = 3  
fib(5) = fib(3) + fib(4) = 2 + 3 = 5

And so on. See “Lists.”

This code has no base case. For example, fib(3) = fib(1) + fib(2) = (fib(-1) + fib(0)) + (fib(0) + fib(1)) = fib(-3) + fib(-2) + . . . , and so on. See “Lists.”

This code needs an additional base case. With this code, fib(2) = fib(0) + fib(1). Since there's no specific definition for fib(0), the code's second line applies, so fib(0) = fib(-2) + fib(-1), and so on. See “Lists.”

With this definition, fib(3) = fib(1 \* 2) = fib(2) = 1. But fib(3) is , not . See “Lists.”

A function named up takes a number, , and returns a list whose values are . Which recursive definition of up is correct?

[Correct Answer] A. up(0) = [0]  
up(n) = concat( up(n-1),[n] )

B. up(0) = [0]  
up(n) = up(n-1)::[n]

C. up(0) = [0]  
up(n) = concat( [n],up(n-1) )

D. up(0) = [0]  
up(n) = (n-1)::up(n)

For example:

up(2) = concat(       up(1)        , [2] )  
      = concat( concat( up(0),[1] ), [2] )  
      = concat( concat(  [0] ,[1] ), [2] )  
      = concat(       [0,1]        , [2] )  
      = [0,1,2]

See “More Recursion Examples.”

With this definition, up(1) = up(0)::[1] = [0]::[1] = [[0], 1]. But up(1) is [0,1], not [[0],1]. See “More Recursion Examples.”

With this definition, up(1) = concat( [1], up(0) ) = concat( [1], [0] ) = [1,0]. But the value of up(1) is [0,1], not [1,0]. See “More Recursion Examples.”

With this definition, up(1) = 0::up(1) = 0::(0::up(1)) = 0::(0::(0::up(1))), and so on. The expansion never ends. See “More Recursion Examples.”

Ackermann's function is defined as follows:

ack(0,x) = x + 1  
ack(n,0) = ack(n-1, 1)  
ack(n,x) = ack(n-1, ack(n,x-1))

Some values of the ack function are too large for any computer to calculate. For example, ack(4,3) is approximately , a number with more than 2,800 digits.

What's the value of ack(1,1)?

[Correct Answer] A. 3

B. 1

C.

D. 0

To find the value of ack(1,1), start by applying the third line in the definition:

ack(1,1) = ack(0, ack(1,0))

Then, apply the second line in the definition:

         = ack(0, ack(0,1))

Then, apply the first line in the definition:

         = ack(0,     2   )

Then, apply the first line again:

         = 3

See “More Recursion Examples.”

The definition of Ackermann's function consists of three lines. When you find the value of ack(1,1), you apply each line at least once. See “More Recursion Examples.”

The definition of Ackermann's function consists of three lines. When you find the value of ack(1,1), you apply each line at least once. See “More Recursion Examples.”

The definition of Ackermann's function consists of three lines. When you find the value of ack(1,1), you apply each line at least once. See “More Recursion Examples.”

Introduction to Functional Programming: Part 5

Formative

What's the value of sqrtMaybe(x-10) >>= minus4Maybe >>= reciprocalMaybe >>= plus13Maybe when x is 10?

[Correct Answer] A. Just 12.75

B. 12.75

C. Nothing

D. 0

sqrtMaybe(10-10) is Just 0.

Binding Just 0 with minus4Maybe yields Just -4.

Binding Just -4 with reciprocalMaybe yields Just -1/4.

Binding Just -1/4 with plus13Maybe yields Just 12.75.

See “More Recursion Examples.”

The result from a Maybe function is a Just value or Nothing. See “More Recursion Examples.”

In this code, the only function that can trigger a sequence of Nothing values is reciprocalMaybe. But the value that we're feeding to reciprocalMaybe isn't zero. So this code doesn't return the Nothing value. See “More Recursion Examples.”

See “More Recursion Examples.”

What's the value of sqrtMaybe(x-1) >>= minus4Maybe >>= reciprocalMaybe >>= plus13Maybe when x is 17?

[Correct Answer] A. Nothing

B. Just 12.75

C. 12.75

D. 0

sqrtMaybe(17-1) is Just 4.

Binding Just 4 with minus4Maybe yields Just 0.

Binding Just 0 with reciprocalMaybe yields Nothing.

Binding Nothing with plus13Maybe yields Nothing.

See “More Recursion Examples.”

See “More Recursion Examples.”

The result from a Maybe function is a Just value or Nothing. See “More Recursion Examples.”

See “More Recursion Examples.”

What's the value of sqrtMaybe(x-10) >>= minus4Maybe >>= reciprocalMaybe >>= plus13Maybe when x is 9?

[Correct Answer] A. Nothing

B. Just Nothing

C. Just

D. Just 12.9230769231

sqrtMaybe(9-10) is Nothing (assuming we're not using imaginary numbers).

Binding Nothing with minus4Maybe yields Nothing.

Binding Nothing with reciprocalMaybe yields Nothing.

Binding Nothing with plus13Maybe yields Nothing.

See “More Recursion Examples.”

In our system, there's no such thing as Just Nothing. See “More Recursion Examples.”

In our system, there's no such thing as Just. There has to be a value after the   
word Just. See “More Recursion Examples.”

See “More Recursion Examples.”

Introduction to Functional Programming: Summative Quiz

Summative

What's the value of foldFromRight(minus, 0, [3, –8, 9])?

[Correct Answer] A. 20

B. 2

C. -4

D. -14

A. Applying minus from right to left, we have (9.

Learning Objective: Understand what distinguishes functional programming from other programming paradigms. Review “Part 1: Getting Started with Functional Programming.”

B. Applying minus from right to left, we don't get the value .

Learning Objective: Understand what distinguishes functional programming from other programming paradigms. Review “Part 1: Getting Started with Functional Programming.”

C. Applying minus from right to left, we don't get the value .

Learning Objective: Understand what distinguishes functional programming from other programming paradigms. Review “Part 1: Getting Started with Functional Programming.”

D. Applying minus from right to left, we don't get the value .

Learning Objective: Understand what distinguishes functional programming from other programming paradigms. Review “Part 1: Getting Started with Functional Programming.”

Let and . What's the value of ?

[Correct Answer] A.

B.

C.

D. None of the other choices are correct.

Learning Objective: Declare and evaluate functions whose parameters or result values (or both) are functions. Review “Part 3: Higher-Order Functions.”

. But isn't the same as .

Learning Objective: Declare and evaluate functions whose parameters or result values (or both) are functions. Review “Part 3: Higher-Order Functions.”

. But isn't the same as .

Learning Objective: Declare and evaluate functions whose parameters or result values (or both) are functions. Review “Part 3: Higher-Order Functions.”

One of the other choices is correct. Find the value of .

Learning Objective: Declare and evaluate functions whose parameters or result values (or both) are functions. Review “Part 3: Higher-Order Functions.”