

Homework #2

Higher Order Functions: Exercises and Processing Data

Overview

Repository Creation

👁️ Click on this link: https://classroom.github.com/a/8G_m2lZh (https://classroom.github.com/a/8G_m2lZh) to accept this assignment in GitHub classroom.

- This will create your homework repository
- Clone it once you've created it.

Description

hoffy.mjs - Write a series of functions that demonstrate the use of the rest operator (or call/apply), and higher order functions

sfmovie.mjs and **report.mjs** - Print out a report analyzing movies filmed at San Francisco, which you will import as a csv file.

See the sample output at the end of these instructions.

Submission Process

You will be given access to a private repository on GitHub. It will contain unit tests, stub files for your code, a `package.json` and a `.eslintrc`.

- The final version of your assignment should be in GitHub.
- **Push** your changes to the homework repository on GitHub.

(4 points) Make at Least 4 Commits

- Commit multiple times throughout your development process.
- Make at least 4 separate commits - (for example, one option may be to make one commit per part in the homework).

Part 1 - Setup and Exercises

For this homework, you'll have files available in your repository, so you'll be cloning first.

The solutions to the following problems can go in the same file - **src/hoffy.mjs**:

Setup

1. go to your github account... and under your profile and organizations
2. find your repository: homework02-yourgithubusername
3. use the appropriate URL to run git clone

```
git clone [YOUR_REPO_URL]
```

Background Info

Implement functions that use JavaScript features such as:

- the rest operator
- the spread operator

- functions as arguments
- functions as return values
- decorators
- optionally call/apply/bind
- optionally arrow functions
- Array methods: 'filter', 'map', 'reduce'

Go through the functions in order; the concepts explored build off one-another, and the functions become more and more challenging to implement as you go on.


Do not use:

- while loops
- for loops
- for ... in loops
- for ... of loops
- forEach method.

There will a small (-2) penalty every time one is used. (Homework is 100 points total)

Steps

1. prep...

-  **NOTE** that if the `.eslint` supplied is not working (it likely isn't), please copy over the `.eslint.cjs` file from your previous assignment(s) (it's a hidden file and starts with `.`) ... or see the edstem thread on how to update the config
- create a `.gitignore` to ignore `node_modules`
- create a `package.json` by using `npm init`
- make sure that `mocha`, `chai`, and `eslint` are still installed (similar to previous assignment)

```
npm install --save-dev mocha
npm install --save-dev eslint
npm install --save-dev chai
npm install --save-dev eslint-plugin-mocha
```

- you'll also need a few additional modules installed locally for the unit tests to run:
 - finally, install `sinon` and `mocha-sinon` locally for mocking `console.log` (these are for unit tests)
 - `npm install --save-dev sinon`
 - `npm install --save-dev mocha-sinon`
2. implement functions below in **hoffy.mjs**
 3. make sure you export your functions as you implement them so that...
 4. you can run tests as you develop these functions (again, the tests are included in the repository): `npx mocha tests/hoffy-test.mjs`
 5. also remember to run `eslint` (there's a `.eslintrc` file included in the repository): `npx eslint src/*`

(40 points) Functions to Implement

(-2 per while, for, forEach, for of, or for in loop used)

`getEvenParam(s1, s2, s3 to sN)`

Parameters:

- `s1, s2, s3 to sN` - any number of string arguments

Returns:

- an array of parameters that have even index (starting from 0)

- if no arguments are passed in, give back an empty array

Description:

Goes through every string argument passed in and picks up the even index parameters (suppose the first parameter has index 0, second has index 1, and so on). No error checking is required; you can assume that every argument is a string or no arguments are passed in at all. If there are no arguments passed in, return an empty array

HINTS:

- use rest parameters!

Example:

```
getEvenParam('foo', 'bar', 'bazzzy', 'qux', 'quxx') // --> ['foo', 'bazzzy', 'quxx']
getEvenParam('foo', 'bar', 'baz', 'qux') // --> ['foo', 'baz']
getEvenParam() // --> []
```

maybe(fn)

Parameters:

- fn - the function to be called

Returns:

- a new function or undefined - the function calls the original function

Description:

maybe will take a function, fn and return an entirely new function that behaves mostly like the original function, fn passed in, but will return undefined if any null or undefined arguments are passed in to fn.

The new function will take the same arguments as the original function (fn). Consequently when the new function is called, it will use the arguments passed to it and call the old function and return the value that's returned from the old function. However, if any of the arguments are undefined or null, the old function is not called, and undefined is returned instead. You can think of it as a way of calling the old function only if all of the arguments are not null or not undefined.

Example:

```
function createFullName(firstName, lastName) {
  return `${firstName} ${lastName}`;
}
maybe(createFullName)('Frederick', 'Functionstein'); // Frederick Functionstein
maybe(createFullName)(null, 'Functionstein');        // undefined
maybe(createFullName)('Freddy', undefined);          // undefined
```

filterWith(fn)

Parameters:

- fn - a callback function that takes in a single argument and returns a value (it will eventually operate on every element in an array)

Returns:

- function - a function that...
 - has 1 parameter, an Array
 - returns a new Array where only elements that cause fn to return true are present (all other elements from the old Array are not included)

Description:

This is different from regular filter. The regular version of filter immediately calls the callback function on every element in an Array to return a new Array of filtered elements. `filterWith`, on the other hand, gives back a function rather than executing the callback immediately (think of the difference between `bind` and `call/apply`). `filterWith` is basically a function that turns another function into a filtering function (a function that works on Arrays).

Example:

```
// original even function that works on Numbers
function even(n) {return n % 2 === 0;}

// create a 'filter' version of the square function
filterWithEven = filterWith(even);

// now square can work on Arrays of Numbers!
console.log(filterWithEven([1, 2, 3, 4])); // [2, 4]

const nums = [1, NaN, 3, NaN, NaN, 6, 7];
const filterNaN = filterWith(n => !isNaN(n));
console.log(filterNaN(nums)); // [1, 3, 6, 7]
```

repeatCall(fn, n, arg)

Parameters:

- `fn` - the function to be called repeatedly
- `n` - the number of times to call function, `fn`
- `arg` - the argument to be passed to `fn`, when it is called

Returns:

- `undefined` (no return value)

Description:

This function demonstrates using functions as an argument or arguments to another function. It calls function, `fn`, `n` times, passing in the argument, `arg` to each invocation / call. It will ignore the return value of function calls. Note that it passes in only one `arg`.

Hint:

Write a recursive function within your function definition to implement repetition.

Examples:

```
repeatCall(console.log, 2, "Hello!");
// prints out:
// Hello!
// Hello!

// calls console.log twice, each time passing in only the first argument

repeatCall(console.log, 2, "foo", "bar", "baz", "qux", "quxx", "corge");

// prints out (again, only 1st arg is passed in):
// foo
// foo
```

largerFn(fn, gn)

Parameters:

- `fn` - candidate function to be returned, takes one integer parameter
- `gn` - candidate function to be returned, takes one integer parameter

Returns:

- `function` - a function that...
 - takes two arguments... an argument for `fn`, and `gn` ... and itself returns yet another function, a "chooser" function
 - this "chooser" function compares the result of calling `fn` and `gn` (`fn` and `gn` each takes one parameter)
 - the "chooser" function returns either `fn` or `gn` depending on whichever function generates larger output (you can assume the value is not going to overflow and if the outputs are equal, return `fn`)

Description:

This function is a decorator. See the slides on the decorator pattern ([../slides/js/higher-order-functions-continued.html](#)) for background information. It builds on top of the example in the slides by actually *modifying* the return value of the original function.

This function wraps the function `fn` and `gn` in another function so that operations can be performed before and after the original function `fn` and `gn` are called. This can be used to modify incoming arguments, modify the return value, or do any other task before or after the function call. Again, we'll be modifying the return value in this case.

Example:

```
// creates two functions, foo and bar
function foo(x) {
  return x * x;
}
function bar(y) {
  return y * y * y;
}
// call largerFn
const decorator = largerFn(foo, bar);
const newFn = decorator(5,3); // 5*5 < 3*3*3 so newFn is bar
console.log(newFn(5)) // -> prints 125
```

limitCallsDecorator(fn, n)

Parameters:

- `fn` - the function to modify (*decorate*)
- `n` - the number of times that `fn` is allowed to be called

Returns:

- `function` - a function that...
 - does the same thing as the original function, `fn` (that is, it calls the original function)
 - but can only be called `n` times
 - after the `n`th call, the original function, `fn` will not be called, and the return value will always be `undefined`

Description:

You'll read from a variable that's available through the closure and use it to keep track of the number of times that a function is called... and prevent the function from being called again if it goes over the `max` number of allowed function calls. Here are the steps you'll go through to implement this:

1. create your decorator (function)
2. create a local variable to keep track of the number of calls
3. create an inner function to return
 - the inner function will check if the number of calls is less than the max number of allowed calls
 - if it's still under max, call the function, `fn` (allow all arguments to be passed), return the return value of calling `fn`, and increment the number of calls
 - if it's over max, just return `undefined` immediately without calling the original function

Example:

```
const limitedParseInt = limitCallsDecorator(parseInt, 3);
limitedParseInt("423") // --> 423
limitedParseInt("423") // --> 423
limitedParseInt("423") // --> 423
limitedParseInt("423") // --> undefined
```

myReadFile(fileName, successFn, errorFn)**Parameters:**

- `fileName` - a string, the name of the file to read from
- `successFunc` - a function to call if file is successfully read
 - function has single argument, the data read from the file as a string
- `errorFunc` - a function to call if error occurs while reading file (such as file does not exist)
 - function has single argument, an error object (the same error object passed to `fs.readFile`'s callback)

Returns:

- undefined (no return value)

Description:

This function gives us an alternative *interface* to `fs.readFile`. `fs.readFile` typically takes a single callback (after the file name) with two arguments: an error object and the data read from the file. This function, instead, takes two callbacks as arguments (both after the file name) – one to be called on success and one to be called on failure. Both callbacks only have one parameter (the data read from the file or an error object). The actual implementation simply calls `fs.readFile`. Note that you can assume that file read in is text, so pass in a second argument to `fs.readFile` to read the data as utf-8:

```
fs.readFile('filename.txt', 'utf-8', callback)
```

Example:

```
// Assuming the contents of the file, tests/words.txt is:
// ant bat
// cat dog emu
// fox
const success = (data) => console.log(data);
const failure = (err) => console.log('Error opening file:', err);

myReadFile('tests/words.txt', success, failure);

// prints out:
// ant bat
// cat dog emu
// fox

myReadFile('tests/fileDoesNotExist.txt', success, failure);
// The error message would be printed out because the file does not exist:
// Error opening file: ...
```

rowsToObjects(data)**Parameters:**

- `data` - an object with the following properties:
 - `headers` - the names of the columns of the data contained in `rows`
 - `rows` - a 2-dimensional Array of data, with the first dimension being rows, and the second being columns

Returns:

- an Array of objects with the original headers (column names) as properties... and values taken from the original data in each row that aligns with the column name

Description:

This converts a 2-d array of data:

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

...into an Array of objects with property names, given the property names as headers.

```
// headers
['a', 'b', 'c'];
```

```
// result
// [{a: 1, b: 2, c: 3}, {a: 4, b: 5, c: 6}, {a: 7, b: 8, c: 9}];
```

The data should come in as an object where the headers an Array in the headers property of the object, and the data is the value in the rows property of the object:

```
{
  headers: ['col1', 'col2'],
  rows: [[1, 2],
         [3, 4]]
}
```

Example:

```
const headers = ['a', 'b', 'c'];
const rows = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
const result = rowsToObjects({headers, rows})
console.log(result);
// [{a: 1, b: 2, c: 3}, {a: 4, b: 5, c: 6}, {a: 7, b: 8, c: 9}];
```

Test, Lint and Commit

Once you're finished with your functions, remember to:

1. make sure all tests are passing
2. make sure that eslint shows no errors
3. commit and push your code!

Part 2 - Drawings and Classes

For this part of the homework, create classes in the file in your repository called **src/drawing.mjs**.

SVG Overview

You'll be implementing classes that represent drawings / parts of a drawing by using SVG.

SVG (<https://developer.mozilla.org/en-US/docs/Web/SVG>) is a markup language for creating vector graphics.

You can think of it as HTML for drawing! (Pssst... it's actually XML). So, some terms used to describe HTML and XML will describe SVG as well:

- **element** - is a part of an SVG image; it's composed to tags, attributes and possibly text / content
- **tag** - is the name of an element surrounded by < and > <rect> ; there can be opening and closing tags in an element: <rect></rect>
- **attribute** - name and value pairs within an opening tag <rect width="100">
 - note that for the example above, the name is unquoted
 - and the value is quoted with double quotes
- **content** - the text surrounded by tags: this is content

Elements can be nested within elements. For example: `<svg xmlns="http://www.w3.org/2000/svg">
<rect width="50" height="50"></rect> </svg>`

SVG markup can be placed in a `.svg` file. An `.svg` file can be opened with browsers, such as Chrome or Firefox. Some file explorers even render SVG directly. SVG can even be embedded directly in an html document (view source on this page, and search for `svg` tags).

svg Element

The container (root element) for an SVG drawing is (big surprise) `svg` :

```
<svg xmlns="http://www.w3.org/2000/svg">  
</svg>
```

- within this container, the x values increase from left to right.
- the y values increase from top to bottom

rect Element

To draw a rectangle, use the `rect` element. Some attributes that you can define on a `rect` element are:

- `x` - the x location of the upper left corner of the rectangle
- `y` - the y location of the upper left corner of the rectangle
- `width` - the width in pixels of the rectangle
- `height` - the height in pixels of the rectangle
- `fill` - the fill color of the rectangle; this can be text, such as `"red"` , `"orange"` , etc.

Note that the `rect` element can be self-closing, but for simplicity in our implementation, we'll create `rect` elements with both an opening and closing tag.

Here's an example of drawing two rectangles:

```
<rect x="25" y="50" width="50" height="100" fill="blue"></rect>  
<rect x="35" y="20" width="50" height="100" fill="orange"></rect>
```

The code above results in this drawing:



text Element

The `text` element allows you to place text in your drawing. Some attributes include:

- `x` - the x location of the upper left corner of the text
- `y` - the y location of the upper left corner of the text
- `fill` - the fill color of the text; this can be a color name, such as `"red"` , `"orange"` , etc.
- `font-size` - the size of the text

Note that the content of a text element (the value between the `text` tags) is what will be displayed. Here's an example:

```
<text x="50" y="70" fill="blue" font-size="70">SVG FTW!</text>
```

The code above results in the following drawing:

SVG FTW!

Implementation

You'll be creating classes that represent generic svg elements, `svg`, `rect`, and `text`. Your implementation:

- **does not require type checking of incoming arguments**
- **should not use**
 - while loops
 - for loops
 - for ... in loops
 - for ... of loops
 - forEach method

There will a small (-2) penalty every time one is used. (Homework is 100 points total)

Create these four classes in `src/drawing.mjs` **and export them**:

1. `GenericElement(name)` - represents a generic SVG element
 - `name` - the name of the element (this name is freeform, so, for example: `svg`, `rect`, or even an element that we haven't covered... `circle`)
2. `RootElement()` - represents an `svg` element
 - should have the attribute: `xmlns="http://www.w3.org/2000/svg"` ...so that it is treated as a valid SVG (XML)
3. `RectangleElement(x, y, width, height, fill)` - represents a `rect` element
 - `x` - the value for the `x` attribute of this `rect` element
 - `y` - the value for the `y` attribute of this `rect` element
 - `width` - the value for the `width` attribute of this `rect` element
 - `height` - the value for the `height` attribute of this `rect` element
 - `fill` - the value for the `fill` attribute of this `rect` element as a color string, such as `"red"`, `"green"`, etc.
4. `TextElement(x, y, fontSize, fill, content)` - represents a `text` element
 - `x` - the value for the `x` attribute of this `text` element
 - `y` - the value for the `y` attribute of this `text` element
 - `fontSize` - the value for the `font-size` attribute of this `text` element
 - `fill` - the value for the `fill` attribute of this `text` element as a color string, such as `"red"`, `"green"`, etc.
 - `content` - content between the `text` tags; the *actual* text

Somewhere in these classes, you should define the following methods (you're free to implement these methods in whatever classes make sense, as long as the example code below works):

- `addAttr(name, value)` - adds an attribute named, `name` to the element called on, with value, `value`
 - `name` - the name of the attribute to be added
 - `value` - the value of the attribute to be added
- `setAttr(name, value)` - set an attribute, `name` to the element called on, with value, `value`. Assume the attribute name always exists when this function is called.
 - `name` - the name of the attribute to be set
 - `value` - the value of the attribute to be set
- `addAttrs(obj)` - adds attributes (keeps existing attributes if already present, but if attribute names are the same, your discretion) to the element called on by using the property names and values of the object passed in: `{x: 100, y: 5}` would translate to `x="100" y="5"`

- `obj` - the object containing properties and value that will be used to create attributes and values on the element
- `removeAttrs(arr)` - remove attributes (remove existing attributes if already present, but if attribute names don't exist, do nothing) to the element called on by using the property names array passed in: `[x,y,z,a]` would remove all attributes `x,y,z,a`, if the `svg` has these attributes.
 - `arr` - an array containing properties that will be removed
- `addChild(child)` - adds an element as a child to the element that this is called on (that is, `child`, is nested in this element)
 - `child` - the element to be nested in the parent element
- `toString()` - returns the string representation of this element, including tags, attributes, text content, and nested elements...
 - for example `<svg xmlns="http://www.w3.org/2000/svg"><text x="50" y="70" fill="blue" font-size="40">SVG FTW!</text> </svg>`
 - you can use any whitespace at your discretion (for example, you can add newlines if you want)
- `write(fileName, cb)` - write the string representation of your element and all its children to a file called `fileName`
 - `fileName` - the path of the file to write to
 - `cb` - function to call when writing is done
 - the callback function can have no parameters
 - use `fs.writeFile` (https://nodejs.org/api/fs.html#fs_fs_writefile_file_data_options_callback) as part of your implementation

Your implementation must follow the requirements listed below:

- you must use `extends` for one or more of your classes
- you can add any properties necessary to make your code work
- an element's attributes can be in any order when the markup is generated (no specific ordering is required from your implementation)
- an element's children can be in any order when the markup is generated (no specific ordering is required from your implementation)
- you can place methods in whatever class you think is appropriate, and you can even repeat code (though you can use inheritance to prevent this) as long as your classes work with the following code:

```
// create root element with fixed width and height
const root = new RootElement();
root.addAttrs({width: 800, height: 170, abc: 200, def: 400});
root.removeAttrs(['abc','def', 'non-existent-attribute']);

// create circle, manually adding attributes, then add to root element
const c = new GenericElement('circle');
c.addAttr('r', 75);
c.addAttr('fill', 'yellow');
c.addAttrs({'cx': 200, 'cy': 80});
root.addChild(c);

// create rectangle, add to root svg element
const r = new RectangleElement(0, 0, 200, 100, 'blue');
root.addChild(r);

// create text, add to root svg element
const t = new TextElement(50, 70, 70, 'red', 'wat is a prototype? 🤖');
root.addChild(t);

// show string version, starting at root element
console.log(root.toString());

// write string version to file, starting at root element
root.write('test.svg', () => console.log('done writing!'));
```

The code above produces the markup below, either via printing to console by converting to a string with `toString` or by writing to a file:

```
<svg xmlns="http://www.w3.org/2000/svg" width="800" height="170">
<circle r="75" fill="yellow" cx="200" cy="80">
</circle>
<rect x="0" y="0" width="200" height="100" fill="blue">
</rect>
<text x="50" y="70" fill="red" font-size="70">wat is a prototype? 🤪
</text>
</svg>
```

Remember that in the resulting markup:

- spacing is at your discretion: you can add / remove newlines, for example
- an element's attributes can be in any order (your discretion)
- an element's children can be in any order (your discretion)

When opened with Chrome or Firefox, you should see:



Part 3 - Processing San Francisco Movie Data

In this part, you'll work with a dataset of movies filmed at San Francisco. You need to read the data from the csv file and convert it to objects in JS, and write a report about the stats of the data.

You'll be using two files for this:

1. `sfmovie.mjs` to create functions at your choice
 - the functions may include reading the file, or a helper function to answer the question
 - it's advised to create one function for each question asked in the report
2. `report.mjs` can be used as a commandline utility. Given an absolute path to a json data file, It should examine the file's contents and print out a report.

Importing Data

- Download the data by going to <https://data.sfgov.org/Culture-and-Recreation/Film-Locations-in-San-Francisco/yitu-d5am> (<https://data.sfgov.org/Culture-and-Recreation/Film-Locations-in-San-Francisco/yitu-d5am>) and clicking the "Export" button on the top right, then the "CSV" button.
 - place it in `data` folder (create this folder if it isn't present)
 - Start by reading in the file (which should have a filename something like `Film_Locations_in_San_Francisco.csv`)
 - make sure that `fs` the module is brought in using ES Modules / `import`, then call the function)
 - **do not use `readFileSync`**
 - See the docs on `fs.readFile` (hint: make sure you specify `utf8` as the second argument)
 - The file that you read in contains reports of movies filmed at San Francisco
 - there's one movie per line
 - each movie is represented by a comma-delimited string
 - there are a couple of options for parsing this data...
 1. you can find a library / npm module to parse csvs (make sure that it's contained in `package.json`)
 - one library, `csv-parse`, supports callback based parsing (as well as streaming, promise and sync)
 - see the docs for `csv-parse` here (<https://csv.js.org/parse/api/callback/>)
 - you can install it with `npm install csv-parse`

- see the example at the end of this section
- 2. alternatively, you can use the following regular expression to split by comma while preserving commas embedded in data
 - assuming `row` is a line of data:
 - `row.split(/,(?=(?:[^\"]*"|"[^"]*"|'[^']*'|'[^\']*')*))/`
 - this may cause an `eslint` warning, but the graders will ignore it
 - note that there some issues with this regex, but it should be adequate for this dataset
 - (this was sourced from one of the answers on this SO post: <https://stackoverflow.com/a/49670696>)
 - each line into an actual JavaScript object, where the properties correspond with the first row of the file (the headers) HINT: use `rowsToObjects(data)` function
 - each resulting object is placed into an `Array` for later processing
 - you can check out some `String` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) and `Array` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)
 - note that the goal of this assignment is to work with higher order functions, so memory efficiency does not need to be taken into consideration
 - All of your parsing can only be done from within the callback function (or the function to be called once data is read from the file) that you supply to `fs.readFile`
 - Examine the resulting `Array` ... (for example, try printing it out!)

Example csv Parse

```
import assert from 'assert';
import { parse } from 'csv-parse';
const input = '#Welcome\n"1","2","3","4\n"a","b","c","d"';
parse(input, {
  comment: '#'
}, function(err, records){
  assert.deepStrictEqual(
    records,
    [ [ '1', '2', '3', '4' ], [ 'a', 'b', 'c', 'd' ] ]
  );
});
```

Examining the Data

- Assuming that your parsed data is in a variable called `data` ...
- `data` will be an `Array` of objects (each line should have been parsed into an object before pushing into the `data` array)
- Each object in `data` contains the column attributes and corresponding values
- **There are several pieces of mis-formatted data starting at line 120. For the sake of simplicity, just ignore them when parsing the data**

Below is a sample listing from the dataset with a description of properties you'd need for your assignment

```
{
  Title: "A Jitney Elopement",
  "Release Year": "1915",
  Locations: "20th and Folsom Streets",
  "Fun Facts": "",
  "Production Company": "The Essanay Film Manufacturing Company",
  Distributor: "General Film Company",
  Director: "Charles Chaplin",
  Writer: "Charles Chaplin",
  "Actor 1": "Charles Chaplin",
  "Actor 2": "Edna Purviance",
  "Actor 3": ""
}
```

Analytics

Once you've implemented helper functions in `sfmovie.mjs`, you can use those functions in `report.mjs`. If it's helpful, you're free to reuse functions from `hoffy.mjs`, in either file, but `for`, `while`, `for in`, `for of`, and `.forEach` are all still off limits.

`report.mjs` is used to print out the information asked below.

`sfmovies.mjs` Functions

Again, each function will have a parameter, `data`, that's in the format of an `Array` of objects, with each object representing movie data.

1. `longestFunFact(data)` - gives back the whole object which has the longest fun fact. (you can just count length of the string in `Fun Facts` column) If there's a tie, return any of them.
2. `getMovies2021(data)` - gives back an array of UNIQUE movie names in 2021
3. `getProductionCompany(data)` - gives back an array of UNIQUE production companies
4. `mostPopularActors(data)` - gives back an `Array` containing the top three actor names, together with most occurrences in the movies (you can either return an object or a nested array, as long as it contains both the actor name and the corresponding occurrence, you should be good). The occurrence should be counted from all three columns "Actor 1", "Actor 2", and "Actor 3". (When counting values in these columns, ignore empty string and undefined values). Okay to count actor if movie appears more than once in dataset. Your discretion on who to include if there is a tie.

Export these functions using `export`.

`report.mjs`

In `report.mjs`:

1. bring in the functions from `sfmovie.mjs`
 - feel free to add more helper functions / utilities
2. if useful, bring functions from `hoffy.mjs`
3. use `process.argv[2]` to retrieve the path specified when `report.mjs` is run on the commandline:
 - `node src/report.mjs /path/to/file`
 - `process.argv[2]` will contain `/path/to/file`
4. find a way to read the contents of this file, parse it, and transform the data into data that can be used for your `sfmovie.mjs` functions
 - Again, using regular expression to parse each attribute for a data entry is recommended.
5. print out a report that includes:
 - The movie that has the longest fun facts and print name and its release year
 - the movies filmed in 2021
 - three production companies (any of three of them)
6. draw a SVG that shows the top 3 popular actors and their occurrences (like a histogram)
 - it should contain 3 rectangle objects
 - the text should contain actor names, and number of occurrences
 - the width should be the same, but the height should be the occurrences of each actor
 - output the SVG in a file named `actors.svg`
 - it's ok if the graph is upside down (because we draw the rectangles from the top to bottom)
 - **when you import `drawing.mjs` to draw `svg`, you can comment out the test code so that it won't log text when you call `report.mjs`**

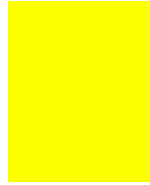
Running the command (from the root of your project):

```
node src/report.mjs data/Film_Locations_in_San_Francisco.csv
```

in terminal on a file would output:

- * The movie Etruscan Smile has the longest fun facts, it was filmed in 2017
- * The movies filmed in 2021 are Venom: Let There Be Carnage, Goliath- Season 4, Shang-Chi and the Legend of the Ten Rings, Clickbait, The Matrix Resurrections.
- * Three of production Companies are: The Essanay Film Manufacturing Company, Metro-Goldwyn-Mayer (MGM), Warner Bros. Pictures

as well as a svg file named `actors.svg` , showing the top 3 actors and their occurrences like below:



Murray Bartlett,144

Jonathan Groff,125

Hugh Laurie,114

Lint, commit and push your code; the next part will make modifications to this existing code (you can overwrite your work in this file directly for the next part).