

Due in the Digital Drop Box by Monday, October 2 at 11:00 PM.

1. The HTML document listed below first prompts for an array of integers. It then prompts for two numerical expressions in x . Next, it uses the `Function` constructor to define two functions. Each has x as its formal parameter and returns the value of one of the expressions entered. Finally, it outputs two tables. The first table gives the results of applying the first constructed function to each element of the array of integers, and the second table gives the results of applying the second function to this array. The screen capture below at right shows the output when the integers 1, 2, 3, 4, 5 are supplied in that order for the array, the first expression supplied is $x+1$, and the second is $x*x$.

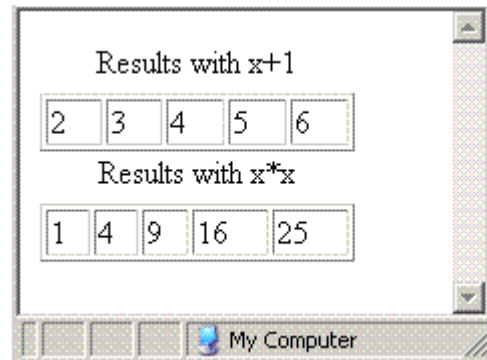
The JavaScript code consists of a single function, `start`, which is invoked when the body of the document is loaded. The code in this function first initializes variable `ar` to the value returned by a call to `inputArray`. Function `inputArray` first initializes local variable `ar` to a new, empty array. A while loop is used to prompt for integers, to convert the strings returned by the prompt to integers, and to assign these integers to successive new elements of `ar`. The condition of the while loop consists of an assignment of the value returned by the prompt to a variable. When the user clicks the **Cancel** button of the prompt box, the prompt, hence the assignment, evaluates to `undefined`, which counts as false, so the loop terminates. So the text in the prompt is

An integer

Cancel to exit

(Use `\n` to get a newline in a dialogue box. Markup is not interpreted in a dialogue box, so `
` will not work.) Within the body of the loop, the value from the prompt is converted to an integer and assigned to `ar` at the index given by a variable that was initialized to 0 and is incremented on each iteration of the loop. Function `inputArray` concludes by returning `ar`.

After initializing `ar`, function `start` then prompts for two numerical expressions in x and uses the `Function` constructor to define functions in terms of these expressions, as described above. Finally, `start` calls function `outApply` twice, once with `ar`, the first string, and the first function (derived from the first string) as arguments, and again with `ar`, the second string, and the second function as arguments. Function `outApply` takes three arguments: an array `ar` of integers, a function `func` with one, numerical formal parameter, and a string `funcStr` that gives the expression used to calculate the value returned by `func`. This function produces the tables shown in the screen capture above (one table per call). The contents of the table caption are produced by concatenating `funcStr` to the end of "Results with ". There is no `thead` element. The table body consists of a single `tr` element. The `td` elements in this row are produced by



a `for/in` loop that ranges a variable over the indices of `ar` and, on each iteration, applies `func` to `ar` indexed by the loop control variable.

The gaps in the listing are labeled with Greek letters. After the listing, these letters are repeated along with descriptions of the code for the corresponding gaps. You can download the file with gaps as part of the zipped file accompanying this assignment.

```

<html>
<head>
<title>HW 4, Prob 1</title>
<script type="text/javascript">
  function start()
  {
    var ar = inputArray(),
        funcStr1 = prompt( "Enter a numerical "
                          + "expression in x",
                          " " ),
        funcStr2 = prompt( "Enter another numerical "
                          + "expression in x",
                          " " ),

        func1 = α _____,
        func2 = β _____;

    outAppl( ar, func1, funcStr1 );
    outAppl( ar, func2, funcStr2 );
  }

  function inputArray()
  {
    γ _____
    _____
    _____
    _____
    _____
  }

  function outAppl( ar, func, funcStr )
  {
    document.writeln( "<table border='1' width='80%'" );
    document.writeln( "<caption>Results with " + funcStr
                      + "</caption>" );
    document.writeln( "<tbody>" );
    document.write( "<tr>" );

    δ _____;

    document.writeln( "</tr>" );
    document.writeln( "</tbody>" );
    document.writeln( "</table>" );
  }

```

Continued

Continued from previous page

```
</script>
</head>
<body onload="start()">
</body>
</html>
```

- α : Use the `Function` constructor to construct a function whose only formal parameter is `x` and whose body consists of a single statement. This statement returns the value of `funcStr1`. (Be sure there is at least one space after `return`.)
- β : Use the `Function` constructor to construct a function identical to the previous one except that it returns the value of `funcStr2`.
- γ : Supply the body of function `inputArray` as described above. (The blank lines do not show the structure of the missing code.)
- δ : Supply the `for/in` loop to produce the `td` cells in the sole row of the table produced by function `outAppl` (as described above).

2. For this problem, we define a prototype (essentially a constructor function) `Store` with the following instance variables:

`name` (expected to be a string)

`address` (expected to be an `Address` object—see below)

`items` (expected to be an array of `Item` objects—see below)

We provide the usual `get` and `set` functionality for the `name` instance variable and the usual `get` functionality for the other two. The `Store()` constructor accepts only one argument, to initialize the `name` instance variable. The `setaddress()` instance method takes the four components of an address and, using them, creates a new `Address` object, which it assigns to the `address` instance variable. The only way to update the `items` instance variable (array) is with a method that uses the information on an item passed to it to construct a new `Item` object, which it adds to the `items` instance array. There is also an instance method to concatenate the information on a store in English-like syntax, an instance method to return an array of the names of the items in a store, and a class method to merge, without repetition, the names of all items in any store in an array of stores. Before discussing the implementation of `Store` in more detail, we discuss the prototypes assumed by `Store` (`Address` and `Item`) as well as some class methods we define for `Array` that are used by `Store`.

You are given file `address.js`, which implements the `Address()` constructor. An `Address` object has the following instance variables (the first three being strings and the last being an integer):

```
street
city
state
zip
```

There are no defaults in the constructor function for any of these instance variables (i.e., if a value for one is not provided, its value is undefined). For each instance variable, a `get` and a `set` method is defined, and the `toString()` instance method forms a string in the usual address format from these four value. You are provided a file `testAddress.js`, which contains code to test the instance methods, and a file `address.html`, which contains script elements linking to `address.js` and `testAddress.js`. To test the `Address()` constructor, you “open” `address.html`.

You are also given file `item.js`, which implements the `Item()` constructor. An `Item` object has the following instance variables:

`name` (a string)

`price` (a floating-point number)

Again, no defaults are provided in the constructor function for these instance variables, but `get` and `set` methods are provided for each. The `toString()` method just returns the name. You are provided a file `testItem.js`, which contains code to test the instance methods, and a file `item.html`, which contains script elements linking to `item.js` and `testItem.js`. To test the `Item()` constructor, “open” `item.html`.

The `mergeNames()` class method for `Store` (see below) uses a `merge()` class method for `Array` that you must define. You are provided a file `array.js`, with gaps

that you fill in, in which `merge()` is defined. To define `merge()`, we define a function `Array_merge()`, which is assigned to the `merge` property of `Array` (not the `merge` property of `Array.prototype`, which would make `merge()` an instance method). The `merge()` method uses the class method `member()` of `Array`. To define this, we again define a function that is assigned to the appropriate property of `Array`. The code for this function is listed in full. Given a value `m` and an array `ar`, it returns `true` if `m` is an element of `ar` (and otherwise returns `false`).

The following is a listing of `array.js`. Method `merge()`, given arrays `ar1` and `ar2`, returns a new array, `ar3`, containing the elements in either `ar1` or `ar2` without repetitions. (A better name might be `union`). The first thing it does is set `ar3` to a copy of `ar1`; the last thing it does is return `ar3`. The gap in the middle is a `for` loop that ranges a variable, say `i`, over the indices of `ar2`. For each value of `i`, it checks whether `ar2[i]` is not a member of `ar3`. If so, it adds `ar2[i]` to `ar3`.

```
/* Array class method member()
 * Give a value 'm' and an array 'ar', return
 * 'true' if 'm' is in 'ar' and otherwise 'false'
 */
function Array_member( m, ar )
{
    var i = 0;

    while ( i < ar.length && ar[i] != m )
        i ++;

    return i != ar.length;
}

Array.member = Array_member;

/* Array class method merge()
 * Give two arrays, 'ar1' and 'ar2', return an array that
 * contains all the elements in either without repetitions
 */
function Array_merge( ar1, ar2 )
{
    var ar3 = ar1.concat();

    _____
    _____
    _____

    return ar3;
}

Array.merge = Array_merge;
```

You are provided a file `testArray.js`, which contains code to test these class methods, and a file `array.html`, which contains script elements linking to `array.js` and `testArray.js`. To test your implementation of `merge()`, “open” `array.html`.

Now, back to `Store`. File `store.js` contains (1) the `Store()` constructor function, (2) the definition of two additional instance methods (`info()` and `getItemNames()`) outside the constructor, and (3) the definition of a class method (`mergeItemNames()`). This file is made available with gaps (see below). The constructor has only one formal parameter, `name`, which is used to initialize the `name` instance variable. The values of the other instance variables are not primitive values or strings and so are always initialized to essentially empty structures. (The only information a `Store` object contains initially is its name.) The `address` instance variable is initialized to an `Address` object all of whose instance variables are undefined, and `items` is initialized to the empty array. The usual `get` and `set` methods are defined for `name`, and typical `get` methods are defined for `address` and `items`. The `set` method for `address` takes the four address components as formal parameters, passes them to the `Address()` constructor, and assigns the new address to the `address` instance variable. We do not define a `set` method for `item`. Instead, we define an `addItem()` instance method that adds one item to the `items` instance array. This method is passed values for the name and price of the new item, passes them to the `Item()` constructor, and adds the new item to the end of the `items` instance array. The `toString()` method just returns the store’s name.

The `info()` instance method returns the concatenation of the four instance variables to provide all the information on the store in something close to an English sentence. It is defined outside the constructor function. We define a function `Store_info()` and assign it to the `info` property of the prototype of `Store`. Instance method `getItemNames()`, which returns an array of all the names of the items in the `items` instance array, is defined similarly. This method initializes local variable `names` to an empty array. It then ranges a variable, say `i`, over the indices of the `items` instance array. For each value of `i`, it adds the value of the `name` instance variable of the i^{th} element in the `items` instance array to the end of the `names` array. After the loop, it returns the array `names`.

The `mergeItemNames()` class method, given an array of `Store` objects, returns an array containing, without repetition, the names of all items in any of the stores. For it, we define a function `Store_mergeItemNames()` and assign it to the `mergeItemNames` property of `Store`. This method takes as its only formal parameter an array `storeAr` of `Store` objects and initializes local variable `ar` to an empty array. It then ranges a variable, say `i`, over the indices of `storeAr`. For each value of `i`, it merges (use `Array.merge()`) `ar` and the array of names of items in the i^{th} element of `storeAr` (invoke the `getItemNames` instance method of `storeAr[i]`); the result of the merge is assigned back to `ar`, which thus works as a sort of accumulator for the loop. After the loop, `ar` is returned.

We here list `store.js` with gaps. The gaps are labeled with Greek letters that are repeated after the listing with descriptions of the code that goes into the corresponding

gaps. We then discuss the test file and the HTML driver and, finally, display the rendering you should get when you “open” the HTML driver with the `store.js` file correctly completed.

```
function Store( name )
{
  this.name = name;
  this.address = new Address();
  this.items = [];
  this.getname = function() { return this.name; };
  this.setname = function( name ) { this.name = name; };
  this.getaddress = function() { return this.address; };
  this.setaddress = function( street, city, state, zip )
  {
    /* A new 'Address' object initialized
     * with the values supplied by the
     * formal parameters */
    α
    _____;
  };
  this.getItems = function() { return this.items; };
  this.addItem = function( name, price )
  {
    /* Add to the 'items' instance array a new
     * 'Item' object initialized with the
     * values supplied by the formal
     * parameters */
    β
    _____;
  };
  this.toString = function() { return this.name; };
}

/* ADDITIONAL INSTANCE METHODS */

/* Instance method info()
 * Return a string that combines the values of the 4 instance
 * variables to provide all the information on the
 * store in something close to an English sentence
 */
function Store_info()
{
  return γ
  _____;
}

Store.prototype.info = Store_info;
```

Continued

Continued from previous page

```

/* Instance method getItemNames()
 * Return an array of all the names of the items in this.items
 */
function Store_getItemNames()
{
  var names = new Array();

  δ
  _____;

  return names;
}

Store.prototype.getItemNames = Store_getItemNames;

/* CLASS METHOD */

/* Class method mergeItemNames()
 * Given an array 'storeAr' of 'Store' objects, return an
 * array containing without repetition the names of all items
 * in any of the stores
 */
function Store_mergeItemNames( storeAr )
{
  var ar = new Array();

  ε
  _____;

  return ar;
}

Store.mergeItemNames = Store_mergeItemNames;

```

- α : Construct a new `Address` object using the four formal parameters here and assign the new object to the `address` instance variable
- β : Construct a new `Item` object using the two formal parameters here and add the new object to the end of the `items` instance array.
- γ : Concatenate together the values of the four instance variables (and a few string literals) to give English-like syntax. See the example rendering below.
- δ : A for loop that ranges a variable, say `i`, over the indices of the `items` instance array. For each value of `i`, the value of the `name` instance variable of the i^{th} element of the `items` instance array is added to the array `names`.
- ϵ : A for loop that ranges a variable, say `i`, over the indices of `storeAr`. For each value of `i`, merge (use `Array.merge()`) `ar` and the names of the elements of the `items` instance variable (use the `getItemNames()` instance method) of the i^{th} element of `storeAr`. Assign the result of the merge back to `ar`.

The following is the test file for Store, `testStore.js`. It first initializes variable `st` to a Store object with the name given. It then uses the `setaddress()` and `addItem()` (twice) instance methods. The `writeln` statement first uses the `toString()` instance method (since `st` occurs where a string is expected), and then uses the various `get` methods (including `getItemNames()`). Next, array `stores` is set to contain three Store elements. Finally, the `mergeItemNames()` class method is invoked to output an array of all the items in these stores without repetition.

```

var st = new Store( "Al's Stop 'n Shop" );
st.setaddress( "2 Oak St.", "McLeansville", "NC", 27301 );
st.addItem( "gum", 1.25 );
st.addItem( "beer", 2.30 );
document.writeln( st, "<br><br>",
                  "Name: ", st.getname(), "<br>",
                  "Address: ", st.getaddress(), "<br>",
                  "Items: ", st.getitems(), "<br>",
                  "Item names: ", st.getItemNames() );
document.writeln( "<br><br>", st.info(), "<br>" );

document.writeln(
"<br>*****<br>" );

var stores = new Array();

stores[0] = new Store( "Al's Place" );
stores[0].setaddress( "2 Oak St.", "McLeansville", "NC", 27301 );
stores[0].addItem( "gum", 1.25 );
stores[0].addItem( "beer", 2.30 );

stores[1] = new Store( "Ed's Place" );
stores[1].setaddress( "3 Pine St.", "Greensboro", "NC", 27411 );
stores[1].addItem( "cookie", 0.25 );
stores[1].addItem( "wine", 4.50 );
stores[1].addItem( "gum", 1.50 );

stores[2] = new Store( "Bo's Place" );
stores[2].setaddress( "4 Elm St.", "Greensboro", "NC", 27402 );
stores[2].addItem( "sandwich", 1.45 );
stores[2].addItem( "whiskey", 6.90 );
stores[2].addItem( "cookie", 0.90 );

document.writeln( "Items at any one of the stores:<br>" );
document.writeln( Store.mergeItemNames( stores ), "<br>" );

```

The following is a listing of the HTML drive, `store.html`. Note that there are links not just to `store.js` and `testStore.js`, but also to `item.js` and `address.js`. This is because Store uses the Item and Address prototypes.

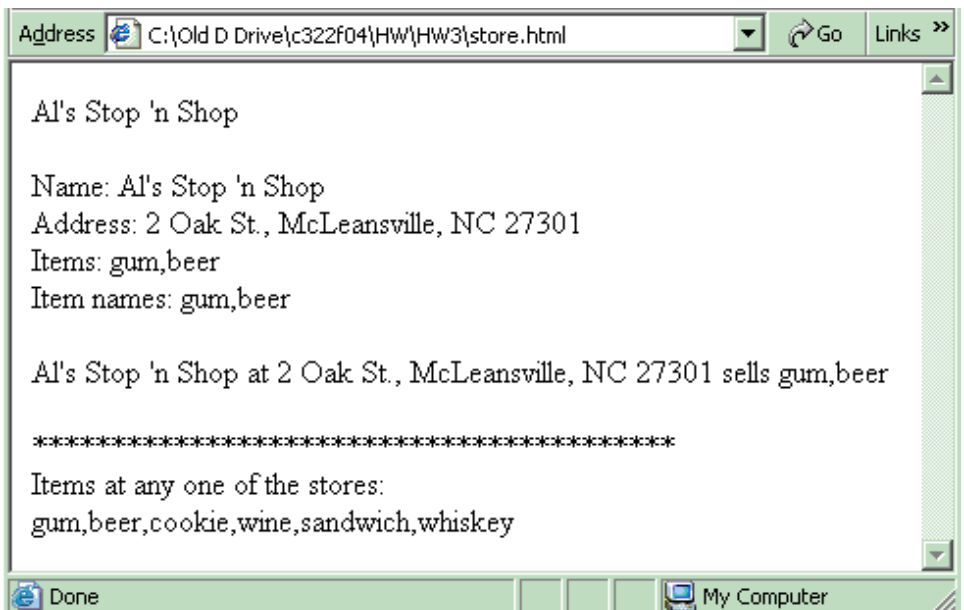
```

<html>
<head>
<title>Store</title>
<script type="text/javascript" src="item.js">
</script>
<script type="text/javascript" src="address.js">
</script>
<script type="text/javascript" src="array.js">
</script>
<script type="text/javascript" src="store.js">
</script>
<script type="text/javascript" src="testStore.js">
</script>
</head>
<body>
</body>
</html>

```

The rendering when `testStore.js` is “opened” is shown at right. You can use this to judge whether your implementation is correct.

You can download the 12 files (an implementation file, a test file, and an HTML driver for each of the three prototypes and the same three for the `Array` class methods) from the Web page where you got this assignment. Recall that two of the files (`array.js` and `store.js` have gaps. When you submit your assignment, zip all the files back together.

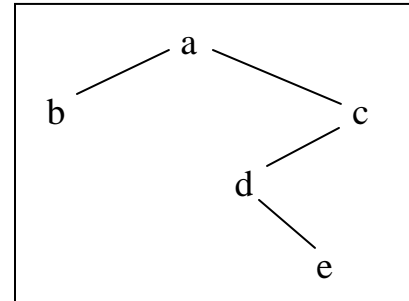


3. The height of a binary tree is the length of its longest branch (root-to-leaf path). Thus, for example, a binary tree whose only node is its root has height 0 and the binary tree shown at right has height 3. As required for consistency, the height of an empty binary tree is -1 . There is a very simple recursive procedure for finding the height of a binary tree. The base case is:

The height of an empty binary tree is -1 .

The recursive case is:

Given a non-empty binary tree T , let $heightL$ be the height of the left subtree of T and $heightR$ be the height of the right subtree of T . Then the height of T is 1 plus the maximum of $heightL$ and $heightR$.



For this problem, you will enhance the code for binary trees on pp. 109 ff. in the notes with the ability to compute height. Recall that we defined a prototype `BTNode` for nodes in a binary tree. We can actually construct binary trees with `BTNodes`, but we also defined a prototype `BinaryTree` to package a binary tree; an instance of `BinaryTree` has a single instance variable, `root`, which references the root node of the binary tree (constructed from `BTNodes`). Recall that we implemented inorder traversal in the notes. We first defined a recursive instance method `inorderTraversal()` for `BTNode`. Then we defined an instance method `inorderTraversal()` for `BinaryTree` that simply checks that the root is non-null then invokes the `inorderTraversal()` instance method of the root node. Here, similarly, we shall define a recursive instance method `height()` for `BTNode` then define an instance method `height()` of `BinaryTree` that returns -1 if the root is null and otherwise returns the value returned by the `height()` instance method of the root.

You can download eight files for this problem from the web page where you got this assignment. Files `btNode.js`, implementing `BTNode`, and `binaryTree.js`, implementing `BinaryTree`, are the implementations shown in the class notes. File `btNodeHeight.js` implements the `height()` instance method of `BTNode` but there are gaps in the code that you must fill in. File `binaryTreeHeight.js`, implementing the `height()` instance method of `BinaryTree`, also has gaps. File `testBTNodeHeight.js` is a file for testing the `height()` instance method of `BTNode`. File `testBTNodeHeight.html` is an HTML document that loads this test file as well as `btNode.js` and `btNodeHeight.js`. File `testBinaryTreeHeight.js` is a file for testing the `height()` instance method of `BinaryTree`. File `testBinaryTreeHeight.html` is an HTML document that loads this test file as well as `btNode.js`, `btNodeHeight.js`, `binaryTree.js`, and `binaryTreeHeight.js`.

Turn in all eight files, zipped together, with the gaps filled in the files that come with gaps.

File Listings

The following is the file `btNodeHeight.js` with gaps.

```
function BTNode_height()
{
  var heightL, heightR;

  α _____
  _____;

  _____;

  β _____
  _____;

  _____;

  return γ _____ + 1;
}

BTNode.prototype.height = BTNode_height;
```

α (4 lines): If the left child of the current node is non-null, then set `heightL` to the result returned by (recursively) invoking the `height()` method of the left child. Otherwise, set `heightL` to `-1`.

β : (4 lines): If the right child of the current node is non-null, then set `heightR` to the result returned by (recursively) invoking the `height()` method of the right child. Otherwise, set `heightR` to `-1`.

γ : the maximum of `heightL` and `heightR`

The following is the file `binaryTreeHeight.js` with gaps.

```
function BinaryTree_height()
{
  α _____
  _____;

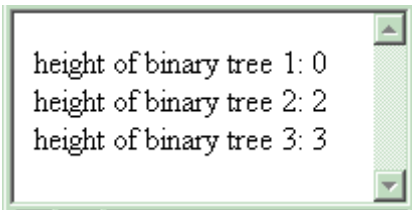
  _____;
}

BinaryTree.prototype.height = BinaryTree_height;
```

α (4 lines): If the root of the current instance is non-null, then return the height of the binary tree of `BTNodes` with that root. Otherwise the binary tree is empty, so return the height of an empty binary tree.

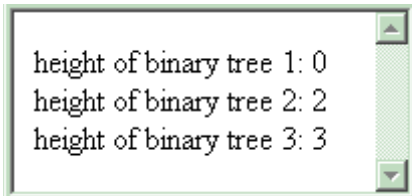
Renderings of the Test Files

The following is the rendering produced when `testBTNodeHeight.html` is loaded.



```
height of binary tree 1: 0  
height of binary tree 2: 2  
height of binary tree 3: 3
```

The following is the rendering produced when `testBinaryTreeHeight.html` is loaded. (It looks just the same as the last.)



```
height of binary tree 1: 0  
height of binary tree 2: 2  
height of binary tree 3: 3
```