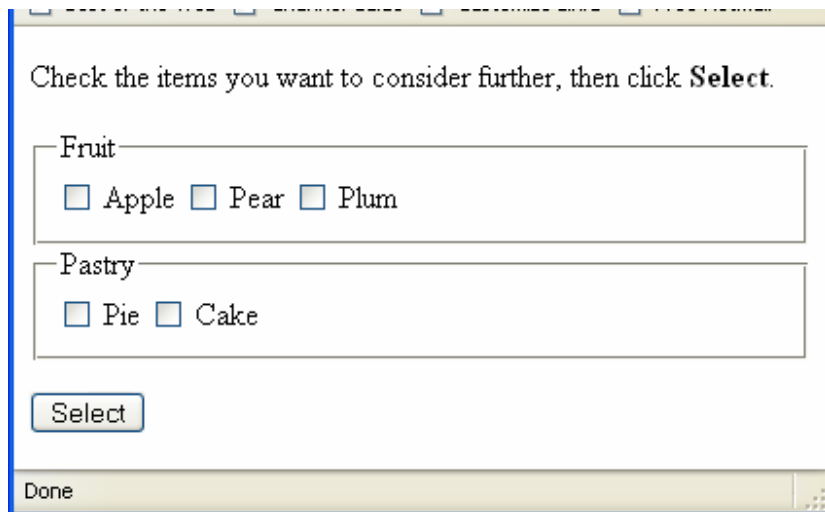


COMP 722 E-commerce Fall 2007 Programming Assignment 4

Due in the digital drop box by Wednesday, Oct. 17 at 11:00 PM.

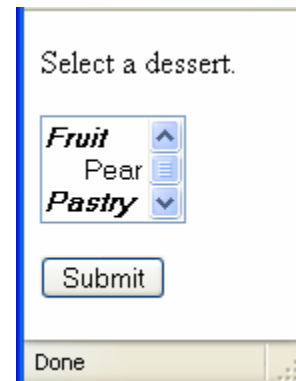
1. For this problem (which cannot be rendered with Internet Explorer), you fill in the gaps in the document listed below. When this document is loaded, it is rendered as shown at right. The entire body consists of a form. The first element in the form is the paragraph at the top of the screenshot. The other elements in the form are two fieldsets and a paragraph containing a button that, when clicked,



The screenshot shows a web browser window with a form. At the top, there is a paragraph: "Check the items you want to consider further, then click **Select**." Below this are two fieldsets. The first fieldset has a legend element "Fruit" and three checkboxes: "Apple", "Pear", and "Plum". The second fieldset has a legend element "Pastry" and two checkboxes: "Pie" and "Cake". Below the fieldsets is a button labeled "Select". At the bottom of the browser window, the status bar shows "Done".

invokes the function `go()`. The first fieldset has a legend element with content **Fruit**. The rest of fieldset consists of three checkboxes paired with label elements. For example, the first box can be checked not only by clicking the box itself but also by clicking the text **Apple**. All three checkboxes have `name="fruit"`. The first checkbox also has `value="apple"` and `id="ap"`, the second has `value="pear"` and `id="pe"`, and the third has `value="plum"` and `id="pl"`. The second fieldset has a legend element with content **Pastry** and has two checkboxes paired with label elements. Both checkboxes have `name="pastry"`. The first also has `value="pie"` and `id="pi"`, and the second has `value="cake"` and `id="ca"`.

Clicking the button invokes `go()`, which removes all the content of the form and adds new content as shown in the screenshot at right. There is now a paragraph, a select element, and submit button inside a second paragraph. The select element has `name="dessert"` and `size="3"`. The select element contains an `optgroup` element for each of the fieldsets containing at least one checked box. The value of the label attribute of an `optgroup` is the text that was the content of the legend element in the corresponding fieldset. The content of an `optgroup` element consists of an `option` element for each checked box in the corresponding fieldset. The value of an `option` element is the value of the corresponding checkbox, and its text content is the text content of the associated label element. The scrollable menu in the last screenshot is the rendering of a select element that results from checking the **Pear** and **Pie** checkboxes; the **Pie** option is not visible since the value of the `size` attribute of the select element is "3".



The screenshot shows a web browser window with a form. At the top, there is a paragraph: "Select a dessert." Below this is a scrollable select menu. The menu has two optgroups: "Fruit" and "Pastry". Under "Fruit", there are two visible options: "Pear" and "Plum". Under "Pastry", there is one visible option: "Pie". Below the select menu is a button labeled "Submit". At the bottom of the browser window, the status bar shows "Done".

When the submit button is clicked, function `echoValue()` is invoked since the form element has `onSubmit="echoValue()"`. This function simply raises an alert

box informing the user of the value selected in the menu. (Note that this page thus implements a two-phase selection process: one first checks options for further consideration then selects a unique option from the resulting restricted menu.) Submitting the form restores the original page since the form has no `action` or `method` attribute.

Most of the code in function `go()` is for constructing the `select` element. To do this, we first set variables `fm` to a reference to the form element and `fsets` to an array of references to the fieldset elements in the form. We then create a `select` element node, assigned to the variable `selNode`, and provide it with attributes `name` with value "dessert" and `size` with value "3". Next, a `for` loop runs `i` from 0 up to one less than the length of the `fsets` array, providing one iteration for each fieldset¹. In the body of the loop (for a fixed value of `i`), variable `chosen` is set to a new empty array, which will contain references to the clicked boxes (`input` elements), and variable `chosenText` is also set to a new empty array, which will contain the text nodes that are the content of the `label` elements corresponding to the clicked boxes. Note that `chosen` and `chosenText` are parallel arrays: the text node in `chosenText[j]` is from the `label` element associated with the `input` element in `chosen[j]`. We also initialize `choices` to an array of references to the `input` elements (checkboxes) within the fieldset referenced by `fsets[i]` and initialize `choicesText` to an array of references to the `label` element within that fieldset; `choices` and `choicesText` are parallel arrays.

Next (still within the loop iterating over the fieldsets), a `for` loop runs `j` from 0 up to one less than the length of `choices`. If `choices[j]` is checked, then we push it onto `chosen` and push the text node that is the sole child of `choicesText[j]` onto `chosenText`.

Then, if `chosen` is not empty (i.e., if at least one checkbox in the corresponding fieldset was checked), then we do the following. We initialize `groupName` to the text in the text node that is the sole child of the `legend` element in the fieldset referenced by `fsets[i]`. And we create an `optgroup` node (with variable `optGNode` storing a reference to it) and provide it with a `label` attribute with the value of `groupName` as its value. Next comes a `for` loop that constructs the `option` elements for the `optgroup` corresponding to the fieldset referenced by `fsets[i]`. This runs `k` from 0 up to one less than the length of `chosen`. For a given checked box `chosen[k]`, it creates an `option` element referenced by `optNode` and provides it with a `value` attribute whose value is the value of the checkbox. The code then makes the corresponding text node a child of `optNode` and makes `optNode` itself a child of the `optgroup` element. Once all the `option` children of the `optgroup` node referenced `optGNode` are in place, this node is made a child of the `select` node, `selNode`.

After all `optgroup` elements corresponding to the fieldsets have been added to the content of the `select` element, it remains to clear the original elements from the form

¹ Even though there are only two fieldsets, you should not unfold this loop into two sequences of code, one for each fieldset in the current implementation, since it will still work if we add or remove fieldsets, and it could be used in other documents with a similar structure.

and add the new elements to it. We first remove all the children of the form, `fm`; this is done by invoking function `removeChildren()`, described below, passing it `fm`. For the first paragraph shown in the second screenshot above, we create a `p` element, set its content to the text “Select a dessert.”, and make it a child of `fm`. We then append the `select` node, `selNode`, as the next child of `fm`. It remains to add to the form a paragraph containing the submit button. So we create a `p` node and assign it to the variable `pSubNode`. We then create an `input` node, provide it with attributes `type` (with value “submit”) and `value` (with value “Submit”), and make it a child of `pSubNode`. Finally, we make this `p` node a child of the form.

Function `removeChild()` (mentioned above for removing the child nodes of the form element) is passed a reference node to an element and removes all that element’s children. One is tempted to use a simple `for` loop that ranges a variable `i` from 0 to one less than the length of `node.childNodes` and, on each iteration, removes `node.childNodes[i]` from the children of `node`. The problem with this is that, when `node.childNodes[0]` is removed, what was `node.childNodes[1]` becomes `node.childNodes[0]` (and the following elements of `node.childNodes` are similarly promoted)². So, before the loop, we set `len` to the length of `node.childNodes` and use it as the upper bound in the loop. In the body of the loop, we remove `node.childNodes[0]` on each iteration.

The following is the document with gaps identified by Greek letters. After the listing, the Greek letters are repeated along with a description of what goes in the corresponding gaps. You can download the document with gaps from the same location where you downloaded this assignment.

```

<html>
<!-- This does not work with IE. -->
<head>
  <title>Problem 1</title>
  <script type="text/javascript">
    function go()
    {
      var fm = α_____ ;
      var fsets = β_____ ;
      var selNode = γ_____ ;
      δ_____ ;
      ε_____ ;
      for ( var i=0; i<fsets.length; i++ ) {
        var chosen = new Array(),
            chosenText = new Array(),
            choices = ζ_____ ,
            choicesText = η_____ ;
      }
    }
  </script>

```

Continued

² This is characteristic of what are called *live* arrays.

Continued from previous page

```

for ( var j=0; j<choices.length; j++ )
  if ( choices[j].checked ) {
    chosen.push( choices[j] );
    θ_____i;
  }

if ( chosen.length ) {
  var groupName =
    ι_____i;
  var optGNode = κ_____i;
  λ_____i;
  for ( var k=0; k<chosen.length; k++ ) {
    var optNode = μ_____i;
    ν_____i;
    ξ_____i;
    ο_____i;
  }
  π_____i;
}
}

removeChildren( fm );

var pNode = ρ_____i;
σ_____i;
τ_____i;
υ_____i;

var pSubNode = φ_____i;
var subNode = χ_____i;
ψ_____i;
ω_____i;
αα_____i;

fm.appendChild( pSubNode );
}

function echoValue()
{
  window.alert( "You have selected " + ββ_____i );
}

```

Continued

Continued from previous page

```

function removeChildren( node )
{
  γγ_____ ;

  _____
  _____ ;
}

</script>
</head>
<body>
<form onSubmit="echoValue()">
  <p>
    Check the items you want to consider further, then click
    <strong>Select</strong>.
  <p>
  <fieldset>
    <legend>Fruit</legend>

    <input type="checkbox" name="fruit" value="apple" id="ap">
      δδ_____

    <input type="checkbox" name="fruit" value="pear" id="pe">
      εε_____

    <input type="checkbox" name="fruit" value="plum" id="pl">
      ζζ_____
  </fieldset>

  <fieldset>
    <legend>Pastry</legend>

    <input type="checkbox" name="pastry" value="pie" id="pi">
      ηη_____

    <input type="checkbox" name="pastry" value="cake" id="ca">
      θθ_____
  </fieldset>

  <p><input type="button" value="Select" onclick="go()"></p>
</form>
</body>
</html>

```

Suggestion: Do not try to fill in all the gaps at once. Copy parts of the file you download to another file, and incrementally implement and test the other file. You can start with the entire body then add one function after another.

- α : A reference to the form
- β : An array of references to the `fieldset` elements
- γ : Create a `select` element.
- δ : Provide the `select` element with a `name` attribute with value “`dessert`”.
- ϵ : Provide the `select` element with a `size` attribute with value “`3`”.
- ζ : An array of the `input` elements within the `fieldset` reference by `fsets[i]`.
- η : An array of the `label` elements within the `fieldset` reference by `fsets[i]`.
- θ : Push the sole child (a text node) of `choicesText[j]` onto the array `chosenText`.
- ι : The text in the text node that is the sole child of the `legend` element in the `fieldset` referenced by `fsets[i]`
- κ : Create an `optgroup` element.
- λ : Provide the `optgroup` element with a `label` attribute whose value is the value of `groupName`.
- μ : Create an `option` element.
- ν : Provide the `option` element with a `value` attribute whose value is the value of `chosen[k]`.
- ξ : Make the text node `chosenText[k]` a child of the `option` element.
- \omicron : Make this `option` element the next child of the `optgroup` element.
- π : Make this `optgroup` element the next child of the `select` element.
- ρ : Create a `p` element.
- σ : Set the text content of this paragraph to “`Select a dessert.`”.
- τ : Make this paragraph the next element in the form.
- υ : Make the `select` element the next element in the form.
- ϕ : Create another paragraph.
- χ : Create an `input` element.
- ψ : Provide the `input` element with a `type` attribute with value “`submit`”.
- ω : Provide the `input` element with a `value` attribute with value “`Submit`”.
- $\alpha\alpha$: Make the `input` element a child of the paragraph.
- $\beta\beta$: The value selected for the `select` element.
- $\gamma\gamma$: The body of the `removeChildren()` function as described above.

δδ: The label associated with the apple checkbox.

εε: The label associated with the pear checkbox.

ζζ: The label associated with the plum checkbox.

ηη: The label associated with the pie checkbox.

θθ: The label associated with the cake checkbox.

2. For this problem (which cannot be rendered with Internet Explorer), you fill in the gaps in the document listed below. When this document is loaded, it is rendered as below (except there are 20, not 19, seconds remaining). The background color of the body is cyan, and that of the table with the prompts and text boxes is yellow. The area at the right with the four lines of text is a textarea. The values in the text boxes are the default values. From the time the page is loaded, the user has 20 seconds to fill in the text boxes with the intended values and to click the submit button, otherwise the form is automatically submitted (with the values currently in the text boxes, some of which the user may have got around to changing). The red number at the bottom of the pages provides a count-down. When it reaches zero, a buzzer goes off and an alert box pops up with the message **Out of time!** When the user dismisses the alert box, the form is submitted, and the data is validated to ensure it is in the correct format. The form data is checked in the top-to-bottom order in which the text boxes appear. If a value is found invalid, the submission is suppressed and an alert box identifying the error pops up. When this is dismissed, the offending text box gains focus, and the user is given 10 seconds to correct the value. Again the red number at the bottom provides a count-down, and the same alert box is raised if the user does not click submit in time; now, however, there is no buzzer. This cycle repeats until all values have the proper format. Since the name of a nonexistent file is used as the value of the `action` attribute of the form element, when the user succeeds in submitting the form, the browser shows a **File not found** message.

The screenshot shows a web browser window with a cyan background. A yellow rectangular area contains the following form elements:

- Prompt: "Your phone number in the format (999) 999-9999:" followed by a text box containing "(800) 334-7245".
- Prompt: "Your city (in North Carolina):" followed by a text box containing "Greensboro".
- Prompt: "The 3-letter abbreviation of your destination airport:" followed by a text box containing "GSO".

To the right of the yellow area is a white text area containing the following text:

```
Previous and current entries
Your phone number: (800) 334-7245
Your city: Greensboro
your destination airport: GSO
```

Below the yellow area are two buttons: "Reset" and "Submit". At the bottom of the cyan area, the text "Seconds remaining: 19" is displayed in red. The browser's status bar at the bottom shows "Done".

The default value for the textarea is the first line displayed in it above. When the page is loaded, a line of text is copied into the text area for each of the three text boxes. A line of text is made by extracting part of the prompt and concatenating it with the value in the corresponding text box (with " : " in between). Later, when a text box loses focus (which generally happens because the user has finished changing a value), a similar line

is copied into the textarea. This provides the user with a history of the values that have been entered for the various text boxes. (This is convenient if, for example, the user decides he had the correct value after all.)

Almost the entire body is a form element. The exception is an embed element (for the buzzer sound) at the bottom of the body:

```
<embed src="alarm1.wav" autostart=false width=0 height=0
      id="mySound" enablejavascript="true">
```

The embed element “displays” output from a file that is executed by a plug-in.

In this case, we make available the WAV file `alarm1.wav`, which is an audio recording of a buzzer. (WAV—or WAVE—is short for Waveform audio format. It is a Microsoft and IBM audio file format standard for storing audio on PCs and is the main format used on Windows systems for raw audio.) Since this embed element has `width=0` and `height=0`, it is invisible to the user. (Alternatively, one could use `hidden="true"`.) When the element is visible, it is rendered as a rectangular area with a play bar. Since we have `autostart=false`, the file will not automatically start playing when the document is loaded. (One can use `loop="true"` to have the sound repeat as long as the browser renders the document.) To be able to control the sound with script, we need to include `enablejavascript="true"`. If `embElt` is a script object corresponding to an embed element in the document body, then `embElt.Play()` starts up the file identified by the `src` attribute.

The first form element is a table, rendered at the top left of the above figure, with three rows and two columns. In each row, a text box is in the second column, and a description of what it contains in the first. The content of the first cell in each row, besides text, contains a span element enclosing what can pass as an abbreviation of the description. This is what the script uses in constructing the lines in the textarea.

To control the widths of the columns, we include in the table before the first row a `colgroup` element. The content of a `colgroup` element is one or more `col` elements, which, top-to-bottom, correspond to the columns taken left-to-right. (The `span` attribute can be used to have a `col` element cover more than one column.) Formatting attributes (such as `width`) in a `col` element apply to all the cells in the corresponding column. In our case, we want the first column to be 190 pixels wide and the second column to be 125 pixels wide.

The values of the name attributes of the text-box `input` elements are, from top to bottom, “phone”, “city”, and “airport”. The values of the `value` attributes are the defaults shown in the above screenshot. Each `input` element also has an intrinsic event attribute whose value is a call to the `copy()` function, passing a reference to the `input` element itself; the event in question is the one that fires when the element loses focus.

After this table element, the form contains the `textarea` element, specifying 12 rows and 13 columns (that is, a width of 13 character positions). Its content is the initial row of text. This element has an `id` attribute with value “echo”.

The `style` attribute in the table specifies that it floats left. With a browser window of a reasonable width, the `textarea` will flow up and to the right of the table. The reset and submit buttons, however, should be rendered below the table. So we put them in a `p` element with `style="clear: left"`.

The final element in the form is a `table` element with a single row having two cells. The first cell contains the text **Seconds remaining:**, and the second cell is empty but will contain the red number giving the count-down; the `id` of this cell is “secs”.

The JavaScript code consists solely of functions except for five global variables:

- `secsLeft` keeps track of how many seconds remain in the initial count-down; it is initialized to 21 but is decremented to 20 before it ever appears in the browser screen.
- `newSecsLeft` is used to re-initialize `secsLeft` when the user is asked to correct improperly formatted data; it is initialized to 11 but again is decremented before it is ever used.
- `firstTimeOut` is a Boolean variable initialized to true and set to false after the buzzer has been sounded once. The code is written to ensure that the buzzer sounds at most once. (This is a hack to prevent the error that we do not understand that occurs when we try to sound the buzzer more than once.)
- `fm` stores a reference to the form element.
- `timer` stores the control object for the repeated execution of the `run()` function.

Function `start()` is called when the page is loaded. It sets `fm` to a reference to the form element and adds the function `validate` as the listener for submit events. It then calls `copy()` three times, passing in succession references to the three input elements. This produces the first three lines of description in the textarea. Finally, `start()` initializes the control that causes `run()` to be called every second and assigns the control object to the variable `timer`.

Function `copy()` is passed a reference `elt` to an input element. Recall that an input element is in the second of two cells in a table row; the `span` element in the first cell contains the abbreviated description that we want to include in the line copied to the textarea. So `copy()` first sets `shortName` to this text. To get to this text from the input element, we need the parent of its parent (going up through `td` to `tr`), and then we find the only `span` descendant of this `tr`. The only child node of this `span` element is a text node; we set `shortName` to the text in this text node. Next, `copy()` sets `val` to the contents of the `value` attribute of the input element. Finally, it appends the following to the text in the textarea:

```
"\n" + shortName + ": " + val
```

Function `run()` first decrements the counter `secsLeft` and copies it as the content of the cell showing the count-down. When `secsLeft` reaches zero, it stops the repeated execution of `run()` (which requires access to the control object `timer`), and, if this is the first time the count-down has reached zero (see the variable `firstTimeOut`), it sets `firstTimeOut` to false and plays the buzzer file. In any case when `secsLeft` reaches zero, it raises the out-of-time alert box and clicks the submit button.

Function `validate()` must first also stop the repeated execution of `run()` (since it might be called because the user clicked the submit button, not because he ran out of time). This function checks the content of each of the text boxes in top-to-bottom order—

the structure is `if ... else if ... else if` For each text box, if the value for that box does not match a certain pattern (specified by a regular expression—note that the entire content of the text box must match the pattern), it does the following:

- Suppress the submission.
- Raise an alert box reminding the user of the proper format.
- Put the offending text box in focus.
- Record that we must start over with the count-down (but now with a shorter period).

Recording that we must start over is done by setting variable `restart` to true, this variable having been initialized to false. If `restart` is true after the validation, then the last thing `validate()` does is call `startOver()`.

Function `startOver()` re-initializes `secsLeft` to the value of `newSecsLeft`. It then again initializes the control that causes `run()` to be called every second and assigns the control object to the variable `timer`.

The following is the document with gaps identified by Greek letters. After the listing, the Greek letters are repeated along with a description of what goes in the corresponding gaps. You can download the document with gaps as well as the WAV file with the buzzer sound from the same location where you downloaded this assignment.

```
<html>
<head>
  <title>Problem 2</title>
  <style type="text/css">
    input.sub {
      background-color: #FFB0B0;
      width: 50;
    }
  </style>
  <script type="text/javascript">
    var secsLeft = 21, // Initial countdown starts at 1 less
        newSecsLeft =11, // Countdown to correct starts at 1 less
        firstTimeOut = true, // The alarm sounds just once
        fm, // A reference to the form element
        timer; // The timer for the countdowns

    function start()
    {
      α _____;
      β _____;
      γ _____;
      _____;
      _____;
      δ _____;
    }
  </script>
</html>
```

Continued

Continued from previous page

```

function run()
{
    ε_____ ;

    if ( ! secsLeft ) {
        ζ_____ ;

        // "Bad NPObjct as private data" error trying to play it 2X
        if ( firstTimeOut ) {
            firstTimeOut = false;
            η_____ ;
        }
        window.alert( "Out of time!" );
        θ_____ ;
    }
}

function validate( event )
{
    ξ_____ ;

    var restart = false;

    if ( ! ι_____ ) {
        κ_____ ;
        window.alert( "The phone number must have the format\n"
            + "(999) 999-9999" );
        λ_____ ;
        restart = true;
    }
    else if ( ! μ_____ ) {
        κ_____ ;
        window.alert( "This is not a valid city name" );
        ν_____ ;
        restart = true;
    }
    else if ( ! ξ_____ ) {
        κ_____ ;
        window.alert( "The airport abbreviation must consist of "
            + "exactly 3 uppercase letters." );
        ο_____ ;
        restart = true;
    }

    if ( restart ) startOver();
}

```

Continued

Continued from previous page

```

function startOver()
{
    secsLeft = newSecsLeft;
    δ_____ ;
}

function copyValue( elt )
{
    var π_____ = _____,
        shortName
        = _____,
        val = ρ_____ ;

    σ_____ "\n" + shortName + ": " + val;
}

</script>
</head>
<body style="background-color: cyan" onload="start()">
<form action="nonExistent.php" method="post">
  <table style="float: left; background-color: yellow">
    τ_____
    _____
    _____
    _____
  <tr height="50">
    <td>
      <span>Your phone number</span>
      in the format<br><tt>(999) 999-9999:</tt>
    </td>
    <td valign="bottom">
      <input type="text" name="phone" size="14" value="(800) 334-7245"
        υ_____ >
    </td>
  </tr>
  <tr height="50">
    <td><span>Your city</span> (in North Carolina):</td>
    <td><input type="text" name="city" size="15" value="Greensboro"
      υ_____ >
    </td>
  </tr>
</form>

```

Continued

Continued from previous page

```

<tr height="50">
  <td>
    The 3-letter abbreviation of
    <span>your destination airport</span>:
  </td>
  <td valign="bottom">
    <input type="text" name="airport" size="3" maxlength="3" value="GSO"
      <u>_____</u>
    </td>
</tr>
</table>

<p>
  <textarea rows="12" cols="33" name="echo">Previous and current entries</textarea>
</p>

<p style="clear: left">
  <input type="reset">
  <input type="submit" value="Submit" width="50" class="sub" id="sButton">
</p>

<table>
  <tr style="font-size: 16pt">
    <td>Seconds remaining:</td>
    <td id="secs" style="color: red"></td>
  </tr>
</table>
</form>

<embed src="alarm1.wav" autostart=false width=0 height=0
  id="mySound" enablejavascript="true">
</body>
</html>

```

Suggestion: Do not try to fill in all the gaps at once. Copy parts of the file you download to another file, and incrementally implement and test the other file. You can start with the entire body then add one function after another.

α : Assign to `fm` a reference to the `form` element.

β : Make function `validate` a listener for submit events for the form.

γ : Invoke the function `copyValue()` three times to write in the text area the information on the phone, city, and airport text boxes.

δ (occurs twice): Start a control that invokes `run()` every second and assign the control object to `timer`.

ϵ : Decrement `secsLeft` (by 1) and assign the resulting value as the text content of the `td` element that displays the countdown number.

ζ (occurs twice): Stop the repeated invocation of `run()`. This requires the control object that's the value of `timer`.

- η: Play the sound file associated with the embed element.
- θ: Click the submit button.
- ι: Check whether the value in the phone text box has the correct format. This value must be matched by the entire pattern. (A match results in a true value for this expression, but the value is complemented to form the condition for the if statement.)
- κ (occurs 3 times): Suppress submission.
- λ: Cause the phone text box to be in focus.
- μ: Check whether the value in the city text box has the correct format: It must start with an uppercase letter, and this must be followed by one or more (upper- or lowercase) letters or spaces. See also the comments under ι.
- ν: Cause the city text box to be in focus.
- ξ: Check whether the value in the city text box has the correct format: exactly three uppercase letters. See also the comments under ι.
- ο: Cause the airport text box to be in focus.
- π (stretches over 3 lines, some parts already present): Assign to shortName the text in the td cell in the same row as the td cell containing the input element that is the referent of elt (the parameter in this function, copyValue). This can be done in several ways, and you do not have to follow the structure suggested here. In any case, you probably want to break the navigation task up into at least two statements to avoid a very long expression on the right-hand side of an assignment.
- ρ: The value for the text box reference by elt (assigned here to the val).
- σ: Append the text here (viz., "\n" + shortName + ": " + val) to the text already in the textarea.
- τ: Include a colgroup element whose content is col elements that specify that the left column in the table has a width of 150 pixels and that the other column has a width of 125pixels.
- υ (occurs 3 times): Cause copyValue to be called when this input element loses focus, and pass it a reference to the input element itself.