Everything New is Old Again

A Brief History of Time
(with Javascript Closures)

Wilmington JS - Aug 21, 2018

David Biesack
VP, API Platforms & Lead API Architect, APITURE


CODE
CLOSED
CODE
CLOSURES

Closures are *core* to JavaScript ...

Closures are *core* to JavaScript ...

... and Node.js

Closures are *core* to JavaScript ...

... and Node.js

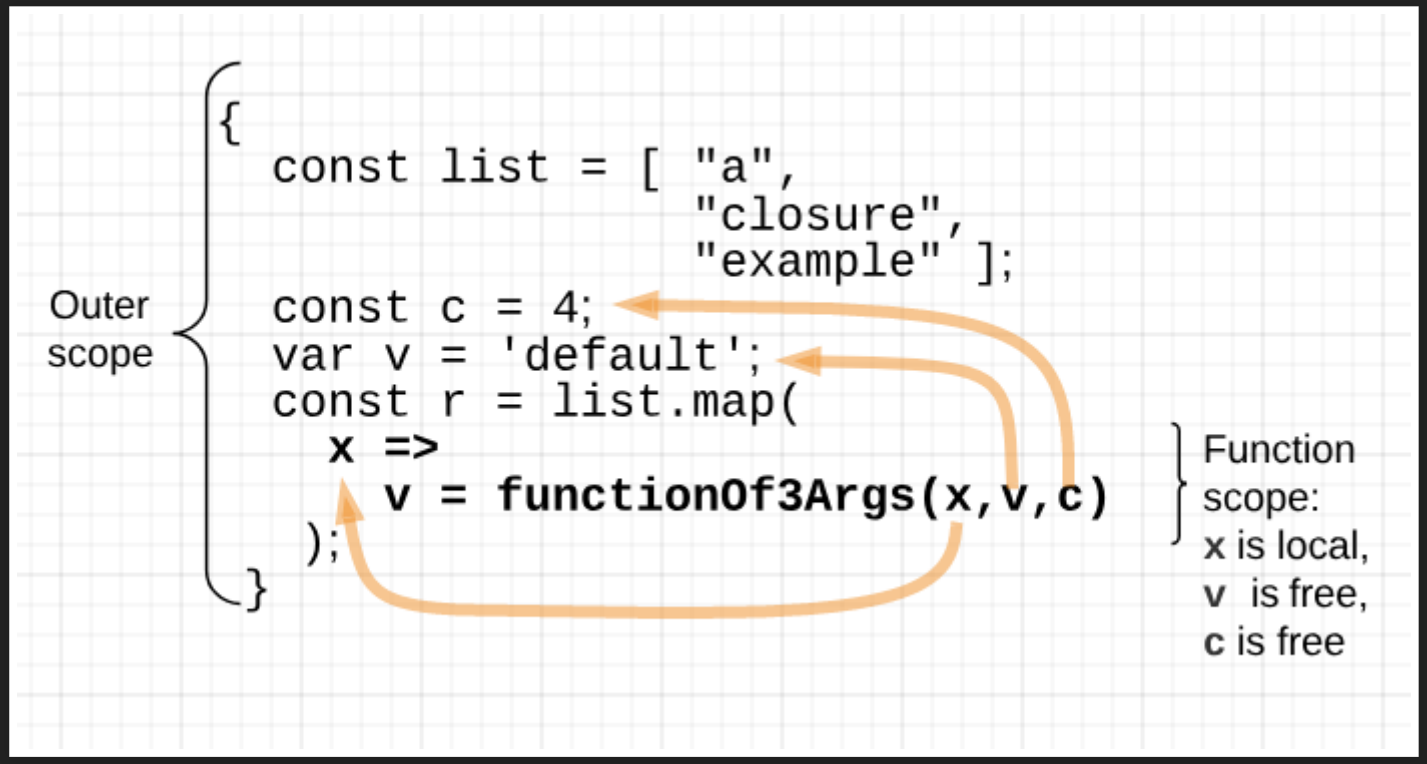
... and functional programming

Closures Explained

A function whose free variables (or open bindings) have been closed (or bound) in its containing lexical environment, resulting in a closed expression (one with no free variables):

```
const list = ["a", "closure", "example"];
var v = 'default';
const c = 4;
const r = list.map(x => v = functionOf3Args(x,v,c));
```

Outer vs. function lexical scope



Necessity is the Mother of Invention

Necessity is the Mother of Invention

Nobody mentions Invention's father...

Necessity is the Mother of Invention

Nobody mentions Invention's father...

Conclusion: Javascript Closures are Necessity's Bastard?

The Neccessity Defense

```
<body onload="document.write('<h1>Everthing New</h1>'
                           + '<p>is Old Again</p>')">
```

The `onload` right hand side is a Javascript function body.

`document` is a free variable within that function body.

`document` is bound outside of the function, but available in the function's *environment*.

The Neccessity Defense, Part II

`Array.prototype.forEach(callback)` requires a callback function :

- `callback(item)`
- `callback(item, index, array)`

But what if you want to call a function that requires three arguments?

... or mutate or collect state on each call?

But what if you want to call a function that requires three arguments?

... or mutate or collect state on each call?

Closures to the rescue:

```
function functionOf3Args(x,v,c) {
  return (x.length > c)
         ? x
         : v;
}
{
  const list = ["a", "closure", "example"];
  const c = 4;
  var v = 'default';
  const r = list.map(x => v = functionOf3Args(x,v,c));
}
```

```
function functionOf3Args(x,v,c) {
  return (x.length > c)
         ? x
         : v;
}
{
  const list = ["a", "closure", "example"];
  const c = 4;
  var v = 'default';
  const r = list.map(x => v = functionOf3Args(x,v,c));
}
```

`x` is bound in the anonymous function

```
function functionOf3Args(x,v,c) {
  return (x.length > c)
         ? x
         : v;
}
{
  const list = ["a", "closure", "example"];
  const c = 4;
  var v = 'default';
  const r = list.map(x => v = functionOf3Args(x,v,c));
}
```

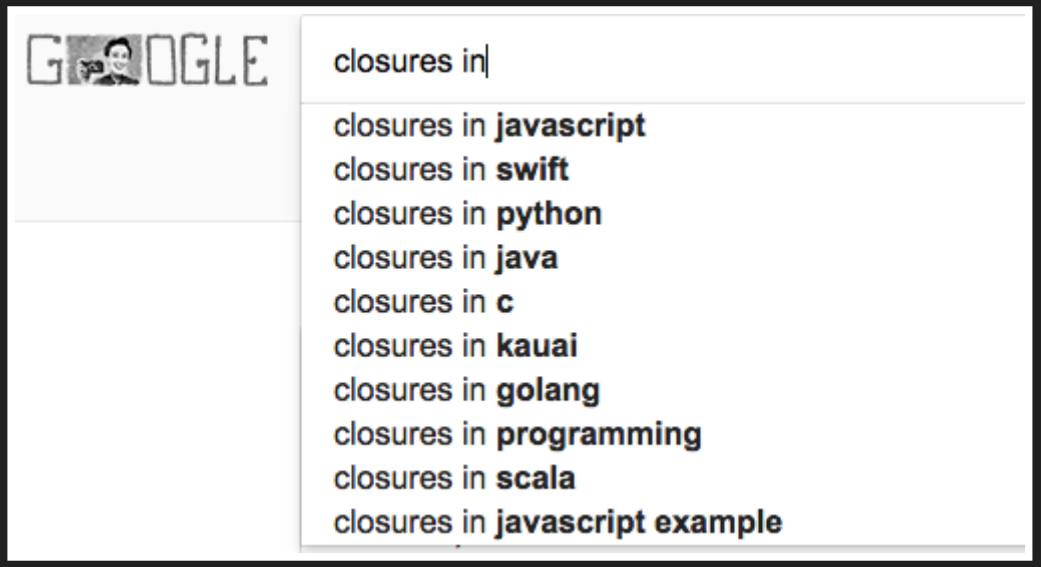
`x` is bound in the anonymous function

`v` and `c` are free in the anonymous function

… But JS is far from the first

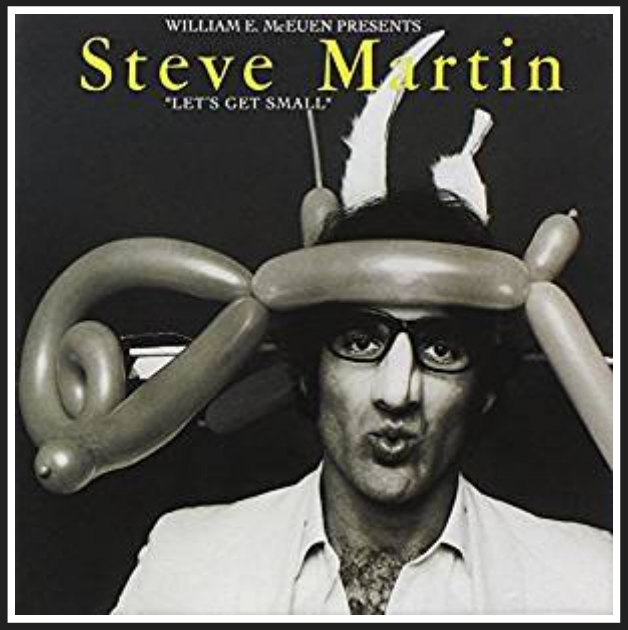
History

Everything New... is Old Again


GOOGLE
closures in
closures in javascript
closures in swift
closures in python
closures in java
closures in c
closures in kauai
closures in golang
closures in programming
closures in scala
closures in javascript example

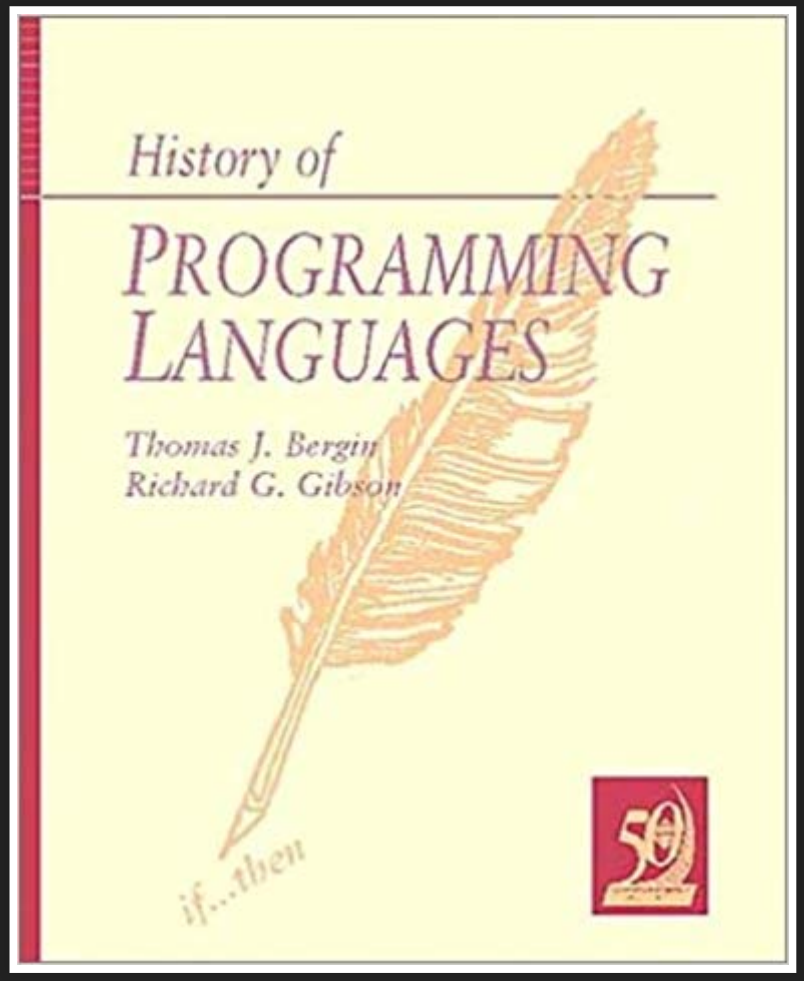
First, Some Historical Context

First, Some Historical Context



~~First, Some Hysterical Context~~

First, Some Historical Context



"Closure" coined by Peter J. Landin in 1964

A Very Brief History of Our Time With Closures

Lisp was created in the late 1950's by John McCarthy and others at M.I.T. One feature of the language was function-valued expressions, signified by lambda. The name lambda was borrowed from a mathematical formalism known as the lambda calculus. [...]

Although Lisp was not based on an effort to model that formalism, lambda plays approximately the same role in Lisp as it does in the lambda calculus: lambda is the syntax for a function-valued expression. [...]

```
(lambda (x) x)
```

Look familiar?

Now we fast-forward to the mid 1970's. [...] A number of popular Lisp dialects were in use including InterLisp, MacLisp, UCI-Lisp, Stanford Lisp 1.6, and U. Utah's Standard Lisp. All of them were dynamically scoped.

It was in this context that Guy Steele and Gerald Jay Sussman developed Scheme, a very simple Lisp dialect.

Neil Gafter, *A Definition of Closures*

Scheme

First languages to adopt closures, 1975

Revised^3 Report on the Algorithmic Language Scheme

(YF) = (F (YF))

```
(define (y f) (lambda () (funcall f (y f))))
```

Scheme

```
(define (functionOf3Args x v c)
  (if (> (string-length x) c)
      x
      v))

(let ( (l '("a" "closure" "example"))
       (v "value")
       (c 4)
      )
   (map (lambda (x) (write-line (functionOf3Args x v c))) l)
)
```

yields:

```
"value"
"closure"
"example"
```

Lisp

"Lisp was originally created as a practical mathematical notation for computer programs, influenced by the notation of Alonzo Church's lambda calculus." - Wikipedia

Lisp

Common LISP: Newer variant of LISP, with lexical (not dynamic) scoping, closures (Guy Steele, again)

```
(defun functionOf3Args (x v c)
  (if (> (string-length x) c)
      x
      v))

(let ( (l '("a" "closure" "example"))
       (v "value")
       (c 4)
      )
   (map (lambda (x) (write-line (functionOf3Args x v c))) l)
)
```

Lambda Calculus

The granddaddy of them all.

Alonzo Church's formalized the mathematics of functions and variable binding with the Lambda Calculus in the 1930's

λx . x

"the smallest universal programming language of the world."

A Tutorial Introduction to the Lambda Calculus, Raúl Rojas

The λ calculus consists of a single transformation rule (variable substitution) and a single function definition scheme. It was introduced in the 1930s by Alonzo Church as a way of formalizing the concept of effective computability. -- Rojas

The λ calculus helps us understand and model computation, primarily through pure functions.

```
 <expression> := <name> | <function> | <application>
   <function> := λ <name>.<expression>
<application> := <expression><expression>
```

λx.x

is a *function*

(λx.x)y

is an *application* of a function to an *argument*

(λx.x)y = [y/x]x = y

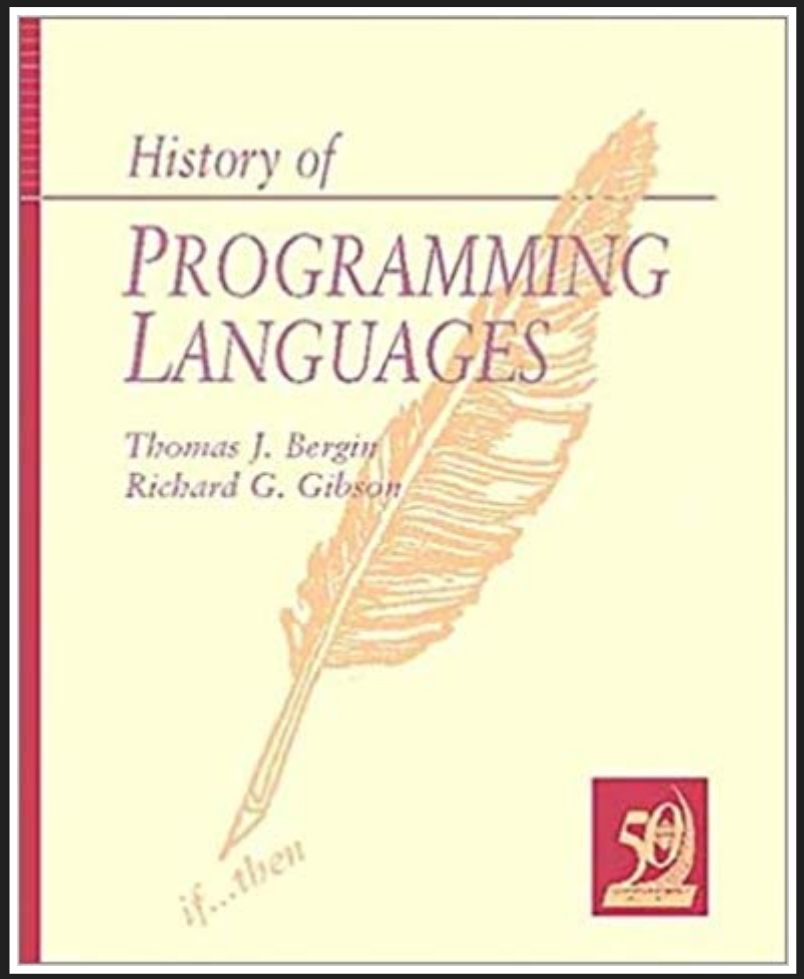
From these roots, a functional model of numbers and even arithmetic and logic functions can be defined in a purely function-based model

- `0` is modeled as a function
- `1` is modeled as the application of a *successor* function to `0`, `S0`
- `2` is modeled as the application of a *successor* function to `1`, `S1` == `SS0`
- ... boolean values true and false, logical operations like less than, greater than
- ... and recursion

A Tutorial Introduction to the Lambda Calculus, Raúl Rojas

Historical Context

This was before programming languages were invented. This was before computers were invented.



Those who remember [the best of] the past of programming languages are *fortunate* to repeat it.

Presented with reveal.js