JavaScript in Ten Minutes

Spencer Tipping

May 9, 2010

Contents

1 Types

JavaScript has nine types. They are:

1. Null – `null`. Chucks a wobbly if you ask it for any attributes; e.g. `null.foo` fails. Never boxed.[1]
2. Undefined – `undefined`. What you get if you ask an object for something it doesn't have; e.g. `document.nonexistent`. Also chucks a wobbly if you ask it for any attributes. Never boxed.
3. Strings – e.g. `'foo'`, `"foo"` (single vs. double quotation marks makes no difference). Sometimes boxed. Instance of `String`.
4. Numbers – e.g. `5`, `3e+10` (all numbers behave as floats – significant for division, but can be truncated by `x >>> 0`). Sometimes boxed. Instance of `Number`.
5. Booleans – `true` and `false`. Sometimes boxed. Instance of `Boolean`.
6. Arrays – e.g. `[1, 2, "foo", [3, 4]]`. Always boxed. Instance of `Array`.
7. Objects – e.g. `{foo: 'bar', bif: [1, 2]}`, which are really just hashtables. Always boxed. Instance of `Object`.
8. Regular expressions – e.g. `/foo\s*([bar]+)/`. Always boxed. Instance of `RegExp`.
9. Functions – e.g. `function (x) {return x + 1}`. Always boxed. Instance of `Function`.

The value `null` is actually almost never produced by JavaScript. The only case you're likely to run across `null` is if you assign it somewhere (most of the time you'll get `undefined` instead – one notable exception is `document.getElementById`, which returns `null` if it can't find an element). Making sparing use of `undefined` and instead using `null` can make bugs much easier to track down.

[1] Boxing is just a way of saying whether something has a pointer. A boxed type is a reference type, and an unboxed type is a value type. In JavaScript, this has additional ramifications as well – see section 3.6.

2 Functions

Functions are first-class lexical closures,[2] just like `lambda`s in Ruby or `sub`s in Perl.[3] They behave pretty much like you'd expect, but there are several really cool things about functions and one really egregious disaster.

2.1 Variadic behavior (a cool thing)

Functions are always variadic.[4] Formal parameters are bound if they're present; otherwise they're `undefined`. For example:

```
(function (x, y) {return x + y}) ('foo')        // => 'fooundefined'
```

The arguments to your function can be accessed in a first-class way, too:

```
var f = function () {return arguments[0] + arguments[1]};
var g = function () {return arguments.length};
f ('foo')                    // => 'fooundefined'
g (null, false, undefined)   // => 3
```

The `arguments` *keyword is not an array!* It just looks like one. In particular, doing any of these will cause problems:

```
arguments.concat ([1, 2, 3])
[1, 2, 3].concat (arguments)
arguments.push ('foo')
arguments.shift ()
```

To get an array from the `arguments` object, you can say `Array.prototype.slice.call (arguments)`. As far as I know that's the best way to go about it.

2.2 Lazy scoping (a cool thing)

Internally, functions use a lexical scoping chain. However, the variables inside a function body aren't resolved until the function is called. This has some really nice advantages, perhaps foremost among them self-reference:

```
var f = function () {return f};
f () === f              // => true
```

[2] First-class in the sense that you can pass them around as values at runtime. You can't reliably introspect them, however, because while you can obtain their source code via `toString` you won't be able to access the values they close over.

[3] Note that block scoping isn't used – the only scopes that get introduced are at function boundaries.

[4] The number of arguments a function accepts is referred to as its *arity*. So a unary function, which is monadic, takes one, a binary function, which is dyadic, takes two, etc. A function that takes any number of arguments is said to be variadic.

<hacker-stuff> Another nice thing is that you can create functions that refer to variables that might never exist. (Some people would consider this a non-feature.) This means that JavaScript can be made to support syntactic macros via the `toString` method:

```
var f = function () {return $0 + $1};
var g = eval (f.toString ().replace (/\$(\d+)/g,
          function (_, digits) {return 'arguments[' + digits + ']'}));
g (5, 6)        // => 11
```

Theoretically by extending this principle one could implement true structural macros, operator overloading, a type system,[5] or other things.</hacker-stuff>

2.3 The meaning of `this` (the egregious disaster)

One would think it is a simple matter to figure out what `this` is, but it's apparently quite challenging, and JavaScript makes it look nearly impossible. Outside of functions (in the global scope, that is), the word `this` refers to the *global object*, which is `window` in a browser. The real question is how it behaves inside a function, and that is determined entirely by how the function is called. Here's how that works:

1. If the function is called alone, e.g. `foo(5)`, then inside that function's body the word `this` will be equivalent to the global object.

2. If the function is called as a method, e.g. `x.foo(5)`, then inside that function's body the word `this` refers to the object, in this case `x`.

3. If the function starts off as a method and then is called alone:

   ```
   var f = x.foo;
   f (5);
   ```

   then `this` will be the global object again. Nothing is remembered about where `f` came from; it is all determined right at the invocation site.

4. If the function is invoked using `apply` or `call`, then `this` points to whatever you set it to (unless you try to set it to `null` or `undefined`, in which case it will be the global object again):

   ```
   var f = function () {return this};
   f.call (4)          // => 4
   f.call (0)          // => 0
   f.call (false)      // => false
   f.call (null)       // => [object global]
   ```

[5] God forbid.

Given this unpredictability, most JavaScript libraries provide a facility to set a function's `this` binding (referred to within JavaScript circles as just a function's binding) to something invocation-invariant. The easiest way to do this is to define a function that proxies arguments using `apply` and closes over the proper value (luckily, closure variables behave normally):

```
var bind = function (f, this_value) {
  return function () {return f.apply (this_value, arguments)};
};
```

The difference between `call` and `apply` is straightforward: `f.call (x, y, z)` is the same as `f.apply (x, [y, z])`, which is the same as `bind (f, x) (y, z)`. That is, the first argument to both `call` and `apply` becomes `this` inside the function, and the rest are passed through. In the case of `apply` the arguments are expected in an array-like thing (`arguments` works here), and in the case of `call` they're passed in as given.

2.3.1 Important consequence: eta-reduction

In most functional programming languages, you can eta-reduce things; that is, if you have a function of the form `function (x) {return f (x)}`, you can just use `f` instead. But in JavaScript that's not always a safe transformation; consider this code:

```
Array.prototype.each = function (f) {
  for (var i = 0, l = this.length; i < l; ++i)
    f (this[i]);
};

var xs = [];
some_array.each (function (x) {xs.push (x)});
```

It might be tempting to rewrite it more concisely as:

```
some_array.each (xs.push);
```

however this latter form will result in a mysterious JavaScript error when the native `Array.push` function finds `this` to be the global object instead of `xs`. The reason should be apparent: when the function is called inside `each`, it is invoked as a function instead of a method. The fact that the function started out as a method on `xs` is forgotten. (Just like case 3 above.)

The simplest way around this is to bind `xs.push` to `xs`:

```
some_array.each (bind (xs.push, xs));
```

3 Gotchas

JavaScript is an awesome language just like Perl is an awesome language and Linux is an awesome operating system. If you know how to use it properly, it will solve all of your problems trivially (well, almost), and if you miss one of its subtleties you'll spend hours hunting down bugs. I've collected the things I've run into here, which should cover most of JavaScript's linguistic pathology.[6]

3.1 Semicolon inference

You won't run into any trouble if you always end lines with semicolons. However, most browsers consider it to be optional, and there is one potential surprise lurking if you choose to omit them.

Most of the time JavaScript does what you mean. The only case where it might not is when you start a line with an open-paren, like this:

```
var x = f
(y = x) (5)
```

JavaScript joins these two lines, forming:

```
var x = f (y = x) (5)
```

The only way around this that I know of is to put a semicolon on the end of the first line.

3.2 Void functions

Every function returns a value. If you don't use a `return` statement, then your function returns `undefined`; otherwise it returns whatever you tell it to. This can be a common source of errors for people used to Ruby or Lisp; for instance,

```
var x = (function (y) {y + 1}) (5);
```

results in `x` being `undefined`. If you're likely to make this slip, there's an Emacs mode called "js2-mode" that identifies functions with no side-effects or return values, and it will catch most of these errors.

3.3 `var`

Be careful how you define a variable. If you leave off the `var` keyword, your variable will be defined in the global scope, which can cause some very subtle bugs:

[6] There is plenty of this pathology despite JavaScript being generally an excellent language. This makes it ideal both for people who want to get things done, and for bug-connoisseurs such as myself.

```
var f = function () {         // f is toplevel, so global
  var x = 5;                  // x is local to f
  y = 6;                      // y is global
};
```

As far as I know, the same is true in both types of `for` loop:

```
for (i = 0; i < 10; ++i)      // i is global
for (var i = 0; i < 10; ++i)  // i is local to the function
for (k in some_object)        // k is global
for (var k in some_object)    // k is local to the function
```

3.4 Lazy scoping and mutability

This is a beautiful disaster. Check this out:

```
var fs = [];
for (var i = 0; i < 3; ++i)
  fs.push (function () {return i});
var xs = [];
for (var j = 0; j < 3; ++j)
  xs.push (fs[j]());
xs          // what will this be?
```

A reasonable conclusion is that `xs` is `[0, 1, 2]`, since those were the values of `i` when the functions were created. However, because of lazy scoping, `xs` will in fact have the value `[3, 3, 3]`. The reason for this is that at the time of invocation, each function will see the current value of `i`, which is 3. (Interestingly, rewriting the second `for` loop to use `i` rather than `j` will produce the expected output!)

The simplest way around this is to introduce a layer of scope for each iteration:

```
for (var i = 0; i < 3; ++i)
  (function (x) {fs.push (function () {return x})}) (i);
```

The inner `x` will be preserved by the reference from the function that gets pushed onto the array, and because it is in a separate scope its value will be invariant with future changes to the outer `i`. (Basically, we're forcing `i` to be evaluated immediately, since it's now a function parameter.) Note, however, that the following code still won't work:

```
for (var i = 0; i < 3; ++i) {
  var x = i;
  fs.push (function () {return x});
}
```

because `x` will be scoped to the nearest enclosing function; so its value will change just as frequently as the value of `i`.

3.5 Equality

Because the == is lame, these are all true in JavaScript:

```
null  == undefined
null  == 0
false == ''
''    == 0
true  == 1
true  == '1'
'1'   == 1
```

So, *never use the == operator unless you really want this behavior*. Instead, use === (whose complement is !==), which behaves sensibly. In particular, === requires both operands to not only be the same-ish, but also be of the same type. It does referential comparison for boxed values and structural comparison for unboxed values (and might fudge some of the sometimes-boxed values; I'm not sure about all of the edge cases). In particular, it's safe to use on strings: `'foo' === 'f' + 'oo'`.

3.6 Boxed vs. unboxed

Boxed values can store properties. Unboxed values will silently fail to store them; for example:

```
var x = 5;
x.foo = 'bar';
x.foo     // => undefined; x is an unboxed number.

var x = new Number (5);
x.foo = 'bar';
x.foo     // => 'bar'; x is a pointer.
```

How does a sometimes-boxed value acquire a box? When you do one of these things:

1. Call its constructor directly, as we did above
2. Set a member of its prototype and refer to `this` inside that method (see section 4)

All HTML objects, whether or not they're somehow native, will be boxed.

3.7 Things that will silently fail or misbehave

JavaScript is very lenient about what you can get away with. In particular, the following are all perfectly legal:

```
[1, 2, 3].foo     // => undefined
[1, 2, 3][4]      // => undefined
1 / 0             // => Infinity
0 * 'foo'         // => NaN
```

This can be very useful. A couple of common idioms are things like these:

```
e.nodeType || (e = document.getElementById (e));
options.foo = options.foo || 5;
```

Also, the language will convert *anything* to a string or number if you use +. All of these expressions are strings:

```
null + [1, 2]             // => 'null1,2'
undefined + [1, 2]        // => 'undefined1,2'
3 + {}                    // => '3[object Object]'
'' + true                 // => 'true'
```

And all of these are numbers:

```
undefined + undefined     // => NaN
undefined + null          // => NaN
null + null               // => 0
{} + {}                   // => NaN
true + true               // => 2
0 + true                  // => 1
```

And some of my favorites:

```
null * false + (true * false) + (true * true)     // => 1
true << true << true                              // => 4
true / null                                       // => Infinity
```

3.8 Things that will loudly fail

There is a point where JavaScript will complain. If you call a non-function, ask for a property of `null` or `undefined`, or refer to a global variable that doesn't exist,[7] then JavaScript will throw a `TypeError` or `ReferenceError`.

3.9 Throwing things

You can `throw` a lot of different things, including unboxed values. This can have some advantages; in this code for instance:

[7] To get around the error for this case, you can say `typeof foo`, where `foo` is the potentially nonexistent global. It will return `'undefined'` if `foo` hasn't been defined (or contains the value `undefined`).

```
try {
  ...
  throw 3;
} catch (n) {
  // n has no stack trace!
}
```

the `throw`/`catch` doesn't compute a stack trace, making exception processing quite a bit faster than usual. But for debugging, it's much better to throw a proper error:

```
try {
  ...
  throw new Error(3);
} catch (e) {
  // e has a stack trace, useful in Firebug among other things
}
```

3.10 Don't use `typeof`

Because it behaves like this:

```
typeof function () {}     // => 'function'
typeof [1, 2, 3]          // => 'object'
typeof {}                 // => 'object'
typeof null               // => 'object'
typeof typeof             // hangs forever in Firefox
```

`typeof` is a really lame way to detect the type of something. Better is to use an object's `constructor` property, like this:

```
(function () {}).constructor      // => Function
[1, 2, 3].constructor             // => Array
({}).constructor                  // => Object
true.constructor                  // => Boolean
null.constructor                  // TypeError: null has no properties
```

In order to defend against `null` and `undefined` (neither of which let you ask for their constructor), you might try to rely on the falsity of these values:

```
x && x.constructor
```

But in fact that will fail for `''`, `0`, and `false`. The only way I know to get around this is to just do the comparison:

```
x === null || x === undefined ? x : x.constructor
```

Alternatively, if you just want to find out whether something is of a given type, you can just use `instanceof`, which never throws an exception.[8]

[8] Well, almost. If you ask for it by putting `null`, `undefined`, or similarly inappropriate things on the right-hand side you'll get a `TypeError`.

3.11 Browser incompatibilities

Generally browsers since IE6 have good compatibility for core language stuff. One notable exception, however, is an IE bug that affects `String.split`:

```
var xs = 'foo bar bif'.split (/(\s+)/);
xs          // on reasonable browsers: ['foo', ' ', 'bar', ' ', 'bif']
xs          // on IE: ['foo', 'bar', 'bif']
```

I'm sure there are other similar bugs out there, though the most common ones to cause problems are generally in the DOM.[9]

4 Prototypes

Are overrated. Prototype-based OOP is almost as miserable as most other forms of OOP, and I never use it. But sometimes you need to do things with prototypes, so here's the basic idea.

Whenever you define a function, it serves two purposes. It can be what every normal programmer assumes a function is – that is, it can take values and return values, or it can be a mutant instance-generating thing that does something completely different. Here's an example:

```
// A normal function:
var f = function (x) {return x + 1};
f (5)     // => 6
```

This is what most people expect. Here's the mutant behavior that almost nobody expects:

```
// A constructor function
var f = function (x) {this.x = x + 1};    // no return!
var i = new f (5);                        // i.x = 6
```

The following things are true at this point:

```
i.constructor === f
i.__proto__ === i.constructor.prototype   // on Firefox, anyway
i instanceof f
typeof i === 'object'
```

The `new` keyword is just a right-associative unary operator, so you can instantiate things first-class:

```
var x = 5;
new x.constructor ();     // Creates a boxed version of x
```

[9] jQuery is your friend here. It's branded as a JavaScript library, but in fact it's a set of enhancements to the DOM to (1) achieve a uniform cross-browser API, and (2) make it easier to retrieve and manipulate nodes.

If you are going to program using this questionable design pattern, then you'll probably want to add methods to things:

```
var f = function (x) {this.x = x};
f.prototype.add_one = function () {++this.x};
var i = new f (5);
i.add_one ();
i.x               // => 7
```

You can find tons of information about this kind of prototype programming online.

4.1 Autoboxing

You might be tempted to try something like this:[10]

```
Boolean.prototype.xor = function (rhs) {return !! this !== !! rhs};
```

And, upon running this code, you'd run into this tragically unfortunate property:

```
false.xor (false)         // => true
```

The reason is that when you treat an unboxed value as an object (e.g. invoke one of its methods), it gets temporarily promoted into a boxed value for the purposes of that method call. This doesn't change its value later, but it does mean that it loses whatever falsity it once had. Depending on the type you're working with, you can convert it back to an unboxed value:

```
function (rhs) {return !! this.valueOf () !== !! rhs};
```

5 A Really Awesome Equality

There is something really important about JavaScript that isn't at all obvious from the way it's used. It is this: The syntax `foo.bar` is, in all situations, identical to `foo['bar']`. You could safely make this transformation to your code ahead of time, whether on value-properties, methods, or anything else. By extension, you can assign non-identifier things to object properties:

```
var foo = [1, 2, 3];
foo['@snorkel!'] = 4;
foo['@snorkel!']    // => 4
```

You can also read properties this way, of course:

[10] `!! x` is just an idiom to make sure that `x` ends up being a boolean. It's a double-negation, and `!` always returns either `true` or `false`.

```
[1, 2, 3]['length']         // => 3
[1, 2, 3]['push']           // => [native function]
```

In fact, this is what the `for (var ... in ...)` syntax was built to do: Enumerate the properties of an object. So, for example:

```
var properties = [];
for (var k in document) properties.push (k);
properties          // => a boatload of strings
```

However, `for ... in` has a dark side. It will do some very weird things when you start modifying prototypes. For example:

```
Object.prototype.foo = 'bar';
var properties = [];
for (var k in {}) properties.push (k);
properties          // => ['foo']
```

To get around this, you should do two things. First, never modify `Object`'s prototype, since everything is an instance of `Object` (including arrays and all other boxed things); and second, use `hasOwnProperty`:

```
Object.prototype.foo = 'bar';
var properties = [], obj = {};
for (var k in obj) obj.hasOwnProperty (k) && properties.push (k);
properties          // => []
```

In particular, never use `for ... in` to iterate through arrays (it returns string indices, not numbers, which causes problems) or strings.

6 Extending JavaScript

JavaScript can do almost anything that other languages can do. However, it might not be very obvious how to go about it.

6.1 Iterators for cool people

Because languages like Ruby showed the world just how passé `for` loops really are, a lot of self-respecting functional programmers don't like to use them. If you're on Firefox, you won't have to; the `Array` prototype includes `map` and `forEach` functions already. But if you're writing cross-browser code and aren't using a library that provides them for you, here is a good way to implement them:

```
Array.prototype.each = Array.prototype.forEach || function (f) {
  for (var i = 0, l = this.length; i < l; ++i)
    f (this[i]);
```

```
  return this;      // convenient for chaining
};

Array.prototype.map = Array.prototype.map || function (f) {
  var ys = [];
  for (var i = 0, l = this.length; i < l; ++i)
    ys.push (f (this[i]));
  return ys;
};
```

As far as I know this is (almost) the fastest way to write these functions. We declare two variables up-front (`i` and `l`) so that the length is cached; JavaScript won't know that `this.length` is invariant with the `for` loop, so it will check it every time if we fail to cache it. This is expensive because due to boxing we'd have a failed hash-lookup on `this` that then dropped down to `this.__proto__`, where it would find the special property `length`. Then, a method call would happen to retrieve `length`.[11]

The only further optimization that could be made is to go through the array backwards (which only works for `each`, since `map` is assumed to preserve order):

```
Array.prototype.each = function (f) {
  for (var i = this.length - 1; i >= 0; --i)
    f (this[i]);
};
```

This ends up being very slightly faster than the first implementation because it changes a floating-point subtraction (required to evaluate < for non-zero quantities) into a sign check, which internally is a bitwise `and` and a zero-predicated jump. Unless your JavaScript engine inlines functions and you're really determined to have killer performance (at which point I would ask why you're using JavaScript in the first place), you probably never need to consider the relative overhead of a non-zero < vs. a zero >=.

You can also define an iterator for objects, but not like this:

```
Object.prototype.each = function (f) {
  // NO NO NO!!! Don't do it this way!
  for (var k in this) this.hasOwnProperty (k) && f (k);
};
```

Much better is to implement a separate `keys` function:

```
var keys = function (o) {
  var xs = [];
```

[11] This gets into how JavaScript presents certain APIs. Internally it has a notion of gettable and settable properties, though there isn't a cross-browser way to create them. But properties such as `length`, `childNodes`, etc. are all really method calls and not field lookups. (Try assigning to one and you'll see.)

```
  for (var k in o) o.hasOwnProperty (k) && xs.push (k);
  return xs;
};
```

This way you avoid polluting the `Object` prototype, which is shared across most other objects.

6.2 Java classes and interfaces

No sane person would ever want to use these. But if you're insane or are being forced to, then the Google Web Toolkit will give you a way to shoot yourself in the foot and turn it into JavaScript.

6.3 Recursive metaclasses

There are different ways to approach this, but a straightforward way is to do something like this:[12]

```
var metaclass = {methods: {
  add_to: function (o) {
    var t = this;
    keys (this.methods).each (function (k) {
      o[k] = bind (t.methods[k], o);        // can't use /this/ here
    });
    return o}}};
metaclass.methods.add_to.call (metaclass, metaclass);
```

At this point, `metaclass` is now itself a `metaclass`. We can start to implement instances of it:

```
var regular_class = metaclass.add_to ({methods: {}});
regular_class.methods.def = function (name, value) {
  this.methods[name] = value;
  return this;
};
regular_class.methods.init = function (o) {
  var instance = o || {methods: {}};
  this.methods.init && this.methods.init.call (instance);
  return this.add_to (instance);
};
regular_class.add_to (regular_class);
```

This is a Ruby-style class where you can define public methods and a constructor. So, for example:

[12] Remember that a class is just a function that produces instances. Nothing about the `new` keyword is necessary to write object-oriented code (thank goodness).

```
var point = regular_class.init ();
point.def ('init', function () {this.x = this.y = 0});
point.def ('distance', function () {
  return Math.sqrt (this.x * this.x + this.y * this.y)});
```

We're using the rather verbose `this.x`, which may offend some Python-eschewing Rubyists. Fortunately, we can use dynamic rewriting to use the `$` where Rubyists would use `@`:[13]

```
var ruby = function (f) {
  return eval (f.toString ().replace (/\$(\w+)/g,
    function (_, name) {return 'this.' + name}));
};

point.def ('init', ruby (function () {$x = $y = 0}));
point.def ('distance', ruby (function () {
  return Math.sqrt ($x * $x + $y * $y)}));
```

And now you can use that class:

```
var p = point.init ();
p.x = 3, p.y = 4;
p.distance ()       // => 5
```

The advantage of using metaclasses is that you can do fun stuff with their structure. For example, suppose that we want to insert method tracing into all of our points for debugging purposes:

```
keys (point.methods).each (function (k) {
  var original = point.methods[k];
  point.methods[k] = function () {
    trace ('Calling method ' + k +
      ' with arguments ' + arguments.join (', '));
    return original.apply (this, arguments);
  };
});
```

Now `trace` (which isn't a JavaScript built-in, so you'd have to define it) would be called each time any method of a `point` instance was called, and it would have access to both the arguments and the state.

6.4 Tail calls and delimited continuations

JavaScript does not do tail-call optimization by default, which is a shame because some browsers have short call stacks (the shortest I'm aware of is 500 frames, which goes by especially quickly when you have bound functions and iterators). Luckily, encoding tail calls in JavaScript is actually really simple:

[13] And, in fact, we could bake this `ruby()` transformation into a metaclass to make it totally transparent if we wanted to.

```
Function.prototype.tail = function () {return [this, arguments]};
Function.prototype.call_with_tco = function () {
  var c      = [this, arguments];
  var escape = arguments[arguments.length - 1];
  while (c[0] !== escape)
    c = c[0].apply (this, c[1]);
  return escape.apply (this, c[1]);
};
```

This is actually an implementation of delimited continuations, since the escape parameter serves the purpose of the final CPS target.[14] So writing a factorial function, for example:

```
// Standard recursive definition
var fact1 = function (n) {
  return n > 0 ? n * fact1 (n - 1) : 1;
};

// Tail-recursive definition
var fact2 = function (n, acc) {
  return n > 0 ? fact2 (n - 1, acc * n) : acc;
};

// With our tail-call mechanism
var fact3 = function (n, acc, k) {
  return n > 0 ? fact3.tail (n - 1, acc * n, k) : k.tail (acc);
};
```

The first two functions can be called normally:

```
fact1 (5)           // => 120
fact2 (5, 1)        // => 120
```

though neither will run in constant stack space. The third one, on the other hand, will if we call it this way:

```
var id = function (x) {return x};
fact3.call_with_tco (5, 1, id)      // => 120
```

The way this tail-call optimization strategy works is that instead of creating new stack frames:

```
fact1(5)
  5 * fact1(4)
    4 * fact1(3)
    ...
```

[14] If this didn't make sense, then see the Wikipedia article on continuation-passing style and delimited continuations. If that doesn't help, then you should probably skip this section.

or even creating hollow ones:

```
fact2(5, 1)
  fact2(4, 5)
    fact2(3, 20)
    ...
```

we pop out of the last stack frame before allocating a new one (treating the array of `[function, args]` as a kind of continuation to be returned):

```
fact3(5, 1, k) -> [fact3, [4, 5, k]]
fact3(4, 5, k) -> [fact3, [3, 20, k]]
fact3(3, 20, k) ...
```

It isn't a bad performance hit, either – the overhead of allocating a two-element array of pointers is minimal.

6.5 Syntactic macros and operator overloading

Lazy scoping lets us do some cool stuff. Let's say we want to define a new syntax form for variable declaration, so that instead of this:

```
var f = function () {
  var y = (function (x) {return x + 1}) (5);
  ...
};
```

we could write this:

```
var f = function () {
  var y = (x + 1).where (x = 5);
  ...
};
```

This can be implemented in terms of regular expressions if we don't mind being woefully incorrect about half the time:

```
var expand_where = function (f) {
  var s = f.toString ();
  return eval (s.replace (/\(([^)]+)\)\.where\(([^)])\)/,
    function (_, body, value) {
      return '(function (' + value.split ('=')[0] + '){return ' +
             body + '}) (' + value.split ('=', 2)[1] + ')';
    }));
};
```

Now we can say this:

```
var f = expand_where (function () {
  var y = (x + 1).where (x = 5);
  ...
});
```

Obviously a proper parser is more appropriate because it wouldn't fail on simple paren boundaries. But the important thing is to realize that a function gives you a way to quote code, just like in Lisp:

```
(defmacro foo (bar) ...)
(foo some-expression)
```

becomes this in JavaScript (assuming the existence of `parse` and `deparse`, which are rather complicated):

```
var defmacro = function (transform) {
  return function (f) {
    return eval (deparse (transform (parse (f.toString ()))));
  };
};
var foo = defmacro (function (parse_tree) {
  return ...;
});
foo (function () {some-expression});
```

This principle can be extended to allow for operator overloading if we write a transformation that rewrites operators into method calls:

```
x << y      // becomes x['<<'](y)
```

Remember that property names aren't restricted to identifiers – so we could overload the << operator for arrays to work like it does in Ruby with:

```
Array.prototype['<<'] = function () {
  for (var i = 0, l = arguments.length; i < l; ++i)
    this.push (arguments[i]);
  return this;
};
```

The only thing that's unfortunate about implementing this stuff in JavaScript rather than Lisp is that JavaScript bakes syntactic constructs into the grammar, so trying to introduce new syntactic forms such as `when` isn't very convenient:

```
expand_when (function () {
  when (foo) {      // compile error; { unexpected
    bar ();
  }
});
```

But anything you can do inside the JavaScript parse tree is fair game.[15]

7 Further reading

I highly recommend reading jQuery (http://jquery.com) for the quality and conscientiousness of the codebase. It's a brilliant piece of work and I've learned a tremendous amount by pawing around through it.

As a shameless plug, I also recommend reading through Divergence (http://spencertipping.com/divergence.js), a library that I'm working on. It's very different from jQuery – much more terse and algorithmic (and has no DOM involvement; I recommend jQuery for that). jQuery uses a more traditional approach, whereas Divergence tends to make heavy use of closures and functional metaprogramming.

[15] Keep in mind that `toString` will sometimes rewrite your function to standard form, so leveraging ambiguities of the syntax isn't helpful. In Firefox, for example, writing expressions with excess parentheses is not useful because those excess parentheses are lost when you call `toString`.