Exercises to Accompany
Introduction to Functional Programming
How to Think Functionally in (Almost) Any Language
with Barry Burd

This list includes three kinds of exercises:

- Exercises marked **General**: Complete these exercises without writing or reading code of any kind, or explore features in a programming language that you may not have seen before.
- Exercises marked **Using pseudocode**: Complete these exercises by reading or writing simplified syntax that doesn't belong to any particular programming language, like the syntax that I use in the course.
- Exercises marked **In your programming language**: Complete these exercises by reading or writing code in a programming language of your choice. (And don't forget to test your code!)

There are exercises for almost every section of the course. For example, Exercises 1.1, 1.2 and 1.3 are for the first section (the section entitled *About This Course*). Exercises 2.1, 2.2 (and so on) are for the second section (the section entitled *Solving a Problem Both Ways*).

Some solutions appear at the end of the list of exercises.

If you have questions, send email to functional@allmycode.com, tweet to @allmycode, or post on Facebook to /allmycode.

Have fun and learn a lot!

1. About This Course

General

1.1. The following program is written in a very old version of the BASIC programming language. In case you're wondering, the percent sign (%) is BASIC's mod operator. So, for example, `6 % 3` is `0` and `7 % 3` is `1`.
Trace through the execution of the program to determine the program's output.
GOTO statements are bad, and this exercise with GOTO statements is intentionally annoying. So have fun with it but don't take it too seriously.

```
10 LET X = 5
25 PRINT X
20 GOTO 90
30 LET X = X + 8
35 PRINT X
35 IF X % 2 = 0 THEN GOTO 60
40 LET X = X * 5
45 PRINT X
50 IF X < 51 THEN GOTO 30
```

```
60 LET X = X - 1
65 PRINT X
70 IF X % 3 = 0 THEN GOTO 30
85 END
90 LET X = X * 2
95 PRINT X
100 GOTO 40
```

1.2. Type the code from Exercise 1.1 into the interpreter at `www.quitebasic.com` to find out if your proposed output is correct.

In your programming language

1.3. *Obfuscated* code is code that's difficult to read. There are good reasons and bad reasons for creating obfuscated code. Some software tools turn readable code into obfuscated code in order to keep the code from being hacked. On the other hand, some obfuscated code is written for fun to show how strange the code can be. People post examples of such code on the Internet.
Search the Internet for fun examples of obfuscated code.

2. Solving a Problem Both Ways

Pseudocode

2.1. Here's a slight modification of the credit card categorization problem (called Problem 1 in the video): Write imperative-style pseudocode to display the items in the Food category whose amounts are $10 or more. In your solution, don't use the word AND. Don't use a symbol that stands for the word AND.

2.2. Write functional-style pseudocode to solve the problem in Exercise 2.1.

In your programming language

2.3. Write and test an imperative-style program to solve the credit card categorization problem (called Problem 1 in the video).

2.4. Write and test an imperative-style program to solve the problem in Exercise 2.1.

2.5. Find out if your language has a feature like the `filter` function. If it does, learn how to use the filter function to solve the credit card categorization problem.

3. Using Filter, Map and Fold

Pseudocode

3.1. Rewrite the following function definitions using lambda notation:

3.1.1. f(x) = x + 1

3.1.2. f(x) = x
This is the identity function – the function that returns whatever you give it.

3.1.3. nameOf(customer) = customer.name

3.2. Evaluate the following lambda expressions:

3.2.1. $(\lambda\ x \rightarrow 6 * x)\ (21)$

3.2.2. $(\lambda\ x \rightarrow x / 2)\ ((\lambda\ x \rightarrow x + 7)\ (19))$

3.3. Evaluate the following expressions

3.3.1. map(timesTwo, [2, 4, 5])
3.3.2. map(timesTwo, [8])
3.3.3. map(timesTwo, [])
3.3.4. map(addOne, map(timesTwo, [2, 2, 4, –3]))
3.3.5. map(timesTwo, map(addOne, [2, 2, 4, –3]))
3.3.6. foldFromLeft(plus, 7, [3, –8 9])
3.3.7. foldFromLeft(minus, 7, [3, –8, 9])
3.3.8. foldFromRight(minus, 7, [3, –8, 9])
3.3.9. foldFromLeft(minus, 7, map(timesTwo, [3, 0, 8]))

3.4. You're given a list of customers. Each customer has a name and an amount. A customer's amount is that customer's outstanding balance. Write imperative-style pseudocode to print the smallest negative balance. For example, if Joe's balance is –$10, Ann's balance is –$2, and Donna's balance is $5, print –2.
(Assume that the list has at least one customer in it, and that customers' balances range between –$1000 and $1000.)

3.5. You have a function (called `max`) that finds the larger of two numbers. Write functional-style pseudocode to solve the problem in Exercise 3.4.

In your programming language

3.6. Find out if your language has features like the `map`, `foldFromLeft` and `foldFromRight` functions.

4. Imperative and Functional Programming Languages

General

4.1. Microsoft Excel has a Filter feature. You can find documentation about this feature by visiting https://support.office.com/en-us/article/Filter-data-in-a-range-or-table-01832226-31b5-4568-8806-38c37dcc180e. Read about this feature, and try using it in Microsoft Excel.

4.2. Microsoft Excel has an Aggregate function, which behaves a bit like our `foldFromLeft` and `foldFromRight` functions. You can find documentation about this function by visiting https://support.office.com/en-us/article/AGGREGATE-function-43b9278e-6aa7-4f17-92b6-e19993fa26df. Read about this function, and try using it in Microsoft Excel.

In your programming language

4.3. Find out how the experts classify your programming language. Is it imperative, purely functional, hybrid, or some other kind of language.

4.4. If your language supports some functional features, read up on those features. Find a few simple sample programs on the web and run them to find out how they behave.

4.5. Many languages can be extended to include functional features that aren't officially part of the language. Groups of developers create tools enabling you to use these additional functional features. Find out if your programming language has such extensions.

5. Pure Functions

Pseudocode

5.1. Which of these pseudocode functions are pure? Which aren't pure? Why?

5.1.1. f(x) = x + x + x

5.1.2.

```
f(x) {
    x = x + 7
    return x
}
```

5.1.3. f(x) = x + current_day_of_the_month
where current_day_of_the_month is a number from 1 to 31

5.1.4.

```
f(x) {
    integer y = 3
    return x + y
}
```

5.1.5.

```
f(x) {
    integer y = random()
    return x + y - y
}
```

5.1.6. length(the_string_s) = number of characters in the_string_s

5.1.7.

```
post(message, URL) {
    add the message to the message board at the URL
}
```

5.2. Which of these expressions are referentially transparent? Which aren't? Why?

5.2.1. inputFromKeyboard(x)
If you execute `y = inputFromKeyboard(x)`, and the user types 7, then the value of y becomes 7.

5.2.2. 7 + 6

5.2.3. f(x) = x + current_day_of_the_month
where current_day_of_the_month is a number from 1 to 31

5.3. Revisit your solution to Problem 3.2 to make sure that nothing in your solution is mutable.

6. Some Benefits of Pure Functions

Pseudocode

6.1. Use pseudocode to write tests for the following two functions. Identify the setup step(s) required to test each function.

6.1.1.

```
mortgage(principle, ratePercent, numYears) {
  rate = ratePercent / 100
  numPayments = numYears * 12
  effectiveAnnualRate = rate / 12
  payment = principal * (effectiveAnnualRate
   / (1 - (1 + effectiveAnnualRate)^(-numPayments))))
}
```

In case you care, my ^ symbol means "to the power of." Also, the monthly payment on a 30 year, $100000.00 mortgage with 5.25% interest is $552.20.

6.1.2.

```
mortgage(principle, numYears) {
  rate = currentRatePercent / 100
  numPayments = numYears * 12
  effectiveAnnualRate = rate / 12
  payment = principal * (effectiveAnnualRate
   / (1 - (1 + effectiveAnnualRate)^(-numPayments))))
}
```

The function in this exercise is almost the same as the function in Exercise 6.1.1. The only difference is that this exercise's function uses a variable (`currentRatePercent`) whose value is set outside of the `mortgage` function and can be modified outside of the `mortgage` function.

6.2. The `mortgage` function in Exercise 6.1.1 might run correctly when you test it with parameters `100000.00`, `5.25`, `30`. Does this mean that the function will run correctly whenever anyone calls the function with parameters `100000.00`, `5.25`, `30`?

6.3. The `mortgage` function in Exercise 6.1.2 might run correctly when you test it with parameters `100000.00`, `30`. Does this mean that the function will run correctly whenever anyone calls the function with parameters `100000.00`, `30`?

6.4. Define `factorial(n) = 1*2*3*...*n`. Here's a pseudocode program to repeatedly calculate `factorial(n)` and count the number of multiplication operations done during the calculation:

```
loop
    input n
    result = 1
    count = 0
    for i from 2 to n do
        result = result * i
        count = count + 1
    print result
    print count
```

The factorial function is pure so it can be memoized. Use memoization to make values of `count` smaller for values of `n` less than or equal to 100.

In your programming language

6.5. Write a program that inputs an integer that's less than or equal to 1000 (call it `n`) and then uses a loop to add up the integers 1 to `n`. The program displays the resulting sum.

6.6. Modify the program of Exercise 6.4 so that the program repeatedly inputs a new value for `n` and then displays the sum of the integers 1 to `n`. The program stops repeating when the user enters 0 for `n`.
To solve this problem, create an array of size 1000 (call it the `totals` array). Put 1 into `totals[1]`. Then put the sum of 1 and 2 into `totals[2]`. Then put the sum of 1, 2 and 3 into `totals[3]`. And so on. What if the user inputs 950 for `n` and then 765 for `n`? Don't recalculate the sum of the numbers 1 to 765. In one step, get that sum from the `totals` array.

7. Avoiding Race Conditions

General

7.1. Ann's parents, Bob and Carol, have a joint bank account. Bob visits an ATM machine. He checks his balance, which is $300. So Bob requests a withdrawal of $100. Then the machine's screen displays a message saying that Bob can't withdraw the money. What went wrong?

7.2. Describe a scenario in which the credit card total problem (called Problem 2 in the video) can suffer from race conditions.

Pseudocode

7.3. What outputs may result from running the following code?

```
x = 0
three times do {
    simultaneously do {
        x = x + 1
    }
    and
    {
        x = x + 1
    }
}
print x
```

8. Efficient Parameter Passing

General

8.1. In an old version of the FORTRAN programming language, any numeric literals that were passed as parameters were stored as variables with values before they were passed. What unwanted consequence can this have for parameter passing?

In your programming language

8.2. There are many ways for languages to implement parameter passing. You've probably used parameter passing in an imperative language, and you probably know the rules that govern parameter passing in your language. But do you know how parameter passing works under the hood? Research this question for the language of your choice.

8.3. Are there different options for passing parameters in your programming language? If so, which of them allow you to modify the values of the parameters in the function call? Which don't?

9. Lazy Evaluation

Pseudocode

9.1. In each part of this problem, evaluate the expressions lazily, and then eagerly. In each case, are the results different?

9.1.1. In this problem, a call to the `print` function returns the number of values that were successfully printed. For example, `print(x, y)` might return the number 2.

```
x = 7
if x < 5 & (print(x) = 1)
    print("x is", x)
```

9.1.2.

```
if theArray has an element with index 10 & theArray[10] = 0
    print("OK")
```

9.1.3. In this problem, ++x behaves as both an instruction and an expression. As an instruction, ++x adds 1 to the value of x. As an expression, the value of ++x is the newly obtained value of x. For example, the code

```
x = 7
print(++x)
print(x)
```

displays the numbers 8 8. With that in mind, evaluate the following code both lazily and eagerly.

```
x = 18
if ++x > 19 & print(++x)
    print("x is", x)
print(x)
```

9.1.4. In this problem, x++ is the same as in Problem 9.1.3.

```
x = 18
if ++x > 18 or print(++x)
    print("x is",x)
print(x)
```

9.1.5. x = (if y notEqualTo 0 then (x / y) else x + 1)
9.1.6. firstElementOf([0, 3. 6. 9, 12])
9.1.7. firstElementOf([0, 3, 6, 9, 12, ...])
9.1.8. sumOfTheNumbersIn([0, 3, 6, 9, 12, ...])

10. Introduction to Higher-Order Functions

General

10.1. Let f(x) = x + 7, let g(x) = x^2 and let h(x) = 1/x.

10.1.1. Find the value of f∘g(5).
10.1.2. Write an arithmetic expression for the function f∘g.
10.1.3. Find the value of g∘f(5).
10.1.4. Write an arithmetic expression for the function g∘f.
10.1.5. Find the value of h∘h(5).
10.1.6. Find the value of g∘f∘h(5).
10.1.7. Write an arithmetic expression for the function g∘f∘h.

10.2. In the video, I describe function composition, denoted by the ∘ symbol. Is composition a higher-order function? Why, or why not?

10.3. If you've taken calculus, you've seen the derivative, denoted by *d/dx*. Explain why the derivative is a higher-order function.

10.4. Symbolic logic has two functions named thereExists and forAll. Here are examples of the use of these two functions:

thereExists(even, [1, 3, 5]) is false
because there are no even numbers in the list [1, 3, 5]
thereExists(even, [1, 2, 5]) is true
because there's an even number in the list [1, 2, 5]
forAll(even, [1, 2, 5]) is false
because not all numbers in the list [1, 2, 5] are even
forAll(even, [2, 6, 10]) is true
because all numbers in the list [1, 2, 5] are even

Are thereExists and forAll higher-order functions? Why or why not?

11. Currying

General

11.1. In the video, I use the notation

```
filter : function, list → list
```

to describe the parameters and result time of the `filter` function. Use similar notation to describe each of the following functions:

11.1.1. Describe the function f in Exercise 10.1.

11.1.2. Describe the function f∘g in Exercise 10.1.1

11.1.3. Describe the result of partially applying 2 to the first argument in the function `add(x, y) = x + y`.

11.1.4. Describe composition (denoted by the ∘ symbol).

11.1.5. Describe the `applyNTimes` function which applies a function n times. For example,

```
applyNTimes(x + 1, 3) = ((x + 1) + 1) + 1
applyNTimes(x * 2, 4) = (((x * 2) * 2) * 2) * 2
```

11.1.6. Describe the `addF` function, which takes two functions, `f` and `g`, and returns another function that sums up the return values from `f` and `g`. For example,

```
addF(x^2, 2*x) = x^2 + 2*x
```

11.1.7. In the video, you start with `add` and, from it, you create `curryAdd`. What are you doing when you go from `add` to `curryAdd`?

12. Closures

Pseudocode

12.1. What's the output of the following code?

```
makeNewFunction(factor) {
  size = 1
  f() = multiply size by factor and return the new size value
  return f
}

increaseA = makeNewFunction(5)
print( increaseA() )
print( increaseA() )
increaseB = makeNewFunction(10)
print( increaseB() )
print( increaseB() )
print( increaseA() )
```

12.2. What's the output of the following code?

```
createGreeting(interjection) {
   f(name) = interjection"," name"!"
   return f
}

formalGreeting = createGreeting("Hello")
casualGreeting = createGreeting("Hi")

print ( formalGreeting("Mr. Williams") )
print ( formalGreeting("Ms. Polywether") )
print ( casualGreeting("Joe") )
```

12.3. What's the output of the following code?

```
delayDisplay(n) {
    f(message) = wait n seconds and then display message
}
g = delayDisplay(1)
h = delayDisplay(5)
h("Goodbye")
g("Hello")
```

13. Introduction to Lists
Pseudocode

13.1. Find `head(tail([6,9,12,8]))`
13.2. Find `tail(tail(tail([6,9,12,8])))`
13.3. Find `construct(10,tail([21,15]))`
13.4. True or false? `tail([18,8])` is equal to `8`
13.5. Find `concatenate(concatenate([1,3],[1,8]),[0,0])`
13.6. With `x = [3,2,19]`, find `construct(head(x),tail(x))`

14. Recursion
and
15. More Recursion Examples
Pseudocode

15.1. Show how to use the `reverse` function from the video to find out if a list of characters is a palindrome.
15.2. Use the `reverse` function from the video to define a `last` function. When applied to a list, the `last` function returns the last value in the list.

```
last([1,7,5]) = 5
last([2]) = 2
```

15.3. Use the `last` function in Exercise 15.2 to write recursive code for a function that creates a list of factorials up to and including the number given to it. For example,

```
factorials[1] = 1
factorials[2] = [1,2]
factorials[3] = [1,2,6]
factorials[4] = [1,2,6,24]
```

15.4. Write recursive code for a function that sums the numbers in a list. For example,

```
sum([1,3,5])=9
sum([]) = 0
```

15.5. A function named `firstNFrom` takes a list and an integer `n`, and returns the first `n` values from the list. For example,

```
firstNFrom([9, 6, 3, 4], 2) = [9, 6]
firstNFrom([9, 6, 3, 4], 1) = [9]
```

Write recursive code for the `firstNFrom` function. (Assume that n is always greater than 0 and less than or equal to the length of the list.)

15.6. A function named `alternates` takes a list and returns a list containing the alternate values from the original list. For example,

```
alternates([2,19,81,4]) = [2,81]
alternates([1,2,7,5,9]) = [1,7,9]
alternates([8]) = [8]
alternates([]) = []
```

Write recursive code for the `alternates` function.

16. Computations that Might Fail

Pseudocode

16.1. Find the value of `sqrtMaybe(x-10) >>= minus4Maybe >>= reciprocalMaybe >>= plus13Maybe` when x = 10.

16.2. Find the value of `sqrtMaybe(x-1) >>= minus4Maybe >>= reciprocalMaybe >>= plus13Maybe` when x = 17.

16.3. Find the value of `sqrtMaybe(x-10) >>= minus4Maybe >>= reciprocalMaybe >>= plus13Maybe` when x = 9.

16.4. Find the value of `reciprocalMaybe(x) >>= sqrtMaybe` when x = 1.

16.5. Find the value of `reciprocalMaybe(x) >>= sqrtMaybe` when x = 0.

16.6. The `defintionMaybe` function takes a string of characters (such as a word) and returns the dictionary definition of that string of characters, if there is one. The `lengthMaybe` function takes a string of characters (such as a word or group of words) and returns the number of characters in the string, if there is one. (Count the blank spaces and punctuation in the string.)

16.6.1. The definition of *house* is *A building whose purpose is to regularly shelter the same person or people*. Find the value of `definitionMaybe(x) >>= lengthMaybe` when x = `"house"`.

16.6.2. The non-word *bxbutw* isn't a real word so it doesn't have a definition. Find the value of `definitionMaybe(x) >>= lengthMaybe` when x = `"bxbutw"`.

16.7. Describe the parameter types, return type, and rule for applying the `bind` function in Exercise 16.6.

17. More Monads

Pseudocode

17.1. Using the functions defined in the video, define a function whose parameter is a person and whose result is a list of the person's great aunts and uncles.

17.2. Ann's small business has 3 employees. A particular employee may or may not have clients. Describe the functions for finding all the clients in Ann's business.

17.3. It's Parents' Appreciation Day for the employees of Ann's small business. (See Exercise 17.2.) Define a function to create a list of parents to invite to the Appreciation Day party. (Don't bother inviting Ann's parents. They're no fun!)

17.4. If you try to perform a computation, and the computation fails, the word `Nothing` in the output with no other explanation might be a bit frustrating. To fix this, imagine a new kind of monad that I call the `MaybeAFloatWithMessage` monad. Like the `Maybe` monad from the video, the `MaybeAFloatWithMessage` monad has one of two things in it:

- If the computation succeeds, the monad contains `Just` a value.
- If the computation fails, the monad contains `ErrorBecauseOf` value.

For example, if you try to divide by 2, you get `Just 1/2`. But if you try to divide by 0, you get `ErrorBecauseOf 0`. The `0` in the result might not be very informative, but it's probably better than nothing.

Describe the details of the `MaybeAFloatWithMessage` monad.

Some Solutions

1. **About This Course**
   1.1. The output is 5 10 50 58 57 65 325 324 332 331

2. **Solving a Problem Both Ways**
   2.1.

```
for each purchase in purchasesList
    if purchase.category == Food
        if purchase.amount >= 10
            print purchase
```

   2.2.

```
hasCategoryFood(purchase)  =  purchase.category == Food
tenOrMore(purchase)        =  purchase.amount >= 10
print(filter(tenOrMore,filter(hasCategoryFood,purchasesList)))
```

3. **Using Filter, Map, and Fold**
   3.1. Rewrite the following function definitions using lambda notation:
      3.1.1. λ x $\rightarrow$ x + 1
      3.1.2. λ x $\rightarrow$ x
      3.1.3. λ customer $\rightarrow$ customer.name
   3.2. Evaluate the following lambda expressions:
      3.2.1. (λ x $\rightarrow$ 6 * x) (21) = 6 * 21 = 126
      3.2.2. (λ x $\rightarrow$ x / 2) ((λ x $\rightarrow$ x + 7) (19)) = (λ x $\rightarrow$ x / 2) (19 + 7) = (λ x $\rightarrow$ x / 2) (26) = 13
   3.3. Evaluate the following expressions
      3.3.1. map(timesTwo, [2, 4, 5]) = [4, 8, 10]
      3.3.2. map(timesTwo, [8]) = [16]
      3.3.3. map(timesTwo, []) = []
      3.3.4. map(addOne, map(timesTwo, [2, 2, 4, –3])) =
      map(addOne, [4, 4, 8, –6]) =
      [5, 5, 9, –5]
      3.3.5. map(timesTwo, map(addOne, [2, 2, 4, –3]))
      map(timesTwo, [3, 3, 5, –2]) =
      [6, 6, 10, –4]
      3.3.6. foldFromLeft(plus, 7, [3, –8 9]) = 7 + 3 + (–8) + 9 = 11
      3.3.7. foldFromLeft(minus, 7, [3, –8, 9]) = ((7 – 3) – (–8)) – 9 = 3
      3.3.8. foldFromRight(minus, 7, [3, –8, 9]) = 3 – ((–8) – (9 – 7)) = 13
      3.3.9. foldFromLeft(minus, 7, map(timesTwo, [3, 0, 8])) =
      foldFromLeft(minus, 7, [6, 0, 16]) = ((7 – 6) – 0) – 16 = –15
   3.4.

```
smallestNegativeBalance = -1000
for each customer in customersList
    if customer.balance < 0
        if customer.balance > smallestNegativeBalance
```

```
            smallestNegativeBalance = customer.balance
print smallestNegativeBalance
```

3.5.

```
getBalance(customer) = customer.balance
isNegative(number) = number < 0
balancesList = map(getBalance, customersList)
negBalancesList = filter(isNegative, balancesList)
smallestNegBal = foldFromLeft(max, -1000, negBalancesList)
print(smallestNegBal)
```

Without naming so many intermediate functions:

```
getBalance(customer) = customer.balance
isNegative(number) = number < 0
print(foldFromLeft(max, -1000,
            filter(isNegative, map(getBalance, customersList))))
```

Without naming any intermediate functions:

```
print(foldFromLeft(max, -1000,
   filter(λ number -> number < 0,
          map(λ customer -> customer.balance , customersList))))
```

5. Pure Functions

5.1.

5.1.1. This function is pure because it doesn't use any value other than its parameter x, and it doesn't modify any value(s) declared outside of itself.

5.1.2. The purity or impurity of this function depends on the way the programming language handles parameters. In some languages, modifying a parameter's value has no effect on the parameter in the calling code. In such a language, the following code would display the value 10 and the function would be pure.

```
x = 10
y = f(x)
print x
```

In other languages, modifying a parameter's value changes the parameter in the calling code. In such a language, the same code would display the value 17 and the function would be impure.

5.1.3. This function is impure because it uses a number that it obtains from the system clock, and the system clock isn't internal to the function.

5.1.4. This function is pure. It doesn't use any information that comes from outside of the function. It defines an additional variable y but that variable is fully contained inside the function. This function is equivalent to the x + 3 function.

5.1.5. This function is impure. The function's return result doesn't depend on the value obtained by calling `random()` so in that sense, the function doesn't really use an outside value. But a subsequent call to `random()` (after exiting a call to `f(x)`) will be different because one call to `random()` has been "used up" by the call to `f(x)`. So, in a subtle sense, this function changes something external to it. So this function is impure.

5.1.6. This function is pure. It doesn't change anything outside of itself, and it's return result depends only on the input parameter (`the_string_s`).

5.1.7. This function is impure. Its execution changes whatever website the URL points to.

5.2.

5.2.1. This expression is referentially opaque (the opposite of referentially transparent). If you call `inputFromKeyboard(x)` twice, the value of `inputFromKeyboard(x)` might be 7 the first time and 123897 the second time.

5.2.2. This expression is referentially transparent. The expression `7 + 6` means 13, no matter where it appears in a program.

5.2.3. This expression is referentially opaque (the opposite of referentially transparent). If you call `f(x)` twice, the value of `f(x)` might be 7 the first time and 31 the second time.

5.3. Here's a copy of one of my solutions to Problem 3.2:

```
getBalance(customer) = customer.balance
isNegative(number) = number < 0
balancesList = map(getBalance, customersList)
negBalancesList = filter(isNegative, balancesList)
smallestNegBal = foldFromLeft(max, -1000, negBalancesList)
print(smallestNegBal)
```

In this solution, notice that none of the program's variables (`balancesList`, `negBalancesList`, `smallestNegBal`) have values that vary. You don't have to think of the expression

```
balancesList = map(getBalance, customersList)
```

as assigning a value to `balancesList`. Instead, you can think of it as the definition of `balancesList`.

6. Some Benefits of Pure Functions

6.1. Testing functions:

6.1.1. This function is pure so the test requires no setup.

```
if mortgage(100000.00, 5.25, 30) = 552.20
    return "Passed"
```

```
        else
            return "Failed"
```

6.1.2. This function isn't pure so, before the test, you have to set up the value of `currentRatePercent`.

```
         currentRatePercent = 5.25
         if mortgage(100000.00, 30) = 552.20
             return "Passed"
         else
             return "Failed"
```

In this example, the setup involves only one statement. The setup for a function in a real-life application typically involves many more statements.

6.2. The function in Exercise 6.1.1 is pure, so it always yields the same result when it runs with parameters `100000.00`, `5.25`, `30`. (I'm ignoring things like differences in the way computers perform arithmetic, or people tripping over power cords while the function executes.) So if the function passes your test with parameters `100000.00`, `5.25`, `30`, the program is guaranteed to run correctly with those parameters.

6.3. The function in Exercise 6.1.2 isn't pure so it doesn't always yeild the same result when it runs with parameters `100000.00`, `30`. The function might be correct when the `currentRatePercent` is 5.25, but not when the `currentRatePercent` is 6.00.

6.4.

```
    largestKnownFactorial = 1
    for n from 1 to 100 do
        knownFactorials[n] = 1
    loop
        input n
        count = 0
        if n > largestKnownFactorial
            for i from largestKnownFactorial + 1 to n
                knownFactorials[i] = knownFactorials[i - 1] * i
                count = count + 1
            largestKnownFactorial = n
        print knownFactorials[n]
        print count
```

Here's a Java program to implement the pseudocode:

```
import java.math.BigInteger;
import java.util.Scanner;

public class Main {
  Scanner keyboard = new Scanner(System.in);
```

```
    public static void main(String[] args) {
      new Main();
    }

    Main() {
      int n;
      BigInteger[] knownFactorials = new BigInteger[101];
      int largestKnownFactorial = 1;
      for (n = 1; n <= 100; n++) {
        knownFactorials[n] = new BigInteger("1");
      }

      for (;;) {
        System.out.print("n: ");
        n = keyboard.nextInt();
        BigInteger result = new BigInteger("1");
        int count = 0;
        for (int i = 2; i <= n; i++) {
          result = result.multiply(
              new BigInteger(Integer.toString(i)));
          count++;
        }
        System.out.println("Without memoization:::Result: "
            + result + " Count: " + count);

        count = 0;
        if (n > largestKnownFactorial) {
          for (int i = largestKnownFactorial
              + 1; i <= n; i++) {
            knownFactorials[i] = knownFactorials[i - 1]
                .multiply(
                    new BigInteger(Integer.toString(i)));
            count++;
          }
          largestKnownFactorial = n;
        }
        System.out.println("With memoization:::Result: "
            + knownFactorials[n] + " Count: " + count);
      }
    }
  }
```

7. Avoiding Race Conditions

7.1. Bob is a victim of a race condition. While Bob was preparing to request $100, Carol was withdrawing money from the same account at a different ATM machine. By the time Bob completed his request, there wasn't enough money in the account to cover Bob's $100 withdrawal.

7.2. Divide the Food purchases into two threads (perhaps two cores on a multi-core processor). Thread A totals up half of the Food purchases while Thread B totals up the other half. The grand total is maintained in a central location that's updated by both Thread A and Thread B. In the end, the grand total is incorrect.
In the functional version of the problem, there's no `total` variable, only a `sum` or `fold` expression. So the code doesn't lend itself to the updating of mutable variables.

7.3. The variable `x` is mutable and there are two simultaneous threads (call them Thread A and Thread B). In a scenario that suffers from no race conditions, Thread A executes all of its statements and then Thread B executes all of its statements. In this scenario, the final value of `x` is 6.
In a scenario that suffers the most from race conditions, the following happens:

Thread A gets the value of `x`, which is 0.
Thread B gets the value of `x`, which is still 0.
Thread A adds 1 to 0 and assigns 1 to `x`.
Thread B adds 1 to 0 and assigns 1 to `x`.
Thread A gets the value of `x`, which is 1.
Thread B gets the value of `x`, which is still 1.
Thread A adds 1 to 1 and assigns 2 to `x`.
Thread B adds 1 to 1 and assigns 2 to `x`.
Thread A gets the value of `x`, which is 2.
Thread B gets the value of `x`, which is still 2.
Thread A adds 1 to 2 and assigns 3 to `x`.
Thread B adds 1 to 2 and assigns 3 to `x`.
The program prints 3.

8. Efficient Parameter Passing

8.1. Consider the following pseudocode:

```
f(2)
print 2
print 2 + 2

f(x) {
  x = x + 1
}
```

In an old version of FORTRAN, the output of this code (written in FORTRAN syntax) would be 3 6, not 2 4.

9. Lazy Evaluation

9.1.1. With lazy evaluation, the code doesn't execute `(print(x) = 1)` because, with x not less than 5, the if condition cannot possibly be true. Because the entire `if` condition is false, so the code doesn't print anything.
With eager evaluation, the code evaluates both `x < 5` and `(print(x) = 1)`. So the code prints the value of x (which is 7). Because the test still makes the `if` statement's condition false, the code doesn't display `x is 7`.
Note: In an extreme, counterproductive version of eager evaluation, the code would evaluate the `print("x is", x)` inside the body of the `if` statement even though the `if` statement's condition is false, and thus display `x is 7`.

9.1.2. Assume that `theArray` has only 4 elements, `theArray[0]`, `theArray[1]`, `theArray[2]`, and `theArray[3]`.
With lazy evaluation, the code never performs the test `theArray[10] = 0` and the call to `print` isn't executed.
With eager evaluation, the code performs the test `theArray[10] = 0` even though the outcome of that test has no effect on the value of the `if` statement's condition. In some languages, the request for the value `theArray[10]` overruns the space allocated to the array (which isn't good). In other languages, the request for `theArray[10]` generates an error.

9.1.3. The expression `++x > 19` adds 1 to x, making the value of x be 19. So that `++x > 19` expression is false.
With lazy evaluation, the code doesn't bother to execute `print(++x)`. And because it's inside the body of the `if` statement, the code doesn't execute `print("x is", x)`. So the only printing the code does is the final `print(x)`, and the entire output is the number 19.
With eager evaluation, the code executes `print(++x)`. Execution of that function call outputs the value `20`. Then the execution of the final print(x) outputs `20` a second time.

9.1.4. The evaluation of the expression `++x > 18` sets the value of x to 19. So `++x > 18` is true. That's enough to make the entire `if` condition be true no matter what comes after the word `or`. So there's no need to evaluate `print(++x)`.
With lazy evaluation, the code doesn't evaluate `print(++x)`, so the call `print("x is",x)` displays `x is 19`, and the final `print(x)` call displays 19 again. So, overall, the output of the code is `x is 19 19`.
With eager evaluation, the code evaluates `print(++x)` even though that evaluation doesn't change the value of the entire `if` statement condition. Evaluation of `print(++x)` changes the value of x to 20. and prints `20`. Then the code executes the other two `print` statements. So, overall the output of the code is `20 x is 20 20`.

9.1.5. Assume that y is equal to 0.
With lazy evaluation, the code ignores the `then (x /y)` part and goes straight to the `else x` part. So the overall effect is like executing `x = x + 1`.
With eager evaluation, the code doesn't ignore the `then (x /y)` part and divides x by 0, which isn't a good thing to do. In some languages, this generates an arithmetic error and the program crashes.

9.1.6. With both lazy and eager evaluation, the value of this expression is 0.

9.1.7. With lazy evaluation, the code figures out what `firstElementOf` means and looks for only the first element of `[0, 3, 6, 9, 12, ... ]` which is 0.
With eager evaluation, the code tries to find all elements of `[0, 3, 6, 9, 12, ... ]` and that takes forever. So the code never gets to look for the first element of the list.

9.1.8. With both lazy and eager evaluation, the code has to find all numbers in the list `[0, 3, 6, 9, 12, ... ]`. That takes forever, so in both cases, the code never comes up with an answer.

10. Introduction to Higher-Order Functions

10.1.

10.1.1. $f \circ g(5) = f(g(5)) = f(25) = 32$

10.1.2. $f \circ g(x) = f(x^2) = x^2 + 7$

10.1.3. $g \circ f(5) = g(12) = 144$

10.1.4. $g \circ f(x) = g(x + 7) = (x + 7)^2$

10.1.5. $h \circ h(5) = h(1/5) = 1/(1/5) = 5$

10.1.6. $g \circ f \circ h(5) = g(f(1/5)) = g(1/5 + 7) = (1/5 + 7)^2$

10.1.7. $g \circ f \circ h(x) = g(f(h(x))) = g(f(1/x)) = g(1/x + 7) = (1/x + 7)^2$

10.2. Composition is a higher-order function because composition take two functions (as its parameters) and returns a third function as its result. For example, in Exercise 10.1.2, composition takes the functions x^2 and $x + 7$ and creates the function $x^2 + 7$.

10.3. The derivative takes a function as its parameter and returns another function as its result. For example, *d*/*dx* takes the function x^2 and returns the function $2x$ as its result.

10.4. The function thereExists takes, as its parameters a function (such as even) and a list (such as [1, 3, 5]). The first parameter, even, is a function because even take a number (such as 1) and returns true or false, depending on whether the number is even or not. Therefore, thereExists is a higher-order function.
Similarly, forAll is a higher-order function.

11. Currying

11.1.1. f : number -> number

11.1.2. f∘g : number -> number

11.1.3. The result is a new function add2, with the formula add2(y) = y + 2.
add2: number -> number

11.1.4. When you compose one function with another function, you get yet another function.
∘ : function, function -> function

11.1.5. The parameter list for `applyNTimes` is a function (such as `x + 1`) and a number (such as 3). The result is a function (represented by an expression such as `((x + 1) + 1) + 1`).

```
applyNTimes : function, number -> function
```

11.1.6. The addF function takes two functions as its parameters and returns yet another function as its result.

```
addF : function, function -> function
```

11.1.7. I didn't focus on this point in the video, but when you go from add to `curryAdd`, you're applying a function that you can call the `curry` function. In this example, the `curry` function takes the add function as its argument and returns the curryAdd function as its result.

```
curry : function -> function
```

12. Closures

12.1. The output is

```
5
25
10
100
125
```

12.2. The output is

```
Hello, Mr. Williams!
Hello, Ms. Polywether!
Hi, Joe!
```

12.3. The output is

`Hello` (after waiting 1 second)

Goodbye (after waiting 4 more seconds)

13. Introduction to Lists

General

13.1. `head(tail([6,9,12,8])) = head([9,12,8]) = 9`

13.2.
```
tail(tail(tail([6,9,12,8]))) = tail(tail([9,12,8]))
    = tail([12,8]) = [8]
```

13.3. `construct(10,tail([21,15])) = construct(10,[8]) = [10,8]`

13.4. false because `tail([18,8])` is equal to `[8]` (the list whose only entry is the number 8) which isn't quite the same as the number 8.

13.5. `concatenate(concatenate([1,3],[1,8]),[0,0]) =`
concatenate([1,3,1,8], [0,0]) = [1,3,1,8,0,0]

13.6. With `x = [3,2,19]`, `construct(head(x),tail(x)) =`
`consttruct(3, [2,9]) = [3,2,9] = x`

14. Recursion

and

15. More Recursion Examples

15.1.

```
isAPalindrome(aList) = (aList equals reverse(aList))
```

15.2.

```
last(aList) = head(reverse(aList))
```

15.3.
```
factorials(1) = [1]
factorials(n) =
     concatenate(factorials(n-1),[last(factorials(n-1))*n])
```
15.4.
```
sum([]) = 0
sum(h::t) = h + sum(t)
```
15.5.
```
firstNFrom(h::t, 1) = [h]
firstNFrom(h::t, n) = construct(h, firstNFrom(t, n-1))
```
15.6.
```
alternates([]) = []
alternates(h::[]) = [h]
alternates(h::t) = h :: (alternates (tail t))
```

16. Computations that Might Fail

16.1.
```
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
```
16.2.
```
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
```
16.3.
```
sqrtMaybe(9-10) is Nothing
Binding Nothing with minus4Maybe yields Nothing
Binding Nothing with reciprocalMaybe yields Nothing
Binding Nothing with plus13Maybe yields Nothing
```
16.4.
```
reciprocalMaybe(1) is Just 1/1 which is Just 1
Binding Just 1 with sqrtMaybe yields Just 1
```
16.5.
```
reciprocalMaybe(0) is Nothing
Binding Nothing with sqrtMaybe yields Nothing
```
16.6.

16.6.1.
```
definitionMaybe("house") is
   Just "A building whose purpose is to regularly shelter the
   same person or people."
Binding Just "A building whose purpose is to regularly shelter
   the same person or people." to lengthMaybe yields Just 75.
```

16.6.2.

`definitionMaybe("bxbutw")` is `Nothing`
Binding `Nothing` to `lengthMaybe` yields `Nothing`.

16.7.

```
bind : Maybe a string, f -> Maybe an integer
```
with
```
f : string -> Maybe an integer
```

The rule for applying this exercise's `bind` is the same as the rule for the `bind` in the video: "If you have `Just y`, apply `f` to y getting `Just f(y)` or `Nothing`. If you have `Nothing`, get `Nothing`."

17. More Monads

17.1.

```
parentsOf >>= parentsOf >>= siblingsOf
```

17.2.

```
employeesOf : person -> list of people
clientsOf : person -> list of people
bind : list of people, f -> list of people
```

The rule for applying this exercise's `bind` is the same as the rule for the `bind` in the video: "Apply `f` to each person in the list, and then `flatten` the resulting list."

17.3.

```
employeesOf >>= parentsOf
```

17.4. Define two functions, `sqrtMaybeMess` and `reciprocalMaybeMess`.

`sqrtMaybeMess(4)` is `Just 2`
`sqrtMaybeMess(-4)` is `ErrorBecauseOf -4`
`reciprocalMaybeMess(5)` is `Just 1/5`
`reciprocalMaybeMess(0)` is `ErrorBecauseOf 0`

The rule for applying `bind` is as follows: "If you have `Just y`, apply `f` to y getting `Just f(y)` or `ErrorBecauseOf y`. If you have `ErrorBecauseOf y`, get `ErrorBecauseOf y`."

So, for example, in the expression `sqrtMaybeMess(x-7) >>= minus4MaybeMess >>= reciprocalMaybeMess >>= plus13MaybeMess` with x = 23,

`sqrtMaybeMess(23-7)` is `Just 4`
Binding `Just 4` with `minus4MaybeMess` yields `Just 0`
Binding `Just 0` with `reciprocalMaybeMess` yields `ErrorBecauseOf 0`
Binding `ErrorBecauseOf 0` with `plus13MaybeMess` yields `ErrorBecauseOf 0`

```
sqrtMaybeMess : float -> MaybeAFloatWithMessage
reciprocalMaybeMess : float -> MaybeAFloatWithMessage
bind : MaybeAFloatWithMessage, f -> MaybeAFloatWithMessage
```