Comp160
Lab 5
Spring 2018

For this lab you'll mainly be working with and extending a program from the book in order to practice the design recipe and good program design habits.

The Movie Theater Profit Program

In Section 2.3 of the book you are presented with a problem about exploring how changes to the ticket prices at a local movie theater affect the theater's profit. Go ahead and find the problem in the book and give it a read.

Now that you've read it, let's get to work.

1. The problem discusses five key pieces of information: the ticket price, the number of people that attend a movie, the amount by which the owner changed the ticket price, the fixed cost to show a movie, and the variable cost to show a movie. For the program given in the book, each one of these pieces of information is represented by a BSL number. Write **data definitions** for each of these pieces of information.

2. Not all BSL numbers have a reasonable interpretation as information in our movie theater problem. For example. BSL allows for complex and imaginary numbers but such numbers make no sense in the context of this problem. For each of your data definitions, add a comment on the kinds of BSL numbers that can and cannot be reasonably interpreted as the kind of information that is covered by the definition. Use examples to illustrate the distinction. At the top of the definition window, below your file header information, briefly comment on what *reasonably interpreted* means to you for this program. Are there numbers that don't fit your definition of reasonable but might fit someone else's?

Following the problem is a four function program that can be used to compute the profit for a given ticket price.

3. Copy the four function definitions into the definitions window. Press run to be sure that DrRacket reads the code as correct and that nothing is out of place.

4. Complete Exercise 27 and Exercise 30[1]. If a constant is the value

[1] Review section 2.4 if you're stuck or confused here.

corresponding to one of your data definitions, then place the constant definition beneath the data definition and name the constant accordingly. Otherwise, come up with a data definition corresponding to the constant and write it above the constant definition.

Notice that the function *profit* makes use of the functions *cost* and *revenue* which in turn make use of the function *attendees*. We call the function *profit* the main, or top-level, function as it solves the main problem we set out to solve. It is the function our theater owner would use to figure out how to maximize his profits. The functions *cost* and *revenue* are helper functions[2] for *profit* and *attendees* is a helper for *revenue* and *cost*.

[2] a.k.a. auxiliary functions

Section 3.3 of your text describes **domain knowledge**. It's a short section. Go read, or re-read, it.

Not let's see if we can't use the idea of domain knowledge to better understand the where and why of these helper functions and how it might help us to achieve a *one task per function* ideal.

5. This program draws on domain Knowledge from the theater owner and from the study of business in general. Below the header at the top of your file, make a list of all the specific pieces of domain knowledge you can find. For each piece of knowledge, comment on its role in the program and whether or not it's specific to this problem or comes from a more general domain, i.e. business. Pay particular attention to whether or not the knowledge lends itself to completing function definitions or to organizing code across different functions and splitting the program into helper functions for the top-level function.

Now turn your attention back to the design recipe. You've dealt with an initial set of data definitions and explored some issues with domain knowledge. It's now time to work on the functions themselves. Step five, **function definitions**, has clearly been done for us. We're going to go back and fill in steps two and three.

6. For each of the four functions, complete step two of the design recipe by writing a **signature** and **purpose statement**[3]. Place these things directly above the function definition. Include the name of the function in the signature as shown in class[4].

[3] We're skipping the **stub** part of step two. We'll practice those later.

[4] name: inputs → output

7. Exercise 28 asks you to determine the profit for five different ticket prices. Let's use these five cases to complete step three of the design process, **functional examples**. For each of the four functions, develop five examples, one for each of the five ticket prices. Use BSL *check-expect* expressions to formulate your examples as unit

tests. Place the tests between the purpose statement and the function definition as seen in Section 3.5 of the text. If you have problems writing the check-expect statements, then first write functional examples as comments like you see in Section 3.1. You are strongly encouraged to first work out the expected results on your own and only then check your work by running the relevant BSL function call in the interaction window[5].

[5] Tests check to the see that a function does what its supposed to do not what it was programmed to do. These two things are not always the same

8. Thus far we've ignored step four of the design recipe, , **funciton templates**. Add to your earlier comments about domain knowledge any observations you might have about ways in which domain knowledge might have led to a template for one or more of the functions. Templates provide a basic skeleton, or outline, for the definition.

9. If for some reason you have not run all of your unit tests by this point, do so now. If all of your tests passed, then this completes step six of the design recipe, **testing**. If some of your tests failed, then double check that you copied the code correctly and that you properly calculated your expected results for each test case.

10. Now that the program is complete and tested, let's put it to work. Complete Exercise 28 by using the program to determine which ticket price will maximize the owner's profit. Write up that price as a sixth example and test for all four of functions.

    Before moving on, let's look at Exercise 29.

11. You do not need to carry out the program modification described in Exercise 29. Instead, comment on which program will be easier to modify and explain your reasoning. Place these comments at the bottom of your definitions window.

Stubs

Below are a series of function signatures. Provide a **stub** for each of them. Once that is done write a single unit test for each using whatever values of the appropriate type you'd like. The tests are just to practice writing tests, function calls, and literals using the different types.

12. foo: Number Number → String

13. fee: Boolean Boolean → Number

14. fye: String → Image

15. fah: Image Number Number → Boolean