Functional Array Funhouse Intensive Exercises

Aaron W. Hsu | *arcfide@sacrideo.us* | *http://www.sacrideo.us*

2nd Edition

Introduction/Walkthrough

Set ⎕IO←0 as the first thing that you do. This is assumed to be the case for the rest of the document.

This walkthrough will help you learn about APL and help you to get up and running with your environment. This is a good first taste of programming with APL and seeing how to write complex problems with minimal effort.

Firstly, let's generate the first 9 natural numbers.

```
      ⍳9
0 1 2 3 4 5 6 7 8
```

And we can turn this into a 3 by 3 matrix.

```
      3 3⍴⍳9
0 1 2
3 4 5
6 7 8
```

And from there we can select specific positions using membership.

```
      (3 3⍴⍳9)∈1 2 3 4 7
0 1 1
1 1 0
0 1 0
```

Now let's save this to a variable (name it) for use.

```
      r←(3 3⍴⍳9)∈1 2 3 4 7
```

Now we can take a bigger matrix with `r` embedded within it.

```
      5 7 ↑ r
0 1 1 0 0 0 0
1 1 0 0 0 0 0
0 1 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

We can rotate this matrix around to shift our picture. Let's start by rotating around the vertical axis.

```
      ¯2⌽5 7↑r
0 0 0 1 1 0 0
0 0 1 1 0 0 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

We can then rotate this around the horizontal axis as well.

```
      ¯1 ⊖ ¯2 ϕ 5 7 ↑ r
0 0 0 0 0 0 0
0 0 0 1 1 0 0
0 0 1 1 0 0 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
```

Let's save this as a `R`.

```
      R←¯1 ⊖ ¯2 ϕ 5 7 ↑ r
```

We can produce a nested vector containing three of these values.

```
      R R R
```

0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0

If we use the Each operator, we can apply multiple rotates, one for each of the `R`'s.

```
      1 0 ¯1 ϕ¨ R R R
```

0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0

We can avoid the duplication of needing to have the 3 `R`'s by enclosing the `R` on the right to create a scalar/boxed array that will be applied to each element on the left using the vertical rotation.

```
      1 0 ¯1 ϕ¨⊂R
```

0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0

We can use the Outer Product operator to apply a horizontal rotation for each of the vertical rotations.

```
      1 0 ¯1 ∘.⊖ ¯1 0 1 ⌽¨ ⊂R
```

0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0

Let's sum up the columns.

```
      +⌿1 0 ¯1 ∘.⊖ 1 0 ¯1 ⌽¨ ⊂R
```

0 0 1 1 0 0 0 0 1 2 1 0 0 0 0 1 3 1 0 0 0 0 1 2 0 0 0 0 0 0 1 0 0 0 0	0 0 0 1 1 0 0 0 0 1 2 1 0 0 0 0 1 3 1 0 0 0 0 1 2 0 0 0 0 0 0 1 0 0 0	0 0 0 0 1 1 0 0 0 0 1 2 1 0 0 0 0 1 3 1 0 0 0 0 1 2 0 0 0 0 0 0 1 0 0

And sum up the rows.

```
      +/+⌿1 0 ¯1 ∘.⊖ 1 0 ¯1 ⌽¨ ⊂R
```

0 0 1 2 2 1 0 0 1 3 4 3 1 0 0 1 4 5 4 1 0 0 1 3 3 2 0 0 0 0 1 1 1 0 0

The above is now a neighbor count, including self, which is the critical computation required for the Game of Life. We can use this now, by checking for which positions have a neighbor count of 3 or 4, which we need to figure out whether the next generation of the Game of Life will have a 1 in it.

```
      3 4 = +/+⌿1 0 ¯1 ∘.⊖ 1 0 ¯1 ⌽¨ ⊂R
```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We need to have a 1 in the future generation if there is a neighbor count of 3, or if there is a neighbor count of 4 and the previous generation already has a 1 in that position. We can do this using Boolean-AND on the 3 and 4 results.

```
      1 R ∧ 3 4 = +/+⌿1 0 ¯1 ∘.⊖ 1 0 ¯1 ⌽¨ ⊂R
```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Now, either of these cases will work, so we use Boolean OR with Inner Product to compose them together as a single unit, and then we want to de-box or disclose the result at the end to get a plan matrix as the end result.

```
      ⊃1 R ∨.∧ 3 4 = +/+⌿1 0 ¯1 ∘.⊖ 1 0 ¯1 ⌽¨ ⊂R
0 0 0 0 0 0 0
0 0 1 1 1 0 0
0 0 1 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 0 0 0
```

Notice that we have this expression written with now explicit loops, no branching, and it is written without regards to the size or shape of R, so it will work for any size R. Let's turn this into a function called life. Notice that we change the variable name with the name for the right argument to a function, which is Omega (⍵).

```
      life←{⊃1 ⍵ ∨.∧ 3 4 = +/+⌿1 0 ¯1 ∘.⊖ 1 0 ¯1 ⌽¨ ⊂⍵}
```

Now let's see the first three generations of a Game of Life.

```
      R (life R) (life life R)
```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	1	1	0	0	0	0	1	1	1	0	0	0	0	1	1	0	0	0
0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0
0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We can make a little function that will run the game of life ⍵ number of times starting with ⍺ as the initial matrix.

```
gen←{(life⍣⍵)⍺}
```

Let's see it working.

```
R∘gen⍣⍳4
```

0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0	0 0 1 1 0 0 0 0 0 1 1 1 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0

And now we can do this on a larger matrix and see it working. Let's make our larger matrix first.

```
RR ← 15 35 ↑ ¯10 ¯20 ↑ R
```

Let's make a picture of this. We'll store this picture in pic and we will open an edit window so that we can see it.

```
pic ← '.⌸'[RR]
)ed pic
```

Now let's make an animation of this. We need to update the pic variable on each iteration, and then we need to put in a delay of an 8th of a second to make sure that we can see it working. We'll discard the return result with an empty function, and run it to a fixed point.

```
{}{pic∘←'.⌸'[⍵] ⋄ _←⎕DL ÷8 ⋄ life ⍵} ⍣≡ RR
```

Philosophy and Big Ideas

> *By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. — A. N. Whitehead*

> *The quantity of meaning compressed into small space by algebraic signs, is another circumstance that facilitates the reasonings we are accustomed to carry on by their aid. — Charles Babbage*

Iverson's Principles of Language Design:

- Ease of expressing constructs arising in problems
- Suggestivity
- Ability to subordinate detail
- Economy
- Amenability to formal proofs

Patterns vs. Anti-patterns Dichotomy

A major problem with progressing in APL is that forward progress and good APL follow a set of design/best practices that directly contradict and conflict with traditional programming wisdom. Indeed, APL design patterns appear as Anti-patterns in most other programming languages.

Macro vs. Micro

Traditional programming problem solving practice encourage breaking down a problem into very small components, and most programming languages are designed around the idea of providing a clear picture of a small piece of code in isolation of the rest of the code. APL instead emphasizes finding ways of expressing solutions that emphasize the macro-view of the problem, and attempts to eliminate as much as possible the “small details” or the decomposition of a problem into parts that are viewed separately or independently of the whole.

Transparency vs. Abstraction

Abstraction is considered harmful. This isn't to say that we don't work with abstractions, or at an abstraction, but the process of programming by which we build progressively upon previous abstractions with new abstractions, eventually resulting in some generic framework that we can then plug in our desired instance and obtain a result is considered to be a harmful practice.

Rather than subordinating detail, abstraction hides it. “Good” abstraction hides it so well that you cannot access it. Good programming languages often provide means of solid abstractions that

prevent introspection beyond the abstraction boundaries, while languages are often criticized for permitting too much leakiness in their abstraction boundaries.

APL on the other hand, embraces transparency of expression, and tends to eschew the ever-increasing introduction of abstraction to solve complexity. Instead, transparency of ideas, and directness of expression serve as the weapons with which the APLer falls upon the enemy called "Complexity" and deals a mortal blow.

Idioms vs. Libraries

Libraries create hard barriers in the abstraction landscape. Like Knuth's view on black box libraries, APLer's generally tend to favor the use of idiomatic implementations of common actions, rather than using library calls, whenever possible.

Data vs. Control Flow

Most programming languages emphasize the use of control flow for managing programming logic. Logic in APL is often most efficiently expressed in a data representation that is then processed using array operations, achieving the end result, rather than to have a more complicated source structure that embeds the logic in the control flow of the program.

Structure vs. Names

Other programming languages usually expect to use long, descriptive names that reference often very distant objects or ideas. The use of names serves as the documentation that allows one to track through the code and have an idea of what is going on.

To the APLer, names, particularly long names, often get in the way of revealing the overall structure of the code, which is considered to be more documentary and elucidatory than the names would have been. Because implementations of ideas are often shorter than descriptive names for the same, the use of structure and relatively short (in terms of spatial distance) data dependencies means that names are often chosen to avoid getting in the way of the structure, rather than providing the primary anchor points for understanding the code.

Concision vs. Verbosity

Common programming practice suggests that those unfamiliar with the code base should still get the general idea of the code, which is the idea of "readability." The result is that usually code is deemed to be more readable when it is more verbose, as the expected reader is assumed to now have an intimate knowledge of the entire code base. Indeed, it is presumed that even programmers that spend their whole lives inside of the code base will not be able to keep the whole of the source in their minds at once because of the complexity and size of most non-trivial programs. This leads to a general favoritism to verbose code over concise code, which is considered to be more self-documented even for the experienced programmer.

APL programmers, on the other hand, often optimize their code so that it is more easily manipulated, and they are more easily able to absorb the code base on the whole. Contrary to traditional wisdom, they often expect that they will be able to see and process significant programs at one time (macro vs. micro) which in turn leads them to favor concision, which allows them to see more of the code, rather than verbosity, which hides more code at a time. Concise code is taken to be more readable both at the small level (because it is easier to play with and explore and understand as a whole) but also at the level of integration, where the more concise code is taken to be easier to integrate into a larger code base and not lose sight of the entire source code.

Implicit vs. Explicit

Often programmers are taught and follow the best practice of being very explicit with types, data encodings, and API boundaries. Often data mappings and control flow are very explicit. APLers tend to prefer to make many things implicit in the code, to make the bigger picture clearer. This leads to code that is often written for special cases, that just happen to also work in the general case, or code that is written implicitly in such a way that it is general, though the specific details of the generality are not expressed anywhere. APLers are also more likely to be willing to endure apparent “type” issues or other “inconsistencies” that might otherwise be solved by being more explicit.

Syntax vs. Semantics

Most programmers are taught that syntax matters less than semantics, and they are often only considered with ensuring that the semantics of their ideas are doing what they want, and they will often ignore syntax or only pay attention to it in so far as is required for their own limited sanity.

APLers on the other hand, have a strong tradition of focusing on syntax over semantics, and often spend considerable time thinking about the experience that one feels with code, and how that code is “read” as well as how the syntax flows and works. Semantics are often given their simplest possible assignments, and more effort is put into how best to work with the syntax, rather than working on more sophisticated semantics.

Enter the Matrix

Sometimes people think that programming in APL is like programming in the Matrix. They're not far off! Let's make the classic Matrix coding animation.

To start with you need a canvas on which you're going to do your work. We're going to use a rectangular matrix to do this. We'll start with a 41 by 80 blank matrix (full of spaces) that we will open up in a new window to see how it works.

```
T←41 80⍴' '
)ed T
```

When you want to update `T` with a new value, you can use `T⍤←Expr` to update the edit window with a new value.

Problem Set

Try to create a canvas of different size and shape and see which one you like best.

1. Create a canvas with a different background text. That is, use something other than a space to fill in the data.
2. Now create a pattern for your background. After you're done, go back to a blank canvas.
3. Use the `⎕AV` system variable to extract the characters in the range [156, 54) to get the characters we will use for our animation.
4. To do our animation, the screen starts by scrolling from the top, so let's start by creating a line (as wide as your canvas) containing random characters from #3.
5. Update your canvas by catenating this random line onto your existing canvas. Notice that there's a problem with the canvas getting bigger. Play with different ways you might keep the canvas the same but still add a new line.
6. You can use the `⍣` operator to apply a function multiple times, so try applying what you've learned in #5 to create a scrolling canvas of APL symbols!
7. This isn't quite good enough, so what's a way to improve it? For one, you can use `⎕DL` to slow down the function so that it runs at the right pace.
8. Also, in the original matrix animation, the lines scroll down independently, rather than scrolling all at the same time. You can do this using randomness and rotations to create this effect. See what you can do.
9. See what you can do to tweak this and get it running smoothly and see how authentic you can make it.

Image Processing

The following are a series of challenges and puzzles to stretch your programming. Some of them are trivial, and very easy. You should be able to get them figured out in only a few minutes and by looking through the documentation. However, some of them are more sophisticated, and will require you to break down the problem into smaller pieces.

Before starting your work, always make sure you have initialized the graphics framework by running `codfns.GfxΔInit`*. If you get an error, remember to execute* `)reset` *to get back to a good state.*

The Basics

Choose a photo from that folder and copy it to your desktop. You can then load these images into your workspace using the `LoadImage` function. This function takes the path to the image as its right argument. Here is an example:

```
winter←codfns.LoadImage 'winter.jpg'
```

You should always give a name to the expression above! If you do not, then your session will print the data of the image onto your session, which is likely to be quite large, and will take a very long time to display. You can now view the image that you have just obtained by using the `Imager` operator:

```
codfns.(Image Display {0}) winter
```

Image Arrays

How does the above work? The `LoadImage` function describes an image array from a file that you give it. Specifically, an image array is a rank 3 cube (an array with three dimensions) wherein each element represents a pixel/color channel in the image. Each element is a color value. Colors are represented as they often are in computers, using their Red-Green-Blue values. This is a vector of three elements, where each element is an integer from 0 to 255 (inclusive) that represents the various colors. White is 255 255 255, whereas black is 0 0 0. All blue is 0 0 255 and all green is 0 255 0. To see an example of this, we can look at a little corner (the upper left corner) of our sample image that we used above and see what the colors are. Let's look at the square that is 5 pixels by 5 pixels in the upper left corner of our image above, we're using `↑` to convert from our rank 3 representation to a rank 2 nested representation:

```
      ]disp ↓5 5↑winter
```

80 83 76	85 88 81	65 66 60	75 76 70	94 95 89
92 91 86	106 105 100	108 107 102	107 108 102	90 91 85
77 72 66	99 94 88	76 73 66	56 56 48	69 72 63
52 45 39	88 81 75	67 64 57	38 38 30	48 51 42
49 45 36	56 52 43	66 63 54	58 58 48	39 42 31

We used the `]disp` command to get a nicer picture of the pixels than we would have without it. Try the above expression without the `]disp` command and see what happens.

We can see the dimensions of the image (how large it is) by using the `⍴` function.

```
      ⍴winter 480 768 3
```

This means that this image has 480 rows and 768 columns. Working with images is all about describing transformations on these pixels values.

Hint on playing around

It's often helpful to play around with small examples to gain an understanding of how something works. In order to do this, you don't want to work on your fully image, as that might be way, way too big for you to play with. Instead, you can take a smaller piece of the image using `↑` to get only the upper left corner of the image and play with that. So, if you write something like this:

```
small←20 20↑myimage
```

It will give a name to the upper left corner of `myimage`. You can then play around, using small instead of `myimage` to make your life easier.

Challenges

1. Pick an image that you want, load it into the workspace with `LoadImage` and then display that image using `Image`.
2. The functions `⌽`, `⊖`, and `⍉` rotate and flip and image in various ways. Use these functions to flip the image horizontally and vertically.
3. Using what you know from the previous challenges, rotate the image 90 degrees counterclockwise and clockwise.
4. You can negate an image by subtracting 255 by the image. Try it out.

5. Take an image and shift all the colors so that the red is the blue, and the blue is the green and the green is the red. Try other ways of shifting.

6. Shift an image's color channels so that the blue and the red channels are swapped.

   This is just an easy special case of rotating pixels.

7. Take an image and mask out different color channels so that you see only the blue channel, or the red channel, or the green channel, or some combination of those. The key here is mask. If you are dealing with an array of numbers, like we are with images, you can mask the array, which gives you an array that is exactly the same, but with some of the numbers being zero. That's exactly what we want to do here. Given a color pixel 28 34 100, we can mask this pixel so that we have only 28 0 0, or 0 34 0, or 0 0 100. How do we do this? With numbers, you just take the original array and multiply it (using `×`) by a boolean array of the same shape that has a one where you want the number to stay the same, and a zero where you want to blank out that number. So, a pixel mask, or a mask for pixels, is a 3-vector containing only ones and zeros. If we wanted to mask out the green channel, which is the middle channel, we could do something like `1 0 1×pixel`. This expression describes a pixel that is the same as pixel but with the green channel masked out. The trick for this challenge is to figure out the appropriate mask for masking out the red, green, or blue channels, and then to mask out each pixel in an image. Once you have the right mask, let's call it M, then you can mask out the whole array by using a combination of `⍤` and `×`.

8. Get an image that represents the magnitude of the pixels by taking the sum of the RGB value at each pixel.

   Recall that we can get the sum of a vector by using `+/`. You just have to figure out how to use that function on each pixel.

   If you have an array of the magnitudes, you can still use `Image` with it.

9. We say that the above is taking the magnitude of an image, where each element is the magnitude of the corresponding pixel value, then we can colorize an image better than just taking each color channel by taking the magnitude of each pixel and using that value for a given color channel and leaving all the others zero or with some default values. Try this; it is an extension of the previous challenge.

   Put another way, you can improve the colorization of your image by using the magnitude of a pixel (its sum) as the color value for whatever color channel you want to use. To do this you need to apply the sum function over each of the pixels, and then derive a new pixel from each of the sums. Thus, you should think about how you can take a single scalar value (which represents your magnitude) and get a pixel from it where one channel has the

magnitude and the others are zero. You might also think about what you might need to do to ensure that the magnitude stays in the appropriate range.

See the ↑ function. You can use this to get a 3-vector from a single scalar number. After that, you can use ⌽ as you please to put the number in the right place.

10. During the Obama campaign a distinctive image manipulation came out. We call it the `ObamaIcon`, and you can see a website demo of it here:

    http://www.obama-me.com/obama-me.php

    Let's actually do an `ObamaIcon` function that will let you `Obamaify` your own images. This is a bit more complicated function with multiple steps in it, so let's break it down.

    This is not the easiest, but it's a really fun one once you get it.

    First, let's get the big idea. Basically, we want to find a way to describe an image array that is the exact same as the original image array we are given except that every pixel is changed into one of four colors. The magnitude of the pixel will determine which color to use. To make our life easy, let's start by naming an array of these four colors.

    ```
    colors←(0 51 76)(217 26 33)(112 150 158)(252 227 166)
    ```

    Later, if you want, you can change what you use for colors to see different effects! For now, let's start with these colors. Now, if we write something like `colors[0]`, we will get the first color, and if we write `colors[2]`, we get the third color, and so forth. This is the trick. We can put any array whose elements are numbers 0 through 3 (inclusive) in between the brackets and describe an array of these colors. The shape of the array is the same as the shape of the array between the brackets. This is a technique called bracket indexing. So, as an example, let's say that we want to create a 3 by 3 image of these colors. Let say that we only want the first three colors in a repeating pattern, so we get three vertical lines of the first three colors. We can write this kind of expression:

    ```
          3 3⍴0 1 2
    0 1 2
    0 1 2
    0 1 2
    ```

    This is going to be our index array. Basically, this is an array that we can use between the brackets to get out a color array. So we can put this between the brackets in `colors[]` and get out an image that is a 3 by 3 image like we wanted:

```
      ]disp ↓colors[3 3⍴0 1 2]
┌→──────┬──────┬──────┐
↓0 51 76│217 26 33│112 150 158│
├~──────→~──────→~──────→┤
│0 51 76│217 26 33│112 150 158│
├~──────→~──────→~──────→┤
│0 51 76│217 26 33│112 150 158│
└~──────→~──────→~──────→┘
```

So, to get our Obama Icon, we only need to figure out a reasonable expression to put between the brackets that will give us the image we want. So, we just need to write something like this:

```
obamafied←colors[???]
```

Only, we don't know what `???` is yet. So let's figure out what to put in place of the `???` there. We want an expression that maps each of the pixels in the original image to one of our obama colors. Here's how we do that. We need to divide the magnitude of the image by 192, and then round all of these values down. See the `⌊` function (called "floor" when used monadically) to do this. So, you want the floor of the division of the magnitude of the image and 192. Or, put another way, the floor of the magnitude of the image divided by 192. Make sure the floor is the last function applied (don't divide the floor of the magnitude of the image by 192). See the previous challenge problems for help on getting the magnitude of the image.

Once you get it, and you have the result named `obamified` or whatever else you want to call it, you can use `Image` to see your work!

You may try using a self-portrait from your phone to Obamify. :-)

11. Here's a fun little game. You can show black and white images very easily with `Image`, because if you use a Boolean matrix (a matrix with only ones and zeros in it) instead of an image cube as the right argument to `Image`, it will automatically use that as a black and white image, where the ones are white pixels and the zeros are black pixels. So, it's easy to create a checkerboard from that. You want to figure out an expression that gives you something like this:

    ```
    0 1 0 1 0 1
    1 0 1 0 1 0
    0 1 0 1 0 1
    1 0 1 0 1 0
    ⋯
    ```

    And so on. You can do this with a clever use of `⍴` and `⌽`. Go ahead, give it a shot.

12. How do you turn an image into a grayed out one? Well, you take the average of every pixel and use that as the new color value for each RGB channel in that pixel. So, let's say we have a

color 50 75 63. The average of these is 62.667, so let's take the floor of that so we have 62. The grayscale value of that pixel is then 62 62 62. So, the trick for grayscaling an image is to describe an image where each pixel is the grayscale pixel of the corresponding pixel in the input array. The easiest way to do this is to create a function that describes the grayscale of a pixel and then apply that to each pixel. You'll need to figure out how to write a function that takes the floor of the average of the pixel and then creates the new RGB pixel from that average. If functions are giving you trouble, just ask the instructor to explain them!

After you have the average, you can just reshape it to a 3-vector to describe the new grayscale pixel.

13. We can create cool image maps of a variety of colors, sort of like abstract art, but it requires a bit of work. This is a cool function, but be prepared! Instead of being doable in a single step like most of these other challenges are, this one requires you to break the whole process down into a number of different steps that each need to be put together in the right way.

    This is the hardest problem here!

    Step 1. Write a function that, given two points on a coordinate plane, describes the distance between these points.

    Step 2. Write a function that, given the size of an image on the right and an integer scalar on the left, called N, describes N random points in an image of the given size.

    Step 3. Write a function that, given the same N as in step 2, describes a vector of N random colors.

    Step 4. Given a vector of random colors, and a vector of points corresponding to those colors, write a function that describes the color of any given point (index) on an image by choosing the color corresponding to the point that is closest to the given point using the distance function above. You may find the dyadic version of `⍳` helpful here. Step 5. The `Voronoi` function itself can be written like so:

```
Voronoi←{
  points←⍺ RandomPoints ⍵ ⋄ colors←RandomColors ⍺
  ↑GetColor¨⍳⍵
}
```

    Since this is a multi-line function, you can only define this function by executing `)ed Voronoi` and then typing the above into the edit window that pops up. After you are done, hit the Esc key to save the definition. Now if you got all of that right, you can create a Voronoi image by trying something like this:

```
12 Voronoi 480 768
```

    This generates a 768 by 480 image with 12 random colors in it.

14. How do you crop an image? Cropping an image is taking a smaller piece of the image and using that as the new image. It's easy to do this with what you know already. You can use the take ↑ function to get a smaller piece of the upper left hand corner of an image, so the trick is to figure out how to rotate the image using the dyadic forms of ⌽ and ⊖ so that the start of the area you want to crop is in the upper left hand corner. It will be helpful to draw a picture of this out and then figure out how to do the rotations.

15. In our checkerboard challenge above we used the built-in scaling of the `Imager` function. But it's really easy to scale something on your own! Using the `/` and `⌿` functions (note, functions, not operators; see the documentation for "replicate"), you can easily scale an image by a given scalar number. See if you can play around with what these two functions do to small arrays, and small matrices. Then, try to do the checkerboard example again.

16. Blurring an image is something that we often want to do. How does it work? Well, there are lots of different ways to blur an image, but the easiest is to take each pixel, and then average that pixel with the rest of the pixels around it. This represents a single blur step. If you continue blurring step after step, the image will become progressively fuzzier. Warning You can actually do this one with very little code, but it requires a lot of thought! Make sure to get help if you are stuck. There is an easy way to get the sum of all neighboring values in a matrix, and we actually use the same technique in writing the game of life. See the rotations and outer products in the Game of Life example, and watch the YouTube video to see how this actually works. Feel free to rewind it plenty of times and ask questions about how it works. It's not easy to figure this out at first, but once you get it, the rest is pretty easy. After you get the sum of all the neighboring values, you can just divide the whole thing by 9 (since there are nine pixels that contribute to the average) and then get the floor of that. There are a few other book keeping things that have to be taken care of, but once you have that, you're basically home free. If you turn this into a function called blur, then you can blur N number of times using the following expression:

    ```
    blurred←blur⍣N⊢image
    ```

    Here, N should be some number greater than or equal to zero. We can actually make an animation of this to watch an image get blurrier by the second:

    ```
    codfns.Image∘blur codfns.Display ≡ img
    ```

Enjoy!

Mathematics

The following are a series of challenges and puzzles to stretch your programming. Some of them are trivial, and very easy. You should be able to get them figured out in only a few minutes and by looking through the documentation. However, some of them are more sophisticated, and will require you to break down the problem into smaller pieces. Most of these pieces fit together, so you can often use what you have learned in one of these puzzles to work on the more complicated puzzles. Computer Science is like that, you start with the simple and build up to something complex.

> *If at any time you want to reassign a variable's value, type)erase, and it'll be cleared up for you. It's often helpful to play around with small examples to gain an understanding of how something works.*

Challenges

We can write `a←1 2 3 4` to assign that array of numbers to the variable `a`. We can sum that array by adding the elements of the array `+/a`. You should see the answer `10`.

```
      a←1 2 3 4
      +/a
10
```

1. Write a function sum that such that sum `5 6 7 8 9` is `35`.
2. We can write `⍳15` to get an array of the first fifteen integers, and `⍴⍳15` to verify that we indeed have fifteen elements in our array. Using `⍳` and sum, write an expression that gives us the sum of the numbers from `0` to `1000`.
3. The expression `⍴a` is 4. What is `⍴⍴a`, and why?
4. The function `÷` works like our standard division. Executing `10÷5` should yield `2`. Executing `a÷2` should yield `0.5 1 1.5 2`. Find a way to, in one expression, divide 2 by 1, 4 by 2, 6 by 3, and 8 by 4.
5. Write a function mean that will take the average of a series of numbers. You should be able to write mean `a` and get back `2.5`. Note, your answer should probably use ⍵, and function notation `{` and `}`.
6. We can also add arrays element-wise: `a+a` should yield `2 4 6 8`.
7. Write a function triple, which takes a single argument and triples it. Executing triple 5 should yield `15`, and triple a should yield `3 6 9 12`. We can also easily build our own addition tables. Instead of adding element by element piecewise we can instead, row by row, add each element to each element of the row:

```
      a∘.+a
2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8
```

Figure out in your own words what's happening in the above. Experiment and verify, then read about the `∘.` operator, called the outer product in the Mastering Dyalog APL book.

8. Build a multiplication table for the elements 0 to 10. The function ⍳ is used to build a array of integers in order. The expression `⍳12` should yield an array of elements from 0 to 11.

9. Build a function `multtable`, which, when given a number as input will build a multiplication table for the numbers from 0 to the given number.

```
      multtable 5
0 0 0  0  0
0 1 2  3  4
0 2 4  6  8
0 3 6  9 12
0 4 8 12 16
```

10. Build a function `cube` that will, when given a number (or an array of numbers) return cube of that number or an array of cubes as appropriate. You may want to use `*`, the exponentiation function.

```
      cube 3
27
      cube a
1 8 27 64
```

At this point we have ourselves a pretty sophisticated calculator. But we have more than that we can do. The factorial function, which we write as `!`, takes a number, and multiplies it by the number one smaller than it, continuing until we reach 1. The expression `!5` is `120`.

The commute operator, written `⍨`, allows us to provide one argument to a function, and have it used as both arguments. That is, we have the following relations:

```
f⍨ A ←→ A f A    B f⍨ A ←→ A f B
```

The expression `×⍨10` should yield `100`. We could write `square←×⍨` to define a function square. This will, when provided a number, multiply that number by itself.

```
      square 10 100
```

The function `!`, like almost all of the built-in functions, has two meanings depending on how many arguments you provide it. If provided with one, it acts like factorial. When provided with two, it calculates the binomial coefficient. The binomial coefficient of two numbers n and k is exactly the number of ways you can choose *k* elements from a set of *n* elements.

That is, the expression `3!5` yields `10`, because given any 5 distinct elements, there are 10 different ways we can choose 3 of them.

11. Without using more than one ω, write a function `bintable` which will, when given an argument will build a table similar to the one we built for multiplication and addition, but instead it should be calculating the binomial coefficient.

```
        bintable 5
1 1 1 1 1
0 1 2 3 4
0 0 1 3 6
0 0 0 1 4
0 0 0 0 1
```

    You should be able to try that function out for successively larger arguments. Notice that, when starting at the upper left, that two 1s in the second column sum to 2 (such as in the third). In the third column, 1 and 2 sum to 3. This triangle thus formed is known as Pascal's Triangle.

    We've seen already division. Let's now try division with remainders. The `|` function, when provided with two arguments, is the remainder of how many times the first goes into the second.

```
2|a 1 0 1 0
```

12. Using your `bintable` function, you should build a new function binmod2table that operates similarly, but displays a table not of the binomial coefficient, but the remainder of the binomial coefficients when divided by 2:

```
binmod2table 5
1 1 1 1 1
0 1 0 1 0
0 0 1 1 0
0 0 0 1 0
0 0 0 0 1
```

    Again, try experimenting with this with various sized arguments. And we have a cool tool to graphically display the answers. Type Iup.Init to get it started. Now, you can easily visualize it.

```
mytable←binmod2table 5
codfns.(Image Display {0}) mytable
```

13. Now, build a related function `binmod3table` and visualize that graphically for different arguments. The pattern you see emerging is one called Sierpinski's Triangle. We've just generated a fractal.

14. Now, figure out a way to write a two-argument function `binmodntable` which will take the size of the table as before, but also take a second argument (from the left) which acts as the quotient (that by which we take the modulo). You could use this to explore and see what matrices and images you can produce.

15. Let's practice something else: randomness and dice rolling. The `?` function, called "roll", is like a roll of the dice.

```
?6
?6
?6
```

    You will likely get different answers. Running `?15p6` will roll 15 dice (0-5 rather than 1-6). How would you roll 15 dice and have the result range between 1 and 6, rather than between 0 and 5?

16. Write a function of two arguments which will roll *n* dice and count how many of the landed on the number m. This, too, should be a function of two arguments. You will probably want to use `=` to test for equality and `+/` as a part of your counting.

17. Write a function of two arguments *m* and *n* which will generate an image of size *n* by *n* of random numbers between 0 and *m*. You will notice that our arrays start at base 0, rather than base 1. This is typically useful in computer science, but occasionally we're interested in sequences starting with 1. One way to get around this is to have an `add1` function, which will add 1 to every element of an array.

18. Write an `add1` function:

```
add1 1 2 3 2 3 4
```

    With our `add1` function, we can start to investigate prime numbers. Prime numbers are positive numbers with exactly two distinct factors: one and itself.

19. Write a function `dividesinto` that takes a number *n*, and returns a list of binary values (1 or 0) for all numbers between 1 and *n* (inclusive) indicating whether that number divides *n*. Consider the example below:

```
      dividesinto 12
1 1 1 1 0 1 0 0 0 0 0 1
```

20. Using your `dividesinto` function, you should be able to write a function `isprime` which will return a 1 or 0 indicating if a number given as input is prime.

21. According to WIKIPEDIA a *fractal* is a mathematical set that exhibits a repeating pattern displayed at every scale. They are interesting because they produce patterns that are very similar to things we can see around us, both in living organisms and landscapes, and they seem to be part of the fundamental fabric of the universe.

The MANDELBROT SET is the set of numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$.

Write an APL function to determine whether items of a complex matrix *c* are inside the set.

22. Solve Project Euler PROBLEM #18:

    By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

```
   3
  7 4
 2 4 6
8 5 9 3
```

    That is, $3 + 7 + 4 + 9 = 23$.

    Find the maximum total from top to bottom of the triangle below:

```
                            75
                          95  64
                        17  47  82
                      18  35  87  10
                    20  04  82  47  65
                  19  01  23  75  03  34
                88  02  77  73  07  63  67
              99  65  04  28  06  16  70  92
            41  41  26  56  83  40  80  70  33
          41  48  72  33  47  32  37  16  94  29
        53  71  44  65  25  43  91  52  97  51  14
      70  11  33  28  77  73  17  78  39  68  17  57
    91  71  52  38  17  14  91  43  58  50  27  29  48
  63  66  04  68  89  53  67  30  73  16  69  87  40  31
04  62  98  27  23  09  70  98  73  93  38  53  60  04  23
```

    NOTE: As there are only 16384 routes, it is possible to solve this problem by trying every route. However, PROBLEM 67, is the same challenge with a triangle containing one-hundred rows; it cannot be solved by brute force, and requires a clever method! 😉

    The smallest solution should be around 12-13 characters.

Biology

The first set of challenges are related to the analysis of DNA, based on problems posed on PROJECT ROSALIND, followed by a look at the Game of Life, as an example of a cellular automaton simulation.

A Rapid Introduction to Molecular Biology

Reference: http://rosalind.info/problems/dna/

Problem. A STRING is simply an ordered collection of symbols selected from some ALPHABET and formed into a word; the LENGTH of a string is the number of symbols that it contains.

An example of a length 21 DNA STRING (whose alphabet contains the symbols `A`, `C`, `G`, and `T`) is `ATGCTTCAGAAAGGTCTTACG`.

Given. A DNA string *s* of length at most 1000 nt.

Challenge. Write an expression or function which returns four integers counting the respective number of times that the symbols `A`, `C`, `G`, and `T` occur in *s*.

Computing GC-content

Reference: http://rosalind.info/problems/gc/

The GC-content of a DNA string is given by the percentage of symbols in the string that are `C` or `G`. For example, the GC-content of `AGCTATAG` is 37.5%.

Challenge. Write an expression or function which computes the GC-content of a DNA string.

k-mers

The term *k-mer* typically refers to all the possible substrings of length *k* that are contained in a string. In computational genomics, *k-mers* refer to all the possible subsequences (of length *k*) from a read obtained through DNA Sequencing.

Challenge. Write a function that takes a character vector as its right argument and *k* (the substring length) as its left argument and returns a vector of the *k-mers* of the original string. For example, if the function were named `kmers`:

```
      4 kmers 'ATCGAAGGTCGT'
```

ATCG	TCGA	CGAA	GAAG	AAGG	AGGT	GGTC	GTCG	TCGT

k-mer Counting

Challenge. Write an APL function named `kmerCount` which:

- takes a character vector left argument representing of a string of text

- takes a character vector right argument representing a *k*-mer pattern.
- returns the number of times that the *k*-mer pattern appears as a substring of the text.

For example:

```
      'ACTAT' kmerCount 'ACAACTATGCATACTATCGGGAACTATCCT'
3
```

Note that:

```
      'ATA' kmerCount 'CGATATATCCATAG'
3
```

Returns 3 (not 2) since we should account for overlapping occurrences of the pattern in the text.

k-mer Composition

Reference: http://rosalind.info/problems/kmer/

For a fixed positive integer *k*, order all possible *k*-mers taken from an underlying alphabet LEXICOGRAPHICALLY. Then the *k*-mer composition of a string *s* can be represented by an ARRAY `A` for which `A[m]` denotes the number of times that the *m*th *k*-mer (with respect to the lexicographic order) appears in *s*.

AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
3	5	0	8	6	2	1	3	2	2	0	4	5	3	7	8

The 2-mer composition of

```
TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTGTCAAAATATCACATGTGCCT
```

Given. A DNA STRING *s* in FASTA FORMAT (having length at most 100 KBP).

Challenge. Write a function which returns the *k*-mer composition of the right argument, for *k*-mers of the length given by the right argument.

```
      s1←'TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTGTCAAAATATCACATGTGCCT'
      2 kmerComp s1
3 5 0 8 6 2 1 3 2 2 0 4 5 3 7 8
```

The Game of Life

The game of life shows how emergent behaviors can arise to form complex systems from simple rules.

You have already typed in a game of life based on the classical rules in the getting started part of the tutorial. However, we are going to try to understand what you wrote, and modify it so that it does other, neat things.

You should read the WIKIPEDIA ARTICLE on the Game of Life to learn more about the rules and how it works. Here is the function that we use for the game of life:

```
life←{⊃1 ⍵∨.∧3 4=+/,1 0 ¯1∘.⊖1 0 ¯1⌽¨⊂⍵}
```

And recall that we can animate this with the `codfns.Display` operator. See the tutorial to walk you through how to do that.

Additional Resources

Dyalog's Game of Life page:	http://dfns.dyalog.com/n_life.htm
YouTube: Game of Life in APL:	https://www.youtube.com/watch?v=a9xAKttWgP4

Challenges

1. Run the Game of Life.
2. Search online for see the resources below for some example configurations for the game of life. Try to create your own variations and study the results. Make sure to give the gliders a try!
3. Try building more complicated configuration by putting a number of your previous forms together. How do you combine multiple arrays together? Try using the `+` function and the `∧` and the `∨` functions and see how they behave differently.
4. Extend your game of life so that instead of having each wall wrap around to the other side, it extends out as if you were viewing only a small part of an infinite plane. The trick here is to first resize the incoming starting configuration to put a border of zeros around the edge, and then to strip off the border of the resulting generation after applying the life function to it.
5. Extend the game of life to work on a cylinder instead of wrapping around on all sides. To do this, you need to do the same thing as in challenge #4, but only on two of the sides, not all of them.
6. Try changing the rules of the game of life. What rules can you change and how would you change the `life` function to use those new rules?

Chemistry

The ideal gas law is a classic example of formulae in Chemistry that help us predict the behavior of "ideal gases." In this domain, you will learn to explore the ideal gas law by simulation and calculation of the basic ideal gas law:

$$PV = nRT$$

This is a great way to learn how to do some basic programming. In this domain you will:

1. Encode the ideal gas law to calculate to observe how changes in one variable are related to changes in another, with all others being fixed;
2. Design a function to graph these functions; and finally,
3. Visualize these effects by overlaying multiple graphs on one another; and finally.

This is a pretty involved set of things to know when you are just starting out, so don't worry about finishing. Just go through and explore, ask questions, and learn. Have fun!

The Formulas

The first thing that we need to have are the set of formulas we will need to calculate. The ideal gas law has four variables in it. We want to come up with functions that allow us to study the relationship between these four variables. To do this we will have the following variations of the gas law:

$$P(V) = \frac{nRT}{V}$$

$$P(T) = \frac{nRT}{V}$$

$$V(P) = \frac{nRT}{P}$$

$$V(T) = \frac{nRT}{P}$$

$$T(P) = \frac{PV}{nR}$$

$$T(V) = \frac{PV}{nR}$$

Each of the above allows us to study two variables `A(B)` where we want to see how `A` changes in relation to `B`. That is, given different values of `B`, what happens to `A`? The variable in the parentheses in the above equations is therefore called the independent variable.

One thing we can do right now is to define `R←8.314`, which is going to be the same in all of these cases. To do that, let's give the name `R` to the appropriate constant.

Encoding the Formulas

The first step in all of this is to define a function to correspond to each of the above formulas. These functions will both look like this:

```
PV←{n T←⍺ ⋄ FORMULA}
PT←{n V←⍺ ⋄ FORMULA}
VP←{n T←⍺ ⋄ FORMULA}
VT←{n P←⍺ ⋄ FORMULA}
TP←{n V←⍺ ⋄ FORMULA}
TV←{n P←⍺ ⋄ FORMULA}
```

Your task is to replace `FORMULA` with the appropriate formulas above, using the functions that you have available. You will only need to use ÷ and × for these functions (they are relatively simple as functions go).

Now let's try them out. If you have properly encoded the formulas, then using them with any array of numbers should work.

```
      100 10 PT ⍳4
0 83.14 166.28 249.42
```

We can see here that this is basically what we expect. Given a constant amount of gas and a constant volume, the pressure increases as the temperature increases. You can play with these formulas a bit to see if they match what you expect.

Working with our formulas

Right now, we've seen that we can use `⍳N` where `N` is some natural number to generate inputs for our `Y` functions. This is not quite good enough though. So, we can now see the values of `PT` or some other function at whole number values, but what about some other finer resolution, such as fractions. Try out different arguments to `⍳` and see if you can understand how it works. You can read the help or ask your instructor after you have played around a bit to see if your intuition matches the documentation.

Challenge 1. Use `⍳` and any other functions you want to generate all of the quarter seconds between 0 and 3. It should output the following:

```
0 0.25 0.5 0.75 1 1.25 1.5 1.75 2 2.25 2.5 2.75
```

Challenge 2. Write a function called Step that describes all of the values from 0 to `N` (given as the left argument) with the given step `S` (given as the right argument). With it, we can generate the same as from challenge 1 like so:

```
      3 Step ÷4
0 0.25 0.5 0.75 1 1.25 1.5 1.75 2 2.25 2.5 2.75
```

Graphing your Functions

Now let's see about generating a visualization of the gas law. We will try to visualize the changes of each variable based on another. The first thing that we will need is a plan. The way that graphs work is to define a box (a matrix) that will hold all of your plot data. The trick then is to figure out how we are going to take a given (X, Y) pair and place it accurately inside of our "bounding box." To do this, let's start by thinking in only a single dimension. Say that we have the numbers from our previous challenges, 0 through 3 in quarter increments. Now, let's say that we wanted to fit each of these numbers into boxes, but we have only three boxes, number 0 through 2. If we are going to put these other numbers into these boxes, in order, then we have to squash some into the same box. If we want to divide them up evenly and in the same order, then box 0 will hold everything less than 1, and box 1 will hold everything less than 2, and box 2 will hold everything else.

This action that we have just done is to take a set of elements and fit them all into a smaller set of elements. We've created a correspondence from the larger set to the smaller set. This is known as scaling down our larger set to the smaller one. When we are graphing, we have to do the same thing. We will have a set of inputs that we need to scale, either up or down, to fit into the box that we want to use to do the graphing.

Challenge 3. The first challenge then, in getting our graphing done is to write a function Scale that will take the size of our box on the left, which we will represent as a single natural number, and all of the elements on the right. It should describe an array with as many elements as the right argument, but with every value scaled to fit within the range 0 to the left argument. The result should have only natural numbers in it. Here's an example using the case above:

```
      3 Scale 3 Step ÷4
0 0 0 0 1 1 1 1 2 2 2 3
```

Challenge 4. With the `Scale` function written, we can write an operator to graph things out that looks something like this:

```
Graph←{
  h w←¯1+H W←⍺ ⋄ img←H W⍴1
  y x←h w Scale¨(⍺⍺ ⍵) ⍵
  img[...]←0
  codfns.(Image Display {0}) img
}
```

Your job is to write replace `...` with the right code. Here's how the above code works. The first line takes the left argument, which is a pair of the height and width of the box, and gives them names, `H` and `W`. It also names `h` and `w` which are the largest index into the box (ask your instructor about base-0 indexing to learn more about this). Now, we create a blank image of just ones first (thus, all white for a black and white image). For the second line, we use `⍺⍺` which will be the `Y` function. We apply `⍺⍺` on the right argument `⍵` of the function. We then `Scale` each (`Scale¨`) of these `X` and `Y` values

using the small height and width. Now `y` and `x` are vectors containing all of the corresponding *x* and *y* values for graphing. The third line is what you want to work on. It will set all of the coordinates described by `...` to zero, which means they will be drawn as a black dot on our graphed window. You job is to replace `...` with an expression that describes all of the pairs of (y, x) that need to be drawn. The `,` function will be of help here.

Remember to execute `codfns.GfxΔInit` before using any graphics functions.

Challenge 5. A frequent problem that you will see if you start on #4 is that your graph might be drawn upside down or flipped around. You can modify your expression so that it draws it using the appropriate origin. The reason that you get this strange behavior is that the origin on the box is in the upper left-hand corner, while the origin of the normal coordinate plan that you use in math and when graphing functions is in the lower left hand corner. Try to think of how you can transform the values of `X` and `Y` to account for this difference.

Challenge 6. Modify your `Graph` function to draw a straight line across starting from the point where the function starts to the end. This will represent the origin and will allow you to easily see the path of the function. You will need to add one additional line to do more drawing, and it will help if you use the `⍳` function.

Challenge 7. It would be nice if we could see the results of multiple graphs overlayed on top of one another so that we could see the relative differences in those graphs. We can do this with our existing `Graph` operator, but the solution might be a little non-obvious. The trick is to create a new function that catenates (see the `,` function) the results of using multiple functions on multiple different values. So, if we call this function `YPT`, it would describe all the `Y` values at, say, 100 moles and a volume of 10, catenated to the `Y` values at, say, the same moles and a volume 30. This is just sticking together a series of solutions one after another. Using the `Graph` operator with `YPT` will produce multiple plots. You should be able to graph multiple plots by executing the following:

```
720 1152 YPT Graph M Step ÷1000
```

You can change the bounding box size on the left and give your own value for `M`. You can also change the step amount on the right and see how that changes the graph.

Challenge 8. It would be annoying if we wanted to plot a number of different variations of `PV` with different size and temperatures all the time if we had to write out the `YPV` function by hand. But we can define an operator that will do this for us. Can you implement an operator that will let us create `Y` functions that we can use with `Graph`? Here's an example of how you would call it.

```
YPV←PV Plot (20 45)(20 30)(20 20)
```

Using this with `Graph` would give us the function plots at the three different temperatures of 45, 30, and 20 degrees. This is a tough one, so I'll note that my solution uses the following functions and operators:

```
⊃,/↑¨⊂
```

Good luck! If you do not know how operators work, as your instructor for a demonstration or read the help documentation on operators.

Challenge 8 is really tricky!

Physics

One of the classic first steps when studying physics is exploring things like gravity and the traditional laws of Physics. You can do all of this just by doing a lot of symbol pushing with mathematics to prove that everything works out. However, you can also use computers to simulate the behavior according to the equations that you have to verify that your math works out. In this domain, we're going to model gravity and cannons and monkeys on the computer, eventually getting a simulation of the behavior. In this domain, we'll do a few things:

1. Encode the formulae to calculate the trajectory of a cannon ball;
2. Design a function to graph this trajectory;
3. Determine the best angle for cannonball distance through simulation; and finally,
4. Visualize the change in trajectory as parameters change.

This is a pretty involved set of things to know when you are just starting out, so don't worry about finishing. Just go through and explore, ask questions, and learn. Have fun!

The Kinematic Formulas

The first thing that we need to have are the set of formulas we will need to calculate the distances of our cannonball. In our case, since we are doing visualizations and the like, we only care about the functions for the X and Y values on a graph with time T as our input. We also don't think in terms of vertical or horizontal velocities when we are firing a cannon. Instead, we think of the angle of the cannon and the muzzle velocity. So, we want to make sure that our formulae are stated in terms of the angle of the cannon and the muzzle velocity, rather than in terms of vertical or horizontal velocity. This requires some simple substitution on the part of the traditional equations, but is otherwise straightforward.

$$x(t) = vt\cos\theta + \frac{1}{2}a_x t^2$$

$$y(t) = vt\sin\theta + \frac{1}{2}a_y t^2$$

The above formulas assume that our cannon is resting directly on the ground. We use v to represent the initial muzzle velocity, and θ to represent the angle of the cannon. In our case, a_y is $-9.81 m/s^2$ for gravity. Because we are assuming that there are no forces applied to the horizontal motion of the cannonball, we assume that a_x is zero.

Encoding the Formulas

The first step in all of this is to get two functions `Xt` and `Yt` that take a vector of times as the right argument and a pair containing the initial muzzle velocity and the angle of the cannon as the left argument. These functions will both look like this:

```
Xt←{Vi theta←⍺ ⋄ FORMULA}
Yt←{Vi theta←⍺ ⋄ FORMULA}
```

Your task is to replace `FORMULA` with the appropriate formulas above, using the functions that you have available. You will only need to use `×`, `○`, `÷`, `+`, and `*` to do this. Read the documentation to see how they work and ask for help if you cannot figure it out. Indeed, the `○` function can be a little tricky, so feel free to ask someone to explain it to you.

Now let's try them out. The most interesting one is `Yt`, which will show us the height of the cannonball for any given time. If you have properly encoded the formula, then it will also work to give it a vector of times.

```
      20 0.7854 Yt ⍳4
0 9.237161597 8.664323194 ¯1.718515208
```

Just from seeing these numbers you can see how the cannonball will climb up and then by the second second (snicker) it will have begun falling. This is with the cannon having an initial velocity of 20 and an angle of 45 degrees.

Wait, what is that 0.7854 that we have there? That's our degrees. We need to give our `theta` value in radians rather than degrees. Let's write a helper that will convert Degrees to Radians for us:

```
Deg2Rads←{...}
```

It's your job to fill in the `...`. Do a search for Radians on Wikipedia to find the conversion formula.

Working with our formulas

Right now, we've seen that we can use `⍳N` where `N` is some natural number to generate inputs for our `Yt` function. This is not quite good enough though. That lets us see the height of the cannonball at each second, but what about half seconds or some other finer resolution.

Challenge 1. Using `⍳` and any other functions you want to generate all the quarter seconds between 0 and 3. It should output the following:

```
0 0.25 0.5 0.75 1 1.25 1.5 1.75 2 2.25 2.5 2.75
```

Challenge 2. Write a function called `Step` that describes all the values from 0 to *N* (given as the left argument) with the given step `S` (given as the right argument). With it, we can generate the same as from challenge 1 like so:

```
      3 Step ÷4
0 0.25 0.5 0.75 1 1.25 1.5 1.75 2 2.25 2.5 2.75
```

Graphing your Functions

Now let's see about generating a visualization of the cannon fire. We will try to visualize the trajectory of the cannonball. The first thing that we will need is a plan. The way that graphs work is to define a box (a matrix) that will hold all your plot data. For any given time, we can use the `Xt` and

`Yt` functions to generate the X and Y pairs for our plot. The trick then is to figure out how we are going to take a given (X, Y) pair and place it accurately inside of our "bounding box." To do this, let's start by thinking in only a single dimension. Say that we have the numbers from our previous challenges, 0 through 3 in quarter increments. Now, let's say that we wanted to fit each of these numbers into boxes, but we have only three boxes, number 0 through 2. If we are going to put these other numbers into these boxes, in order, then we have to squash some into the same box. If we want to divide them up evening and in the same order, then box 0 will hold everything less than 1, and box 1 will hold everything less than 2, and box 2 will hold everything else.

This action that we have just done is to take a set of elements and fit them all into a smaller set of elements. We've created a correspondence from the larger set to the smaller set. This is known as scaling down our larger set to the smaller one. When we are graphing, we have to do the same thing. We will have a set of inputs that we need to scale, either up or down, to fit into the box that we want to use to do the graphing.

Challenge 3. The first challenge then, in getting our graphing done is to write a function `Scale` that will take the size of our box on the left, which we will represent as a single natural number, and all of the elements on the right. It should describe an array with as many elements as the right argument, but with every value scaled to fit within the range 0 to the left argument. The result should have only natural numbers in it. Here's an example using the case above:

```
      3 Scale 3 Step ÷4
0 0 0 0 1 1 1 1 2 2 2 3
```

Challenge 4. With the `Scale` function written, we can write an operator to graph things out that looks something like this:

```
Graph←{
  h w←¯1+H W←⍺ ⋄ img←H W⍴1
  y x←h w Scale¨(⍺⍺ ⍵) (⍵⍵ ⍵)
  img[...]←0
  codfns.(Image Display {0}) img
}
```

Your job is to write replace `...` with the right code. Here's how the above code works. The first line takes the left argument, which is a pair of the height and width of the box, and gives them names, `H` and `W`. It also names `h` and `w` which are the largest index into the box (ask your instructor about base-0 indexing to learn more about this). Now, we create a blank image of just ones first (thus, all white for a black and white image). For the second line, we use `⍺⍺` which will be the `Y` function and `⍵⍵` which will be the `X` function. We run both of these on the right argument `⍵` of the function. We then `Scale` each (`Scale¨`) of these values using the small height and width. Now `y` and `x` are vectors containing all of the corresponding x and y values for graphing. The third line is what you want to work on. It will set all of the coordinates described by `...` to zero, which means they will be drawn

as a black dot on our graphed window. Your job is to replace `...` with an expression that describes all of the pairs of (y, x) that need to be drawn. The `,` function will be of help here.

Remember to execute `codfns.GfxΔInit` before using any graphics functions.

Challenge 5. A frequent problem that you will see if you start on #4 is that your graph might be drawn upside down or flipped around. You can modify your expression so that it draws it using the appropriate origin. The reason that you get this strange behavior is that the origin on the box is in the upper left-hand corner, while the origin of the normal coordinate plan that you use in math and when graphing functions is in the lower left hand corner. Try to think of how you can transform the values of `x` and `y` to account for this difference.

Challenge 6. Modify your `Graph` function to draw a straight line across starting from the point where the cannon fires to the end. This will represent the ground and will allow you to easily see where the cannonball strikes the ground after being fired. You will need to add one additional line to do more drawing, and it will help if you use the `⍳` function.

Challenge 7. It would be nice if we could see the results of multiple graphs overlaid on top of one another so that we could see the relative differences in those graphs. We can do this with our existing `Graph` operator, but the solution might be a little non-obvious. The trick is to create two new functions `Y` and `X` that catenate (see the , function) the results of using `Yt` and `Xt` on multiple different values. So, `Y` would describe all the `Y` values at, say, velocity 20 and angle 45, catenated to the `Y` values at, say, velocity 20 and angle 30. You would do the same with the `X` values in `X`. This is just sticking together a series of solutions one after another. If you make sure that the `X` and `Y` values match one another then using the Graph operator with `Y` and `X` will produce multiple plots. You should be able to graph multiple plots by executing the following:

```
720 1152 (Y Graph X) M Step ÷1000
```

You can change the bounding box size on the left and give your own value for `M`. You can also change the step amount on the right and see how that changes the graph.

Challenge 8. It would be really annoying if we wanted to plot a number of different variations of `Yt` with different velocities and angles all the time if we had to write out the `Y` function by hand. But we can define an operator that will do this for us. Can you implement an operator that will let us create `Y` or `X` functions that we can use with `Graph`? Here's an example of how you would call it.

This one is really tricky!

```
Y←Yt Plot (20 45)(20 30)(20 20)
X←Xt Plot (20 45)(20 30)(20 20)
```

Using this with `Graph` would give us the cannonball trajectories at the three different angles of 45, 30, and 20 degrees, each shot with the same muzzle velocity. This is a tough one, so I'll note that my solution uses the following functions and operators:

```
⊃,/↑¨⊂
```

Good luck!

Finding the best angle

One of the great things about using computers and knowing how to program is that we can use them to try out interesting elements and study interesting phenomenon in ways that were impossible without them. As an example, if you have finished the #8 challenge assignment, can you use that to try to see what the ideal cannon angle is for shooting a cannon ball the maximum distance? Is it the same regardless of the velocity? Give it a try and see what happens.

Challenge 9. Write a function to figure out the distance that the cannonball travels before hitting the ground given the inputs that it has. If the inputs describe a trajectory that does not have the cannonball below the dirt at some point, you can feel free to just use the greatest distance of those inputs given, you don't have to worry about figuring output when the cannon ball hits the ground. You may find the dyadic form of `⍳` to be useful here. It's called "find."

Challenge 10. Graph this distance function. You can use the `⊢` identity function as the right argument to `Graph` in this case, because we are graphing along time. Do you get what you expect?

Challenge 11. Use the `Distance` function and the `Plot` function to experimentally determine the best angle for the maximum distance of the cannon to a tenth of a degree.

2-D Heat Equation Simulation

The HEAT EQUATION can be discretized and computed as a stencil computation. You can look at the Stencil (`⌺`) operator for more information on how you can easily compute a stencil with APL.

Challenge 12. Write the stencil computation for the Heat Equation.

Challenge 13. Simulate a sample heat diffusion visually.

Graphs

In this domain we're going to be playing with graphs and how graphs work. Specifically, you're going to be building up code to demonstrate one of the famous theorems surrounding graphs, that of graph coloring. For more information on graph coloring see here:

https://en.wikipedia.org/wiki/Graph_coloring

We've been given the graph we're going to be working in this format (silly Schemers):

```
(define middle-earth
  '((lindon eriador forodwaith)
    (forodwaith lindon rhovanion eriador)
    (eriador lindon forodwaith rhovanion enedwaith)
    (rhovanion forodwaith eriador enedwaith rohan rhun)
    (enedwaith eriador rhovanion rohan gondor)
    (rohan enedwaith rhovanion rhun gondor mordor)
    (gondor enedwaith rohan mordor)
    (rhun rohan rhovanion khand mordor)
    (mordor gondor rohan rhun khand harad)
    (khand mordor rhun harad)
    (harad mordor khand)))
```

I think someone seems to like Tolkien! Let's start by getting this into a file that we can read. Once you have that, you can use the `]cd` feature to change into the appropriate directory to read the file, or you can just use the full path if you need to.

Problems

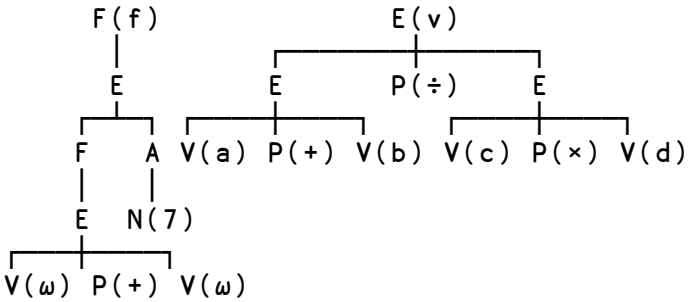
1. Copy the `utf8get` function from the `dfns` workspace into your local workspace so that you can read in the file. Use the `)copy` system command.
2. Use `utf8get` to read the file into a variable that you can work with.
3. Now that we have this, let's see if we can get it into a format that makes a little more sense, start by seeing if you can partition the vector into lines. You'll probably need to look at `⎕UCS` to see how to get the Unicode characters for line endings, and you'll want to find the right primitives for partitioning a string.
4. Now try to convert that nested vector of strings into a character matrix.
5. If you've done this right you'll end up with some lines that you don't need, so drop off the top line and any trailing lines that are blank.
6. Now see about stripping out the parentheses, since these aren't necessary any more.
7. Once you have that removed, you just need to remove the leading whitespace. This is an idiomatic question in APL, and it might be worthwhile to check out the APL Idiom library to see how to do this, or see if you can imagine how you would do it yourself with Scan.

8. Now see if you can convert this matrix of characters into a matrix of strings where each of the strings is one of the names in the set.
9. After you have done this, the first column can be considered the set of all the names, so we can give that it's own name and store it separately.
10. Now take the original matrix and just extract out the connections without the initial associations. If you do this right, the rows in this new matrix will correspond to the elements in the vector you created in #9.
11. Now see if you can build up an adjacency matrix from these two arrays you created in #9 and #10 by testing for membership. You may have to use some operators.
12. This adjacency matrix is what we need to use for our graph coloring. See if you can figure out how to extract the neighbors for any given node.
13. Now see if you can use reduction to do graph coloring by working over a vector of colors for each node, in order. This could be tricky, but play around with it. Recall from Graph theory that you only need four colors total to achieve a coloring in these cases.

Trees/XML/HTML Processing

Manipulating Trees

APL isn't just for flat data. APL has been used with good success on nested and tree data as well. Let's play with that. Consider the following two trees as our working examples:



We'll call the left tree the `F` tree and the right tree the `E` tree. These represent simple ASTs as you might work with in a compiler.

Challenges

1. The first question to ask is how we would want to represent these ASTs in an array language. There are a few different options. Explore using a vector of the depths of the AST, as well as a nested representation. Try a parent vector representation as well.
2. For the rest of these problems, we will be using a depth vector representation in depth-first pre-order traversal. Create and name a vector for each of the above trees to work with.
3. Create equivalent vectors for the type and values of the above trees.
4. Now create a single "AST Matrix" for each tree that contains a column for each of the above fields (depth, type, and value).
5. What is an expression for the count of the nodes in the AST using either the vector or matrix representation?
6. What is an expression for the maximum depth of the AST?
7. Write an expression to extract out the leaf nodes of the AST.
8. Write an expression to change the `V` nodes into `N` nodes, contain some integers instead of variable references.
9. Write an expression to get all of the nodes of a given type from the AST.
10. Write an expression to convert a depth vector to a Boolean matrix representation of the same. For example, the above `F` tree would have the following Boolean depth matrix:

```
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 1
0 0 0 0 1
0 0 1 0 0
0 0 0 1 0
```

Parsing XML

Reference: http://help.dyalog.com/16.0/Content/Language/System%20Functions/xml.htm

Challenges

Extract the data mentioned in https://stackoverflow.com/questions/17056412/parsing-xml-for-deeply-nested-data. A cut-down version of this data, possibly more suitable for experimentation, is listed below:

```
<element1>
    <element2>
        <elementIAmInterestedIn attribute="item 1">
            <element5>
                <otherElementIAmInterestedIn>
                    <data1>text11</data1>
                    <data2>text12</data2>
                </otherElementIAmInterestedIn>
            </element5>
        </elementIAmInterestedIn>
            <elementIAmInterestedIn attribute="item 2">
                <otherElementIAmInterestedIn>
                    <data1>text21</data1>
                </otherElementIAmInterestedIn>
            </elementIAmInterestedIn>
        <elementIAmInterestedIn attribute="item 3">
            <element4>
                <otherElementIAmInterestedIn>
                    <data1>text31</data1>
                    <data2>text32</data2>
                </otherElementIAmInterestedIn>
            </element4>
        </elementIAmInterestedIn>
    </element2>
</element1>
```

Parsing JSON

Reference: http://help.dyalog.com/16.0/Content/Language/System%20Functions/json.htm

Challenges

1. Extract the name and mass of each meteorite logged in https://data.nasa.gov/resource/y77d-th95.json
2. Compute the average and median mass of the recorded meteorites.
3. Extract the ticker data for MSFT from https://www.google.com/finance?q=MSFT&output=json .
4. Use this data to plot the ticker values for MSFT.

Parsing

1. There is a classic APL expression for the computation of the depth of nesting of a parenthesized expression. See if you can figure this one out.
2. See if it works with multiple types of bracketing characters and whether it will work with mixed bracketing characters. What information can you get from this? Are you able to tell if you have unbalanced brackets?
3. Write an expression to tell you what sorts of errors you have in your parentheses, including those of mixed bracket types.
4. Now try extending this idea to handle the parsing of simple single character arithmetic expressions with the standard precedence order.
5. Take the work you have done in #4 and use it to create a `⎕XML` format matrix representing your expression.
6. Now spit this out as XML.
7. Have a go at tokenizing S-exprs.
8. If you can do #7, see if you're up for handling parsing them.
9. Read up on APL Two by Two parsing techniques and see if you can use it to parse the above expressions.
10. Read up on PEG grammars and do the same parsing as above, but this time with PEG parsing combinators that you have written in APL.

Statistics

Here are some basic challenges for doing work with random distributions.

Challenge 1. Write a function to generate n random numbers on the range $[0, n)$ with a uniform distribution.

Challenge 2. Write a new function that will generate n random numbers on a range $[a, b)$ for any arbitrary integers a and b with uniform distribution.

Challenge 3. Write a function to display a histogram of the distribution of random numbers using text.

Challenge 4. Adapt your histogram display to use the `codfns.Histogram` graphics functionality.

Challenge 5. Generate a live display that updates a histogram while generating random values on a uniform distribution so that you can visualize the distribution while the numbers are being generated.

Challenge 6. Write a function equivalent to #1 but with a normal distribution.

Challenge 7. Write a function equivalent to #2 but with a normal distribution.

Challenge 8. Write some functions to demonstrate the effects that common arithmetic operations have on the distribution of random numbers and visualize this graphically.

Databases

Games