Chapter 1

Foundations

1.1 Expressions

1.1.1 What Is An Expression?

In mathematics, an expression is an evaluable collection of symbols constructed with the concept of correctness within a context. In other words, an expression is something which can be evaluated given the right set of symbols and values, which will result in a new meaningful result.

Although this is a rather abstract way of thinking about an expression, it is the basis for how computer science built its own idea of an expression. Even with the great generality which comes with a mathematical definition, we can actually apply this even to everyday programming fairly directly.

If we consider an expression in computer programming, it lines up quite nicely with our mathematical counterpart: a composition of values, operators, constants and functions which is evaluated at run time, and returns a value.

The important idea to consider when we talk about the combination of values, operators, constants and functions, we really are aligning ourselves with the mathematical idea of symbols and evaluable construction. In other words, any expression in computer programming will have some notion of how to correctly assemble language constructs into a new idea which can be compiled or interpreted and executed to give us a result.

Although an expression may return a result and that result will always come from the combination of parts in our expression, there is no assumed notion that our result will be meaningful or useful to us. We will worry about that a little later, though.

In many modern languages, not every line of code is an expression. If we consider flow control structures like if/else and looping blocks, the construct is not, itself, and expression. Although having language structures which are not expressions is perfectly acceptable, it introduces challenges when writing functional code.

1.1.2 Pure Expressions

In the book The Little Schemer, Friedman and Felleisen introduce the concept of s-expressions. In that same vein, this chapter will introduce expressions in Javascript which I will refer to as pure expressions. Although this is not a formal name in the language specification, I believe it will help to differentiate the work we are doing here from other non-expression code which could be written here or in other code. Let's have a look at defining a pure expression.

A pure expression is an expression in Javascript which must adhere to the definition of a computer science definition of an expression. This means, a pure expression must be composed of values, constants, operators and functions. From this we can infer that the following expressions are, in fact, pure expressions as well.

```
17 + 32
'I am a string'.split(' ')
```

Each of these expressions returns a result however, neither of them modify the global working space or produce other environmental effects which are not immediately visible from their output. Expressions like these are important for constructing a functional methodology within Javascript which will help simplify our programs and minimize the number of places unexpected behaviors might sneak in and introduce difficult to diagnose problems.

In contrast to the pure expressions in our example, let's take a look at behaviors which are not expressions. These examples are easy to understand since they do not return a result from their action. Although these behaviors are not expressions, it does not render them useless or unimportant, but it is critical to understand how expressions differ from non-expression behaviors.

```
console.log(42)
var y = 'a value'
```

Clearly, logging a value to the console does not return any meaningful result regarding its action; instead we only get a side effect and nothing more. In much the same way, when we declare a variable with an initial value, we also get no return value. The variable declaration and assignment only affects the environment.

Although logging to the console and declaring variables are not expressions, these are very common behaviors in Javascript programs. Our goal through exploring functional programming in Javascript with pure expressions is to reduce the need for non-expression behaviors. Instead, we should consider code using pure expressions to be the norm, leaving code written in a non-expression way to be limited to particular cases as the need arises.

Our last examples in the world of pure expressions is code which meets the criteria for expressions but are not, in fact, pure expressions. Since pure expressions are a restriction against the definition of expressions, it is possible to write code which meets one definition without meeting the other. Some of the most common examples which behave this way are actually built in functions and core behaviors people rely on every day as they develop applications.

```
function add (a, b) { return a + b; }
myArray.pop()
myArray.reverse();
```

Each of the expressions above actually have a side effect. The most obvious are Array.prototype.pop and Array.prototype.reverse which both return an operation result, but also modify the array they are working on. For the experienced Javascript developer, the side effects which accompany array operations are well known and are completely unsurprising. Unfortunately, for people who are just coming to the language, the fact that some operations modify the array while others do not can be a source of surprise and frustration.

In much the same way that array operations modify the program environment, function declarations also change the environment in a big way. When a function is defined explicitly, it impacts the local scope closest to itself and all enclosing scopes. This modification is, usually, desirable, though it may be surprising to some that creating a named function is, in fact, an expression.

We can, just as easily, create a named function and assign it to a variable in the same line. This action of defining a function and returning a result makes named function declaration an expression, but it definitely excludes it from the pure expression space. Below is an example of function creation and assignment using our add function code from above.

```
var addVar = function add (a, b) { return a + b; };
addVar(5, 6); // 11
```

At this point we should note calling addVar is actually a pure expression, while the assignment of the variable above is not. We can, however, change our named function code into a pure expression with the addition of parentheses around the function declaration. This will reduce our impact on the local scope to the assignment of a single variable. More than that, we will actually isolate the function from our variable name allowing us to do a little name trickery which may be useful later.

```
(function add (a, b) { return a + b; });
add(7, 9); // undefined is not a function ...

var add = (function add (a, b) { return a + b; });
add(7, 9); // 16
```

We can see how wrapping actions in parentheses isolates them from the scope around them. This means we can create an ad-hoc scope around a behavior when we want to execute a line of code and limit its impact on the scope around it. This doesn't isolate our code entirely, but it can reduce unwanted impact. What we have ultimately accomplished is generating a named function without permanently attaching it to the surrounding scope, creating a pure expression where there previously was not one.

1.1.3 Using Pure Expressions

Pure expressions are introduced first in order to set the stage for how we will work to construct programs functionally in Javascript. It is important to understand how pure expressions are built and used as they will help us to build programs which are declarative, clear and concise. Ideally, while we avoid verbosity we also will avoid unnecessary terseness, avoiding code golf simply because we can write something in a single line.

The very nature of functional programming is not simply to write procedural code wrapped in functions, or to exclude the classical object oriented design world simply because we can. Instead we should aim to construct programs which are clear, error-free and declarative. Pure expressions are the means to produce code which brings us closer to that goal.

In order to see what we can achieve with functional programming, using pure expressions, we will perform a simple task. For now we will assume to simply write code in an imperative way in order to construct pure expressions we can use to accomplish small tasks. First, we will look at adding numbers together. We will take an array of numbers and simply add them together.

```
function sum (nums) {
    var result = 0;

    for(var i = 0; i < nums.length; i++) {
        result += nums[i];
    }

    return result;
}

sum([1, 2, 3, 4, 5]); // 15
sum([9, 10, 11]); // 30
```

Clearly sum is not written in a way which uses pure expressions however calling sum is a pure expression since there is a single input, a single output and the environment is not affected by the execution. One of the most important ideas to understand from this is, even if the tools are not immediately available to work exclusively with pure expressions, if we can bundle up the pieces of our code and wrap them to behave as pure expressions, we are a step closer to the goal. We can always come back and improve on an old idea.

Let's take a look at one more piece of code. We will first write it imperatively and then we will do a little work to introduce expressions into the code. We will not look at all of the intermediate steps. Instead it is assumed the move from one implementation to the other is simple enough it should be clear.

```
function getEvens (nums) {
    var result = [];
    for(var i = 0; i < nums.length; i++) {
        if(nums[i] % 2 === 0) {
            result.push(nums[i]);
        }
    }

    return result;
}

getEvens([1, 2, 3, 4, 5]); // [2, 4]
getEvens([1, 3, 5, 7]); // []
```

This kind of development is moving toward a pyramid of doom. Instead, let's rewrite this to be a little more declarative and introduce a pure expression. Although this code will be more modular, it is unlikely our new code could be considered beautiful. Nonetheless, let's have a look.

```
function insertIfEven(result, num) {
    if(num % 2 === 0) { return result.concat([num]); }
    else { return result; }
}

function getEvens (nums) {
    var result = [];

    for(var i = 0; i < nums.length; i++) {
        result = insertIfEven(result, nums[i]);
    }

    return nums;
}

getEvens([1, 2, 3, 4, 5]); // [2, 4]
getEvens([1, 3, 5, 7]); // []
```

Our new code executes and produces the same output, but the implementation is a little different. By introducing the function insertIfEven, we have created a new pure expression which can be used. The getEvens function now updates result on each pass, but our only necessary detail is that we know the value is inserted into an array when even. This kind of declarative programming helps keep the code clear on intent and reduces the amount of mental gymnastics we need to perform at a single level.

1.2 Function Composition

Up to this point we have looked at pure expressions. We can refine our discussion and look at functions which adhere to this same notion of purity. In other words, if we have a pure expressions which involves calling exactly one function, that function must also be pure.

Pure functions are important because, even if they don't exclusively employ pure expressions within the function body, they result in a new way to create pure expressions, which provide all the same benefits we discussed previously. By being able to create pure expressions through calling pure functions, we can begin to construct larger programs from smaller pieces of functionality.

We can create two simple functions which perform basic addition and division, let's call them add and divide. Once we have these new functions, we can start to devise a larger program from them.

```
function add (a, b) {
    return a + b;
}

function divide (a, b) {
    return a / b;
}
```

These functions are, clearly, trivial and could easily be written directly into code anywhere, but they provide the foundation for a very important idea we will explore, function composition. Before we can understand function composition, let's first look at function application.

1.2.1 Function Application

Function application is effectively a mapping from one set to another. A function may be as simple as a hash table lookup In other words if we take the set of numbers 0, 1, 2, 3, 4, and apply the function double(), we get a new set 0, 2, 4, 6, 8, which could employ a simple lookup, or we could actually perform a mathematical operation. Below is the same function written three different ways.

```
function double (x) {
    var doubles = [0, 2, 4, 6, 8];
    return doubles[x];
}

function double (x) {
    return x + x;
}

function double (x) {
    return 2 * x;
}
```