LEARN FUNCTIONAL PROGRAMMING BY BUILDING A SPREADSHEET

Introduction:

Functional Programming is a popular approach to software development. In Functional Programming, developers organize code into smaller functions, then combine those functions to build complex programs.

In this spreadsheet application project, you'll learn about parsing and evaluating mathematical expressions, implementing spreadsheet functions, handling cell references, and creating interactive web interfaces. You'll learn how to dynamically update the page based on user input.

This project will cover concepts like the `map()`, `find()`, and `includes()` methods and the `parseInt()` function.

Step 1:

Your project starts with a basic HTML container and some corresponding CSS. Your first task will be to programmatically generate the cells for your spreadsheet.

The global `window` object represents the browser window (or tab). It has an `onload` property which allows you to define behavior when the window has loaded the entire page, including stylesheets and scripts.

Start by setting the `onload` property of `window` to an arrow function with no parameters. In the function, declare a `container` variable and assign it the value of getting the element by the `id` of `"container"`.

Step 2:

Functions are ideal for reusable logic. When a function itself needs to reuse logic, you can declare a nested function to handle that logic. Here is an example of a nested function:

Example Code:

```
const outer = () => {

  const inner = () => {

  };

};
```

Declare a nested `createLabel` function using arrow syntax. It should take a `name` parameter.

Step 3:

Remember that the `document` object has a `.createElement()` method which allows you to dynamically create new HTML elements.

In your `createLabel` function, declare a `label` variable and assign it a new `div` element.

Step 4:

Set the `className` of the `label` element to `"label"`, and set the `textContent` to the `name` parameter.

Step 5:

Finally, use the `.appendChild()` method to add your `label` element to the `container` element.

Step 6:

You will need a function to generate a range of numbers.

Declare an empty `range` function which takes a `start` and `end` parameter. Use the `Array()` constructor and implicitly return an empty array.

Step 7:

Your array will need to be the size of the range. You can calculate this by finding the difference between `end` and `start`, and adding `1` to the result.

Pass this calculation as the argument for your `Array()` constructor.

Step 8:

The `Array()` constructor has a `.fill()` method which can be used to fill an array with a value. You can use this to fill your array with the `start` value.

Chain the `.fill()` method to your `Array()` constructor, and pass it the `start` value.

Step 9:

Currently your `range` function returns an array with the correct length, but all of the values are the value of `start`. To fix this, chain the `.map()` method to your `.fill()` method.

Pass the `.map()` method a callback which takes `element` and `index` as parameters and returns the sum of those parameters.

Step 10:

Now that you have a `range` function, you can use it to create a range of letters as well.

Declare a `charRange` function using `const` and arrow syntax. It should take a `start` and `end` parameter. The function should implicitly return the result of calling `range()` with `start` and `end` as the arguments.

Step 11:

Your `range` function expects numbers, but your `start` and `end` values will be strings (specifically, they will be single characters such as `A`).

Convert your `start` and `end` values in your `range()` call to numbers by using the `.charCodeAt()` method on them, passing the number `0` as the argument to that method.

Step 12:

`range()` will return an array of numbers, which you need to convert back into characters. Chain the `.map()` method to your `range()` call.

Pass a callback function that takes `code` as the parameter and implicitly returns the value of passing `code` to the `String.fromCharCode()` method.

Step 13:

Now that your helper functions are complete, back in your `onload` event handler you should declare a `letters` variable. Assign it the result of calling `charRange()` with the letters `"A"` and `"J"` as arguments.

Step 14:

Now call the `.forEach()` method of your `letters` array, and pass your `createLabel` function reference as the callback.

You should see some letters appear across the top of your spreadsheet.

Step 15:

Remember that `range()` returns an array, so you can chain array methods directly to the function call.

Call `range()` with `1` and `99` as the arguments, and chain the `.forEach()` method. Pass the `.forEach()` method an empty callback which takes `number` as the parameter.

Step 16:

In your callback, you will need to make two function calls. Start by calling `createLabel()` and pass `number` as the argument. You should see some numbers appear in your spreadsheet.

Then call the `.forEach()` method on your `letters` array. Pass an empty callback function which takes a `letter` parameter.

Step 17:

Now in your nested `.forEach()` call, declare an `input` variable. Use the `.createElement()` method of the `document` object to create an `input` element. Set the `type` attribute to `"text"` and the `id` attribute to `letter + number`.

Step 18:

In earlier projects you learned about the `setAttribute` method. Another way to update an attribute in JavaScript is to use the following syntax:

Example Code:

```
el.attribute = value;
```

The property names for hyphenated HTML attribute values, such as `aria-label`, follow camel case, becoming `ariaLabel`.

Example Code:

```
el.ariaLabel = "Aria Label Value";
```

Set the `aria-label` attribute for the `input` element to the same value as the `id` attribute.

Step 19:

Append the `input` element to your `container` element as a child.

You should now be able to see the cells of your spreadsheet.

Step 20:

Most spreadsheet programs include built-in functions for calculation.

Declare a `sum` function that takes a `nums` parameter, which will be an array of numbers. It should return the result of calling `reduce` on the array to sum all of the numbers.

Step 21:

Declare an `isEven` function, which takes a `num` parameter and returns `true` if the number is even, and `false` otherwise. Use the modulo operator `%` to determine if a number is even or odd.

Step 22:

Declare an `average` function which takes an array of numbers as the `nums` parameter. It should return the average of all the numbers in the array.

The average can be calculated by dividing the sum of all the numbers in the array by the length of the array. Remember that you have a `sum` function you can use.

Step 23:

Your next function will calculate the median value of an array of numbers. Start by declaring a `median` arrow function that takes a `nums` parameter.

In the function, declare a `sorted` variable and assign it the value of sorting a copy of the `nums` array.

You should use the `slice()` method for creating a shallow copy of the array.

Step 24:

Now declare a `length` variable and assign it the length of your sorted array, and a `middle` variable that has the value of the length divided by `2`, subtracted by `1`.

Step 25:

Check if `length` is even using your `isEven` function. If it is, return the average of the number at the `middle` index and the number after that. If it's odd, return the number at the `middle` index – you'll need to round the `middle` value up.

Step 26:

Object properties consist of key/value pairs. You can use shorthand property names when declaring an object literal. When using the shorthand property name syntax, the name of the variable becomes the property key and its value the property value.

The following example declares a `user` object with the properties `userId`, `firstName`, and `loggedIn`.

Example Code:

```
const userId = 1;

const firstName = "John";

const loggedIn = true;


const user = {

  userId,

  firstName,

  loggedIn,

};


console.log(user); // { userId: 1, firstName: 'John', loggedIn: true }
```

To keep track of all of your spreadsheet's functions, declare a spreadsheetFunctions object. Using the shorthand notation syntax, set sum, average, and median as properties on the spreadsheetFunctions object.

Step 27:

Now you can start using your spreadsheet functions. Begin by declaring an update arrow function. It should take an event parameter.

Step 28:

In your window.onload function, you need to tell your input elements to call the update function when the value changes. You can do this by directly setting the onchange property.

Set the onchange property to be a reference to your update function.

Step 29:

Since your update event is running as a change event listener, the event parameter will be a change event.

The target property of the change event represents the element that changed. Assign the target property to a new variable called element.

Step 30:

Because the change event is triggering on an input element, the element will have a value property that represents the current value of the input.

Assign the value property of element to a new variable called value, and use .replace() to remove all whitespace.

Step 31:

Now you need to check if the `value` does not include the `id` of the element. Create an `if` condition to do so.

Step 32:

Spreadsheet software typically uses `=` at the beginning of a cell to indicate a calculation should be used, and spreadsheet functions should be evaluated.

Use the `&&` operator to add a second condition to your `if` statement that also checks if the first character of `value` is `"="`.

Step 33:

In order to run your spreadsheet functions, you need to be able to parse and evaluate the input string. This is a great time to use another function.

Declare an `evalFormula` arrow function which takes the parameters `x` and `cells`.

Step 34:

In order to run your spreadsheet functions, you need to be able to parse and evaluate the input string. This is a great time to use another function.

Declare an `evalFormula` arrow function which takes the parameters `x` and `cells`.

Step 35:

Your idToText function currently returns an input element. Update it to return the value of that input element.

Step 36:

You need to be able to match cell ranges in a formula. Cell ranges can look like A1:B12 or A3:A25. You can use a regular expression to match these patterns.

Start by declaring a rangeRegex variable and assign it a regular expression that matches A through J (the range of columns in your spreadsheet). Use a capture group with a character class to achieve this.

Step 37:

After matching a cell letter successfully, your rangeRegex needs to match the cell number. Cell numbers in your sheet range from 1 to 99.

Add a capture group after your letter capture group. Your new capture group should match one or two digits – the first digit should be 1 through 9, and the second digit should be 0 through 9. The second digit should be optional.

Step 38:

Ranges are separated by a colon. After your two capture groups, your rangeRegex should look for a colon.

Step 39:

After your rangeRegex finds the :, it needs to look for the same letter and number pattern as it did before.

Copy your two existing capture groups and paste them after the colon.

Step 40:

Finally, make your rangeRegex global and case-insensitive.

Step 41:

Declare a rangeFromString arrow function that takes two parameters, num1 and num2. The function should implicitly return the result of calling range with num1 and num2 as arguments.

To be safe, parse num1 and num2 into integers as you pass them into range.

Step 42:

Declare a function elemValue which takes a num parameter. The function should be empty.

Step 43:

In your elemValue function, declare a function called inner which takes a character parameter.

Then, return your inner function.

Step 44:

In your inner function, return the result of calling idToText with character + num as the argument.

Step 45:

The concept of returning a function within a function is called currying. This approach allows you to create a variable that holds a function to be called later, but with a reference to the parameters of the outer function call.

For example:

Example Code

```
const innerOne = elemValue(1);

const final = innerOne("A");
```

innerOne would be your inner function, with num set to 1, and final would have the value of the cell with the id of "A1". This is possible because functions have access to all variables declared at their creation. This is called closure.

You'll get some more practice with this. Declare a function called addCharacters which takes a character1 parameter.

Step 46:

In your elemValue function, you explicitly declared a function called inner and returned it. However, because you are using arrow syntax, you can implicitly return a function. For example:

Example Code:

```
const curry = soup => veggies => {};
```

`curry` is a function which takes a `soup` parameter and returns a function which takes a `veggies` parameter. Using this syntax, update your `addCharacters` function to return an empty function which takes a `character2` parameter.

Step 47:

Your inner functions can also return a function. Using the same arrow syntax, update your `addCharacters` function to return a third function which takes a `num` parameter.

Step 48:

Now update your innermost function in the `addCharacters` chain to implicitly return the result of calling `charRange()` with `character1` and `character2` as the arguments.

Step 49:

Use the same syntax as your `addCharacters` function to update your `elemValue` function. It should no longer declare `inner`, but should return the function implicitly.

Step 50:

Your `addCharacters` function ultimately returns a range of characters. You want it to return an array of cell ids. Chain the `.map()` method to your `charRange()` call. Do not pass a callback function yet.

Step 51:

You can pass a function reference as a callback parameter. A function reference is a function name without the parentheses. For example:

Example Code:

```
const myFunc = (val) => `value: ${val}`;

const array = [1, 2, 3];

const newArray = array.map(myFunc);
```

The `.map()` method here will call the `myFunc` function, passing the same arguments that a `.map()` callback takes. The first argument is the value of the array at the current iteration, so `newArray` would be `[value: 1, value: 2, value: 3]`.

Pass a reference to your `elemValue` function as the callback to your `.map()` method.

Step 52:

Because `elemValue` returns a function, your `addCharacters` function ultimately returns an array of function references. You want the `.map()` method to run the inner function of your `elemValue` function, which means you need to call `elemValue` instead of reference it. Pass `num` as the argument to your `elemValue` function.

Step 53:

Declare a `rangeExpanded` variable and assign it the result of calling the `.replace()` method of your `x` parameter. Pass the `rangeRegex` variable as the argument.

Step 54:

The second argument to the `.replace()` method does not have to be a string. You can instead pass a callback function to run more complex logic on the matched string.

The callback function takes a few parameters. The first is the matched string. Pass an empty callback function to your `.replace()` call, and give it a `match` parameter.

Step 55:

The callback function should have a parameter for each capture group in the regular expression. In your case, `rangeRegex` has four capture groups: the first letter, the first numbers, the second letter, and the second numbers.

Give your callback function four more parameters to match those capture groups: `char1`, `num1`, `char2`, and `num2`. `char` will be short for `character`.

Step 56:

Have your callback implicitly return the result of calling `rangeFromString()` with `num1` and `num2` as the arguments.

Step 57:

Call the `.map()` method on your `rangeFromString()` call, passing a reference to `addCharacters` as the callback function.

Step 58:

`addCharacters` returns a function, so you'll want to call it. Pass `char1` as the argument.

Step 59:

Your `addCharacters(char1)` is also returning a function, which returns another function. You need to make another function call to access that innermost function reference for the `.map()` callback. JavaScript allows you to immediately invoke returned functions:

Example Code:

```
myFunc(1)("hi");
```

Immediately invoke the function returned from your `addCharacters(char1)` call, and pass `char2` as the argument.

Step 60:

Now that your `.map()` function is receiving the returned `num => charRange(...).map(...)` function reference from the curried `addCharacters` calls, it will properly iterate over the elements and pass each element as `n` to that function.

You'll notice that you are not using your `match` parameter. In JavaScript, it is common convention to prefix an unused parameter with an underscore `_`. You could also leave the parameter empty like so: `(, char1)` but it is often clearer to name the parameter for future readability.

Prefix your `match` parameter with an underscore.

Step 61:

Declare a variable `cellRegex` to match cell references. It should match a letter from `A` to `J`, followed by a digit from `1` to `9`, and an optional digit from `0` to `9`. Make the regular expression case-insensitive and global.

Step 62:

Declare a `cellExpanded` variable and assign it the value of calling `.replace()` on your `rangeExpanded` variable. Pass it your `cellRegex` and an empty callback function. The callback function should take a `match` parameter.

Step 63:

Update your callback function to return the result of calling `idToText()` with `match` as the argument. Remember that your regular expression is case-insensitive, so you will need to call `toUpperCase()` on `match` before passing it to `idToText()`.

Step 64:

In mathematics, an infix is a mathematical operator that appears between its two operands. For example, `1 + 2` is an infix expression.

To parse these expressions, you will need to map the symbols to relevant functions. Declare an `infixToFunction` variable, and assign it an empty object.

Step 65:

Object values do not have to be primitive types, like a string or a number. They can also be functions.

Give your `infixToFunction` object a `+` property. That property should be a function that takes an `x` and `y` parameter and implicitly returns the sum of those two parameters.

Because `+` is not alphanumeric, you'll need to wrap it in quotes for your property.

Step 66:

Now create a `-` property that is a function that takes an `x` and `y` parameter and implicitly returns the result of subtracting `y` from `x`.

Step 67:

Following the same pattern, add a property for multiplication `*` and division `/` with the appropriate functions.

Step 68:

Now that you have your infix functions, you need a way to evaluate them. Declare an `infixEval` function which takes two parameters, `str` and `regex`. It should implicitly return the `.replace()` method of `str`, with `regex` and an empty callback as the arguments.

Step 69:

Your callback needs four parameters. `match`, `arg1`, `operator`, and `arg2`.

You will not be using the `match` parameter, so remember to prefix it.

Step 70:

The `regex` you will be passing to your `infixEval` function will match two numbers with an operator between them. The first number will be assigned to `arg1` in the callback, the second to `arg2`, and the operator to `operator`.

Have your callback function implicitly return the operator property of your infixToFunction object. Remember that operator is a variable which holds the property name, not the actual property name.

Step 71:

infixToFunction[operator] returns a function. Call that function directly, passing arg1 and arg2 as the arguments.

Step 72:

You have a slight bug. arg1 and arg2 are strings, not numbers. infixToFunction['+']("1", "2") would return 12, which is not mathematically correct.

Wrap each of your infixToFunction[operator] arguments in a parseFloat() call.

Step 73:

Now that you can evaluate mathematical expressions, you need to account for order of operations. Declare a highPrecedence function that takes a str parameter.

Step 74:

In your highPrecedence function, declare a variable using const and assign it a regex that checks if the string passed to the str parameter matches the pattern of a number followed by a * or / operator followed by another number.

Your function should return a boolean value. Remember that you can use the test() method for this.

Step 75:

You should use `console.log()` to print the result of calling the `highPrecedence` function with the string `"5*3"`.

Step 76:

Remove both the `console.log()` with your `highPrecedence` call, and the `return` statement from your `highPrecedence` function.

Step 77:

Now that you have a regular expression to match multiplication or division, you can evaluate that expression.

Declare a `str2` variable and assign it the result of calling `infixEval` with `str` and `regex` as arguments.

Step 78:

Your `infixEval` function will only evaluate the first multiplication or division operation, because `regex` isn't global. This means you'll want to use a recursive approach to evaluate the entire string.

If `infixEval` does not find any matches, it will return the `str` value as-is. Using a ternary expression, check if `str2` is equal to `str`. If it is, return `str`, otherwise return the result of calling `highPrecedence()` on `str2`.

Step 79:

Now you can start applying your function parsing logic to a string. Declare a function called `applyFunction`, which takes a `str` parameter.

Step 80:

First you need to handle the higher precedence operators. Declare a `noHigh` variable, and assign it the result of calling `highPrecedence()` with `str` as an argument.

Step 81:

Now that you've parsed and evaluated the multiplication and division operators, you need to do the same with the addition and subtraction operators.

Declare an `infix` variable, and assign it a regular expression that matches a number (including decimal numbers) followed by a `+` or `-` operator followed by another number.

Step 82:

Declare a `str2` variable, and assign it the result of calling `infixEval()` with `noHigh` and `infix` as arguments.

Step 83:

Declare a `functionCall` variable, and assign it this regular expression: `/([a-z0-9]*)\(([0-9., ]*)\)(?!.*\()/i`

This expression will look for function calls like `sum(1, 4)`.

Step 84:

Declare a toNumberList function that takes an args parameter and implicitly returns the result of splitting the args by commas. Then chain a map method to your split method and pass in parseFloat as the argument to the map method.

Step 85:

Declare an apply function that takes a fn and args parameter.

Step 86:

The fn parameter will be passed the name of a function, such as "SUM". Update apply to implicitly return the function from your spreadsheetFunctions object using the fn variable as the key for the object access.

Remember that fn might not contain a lowercase string, so you'll need to convert it to a lowercase string.

Step 87:

Your apply function is returning the spreadsheet function, but not actually applying it. Update apply to call the function. Pass in the result of calling toNumberList with args as an argument.

Step 88:

Now your applyFunction needs to return a result. Return the result of calling the .replace() method on str2. Pass your functionCall regex and an empty callback.

Step 89:

Update the callback function to take `match`, `fn`, and `args` as parameters. It should implicitly return the result of checking whether `spreadsheetFunctions` has its own property of `fn`.

Remember to make `fn` lower case.

To check if a property on a given object exists or not, you can use the hasOwnProperty() method.

The `hasOwnProperty()` method returns `true` or `false` depending on if the property is found on the object or not.

Here is an example of how to use the `hasOwnProperty()` method:

Example Code:

```
const developerObj = {

  name: 'John',

  age: 34,

}



developerObj.hasOwnProperty('name'); // true

developerObj.hasOwnProperty('salary'); // false
```

Step 90:

Use the ternary operator to turn your `.hasOwnProperty()` call into the condition. If the object has the property, return the result of calling `apply` with `fn` and `args` as arguments. Otherwise, return `match`.

Step 91:

Back at the beginning of this project, you created the inventory array. Add the newWeapon to the end of the inventory array using the push() method.

In the previous project, you learned how to work with the push method like this:

Example Code:

```
const myArray = [];

myArray.push("new item");

// myArray is now ["new item"]
```

Step 92:

Like you did with your highPrecedence() function, your evalFormula() function needs to ensure it has evaluated and replaced everything.

Use a ternary to check if functionExpanded is equal to the original string x. If it is, return functionExpanded, otherwise return the result of calling evalFormula() again with functionExpanded and cells as arguments.

Step 93:

Now your update() function can actually evaluate formulas. Remember that you wrote the if condition to check that a function was called.

Inside your if statement, set the value of the element to be the result of your evalFormula() function. Do not pass any arguments yet.

Step 94:

The first argument for your evalFormula call needs to be the contents of the cell (which you stored in value). However, the contents start with an = character to trigger the function, so you need to pass the substring of value starting at index 1.

Step 95:

You can quickly get all cells from your page by getting the #container element by its id and accessing the children property of the result. Pass that to your evalFormula() call as the second parameter.

Step 96:

Unfortunately, that children property is returning a collection of elements, which is array-like but not an array. Wrap your second argument in Array.from() to convert it to an array.

Step 97:

Your spreadsheet is now functional. However, you don't have support for very many formulas.

Add an even property to your spreadsheetFunctions. It should take a nums parameter, and return the result of filtering the nums array to only include even numbers. Use a reference to your isEven function to help.

Step 98:

Add a firsttwo property which takes a nums parameter and returns the first two elements of the nums array in order. Then add a lasttwo property which returns the last two elements of the nums array in order.

Step 99:

Add a has2 property which returns whether the nums array has 2 in the values, and an increment property which returns nums with every value incremented by one.

Step 100:

Add a someeven property to your spreadsheetFunctions - use the .some() method to check if any element in the array is even.

Step 101:

Arrays have an .every() method. Like the .some() method, .every() accepts a callback function which should take an element of the array as the argument. The .every() method will return true if the callback function returns true for all elements in the array.

Here is an example of a .every() method call to check if all elements in the array are uppercase letters.

Example Code:

```
const arr = ["A", "b", "C"];

arr.every(letter => letter === letter.toUpperCase());
```

Add an everyeven property to your spreadsheetFunctions - use the .every() method to check whether all array elements are even.

Step 102:

Create a `random` property. This property should use the first two numbers from an array to generate a random whole number. The range for this number starts at the first number (inclusive) of the two and ends just before the sum of these two numbers. Use the `Math.floor()` and `Math.random()` methods for the calculation.

Step 103:

Add a `range` property which generates a range from the first number in `nums` to the second number in `nums`. Remember that you have a `range` function you can reuse here.

Step 104:

The last function has a few approaches to implement, and you are free to choose whichever approach you would like.

Add a `nodupes` property which returns `nums` with all duplicate values removed. For example, `[2, 1, 2, 5, 3, 2, 7]` should return `[2, 1, 5, 3, 7]`.

Step 105:

Finally, to handle potential edge cases, add an empty string property (you will need to use quotes) which is a function that takes a single argument and returns that argument.

With that, your spreadsheet project is now complete. You are welcome to experiment with adding support for even more functions.