This fourth project will give you experience writing a program that involves branching (if). This project will use the same development process as in Programming Projects #2 & #3, and will be subject to the same rules and grading criteria, with some new criteria added. This specification is less detailed than for previous programming projects. **YOU ARE EXPECTED TO KNOW HOW TO PERFORM BASIC TASKS WITHOUT HAVING TO BE TOLD EXPLICITLY.**

You’ve just opened a Tex Mex restaurant.

Each meal your restaurant serves can consist of an entree, a side dish, a dessert, and a drink.

Your Tex Mex restaurant serves several entrees (enchilada, burrito, quesadilla), several side dishes (beans, rice, tortillas), several dessert (sopapillas, churro, tres leches cake), and several drinks (coffee, ice tea, soda).

A customer can order **AT MOST** one item from each category; for example, they can order an enchilada as their entree, beans as their side dish, sopapillas as their dessert, and coffee as their drink. But, they **AREN’T REQUIRED** to order one of each category; for example, they can order no entree, no side dish and no dessert at all, just a cup of coffee.

The prices are:

<table>
<thead>
<tr>
<th>Item</th>
<th>Price</th>
<th>Item</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enchilada</td>
<td>$12.75</td>
<td>Coffee</td>
<td>$4.50</td>
</tr>
<tr>
<td>Burrito</td>
<td>$10.50</td>
<td>Ice Tea</td>
<td>$2.75</td>
</tr>
<tr>
<td>Quesadilla</td>
<td>$8.25</td>
<td>Soda</td>
<td>$2.25</td>
</tr>
<tr>
<td>Beans</td>
<td>$2.00</td>
<td>Sopapillas</td>
<td>$5.00</td>
</tr>
<tr>
<td>Rice</td>
<td>$1.75</td>
<td>Churro</td>
<td>$5.75</td>
</tr>
<tr>
<td>Tortillas</td>
<td>$1.00</td>
<td>Tres Leches Cake</td>
<td>$6.50</td>
</tr>
</tbody>
</table>

Your community’s sales tax rate is 8.75%. A tip of 17.5% is automatically added to every bill.

**NOTE:** You are welcome to substitute other items, as long as you do so consistently and continue to follow the rules. For example, you could change burrito to chile relleno, enchilada to fajitas, etc, but the number and categories of items, and the prices, **MUST** remain the same.

**IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!**
To get full credit, you **MUST** use `if` blocks properly.
I. PROJECT DESCRIPTION

Write a program that simulates greeting a customer, taking their order, calculating the bill and printing the check. The computer will be the wait staff, and the user will be the customer.

EXTRA PREPROCESSOR DIRECTIVE

Your program MUST start with the following TWO preprocessor directives, in this order:
#include <stdio.h>
#include <stdlib.h>

II. STRUCTURE OF THE PROGRAM

The program body MUST be broken into four subsections:
1. Greeting subsection: Greet the customer (user).
2. Input subsection: Prompt for and input their order, one item at a time, IDIOTPROOFING EACH VALUE AS SOON AS IT IS INPUT.
3. Calculation subsection: Determine the price of each item, the subtotal, the tax amount, the tip amount, and the grand total.
4. Output subsection: Output their bill in receipt form.

Please note that you are ABSOLUTELY FORBIDDEN to have:
- ANY executable statements in your declaration section;
- ANY declarations in your execution section (body);
- ANY inputs or calculations in your greeting subsection;
- ANY calculations, or outputs other than prompts and idiotproofing error messages, in your input subsection;
- ANY inputs or outputs in your calculation subsection;
- ANY inputs or calculations in your output subsection.

That is, the subsections MUST BE COMPLETELY SEPARATE, and MUST BE CLEARLY LABELED WITH COMMENTS.

For this programming project, if blocks are not considered to be inputs, nor calculations, nor outputs; that is, in principle you may have an if block in ANY subsection of the program body. However, statements inside the clauses of an if block MUST follow the rules above.

III. IMPLEMENTATION ORDER

Because the program will be complicated, you are STRONGLY advised to IMPLEMENT ONE PART AT A TIME, thoroughly test and debug it, and then go on to the next part. Also, it would probably be best to implement the subsections, NOT in the order in which they appear in the program (as described above), but rather in the following order:

1. Greeting subsection.
2. Input subsection (developed one item at a time). NOTE: If you’re unclear on how to idiotproof, then you can skip the idiotproofing during initial implementation and then develop the idiotproofing code later.
3. Output subsection (developed one item at a time). Note that, at this stage, some or all of your outputs will be garbage, because you haven’t yet written the calculation subsection.
4. Calculation subsection (developed one item at a time), located between the input and output subsections.
IV. DETAILS OF THE PROGRAM STRUCTURE

A. Greet the Customer

Welcome the customer to the restaurant.

B. Input the Customer’s Order

1. **Ask (prompt) the customer for their entree item choice**, giving them a list of entree items to choose from. You can use integer-valued codes to represent the entree items, and you may choose any **REASONABLE** values for coding these items. So, when the program prompts them for their entree item choice, the **OUTPUT** might be something like:

   What entree item would you like?
   Please enter:
   0 for no entree
   1 for enchilada
   2 for burrito
   3 for quesadilla

2. **Input** the customer’s entree item choice.

3. **IDIOTPROOF** the entree item choice, to ensure that the value that the user has input is one of the values listed in the prompt. (See the Grading Criteria for details.)

4. **Ask (prompt) the customer for their side dish item choice**, giving them a list of side dish items to choose from. Again, you can use integer-valued codes to represent the side dish items, and you may choose any **REASONABLE** values for coding these side dish items. So, when the program prompts them for their side dish item, the **OUTPUT** might be something like:

   What side dish would you like?
   Please enter:
   0 for no side dish
   1 for beans
   2 for rice
   3 for tortillas

5. **Input** the customer’s side dish item choice.

6. **IDIOTPROOF** the side dish item choice, to ensure that the value that the user has input is one of the values listed in the prompt. (See the Grading Criteria for details.)
7. **Ask (prompt) the customer for their dessert item choice**, giving them a list of dessert item choices to choose from. Again, you can use integer-valued codes to represent the dessert item choices, and you may choose any **REASONABLE** values for coding these items. So, when the program prompts them for their dessert item choice, the **OUTPUT** might be something like:

What dessert item would you like?
Please enter:
   0 for no dessert
   1 for sopapillas
   2 for churro
   3 for tres leches cake

8. **Input** the customer’s dessert item choice.

9. **IDIOTPROOF** the dessert item choice, to ensure that the value that the user has input is one of the values listed in the prompt. (See the Grading Criteria for details.)

10. **Ask (prompt) the customer for their drink item choice.** Again, you can use integer-valued codes to represent the drink item choices, and you may choose any **REASONABLE** values for coding these items.

11. **Input** the customer’s drink item choice.

12. **IDIOTPROOF** the drink item choice, to ensure that the value that the user has input is one of the values listed in the prompt. (See the Grading Criteria for details.)

**NOTE:** YOU MUST FULLY IDIOTPROOF EVERY INPUT THAT NEEDS IDIOTPROOFING. YOU ARE RESPONSIBLE FOR DETERMINING ALL POSSIBLE FORMS OF IDIOCY. Idiotproofing error messages **MUST** be **HELPFUL** and sufficiently detailed that even an idiot could figure out **SPECIFICALLY** what they’ve done wrong.

**IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!**

**ADVICE:** Avoid using ambiguous names for variables and named constants. Specifically, in this project, **DON’T** use names such as drink or coffee. Instead, use names that **CLEARLY** state the **ROLE** of the variable or named constant, such as ```drink_item_code``` or ```coffee_price```.

**NOTE:** If the customer chooses to buy nothing — no entree, no side dish, no dessert, no drink — then thank them and **EXIT** the program with a return code of zero (using an appropriate named constant), rather than calculating and printing an empty bill.
C. Calculate the Bill

1. **Entree price:** This value isn’t calculated as such, but is obtained from the entree item choice. (See VI.B.1.)
2. **Side dish price:** This value isn’t calculated as such, but is obtained from the side dish item choice.
3. **Dessert price:** This value isn’t calculated as such, but is obtained from the dessert item choice.
4. **Drink price:** This value isn’t calculated as such, but is obtained from the drink item choice.
5. **Subtotal:** Calculate the subtotal of the food and drink prices.
6. **Tax amount:** Calculate the amount of tax on the subtotal. Both food and drink are taxed at the same tax rate (8.75%).
7. **Tip amount:** The restaurant automatically adds a tip of 17.5% of the total food and drink bill, excluding tax — that is, the tip is calculated based on the sum of the entree price, the side dish price, the dessert price and the drink price, NOT on the sum of the entree price, the side dish price, the dessert price, the drink price and the tax amount.
8. **Grand Total:** The grand total bill is the sum of the subtotal, the tax amount, and the tip amount.

D. Output the Bill

1. Present the customer with an itemized bill in receipt form, using the placeholder below. For example:

   `-----------------------------------------
   Enrique’s Tex Mex Bill
   Your order was:
   Enchilada: $12.75
   Beans: $ 2.00
   Sopapillas: $ 5.00
   Coffee: $ 4.50
   ----------------------------------
   Food Total: $17.25
   Tax: $ 1.42
   Tip: $ 3.11
   ----------------------------------
   Grand Total: $21.78
   Thank you for visiting Enrique’s Tex Mex!
   ` (Substitute the name of your restaurant at the top and bottom.)
2. For all of the outputs in the itemized list (from the entree through the grand total), use the `printf` placeholder `%5.2f`, like so:

   `printf(" Enchilada: $%5.2f\n", entree_price);`

   The conversion format in the placeholder tells the compiler that the `printf` statement will output some literal text, followed by a floating point number that takes up at least five spaces, two of which are to the right of the decimal point.
3. Item names **MUST** line up on the left side of the bill, dollar signs **MUST** line up, and the `printf` placeholder will cause prices to be flush to the right of the line. For the conversion format to work, **all dollar amounts MUST be float.**

4. List the bill entries in the order shown, using the name of each menu choice (table, page 1).

5. In some cases, the exact tax amount and/or tip amount will have more than two digits to the right of the decimal point. We will accept results within five cents of the exactly correct bill.

**V. RUNS**

In your script, run the program 6 times, using the following inputs, in the following order:

1. no entree, no side dish, no dessert, no drink
2. enchilada, beans, sopapillas, coffee
3. burrito, rice, churro, ice tea
4. quesadilla, tortillas, tres leches cake, soda
5. enchilada, no side dish, no dessert, ice tea
6. no entree, no side dish, no dessert, coffee

In addition, **RUN THE PROGRAM ONCE FOR EACH POSSIBLE CASE OF IDIOCY** that a user might exhibit; that is, you **MUST** have runs that **COMPLETELY TEST EACH AND EVERY IDIOTPROOF CHECK. YOU ARE RESPONSIBLE FOR DETERMINING ALL POSSIBLE FORMS OF IDIOCY.** In your script file, the idiotproof test runs **MUST** occur **AFTER** the runs listed above.

**ADVICE:** Calculate each (non-idiotproof) run’s result by hand, then compare your hand-calculated values to the output of the program, to determine whether the program is running correctly.

**VI. GRADING CRITERIA**

**A. SUBJECTIVE GRADING OF COMMENTS IN THE PROGRAM BODY**

In previous programming projects, one of the grading criteria for comments in the program body has been that **EVERY** statement in the program body had to be preceded by a clear, helpful explanatory comment.

- For PP#4 and beyond, you may choose to write fewer comments than this (though still in the format described in the PP#2 specification), in which case **YOU AGREE TO ACCEPT WITHOUT ARGUMENT** the graders’ **SUBJECTIVE** opinion on whether the amount and nature of your comments is sufficient.
- Alternatively, you may choose to continue to comply with the old criterion, preceding **EVERY** statement in the program body with a clear, helpful explanatory comment, in which case you are guaranteed to get full credit for this aspect of documentation in the program body (assuming that your comments comply with the original grading criteria for comments in the PP#2 specification).
B. NEW GRADING CRITERIA

1. Format of if statements, else if statements and else statements: 
   For each if statement, the if keyword **MUST** be followed by a blank space and then 
   the open parenthesis that begins the if condition. After the close parenthesis that ends the 
   if condition, there **MUST** be a blank space, followed by the block open, on the same line 
   as the last line of the if condition. 
   For each else if statement, the same. 
   For each else statement, there MUST be a single blank space between the else keyword 
   and the block open, which MUST be on the same line as the else keyword. For example: 

   ```c
   if (entree_item_code == no_item_code) {
       entree_price = no_item_price;
   } /* if (entree_item_code == no_item_code) */
   else if (entree_item_code == enchilada_item_code) {
       entree_price = enchilada_price;
   } /* if (entree_item_code == enchilada_item_code) */
   else if (entree_item_code == burrito_item_code) {
       entree_price = burrito_price;
   } /* if (entree_item_code == burrito_item_code) */
   else if (entree_item_code == quesadilla_item_code) {
       entree_price = quesadilla_price;
   } /* if (entree_item_code == quesadilla_item_code) */
   ```

2. Format of if conditions and else if conditions: 
   For each if statement and each else if statement, in the condition, any binary operators — including relational operators such as == and Boolean operators such as && — MUST be surrounded by one or more blank spaces on each side. Unary operators such as ! MUSTN'T be surrounded by blank spaces. For example: 

   ```c
   if ((dessert_item_code != no_item_code) && 
       (dessert_item_code != sopapillas_item_code) && 
       (dessert_item_code != churro_item_code) && 
       (dessert_item_code != tres_leches_cake_item_code)) {
       printf("ERROR: unknown dessert item code %d.\n",
              dessert_item_code);
       exit(program_failure_code);
   } /* if ((dessert_item_code != no_item_code) && ...) */
   ```
3. **Indenting OF if blocks:**
For a given if block, the if statement, all else if statements (if any), the else statement (if any) and the block closes associated with these statements **MUST** all line up, both with each other and with other statements as appropriate. For example:

```c
if (drink_item_code == coffee_item_code) {
    printf(" Coffee: $%5.2f\n",
            drink_price);
} /* if (drink_item_code == coffee_item_code) */
else if (drink_item_code == ice_tea_item_code) {
    printf(" Ice Tea: $%5.2f\n",
            drink_price);
} /* if (drink_item_code == ice_tea_item_code) */
else if (drink_item_code == soda_item_code) {
    printf(" Soda: $%5.2f\n",
            drink_price);
} /* if (drink_item_code == soda_item_code) */
printf(" ----------------------------------\n");
```

4. **Indenting INSIDE if blocks:**
For a given if block, all statements **INSIDE** any clause of the if block **MUST** be indented **FOUR SPACES** farther than the associated if statement. This applies especially to if blocks nested inside other if blocks. For an example, see above.

5. **Commenting if blocks:** You **MUST** follow all of the rules that are specified in the packet titled “Commenting if Blocks,” which will be posted on the course website soon.

6. **No Text on the Same Line After a Block Open:** On any line that has a block open, you are **ABSOLUTELY FORBIDDEN** to have **ANY** source code text **AFTER** the block open; that is, the block open **MUST** be the **FINAL** character on that line.

7. **Text on the Same Line After a Block Close:** On any line that has a block close, you are **ABSOLUTELY FORBIDDEN** to have **ANY** source code text **AFTER** the block close **EXCEPT** the comment on the same line that labels that block close (see grading criterion #5).
8. **Idiotproofing**

(a) **ALL** inputs **MUST** include idiotproofing checks. Here’s a typical idiotproofing check:

```c
if ((entree_item_code != no_item_code) &&
    (entree_item_code != enchilada_item_code) &&
    (entree_item_code != burrito_item_code) &&
    (entree_item_code != quesadilla_item_code)) {
    printf("ERROR: unknown entree item code \%d.\n",
            entree_item_code);
    exit(program_failure_code);
}
```

Notice the `exit` statement, which causes the program to immediately terminate. **ALL** idiotproofing checks **MUST** include an `exit` statement. Notice that the `exit` statement is **INSIDE** the `if` block, and should be indented appropriately (see above). Also, your program **MUST** have a named constant `program_failure_code` whose value is \-1.

(b) Each idiotproofing check **MUST** occur **IMMEDIATELY** after the associated `scanf` statement. **ALL IDIOTPROOFING MUST BE COMPLETED BEFORE ANY CALCULATIONS ARE PERFORMED;** that is, idiotproofing belongs in the input subsection.

(c) Idiotproof error messages **MUST CLEARLY AND UNAMBIGUOUSLY** state the nature of the error. Thus, no two error messages should be the same.

(d) You **DON'T** have to idiotproof the data type. For example, when inputting the entree item code, you **DON'T** have to check whether the user input something like 2.73. Idiotproofing the data type is far beyond the scope of material covered in this course. In fact, you’d probably have to be in at least the second programming course for CS majors before you’d have covered that kind of material.

**VII. WHAT TO SUBMIT**

Submit cover, summary essay, script, checklist (if any) and extra credit bonus form (if any) in the usual style, format and order, and upload your source file and script file to Canvas in the usual way.
EXTRA CREDIT

(A) HELP SESSION BONUS EXTRA CREDIT

You can receive an extra credit bonus of as much as 2.5% of the total value of PP#4 as follows:

1. Attend at least one regularly scheduled CS1313 help session for at least 30 minutes, through Wed March 27.

2. During the regularly scheduled help session that you attend, work on CS1313 assignments (ideally PP#4, but any CS1313 assignment is acceptable). **YOU CANNOT GET EXTRA CREDIT IF YOU DON’T WORK ON CS1313 ASSIGNMENTS DURING THE HELP SESSION.**

3. Before you leave the regularly scheduled help session, fill out **BOTH** halves of the form on the last page of this project specification and have the help session leader (instructor or TA) sign **BOTH** halves. **THE FORM CANNOT BE SIGNED UNTIL IT IS COMPLETELY FILLED OUT IN INK.** Use of pencil on these forms is **ABSOLUTELY FORBIDDEN.**

4. If you leave the help session without getting the form signed, you **CANNOT** get extra credit for attending that help session; your form **CANNOT** be signed later.

5. Attach the bottom half of the form to your PP#4 paper submission, at the **VERY END**, and keep the top half for your own records.

**VALUE OF THE HELP SESSION EXTRA CREDIT BONUS:**

- for attending a regularly scheduled help session Mon March 4 - Wed March 6: 2.5% of the total value of PP#4;
- for attending a regularly scheduled help session Mon March 11 - Wed March 27: 1.25% of the total value of PP#4.

(B) CHECKLIST BONUS EXTRA CREDIT

You can receive an extra credit bonus of up to 2.5% of the total value of Programming Project #4 by doing the following:

1. Print out the checklist on pages 11-14 of this PP#4 specification. You **MUST** provide the ENTIRE checklist, all 4 pages (preferably double sided on 2 sheets of paper).

2. Complete the checklist, by checking the checkbox for every item that you have performed as described.

3. Include the checklist in your paper submission, as described in this PP#4 specification, page 9, section VII.

**VALUE OF THE CHECKLIST EXTRA CREDIT BONUS:**

2.5% of the total value of PP#4

**NOTES:**
- For the checklist bonus extra credit, if you mark any of the items **INCORRECTLY** (that is, you said that you did do something that you actually didn’t do), then we reserve the right to decide to reduce or eliminate some or all of the checklist bonus extra credit for PP#4, at our sole discretion.
- You can only get each extra credit bonus **ONCE** per programming project that offers it.
- These extra credit bonus items **WON’T** be available on any other programming project unless explicitly stated so in that project’s specification.
PP#4 CHECKLIST (Print this checklist, check all that apply, and include this in your paper submission, for up to 2.5% BONUS EXTRA CREDIT.)

NAME ____________________________ LAB SECTION __________

☐ include directives: I used the correct include directives, in the correct order (as described in the PP#4 specification, page 2, section I).

☐ Named constants: I declared several named constants, for prices, item codes, tax rate and tip rate (as described in the PP#4 specification, page 1, and page 3, item IV.B.1).

☐ Named constant names and variable names: My named constant names and variable names are sufficiently specific that I can easily tell which are prices, which are item codes and so on, for example coffee_price and drink_item_code (as described in the PP#4 specification, page 4, section IV.B).

☐ Program structure: I used the correct program structure (as described in the PP#4 specification, page 2, section II).

☐ Implementation order: I implemented the subsections of my program’s execution section (body) in the correct order (as described in the PP#4 specification, page 1, and page 3, item IV.B.1).

☐ Greeting: I wrote an appropriate greeting (as described in the PP#4 specification, page 3, item IV.A).

☐ Entree prompt: I wrote an appropriate prompt for the entree item code (as described in the PP#4 specification, page 3, item IV.B.1).

☐ Entree input: I wrote an appropriate input for the entree item code (as described in the PP#4 specification, page 3, item IV.B.2).

☐ Entree idiotproof: I wrote an appropriate idiotproof for the entree item code (as described in the PP#4 specification, page 3, item IV.B.3, and the example on page 9, grading criterion 8a).

☐ Side dish prompt: I wrote an appropriate prompt for the side dish item code (as described in the PP#4 specification, page 3, item IV.B.4).

☐ Side dish input: I wrote an appropriate input for the side dish item code (as described in the PP#4 specification, page 3, item IV.B.5).

☐ Side dish idiotproof: I wrote an appropriate idiotproof for the side dish item code (as described in the PP#4 specification, page 3, item IV.B.6, and the example on page 9, grading criterion 8a).

☐ Dessert prompt: I wrote an appropriate prompt for the dessert item code (as described in the PP#4 specification, page 4, item IV.B.7).

☐ Dessert input: I wrote an appropriate input for the dessert item code (as described in the PP#4 specification, page 4, item IV.B.8).

☐ Dessert idiotproof: I wrote an appropriate idiotproof for the dessert item code (as described in the PP#4 specification, page 4, item IV.B.9, and the example on page 7, grading criterion 2).

☐ Drink prompt: I wrote an appropriate prompt for the drink item code (as described in the PP#4 specification, page 4, item IV.B.10).

☐ Drink input: I wrote an appropriate input for the drink item code (as described in the PP#4 specification, page 4, item IV.B.11).
Drink idiotproof: I wrote an appropriate idiotproof for the drink item code (as described in the PP#4 specification, page 4, item IV.B.12, and the example on page 7, grading criterion 2).

Test for buying nothing: I wrote appropriate code to test whether the user ordered nothing, in which case they would be thanked, and the program would exit without printing a bill (as described in the PP#4 specification, page 4, section IV.B).

Determine entree price: I wrote appropriate code to determine the entree price (as described in the PP#4 specification, page 5, item IV.C.1, and the example on page 7, grading criterion 1).

Determine side dish price: I wrote appropriate code to determine the side dish price (as described in the PP#4 specification, page 5, item IV.C.2, and the example on page 7, grading criterion 1).

Determine dessert price: I wrote appropriate code to determine the dessert price (as described in the PP#4 specification, page 5, item IV.C.3, and the example on page 7, grading criterion 1).

Determine drink price: I wrote appropriate code to determine the drink price (as described in the PP#4 specification, page 5, item IV.C.4, and the example on page 7, grading criterion 1).

Calculate subtotal: I wrote appropriate code to calculate the subtotal (as described in the PP#4 specification, page 5, item IV.C.5).

Calculate tax amount: I wrote appropriate code to calculate the tax amount (as described in the PP#4 specification, page 5, item IV.C.6).

Calculate tip amount: I wrote appropriate code to calculate the tip amount (as described in the PP#4 specification, page 5, item IV.C.7).

Calculate grand total: I wrote appropriate code to calculate the grand total (as described in the PP#4 specification, page 5, item IV.C.8).

Output bill header: I wrote appropriate code to output the header of the bill (as described in the PP#4 specification, page 5, item IV.D.1).

Output entree and its price: I wrote appropriate code to output the entree and its price (as described in the PP#4 specification, page 5, item IV.D.1, and the example on page 8, grading criterion 3).

Output side dish and its price: I wrote appropriate code to output the side dish and its price (as described in the PP#4 specification, page 5, item IV.D.1, and the example on page 8, grading criterion 3).

Output dessert and its price: I wrote appropriate code to output the dessert and its price (as described in the PP#4 specification, page 5, item IV.D.1, and the example on page 8, grading criterion 3).

Output drink and its price: I wrote appropriate code to output the drink and its price (as described in the PP#4 specification, page 5, item IV.D.1, and the example on page 8, grading criterion 3).

Output subtotal: I wrote appropriate code to output the subtotal (as described in the PP#4 specification, page 5, item IV.D.1).

Output tax amount: I wrote appropriate code to output the tax amount (as described in the PP#4 specification, page 5, item IV.D.1).
Output tip amount: I wrote appropriate code to output the tip amount (as described in the PP#4 specification, page 5, item IV.D.1).

Output grand total: I wrote appropriate code to output the grand total (as described in the PP#4 specification, page 5, item IV.D.1).

Output bill footer: I wrote appropriate code to output the footer of the bill (as described in the PP#4 specification, page 5, item IV.D.1).

Format of dollar figures in output: In my outputs of various dollar amounts, I used the appropriate conversion format in my placeholder (as described in the PP#4 specification, page 5, item IV.D.2).

Dollar figures in output line up: In my outputs of various dollar amounts, my dollar figures line up properly (as described in the PP#4 specification, page 6, item IV.D.3).

Order of outputs: In my outputs of various dollar amounts, I output them in the appropriate order (as described in the PP#4 specification, page 6, item IV.D.4).

Accuracy of outputs: In my outputs of various dollar amounts, my outputs are correct to within 5 cents (as described in the PP#4 specification, page 6, item IV.D.5).

Runs: In my script file, I did the correct runs in the correct order (as described in the PP#4 specification, page 6, section V).

Idiotproof runs: In my script file, I did all appropriate idiotproof runs, and my idiotproof runs all come after my regular runs (as described in the PP#4 specification, page 6, section V).

Calculation by hand: To verify that my program is correct, I calculated by hand the correct bill for each test run, and checked that against my runs (as described in the PP#4 specification, page 6, section V).

Format of if statements: In my program, my if statements have the correct format (as described in the PP#4 specification, page 7, grading criterion 1).

Format of if conditions and else if conditions: In my program, my if conditions and my else if conditions have the correct format (as described in the PP#4 specification, page 7, grading criterion 2).

Indenting OF if blocks: In my program, my if blocks are properly indented (as described in the PP#4 specification, page 8, grading criterion 3).

Indenting INSIDE if blocks: In my program, statements inside my if blocks are properly indented (as described in the PP#4 specification, page 8, grading criterion 4).

Commenting if blocks: In my program, the block closes of my if blocks are properly labeled with comments on the same line (as described in the document “Commenting if Blocks”).

No source code text on same line after a block open: In my program, my if statements have no source code text on the same line after the block open (as described in the PP#4 specification, page 8, grading criterion 6).

Only comment text on same line after a block close: In my program, my if block closes have no source code text on the same line after the block close except the comment that labels the block close (as described in the PP#4 specification, page 8, grading criterion 7).

Idiotproofing of all inputs: In my program, every input has an idiotproof (as described in the PP#4 specification, page 9, grading criterion 8a).

Idiotproofing exit statement: In my program, every idiotproof has an exit statement (as described in the PP#4 specification, page 9, grading criterion 8a).
Idiotproofing exit statement inside if block: In my program, every idiotproof’s exit statement is inside the idiotproof if block (as described in the PP#4 specification, page 9, grading criterion 8a).

Idiotproofing exit statement indented properly: In my program, every idiotproof’s exit statement is indented properly (as described in the PP#4 specification, page 9, grading criterion 8a).

Idiotproofing exit statement uses program_failure_code: In my program, every idiotproof’s exit statement takes the argument program_failure_code, which is initialized to the correct value (as described in the PP#4 specification, page 9, grading criterion 8a).

Idiotproofing immediately after input: In my program, every idiotproof occurs immediately after the associated input (as described in the PP#4 specification, page 9, grading criterion 8b).

Idiotproofing error messages: In my program, every idiotproof error message is unique and unambiguous (as described in the PP#4 specification, page 9, grading criterion 8c).

Paper submission: My paper submission has the correct printouts in the correct order (as described in the PP#4 specification, page 9, section VII).

Uploads: I’ve uploaded the correct files to the Canvas PP#4 dropbox (as described in the PP#4 specification, page 9, section VII).
CS1313 PROGRAMMING PROJECT #4 BONUS REQUEST FORM

Name ____________________________ Lab ____________
Help Session Date ________________
Help Session Time (Arrive) ___________ Help Session Time (Depart) ________________

Instructor Signature ____________________________

Keep this copy for your records.

Submit this copy.
In your submission, attach this copy at the VERY END.
If you put this in the wrong place in your submission, then you WON’T get the extra credit.