廣東工業大學

本科毕业设计（论文）

外文参考文献译文及原文

学　　院 物理与光电工程学院

专　　业 电子科学与技术

年级班别 2013 级(3)班

学　　号 3113008339

学生姓名 郭桐汕

指导教师 陈国鼎

2017 年 6 月

外文参考文献及原文

郭桐汕

物理与光电工程学院

摘　　要

这是“软件编写”系列文章的第三部分，该系列主要阐述如何在 JavaScript ES6+ 中从零开始学习函数式编程和组合化软件（compositional software）技术（译注：关于软件可组合性的概念，参见维基百科 Composability）。后续还有更多精彩内容，敬请期待！

关键词：JavaScript，EcmaScript 6，JS 函数，JS 函数组合

Abstract

This is part of the “Composing Software” series on learning functional programming and compositional software techniques in JavaScript ES6+ from the ground up. Stay tuned. There’s a lot more of this to come!

Key words：JavaScript，EcmaScript 6, JS Function,JS Function Composition

目　录

1 绪论

对于不熟悉 JavaScript 或 ES6+ 的读者，这里做一个简短的介绍。无论你是 JavaScript 开发新手还是有经验的老兵，你都可能学到一些新东西。以下内容仅是浅尝辄止，吊吊大家的兴致。如果想知道更多，还需深入学习。敬请期待吧。

学习编程最好的方法就是动手编程。我建议您使用交互式 JavaScript 编程环境（如 CodePen 或 Babel REPL）。

或者，您也可以使用 Node 或浏览器控制台 REPL。

2 表达式和值

2.1 表达式和值的含义

表达式是可以求得数据值的代码块。

下面这些都是 JavaScript 中合法的表达式：

```
7;

7 + 1; // 8

7 * 2; // 14

'Hello'; // Hello
```

表达式的值可以被赋予一个名称。执行此操作时，表达式首先被计算，取得的结果值被赋值给该名称。对于这一点我们将使用 `const` 关键字。这不是唯一的方式，但这将是你使用最多的，所以目前我们就可以坚持使用 `const`。

```
const hello = 'Hello';
hello; // Hello
```

3.var、let 和 const

3.1 javascript 变量的声明

JavaScript 支持另外两种变量声明关键字：`var`，还有 `let`。我喜欢根据选择的顺序来考虑它们。默认情况下，我选择最严格的声明方式：`const`。用 `const` 关键字声明的变量不能被重新赋值。最终值必须在声明时分配。这可能听起来很严格，但限制是一件好事。这是个标识在提醒你“赋给这个名称的值将不会改变”。它可以帮你全面了解这个名称的意义，而无需阅读整个函数或块级作用域。

有时，给变量重新赋值很有用。比如，如果你正在写一个手动的强制性迭代，而不是一个更具功能性的方法，你可以迭代一个用 `let` 赋值的计数器。

因为 `var` 能告诉你很少关于这个变量的信息，所以它是最无力的声明标识。自从开始用 ES6，我就再也没在实际软件项目中有意使用 `var` 作声明了。

3.2 javascript 常量的声明

注意一下，一个变量一旦用 `let` 或 `const` 声明，任何再次声明的尝试都将导致报错。如果你在 REPL（读取-求值-输出循环）环境中更喜欢多一些实验性和灵活性，那么建议你使用 `var` 声明变量，与 `let` 和 `const` 不同，使用 `var` 重新声明变量是合法的。

本文将使用 const 来让您习惯于为实际程序中用 `const`，而出于试验的目的自由切换回 `var`。

4.数据类型

4.1 除了 Number 和 String 之外的数据类型

目前为止我们见到了两种数据类型：数字和字符串。JavaScript 也有布尔值（`true` 或 `false`）、数组、对象等。稍后我们再看其他类型。

数组是一系列值的有序列表。可以把它比作一个能够装很多元素的容器。这是一个数组字面量：

```
[1, 2, 3];
```

当然，它也是一个可被赋予名称的表达式：

```
const arr = [1, 2, 3];
```

在 JavaScript 中，对象是一系列键值对的集合。它也有字面量：

```
{
  key: 'value'
}
```

当然，你也可以给对象赋予名称：

```
const foo = {
  bar: 'bar'
}
```

如果你想将现有变量赋值给同名的对象属性，这有个捷径。你可以仅输入变量名，而不用同时提供一个键和一个值：

```
const a = 'a';
const oldA = { a: a }; // 长而冗余的写法
const oA = { a }; // 短小精悍！
```

为了好玩而已，让我们再来一次：

```
const b = 'b';
const oB = { b };
```

对象可以轻松合并到新的对象中：

```
const c = {...oA, ...oB}; // { a: 'a', b: 'b' }
```

这些点是对象扩展运算符。它迭代 `oA` 的属性并分配到新的对象中，`oB` 也是一样，在新对象中已经存在的键都会被重写。在撰写本文时，对象扩展是一个新的试验特性，可能还没有被所有主流浏览器支持，但如果你那不能用，还可以用 `Object.assign()` 替代：

```
const d = Object.assign({}, oA, oB); // { a: 'a', b: 'b' }
```

这个 `Object.assign()` 的例子代码很少，如果你想合并很多对象，它甚至可以节省一些打字。注意当你使用 `Object.assign()` 时，你必须传一个目标对象作为第一个参数。它就是那个源对象的属性将被复制过去的对象。如果你忘了传，第一个参数传递的对象将被改变。

以我的经验，改变一个已经存在的对象而不创建一个新的对象常常引发 bug。至少至少，它很容易出错。要小心使用 `Object.assign()`。

5.解构

对象和数组都支持解构，这意味着你可以从中提取值分配给命过名的变量：

```
const [t, u] = ['a', 'b'];
t; // 'a'
u; // 'b'

const blep = {
   blop: 'blop'
};

//  下面等同于：
// const blop = blep.blop;
const { blop } = blep;
blop; // 'blop'
```

和上面数组的例子类似，你可以一次解构多次分配。下面这行你在大量的 Redux 项目中都能见到。

```
const { type, payload } = action;
```

下面是它在一个 reducer（后面的话题再详细说） 的上下文中的使用方法。

```
const myReducer = (state = {}, action = {}) => {
   const { type, payload } = action;
   switch (type) {
      case 'FOO': return Object.assign({}, state, payload);
      default: return state;
   }
};
```

如果不想为新绑定使用不同的名称，你可以分配一个新名称：

```
const { blop: bloop } = blep;
bloop; // 'blop'
```

读作：把 `blep.blop` 分配给 `bloop`。

6.比较运算符和三元表达式

6.1 严格的等与不等

你可以用严格的相等操作符（有时称为“三等于”）来比较数据值：

```
3 + 1 === 4; // true
```

6.2 宽松的等与不等

还有另外一种宽松的相等操作符。它正式地被称为“等于”运算符。非正式地可以叫“双等于”。双等于有一两个有效的用例，但大多数时候默认使用 `===` 操作符是更好的选择。

6.3 其它比较操作符

> 大于
< 小于
>= 大于或等于
<= 小于或等于
!= 不等于
!== 严格不等于
&& 逻辑与
|| 逻辑或

6.4 三元表达式

三元表达式是一个可以让你使用一个比较器来问问题的表达式，运算出的不同答案取决于表达式是否为真:

```
14 - 7 === 7 ? 'Yep!' : 'Nope.'; // Yep!
```

7.函数

7.1 ES6 箭头函数

JavaScript 支持函数表达式，函数可以这样分配名称：

```
const double = x => x * 2;
```

这和数学表达式 `f(x) = 2x` 是一个意思。大声说出来，这个函数读作 `x` 的 `f` 等于 `2x`。这个函数只有当你用一个具体的 `x` 的值应用它的时候才有意思。在其它方程式里面你写 `f(2)`，就等同于 `4`。

换种说话就是 `f(2) = 4`。您可以将数学函数视为从输入到输出的映射。这个例子里 `f(x)` 是输入数值 `x` 到相应的输出数值的映射，等于输入数值和 `2` 的乘积。

在 JavaScript 中，函数表达式的值是函数本身：

```
double; // [Function: double]
```

你可以使用 `.toString()` 方法看到这个函数的定义。

```
double.toString(); // 'x => x * 2'
```

如果要将函数应用于某些参数，则必须使用函数调用来调用它。函数调用会接收参数并且计算一个返回值。

你可以使用 `<functionName>(argument1, argument2, ...rest)` 调用一个函数。比如调用我们的 double 函数，就加一对括号并传进去一个值：

```
double(2); // 4
```

和一些函数式语言不同，这对括号是有意义的。没有它们，函数将不会被调用。

```
double 4; // SyntaxError: Unexpected number
```

7.1 函数签名

函数的签名可以包含以下内容：

1. 一个 **可选的** 函数名。
2. 在括号里的一组参数。 参数的命名是可选的。
3. 返回值的类型。

JavaScript 的签名无需指定类型。JavaScript 引擎将会在运行时断定类型。如果你提供足够的线索，签名信息也可以通过开发工具推断出来，比如一些 IDE（集成开发环境）和使用数据流分析的 Tern.js。

JavaScript 缺少它自己的函数签名语法，所以有几个竞争标准：JSDoc 在历史上非常流行，但它太过笨拙臃肿，没有人会不厌其烦地维护更新文档与代码同步，所以很多 JS 开发者都弃坑了。

TypeScript 和 Flow 是目前的大竞争者。这二者都不能让我确定地知道怎么表达我需要的一切，所以我使用 Rtype，仅仅用于写文档。一些人倒退回 Haskell 的 curry-only Hindley–Milner 类型系统。如果仅用于文档，我很乐意看到 JavaScript 能有一个好的标记系统标准，但目前为止，我觉得当前的解决方案没有能胜任这个任务的。现在，怪异的类型标记即使和你在用的不尽相同，也就将就先用着吧。

functionName(param1: Type, param2: Type) => Type

double 函数的签名是：

double(x: n) => n

尽管事实上 JavaScript 不需要注释签名，知道何为签名和它意味着什么依然很重要，它有助于你高效地交流函数是如何使用和如何构建的。大多数可重复使用的函数构建工具都需要你传入同样类型签名的函数。

7.2 默认参数值

JavaScript 支持默认参数值。下面这个函数类似一个恒等函数（以你传入参数为返回值的函数），一旦你用 undefined 调用它，或者根本不传入参数——它就会返回 0，来替代：

const orZero = (n = 0) => n;

如上，若想设置默认值，只需在传入参数时带上 `=` 操作符，比如 `n = 0`。当你用这种方式传入默认值，像 Tern.js、Flow、或者 TypeScript 这些类型检测工具可以自行推断函数的类型签名，甚至你不需要刻意声明类型注解。

结果就是这样，在你的编辑器或者 IDE 中安装正确的插件，在你输入函数调用时，你可以看见内联显示的函数签名。依据它的调用签名，函数的使用方法也一目了然。无论起不起作用，使用默认值可以让你写出更具可读性的代码。

注意： 使用默认值的参数不会增加函数的 `.length` 属性，比如使用依赖 `.length` 值的自动柯里化会抛出不可用异常。如果你碰上它，一些柯里化工具（比如 `lodash/curry`）允许你传入自定义参数来绕开这个限制。

7.3 命名参数

JavaScript 函数可以传入对象字面量作为参数，并且使用对象解构来分配参数标识，这样做可以达到命名参数的同样效果。注意，你也可以使用默认参数特性传入默认值。

```
const createUser = ({
  name = 'Anonymous',
  avatarThumbnail = '/avatars/anonymous.png'
}) => ({
  name,
  avatarThumbnail
});

const george = createUser({
  name: 'George',
  avatarThumbnail: 'avatars/shades-emoji.png'
});

george;
/*
{
  name: 'George',
  avatarThumbnail: 'avatars/shades-emoji.png'
}
*/
```

7.4 剩余和展开

JavaScript 中函数共有的一个特性是可以在函数参数中使用剩余操作符 `...` 来将一组剩余的参数聚集到一起。

例如下面这个函数简单地丢弃第一个参数，返回其余的参数：

```
const aTail = (head, ...tail) => tail;
aTail(1, 2, 3); // [2, 3]
```

剩余参数将各个元素组成一个数组。而展开操作恰恰相反：它将一个数组中的元素扩展为独立元素。研究一下这个：

```
const shiftToLast = (head, ...tail) => [...tail, head];
shiftToLast(1, 2, 3); // [2, 3, 1]
```

JavaScript 数组在使用扩展操作符的时候会调用一个迭代器，对于数组中的每一个元素，迭代器都会传递一个值。在 [...tail, head] 表达式中，迭代器按顺序从 `tail` 数组中拷贝到一个刚刚创建的新的数组。之前 head 已经是一个独立元素了，我们只需把它放到数组的末端，就完成了。

7.5 柯里化

可以通过返回另一个函数来实现柯里化（Curry）和偏应用（partial application）：

```
const highpass = cutoff => n => n >= cutoff;
const gt4 = highpass(4); // highpass() 返回了一个新函数
```

你可以不使用箭头函数。JavaScript 也有一个 function 关键字。我们使用箭头函数是因为 function 关键字需要打更多的字。
这种写法和上面的 highPass() 定义是一样的：

```
const highpass = function highpass(cutoff) {
   return function (n) {
      return n >= cutoff;
   };
};
```

JavaScript 中箭头的大致意义就是“函数”。使用不同种的方式声明，函数行为会有一些重要的不同点（=> 缺少了它自己的 this，不能作为构造函数），但当我们遇见那就知道不同之处了。现在，当你看见 x => x，想到的是 “一个携带 x 并且返回 x 的函数”。所以 const highpass = cutoff => n => n >= cutoff; 可以这样读：“highpass 是一个携带 cutoff 返回一个携带 n 并返回结果 n >= cutoff 的函数的函数”

既然 `highpass()` 返回一个函数，你可以使用它创建一个更独特的函数：

```
const gt4 = highpass(4);
gt4(6); // true
gt4(3); // false
```

自动柯里化函数，有利于获得最大的灵活性。比如你有一个函数 add3():

```
const add3 = curry((a, b, c) => a + b + c);
```

使用自动柯里化，你可以有很多种不同方法使用它，它将根据你传入多少个参数返回正确结果：

```
add3(1, 2, 3); // 6
add3(1, 2)(3); // 6
add3(1)(2, 3); // 6
add3(1)(2)(3); // 6
```

令 Haskell 粉遗憾的是，JavaScript 没有内置自动柯里化机制，但你可以从 Lodash 引入：

$ npm install --save lodash

然后在你的模块里:

import curry from 'lodash/curry';

或者你可以使用下面这个魔性写法:

```
// 精简的递归自动柯里化
const curry = (
  f, arr = []
) => (...args) => (
  a => a.length === f.length ?
    f(...a) :
    curry(f, a)
)([...arr, ...args]);
```

7.6 函数组合

当然你能够开始组合函数了。组合函数是传入一个函数的返回值作为参数给另一个函数的过程。用数学符号标识：

f . g

翻译成 JavaScript:

f(g(x))

这是从内到外地求值：

1. `x` 是被求数值
2. `g()` 应用给 `x`
3. `f()` 应用给 `g(x)` 的返回值

例如:

```
const inc = n => n + 1;
inc(double(2)); // 5
```

数值 `2` 被传入 double()，求得 `4`。 `4` 被传入 inc() 求得 `5`。

你可以给函数传入任何表达式作为参数。表达式在函数应用之前被计算:

```
inc(double(2) * double(2)); // 17
```

既然 double(2) 求得 4，你可以读作 inc(4 * 4)，然后计算得 inc(16)，然后求得 17。函数组合是函数式编程的核心。我们后面还会介绍很多。

8.数组

8.1 数组内置方法

数组有一些内置方法。方法是指对象关联的函数，通常是这个对象的属性：

```
const arr = [1, 2, 3];
arr.map(double); // [2, 4, 6]
```

这个例子里，`arr` 是对象，`.map()` 是一个以函数为值的对象属性。当你调用它，这个函数会被应用给参数，和一个特别的参数叫做 `this`，`this` 在方法被调用之时自动设置。这个 `this` 的存在使 `.map()` 能够访问数组的内容。

注意我们传递给 `map` 的是 `double` 函数而不是直接调用。因为 `map` 携带一个函数作为参数并将函数应用给数组的每一个元素。它返回一个包含了 `double()` 返回值的新的数组。

注意原始的 arr 值没有改变：
arr; // [1, 2, 3]

8.2 方法链

你也可以链式调用方法。方法链是指在函数返回值上直接调用方法的过程，在此期间不需要给返回值命名：

```
const arr = [1, 2, 3];
arr.map(double).map(double); // [4, 8, 12]
```

返回布尔值（true 或 false）的函数叫做 断言（predicate）。.filter() 方法携带断言并返回一个新的数组，新数组中只包含传入断言函数（返回 true）的元素：

[2, 4, 6].filter(gt4); // [4, 6]

你常常会想要从一个列表选择一些元素，然后把这些元素序列化到一个新列表中：

[2, 4, 6].filter(gt4).map(double); [8, 12]

注意：后面的文章你将看到使用叫做 transducer 东西更高效地同时选择元素并序列化，不过这之前还有一些其他东西要了解。

总　结

如果你现在有点发懵，不必担心。我们仅仅概览了一下很多事情的表面，它们尚需大量的解释和总结。很快我们就会回过头来，深入探讨其中的一些话题。

I. introduction

For those unfamiliar with JavaScript or ES6+, this is intended as a brief introduction. Whether you're a beginner or experienced JavaScript developer, you may learn something new.

The following is only meant to scratch the surface and get you excited. If you want to know more, you'll just have to explore deeper. There's a lot more ahead.

The best way to learn to code is to code. I recommend that you follow along using an interactive JavaScript programming environment such as CodePenor the Babel REPL.

Alternatively, you can get away with using the Node or browser console REPLs.

II. Expressions and Values

An expression is a chunk of code that evaluates to a value.
The following are all valid expressions in JavaScript:

```
7;
7 + 1; // 8
7 * 2; // 14
'Hello'; // Hello
```

The value of an expression can be given a name. When you do so, the expression is evaluated first, and the resulting value is assigned to the name. For this, we'll use the const keyword. It's not the only way, but it's the one you'll use most, so we'll stick with const for now:

```
const hello = 'Hello';
hello; // Hello
```

III. var, let, and const

JavaScript supports two more variable declaration keywords: var, and let. I like to think of them in terms of order of selection. By default, I select the strictest declaration: const. A variable declared with the const keyword can't be reassigned. The final value must be assigned at declaration time. This may sound rigid, but the restriction is a good thing. It's a signal that tells you, "the value assigned to this name is not going to change". It helps you fully understand what the name means right away, without needing to read the whole function or block scope.

Sometimes it’s useful to reassign variables. For example, if you’re using manual, imperative iteration rather than a more functional approach, you can iterate a counter assigned with let.

Because var tells you the least about the variable, it is the weakest signal. Since I started using ES6, I have never intentionally declared a var in a real software project.

Be aware that once a variable is declared with let or const, any attempt to declare it again will result in an error. If you prefer more experimental flexibility in the REPL (Read, Eval, Print Loop) environment, you may use var instead of const to declare variables. Redeclaring var is allowed.

This text will use const in order to get you in the habit of defaulting to const for actual programs, but feel free to substitute var for the purpose of interactive experimentation.

IV. Types

So far we've seen two types: numbers and strings. JavaScript also has booleans (true or false), arrays, objects, and more. We'll get to other types later.

An array is an ordered list of values. Think of it as a box that can hold many items. Here's the array literal notation:

```
[1, 2, 3];
```

Of course, that's an expression which can be given a name:

```
const arr = [1, 2, 3];
```

An object in JavaScript is a collection of key: value pairs. It also has a literal notation:

```
{
  key: 'value'
}
```

And of course, you can assign an object to a name:

```
const foo = {
  bar: 'bar'
}
```

If you want to assign existing variables to object property keys of the same name, there's a shortcut for that. You can just type the variable name instead of providing both a key and a value:

```
const a = 'a';
const oldA = { a: a }; // long, redundant way
const oA = { a }; // short an sweet!
Just for fun, let's do that again:
const b = 'b';
const oB = { b };
```

Objects can be easily composed together into new objects:

```
const c = {...oA, ...oB}; // { a: 'a', b: 'b' }
```

Those dots are the object spread operator. It iterates over the properties in oA and assigns them to the new object, then does the same for oB, overriding any keys that already exist on the new object. As of this writing, object spread is a new, experimental feature that may not be

available in all the popular browsers yet, but if it's not working for you, there is a substitute: Object.assign():

```
const d = Object.assign({}, oA, oB); // { a: 'a', b: 'b' }
```

Only a little more typing in the Object.assign() example, and if you're composing lots of objects, it may even save you some typing. Note that when you use Object.assign(), you must pass a destination object as the first parameter. It is the object that properties will be copied to. If you forget, and omit the destination object, the object you pass in the first argument will be mutated.

In my experience, mutating an existing object rather than creating a new object is usually a bug. At the very least, it is error-prone. Be careful with Object.assign().

V.Destructuring

Both objects and arrays support destructuring, meaning that you can extract values from them and assign them to named variables:

```
const [t, u] = ['a', 'b'];
t; // 'a'
u; // 'b'
const blep = {
   blop: 'blop'
};

// The following is equivalent to:
// const blop = blep.blop;
const { blop } = blep;
blop; // 'blop'
```

As with the array example above, you can destructure to multiple assignments at once. Here's a line you'll see in lots of Redux projects:

```
const { type, payload } = action;
```

Here's how it's used in the context of a reducer (much more on that topic coming later):

```
const myReducer = (state = {}, action = {}) => {
const { type, payload } = action;
switch (type) {
      case 'FOO': return Object.assign({}, state, payload);
     default: return state;
   }
};
```

If you don't want to use a different name for the new binding, you can assign a new name:

```
const { blop: bloop } = blep;
bloop; // 'blop'
```

Read: Assign blep.blop as bloop.

VI. Comparisons and Ternaries

You can compare values with the strict equality operator (sometimes called “triple equals”):

```
3 + 1 === 4; // true
```

There’s also a sloppy equality operator. It’s formally known as the “Equal” operator. Informally, “double equals”. Double equals has a valid use-case or two, but it’s almost always better to default to the === operator, instead.

Other comparison operators include:

> Greater than
< Less than
>= Greater than or equal to
<= Less than or equal to
!= Not equal
!== Not strict equal
&& Logical and
|| Logical or

A ternary expression is an expression that lets you ask a question using a comparator, and evaluates to a different answer depending on whether or not the expression is truthy:

```
14 - 7 === 7 ? 'Yep!' : 'Nope.'; // Yep!
```

VII. Functions

JavaScript has function expressions, which can be assigned to names:

```
const double = x => x * 2;
```

This means the same thing as the mathematical function f(x) = 2x. Spoken out loud, that function reads f of x equals 2x. This function is only interesting when you apply it to a specific value of x. To use the function in other equations, you'd write f(2), which has the same meaning as 4.

In other words, f(2) = 4. You can think of a math function as a mapping from inputs to outputs. f(x) in this case is a mapping of input values for xto corresponding output values equal to the product of the input value and 2.

In JavaScript, the value of a function expression is the function itself:

```
double; // [Function: double]
```

You can see the function definition using the .toString() method:

```
double.toString(); // 'x => x * 2'
```

If you want to apply a function to some arguments, you must invoke it with a function call. A function call applies a function to its arguments and evaluates to a return value.

You can invoke a function using <functionName>(argument1, argument2, ...rest). For example, to invoke our double function, just add the parentheses and pass in a value to double:

```
double(2); // 4
```

Unlike some functional languages, those parentheses are meaningful. Without them, the function won't be called:

```
double 4; // SyntaxError: Unexpected number
```

VII.I Signatures

Functions have signatures, which consist of:

- An optional function name.
- A list of parameter types, in parentheses. The parameters may optionally be named.
- The type of the return value.

Type signatures don't need to be specified in JavaScript. The JavaScript engine will

figure out the types at runtime. If you provide enough clues, the signature can also be inferred by developer tools such as IDEs (Integrated Development Environment) and Tern.js using data flow analysis.

JavaScript lacks its own function signature notation, so there are a few competing standards: JSDoc has been very popular historically, but it's awkwardly verbose, and nobody bothers to keep the doc comments up-to-date with the code, so many JS developers have stopped using it.

TypeScript and Flow are currently the big contenders. I'm not sure how to express everything I need in either of those, so I use Rtype, for documentation purposes only. Some people fall back on Haskell's curry-only Hindley–Milner types. I'd love to see a good notation system standardized for JavaScript, if only for documentation purposes, but I don't think any of the current solutions are up to the task, at present. For now, squint and do your best to keep up with the weird type signatures which probably look slightly different from whatever you're using.

```
functionName(param1: Type, param2: Type) => Type
```

The signature for double is:

```
double(x: n) => n
```

In spite of the fact that JavaScript doesn't require signatures to be annotated, knowing what signatures are and what they mean will still be important in order to communicate efficiently about how functions are used, and how functions are composed. Most reusable function composition utilities require you to pass functions which share the same type signature.

VII.II Default Parameter Values

JavaScript supports default parameter values. The following function works like an identity function (a function which returns the same value you pass in), unless you call it with undefined, or simply pass no argument at all -- then it returns zero, instead:

```
const orZero = (n = 0) => n;
```

To set a default, simply assign it to the parameter with the = operator in the function signature, as in n = 0, above. When you assign default values in this way, type inference tools such as Tern.js, Flow, or TypeScript can infer the type signature of your function automatically, even if you don't explicitly declare type annotations.

The result is that, with the right plugins installed in your editor or IDE, you'll be able to see function signatures displayed inline as you're typing function calls. You'll also be able to understand how to use a function at a glance based on its call signature. Using default assignments wherever it makes sense can help you write more self-documenting code.

Note: Parameters with defaults don't count toward the function's .lengthproperty, which will

throw off utilities such as autocurry which depend on the .length value. Some curry utilities (such as lodash/curry) allow you to pass a custom arity to work around this limitation if you bump into it.

VII.III Named Arguments

JavaScript functions can take object literals as arguments and use destructuring assignment in the parameter signature in order to achieve the equivalent of named arguments. Notice, you can also assign default values to parameters using the default parameter feature:

```
const createUser = ({
  name = 'Anonymous',
  avatarThumbnail = '/avatars/anonymous.png'
}) => ({
  name,
  avatarThumbnail
});
const george = createUser({
  name: 'George',
  avatarThumbnail: 'avatars/shades-emoji.png'
});
george;
/*
{
  name: 'George',
  avatarThumbnail: 'avatars/shades-emoji.png'
}
*/
```

VII.IV Rest and Spread

A common feature of functions in JavaScript is the ability to gather together a group of remaining arguments in the functions signature using the rest operator: ...
For example, the following function simply discards the first argument and returns the rest as an array:

```
const aTail = (head, ...tail) => tail;
aTail(1, 2, 3); // [2, 3]
```

Rest gathers individual elements together into an array. Spread does the opposite: it spreads the elements from an array to individual elements. Consider this:

```
const shiftToLast = (head, ...tail) => [...tail, head];
shiftToLast(1, 2, 3); // [2, 3, 1]
```

Arrays in JavaScript have an iterator that gets invoked when the spread operator is used. For each item in the array, the iterator delivers a value. In the expression, [...tail, head], the iterator copies each element in order from the tail array into the new array created by the surrounding literal notation. Since the head is already an individual element, we just plop it onto the end of the array and we're done.

VII.V Currying

Curry and partial application can be enabled by returning another function:

```
const highpass = cutoff => n => n >= cutoff;
const gt4 = highpass(4); // highpass() returns a new function
```

You don’t have to use arrow functions. JavaScript also has a functionkeyword. We're using arrow functions because the function keyword is a lot more typing. This is equivalent to the highPass() definition, above:

```
const highpass = function highpass(cutoff) {
   return function (n) {
      return n >= cutoff;
   };
};
```

The arrow in JavaScript roughly means “function”. There are some important differences in function behavior depending on which kind of function you use (=> lacks its own this, and can't be used as a constructor), but we'll get to those differences when we get there. For now, when you see x => x, think "a function that takes x and returns x". So you can read const highpass = cutoff => n => n >= cutoff; as:

“highpass is a function which takes cutoff and returns a function which takes n and returns the result of n >= cutoff".

Since highpass() returns a function, you can use it to create a more specialized function:

```
const gt4 = highpass(4);
gt4(6); // true
gt4(3); // false
```

Autocurry lets you curry functions automatically, for maximal flexibility. Say you have a function add3():

```
const add3 = curry((a, b, c) => a + b + c);
```

With autocurry, you can use it in several different ways, and it will return the right thing

depending on how many arguments you pass in:

```
add3(1, 2, 3); // 6
add3(1, 2)(3); // 6
add3(1)(2, 3); // 6
add3(1)(2)(3); // 6
```

Sorry Haskell fans, JavaScript lacks a built-in autocurry mechanism, but you can import one from Lodash:

```
$ npm install --save lodash
```

Then, in your modules:

```
import curry from 'lodash/curry';
```

Or, you can use the following magic spell:

```
// Tiny, recursive autocurry
const curry = (
  f, arr = []
) => (...args) => (
  a => a.length === f.length ?
    f(...a) :
    curry(f, a)
)([...arr, ...args]);
```

VII.VI Function Composition

Of course you can compose functions. Function composition is the process of passing the return value of one function as an argument to another function. In mathematical notation:

```
f . g
```

Which translates to this in JavaScript:

```
f(g(x))
```

It's evaluated from the inside out:

- x is evaluated
- g() is applied to x
- f() is applied to the return value of g(x)

For example:

```
const inc = n => n + 1;
inc(double(2)); // 5
```

The value 2 is passed into double(), which produces 4. 4 is passed into inc() which evaluates to 5.

You can pass any expression as an argument to a function. The expression will be evaluated before the function is applied:

```
inc(double(2) * double(2)); // 17
```

Since double(2) evaluates to 4, you can read that as inc(4 * 4) which evaluates to inc(16) which then evaluates to 17.

Function composition is central to functional programming. We'll have a lot more on it later.

VIII. Arrays

VIII.I Array.prototype

Arrays have some built-in methods. A method is a function associated with an object: usually a property of the associated object:

```
const arr = [1, 2, 3];
arr.map(double); // [2, 4, 6]
```

In this case, arr is the object, .map() is a property of the object with a function for a value. When you invoke it, the function gets applied to the arguments, as well as a special parameter called this, which gets automatically set when the method is invoked. The this value is how .map() gets access to the contents of the array.

Note that we're passing the double function as a value into map rather than calling it. That's because map takes a function as an argument and applies it to each item in the array. It returns a new array containing the values returned by double().

Note that the original arr value is unchanged:

```
arr; // [1, 2, 3]
```

VIII.II Method Chaining

You can also chain method calls. Method chaining is the process of directly calling a method on the return value of a function, without needing to refer to the return value by name:

```
const arr = [1, 2, 3];
arr.map(double).map(double); // [4, 8, 12]
```

A predicate is a function that returns a boolean value (true or false). The .filter() method takes a predicate and returns a new list, selecting only the items that pass the predicate (return true) to be included in the new list:

```
[2, 4, 6].filter(gt4); // [4, 6]
```

Frequently, you'll want to select items from a list, and then map those items to a new list:

```
[2, 4, 6].filter(gt4).map(double); [8, 12]
```

Note: Later in this text, you'll see a more efficient way to select and map at the same time using something called a transducer, but there are other things to explore first.

Conclusion

If your head is spinning right now, don’t worry. We barely scratched the surface of a lot of things that deserve a lot more exploration and consideration. We’ll circle back and explore some of these topics in much more depth, soon.