Runtime Funtimes

what runtimes give us
techniques to tackle what they don't

cdleary @ nodevember '14

We'll cover:

What's a runtime?

Techniques to add new capabilities

Random acts of hackery

Standard disclaimer: don't speak for my employer

JavaScript is the **universal runtime**

I know I've heard someone say that before…

JS is a **General Purpose** Programming Language

Why would you ever need anything else?

JS is **just** like **LISP**

Are you sure about that?

```
var lispSuperPowers =
    require('./lisp-dsl-defn');

function greet() {
  "use lisp"
  {define
    salutation
    {list-ref
      {list
        "Hi"
        "Hello"}
      {random
        2}}}
  {define
    greet
      {name}
      {string-append
        salutation
        ", "
        name}}
  {greet
    "Chris"}
}

exports.greet =
    lispSuperPowers(greet)
```

(whole module)

Show of hands:

who thinks this can work

with “stock” node.js

(no C extensions)

github.com/cdleary/nodevember-hacks

We'll get back to how that works…
I really came to this
JS conference to talk about
CPU instructions
(only half serious)

Instructions are divided into **syscalls** that ask the OS for stuff & **everything else** (e.g. add)

C program
compiler-generated
executable code

my_program.exe

function calls
(e.g. malloc,
send, recv)

C runtime
syscall-oriented
or pre-canned
routines

libc.so, libm.so, libpthread.so, libdl.so

C program that just **computes** “fancy” exit codes would require **no syscalls**

But that would be super boring

JS program
source text

my_program.js

“run me!”

d8.exe

JS runtime
syscall-oriented
or pre-canned
routines

JIT codegen

generates
code to run

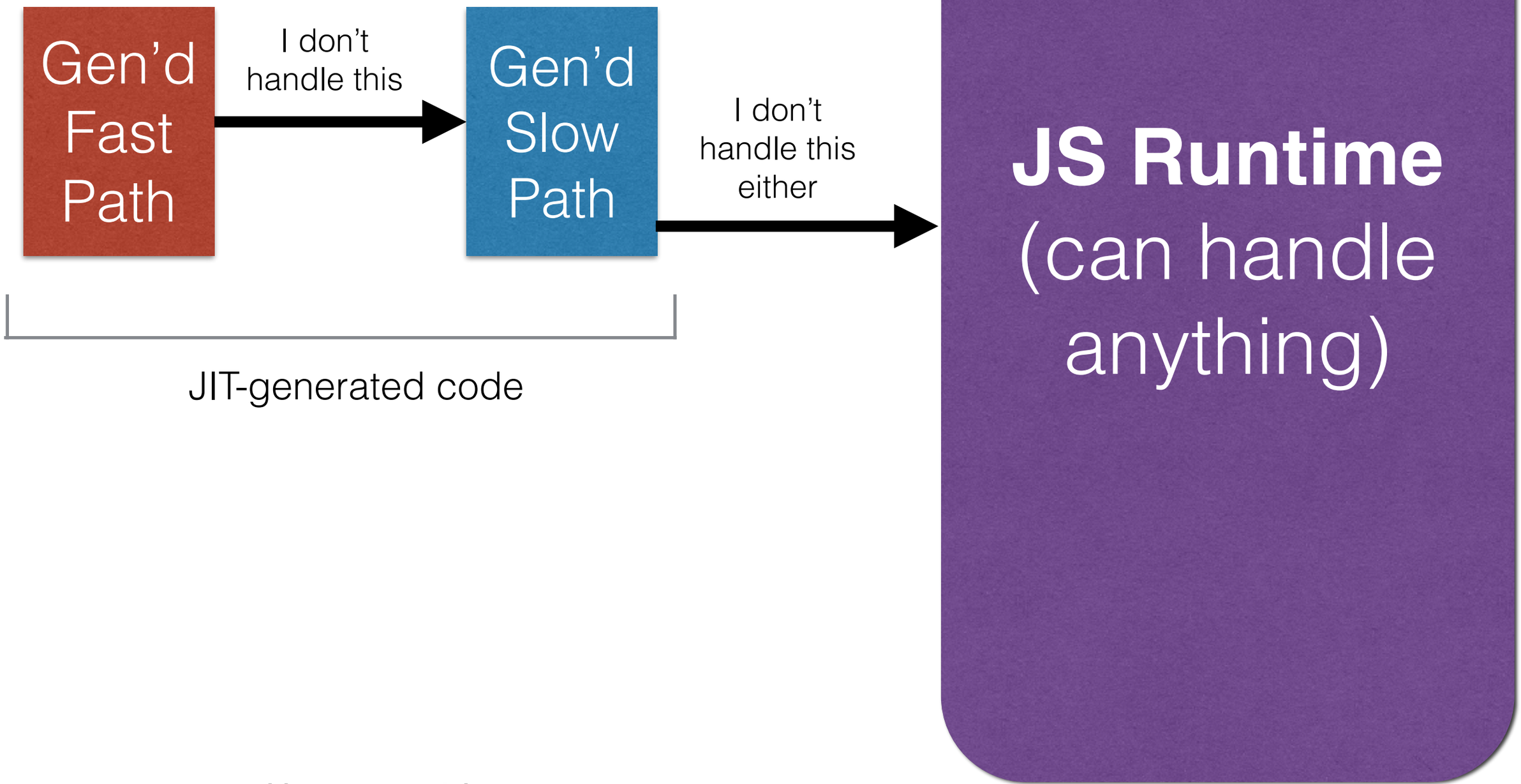
function calls
(e.g. Math.random)
object allocation

Generated
executable code

w/o compilation step (late binding), runtimes are **even more critical**

Runtimes are also **fallbacks** for **fast path assumptions**

Violating these results in (potentially severe) perf penalties



more: http://goo.gl/Msxqdx

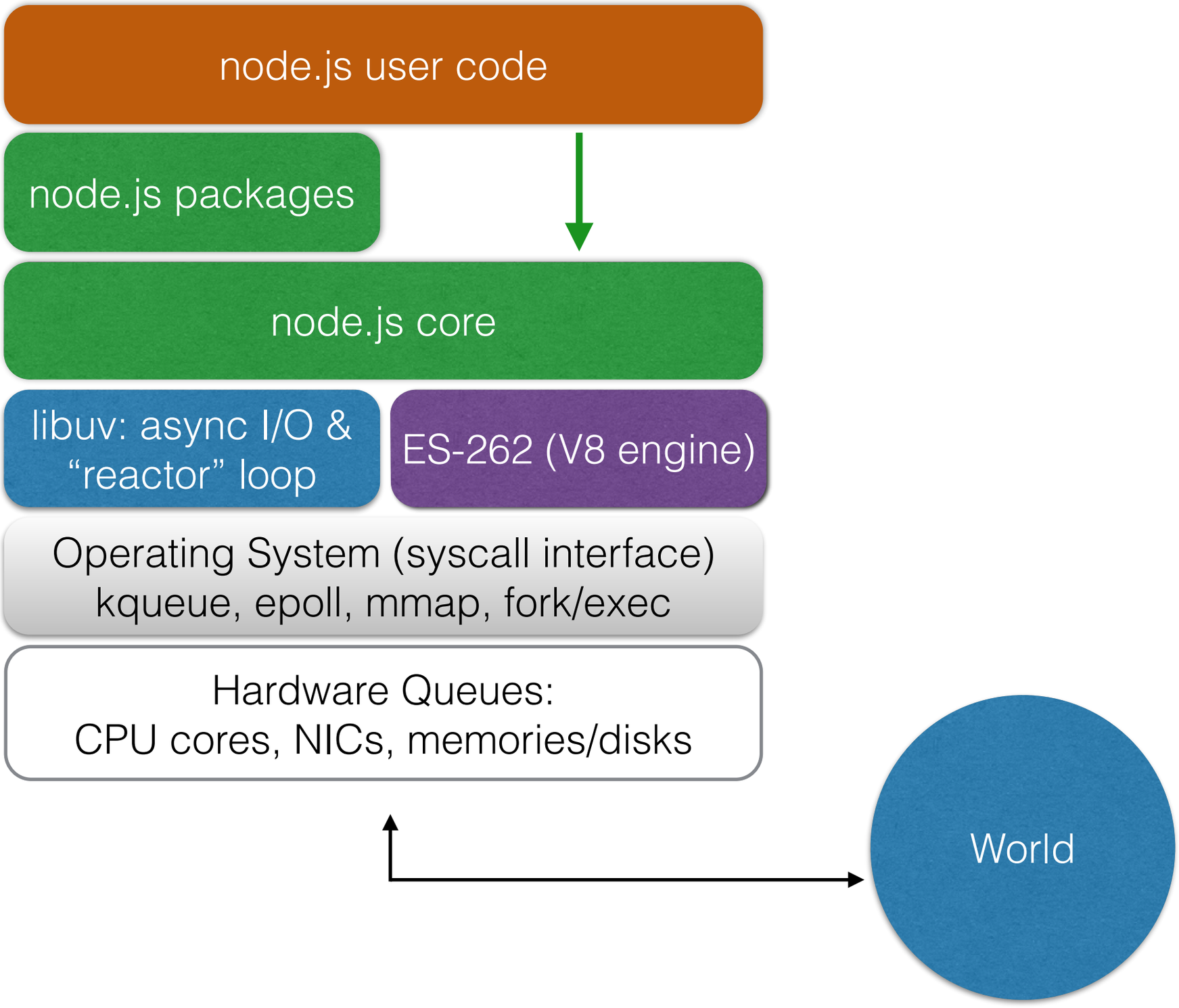
But did you know?

ES-262 is **useless on its own**

Does not prescribe I/O

Not *even* return codes

The **core** JS runtime is
extended by node
to make it (async) I/O capable


node.js user code
node.js packages
node.js core
libuv: async I/O &
“reactor” loop
ES-262 (V8 engine)
Operating System (syscall interface)
kqueue, epoll, mmap, fork/exec
Hardware Queues:
CPU cores, NICs, memories/disks
World

Runtimes are windows to **the world** Program's **execution environment**

I/O, I/O, off to work we go

Runtimes are built to enable

semantics of interest

op a user in our environment wants to perform is **X**

e.g. `addproperty`, `receive`

node.js runtime is more than
V8 with libuv bolted on

defines & supports conceptual constructs
Streams, EventEmitters, Buffers, …

Key insight:
you can **change**
or **play into** what the **platform**
you're building on **wants**

“When all you’ve got is a hammer…
use that hammer to hammer itself into a better hammer.”

–MC Hammer

Sorry, I forgot,
he hammered his name into a
more **condensed**, **memorable**
form…

“When all you’ve got is a hammer…
use that hammer to hammer itself into a better hammer.”

–MC Hammer

“When all you’ve got is a hammer…
use that hammer to hammer itself into a better hammer.”

~~**–MC Hammer**~~

–Hammer

(It’s true, check Wikipedia)

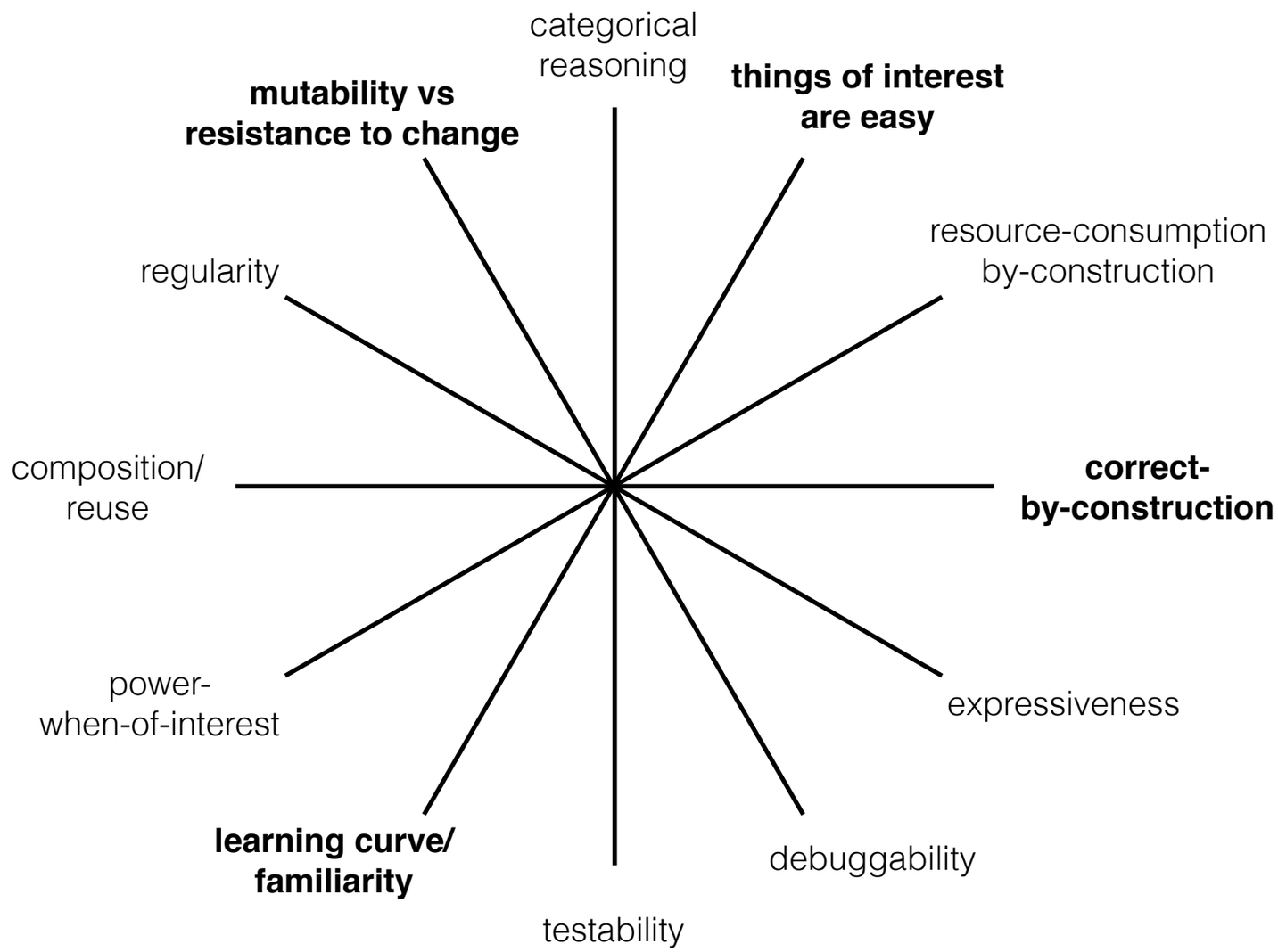
(Except the quote, I totally made that up)

Technique Rundown

Technique	Approach
Modules	**extend lightly**
RPCs	**bridge**, potentially to other platforms
DSLs	**constrict**, specialize, raise
“Declarative Sandboxes”	**constrict w/full language** expressiveness
Macros/Desugaring (sweet.js)	**embrace, extend**
Binary Extensions Hosted FFI Bindings	**extend deeply**, control tradeoffs
Hosted Natives (JIT Hinting)	**symbiotic** optimization
Transpilers	**reface**, change emphasis

Takeaway: there's a slew of techniques available to help us achieve our goals

All these techniques are useful because **programmability** has **no silver bullet**
Therefore, it's good to have options



Crux:

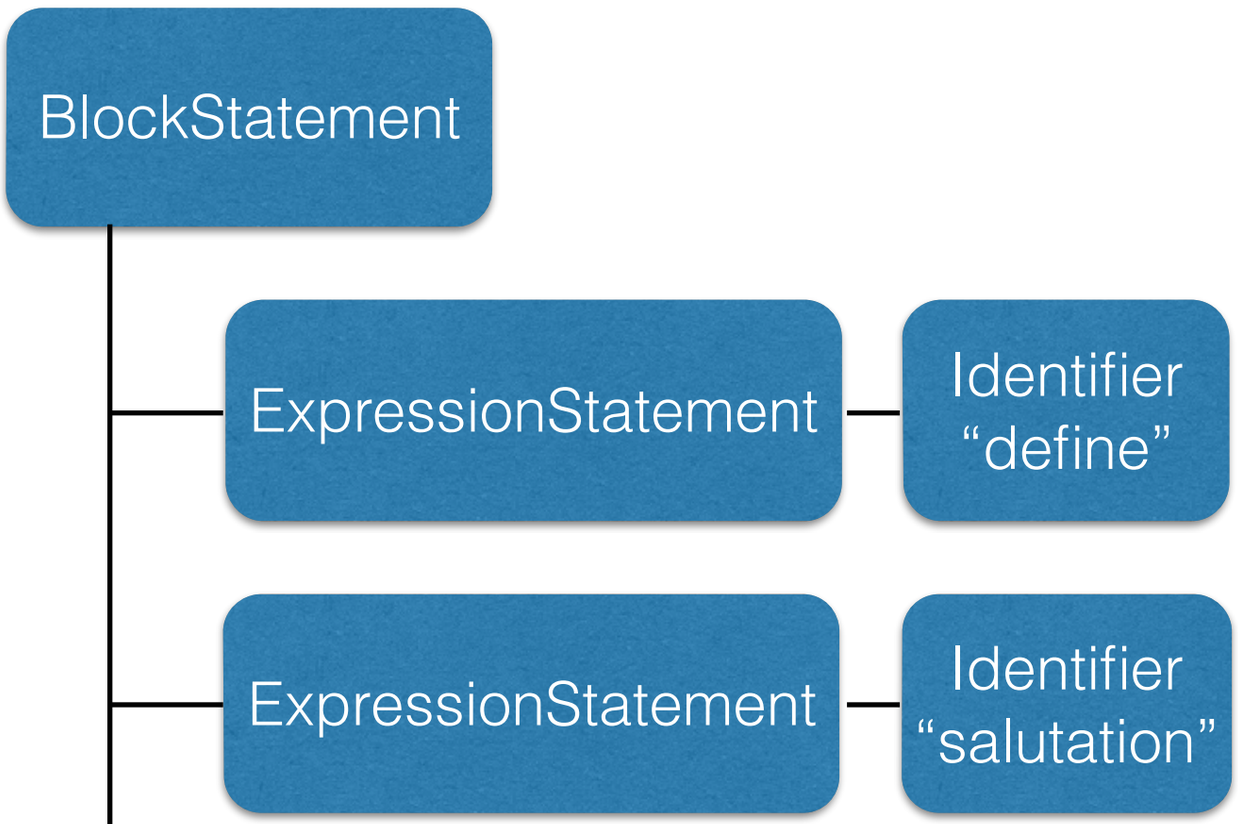
esprima.parse
new Function

```
var lispSuperPowers =
    require('./lisp-dsl-defn');

function greet() {
  "use lisp"
  {define
    salutation
    {list-ref
      {list
        "Hi"
        "Hello"}
      {random
        2}}}
  {define
    greet
      {name}
      {string-append
        salutation
        ", "
        name}}
  {greet
    "Chris"}
}

exports.greet =
    lispSuperPowers(greet)
```

Parse trees




```
{
 "type": "BlockStatement",
 "body": [{
     "type": "ExpressionStatement",
     "expression": {
         "type": "Identifier",
         "name": "define"
     }
 }, {
     "type": "ExpressionStatement",
     "expression": {
         "type": "Identifier",
         "name": "salutation"
     }
 },
 …
}
```

Match against **limited** patterns of
JS parse nodes, reusing the JS parser

Either **rewrite** and **new Function**
or **interpret** that limited parse tree

Transform any syntactically valid JS

Thinking of a use case yet?

µservice:

do one thing, do it well, **pipe** together

“I KNOW THIS, IT’S **UNIX**!”

My dataflow DSL on top of Seneca μservices

seneca.act w/continuation

```
var main = senecaPipeline(function runPipeline() {
  serve&
  (snapshot2 | snapshot-pair-motion | push-update)*N
});
```

pipeline "snowballs" command results

async.forever

DSLs could be
operator overloading from hell,
but provided **semantics** should be
“whitelist based”

A well done DSL has a very short spec

function: short, understandable
class: Single Responsibility Principle
***μ*service**: performs a single function

One person's μservice
Is another person's λ

More?
write in **sync** DSL
lower to **async** program
across processes in `cluster`?

Going farther:

source maps

auto-transform hooks

CPS transform

partial eval

```
var lispSuperPowers =
    require('./lisp-dsl-defn');

function greet() {
  "use lisp"
  {define
    salutation
    {list-ref
      {list
        "Hi"
        "Hello"}
      {random
        2}}}
  {define
    greet
      {name}
      {string-append
        salutation
        ", "
        name}}
  {greet
    "Chris"}
}

exports.greet =
    lispSuperPowers(greet)
```

Takeaway: you **don't need a whole compiler** to make your runtime better

Enabling factor: μservices permit tinkering, are rewrite-friendly: amenable to DSL usage

```
npm install [mynotationhere].js
```

With that one technique covered,
back to the bigger picture

Runtimes **aren't one-style limited**
Server-side has a track record of doing **very fancy things when performance really counts**

DSLs w/perf oriented transforms
AoT compilation of Java
Erlang on Xen (look ma! no OS!)
Vortex-style WPO analysis

Runtimes **aren't one-node limited**

There are **distributed** runtime services

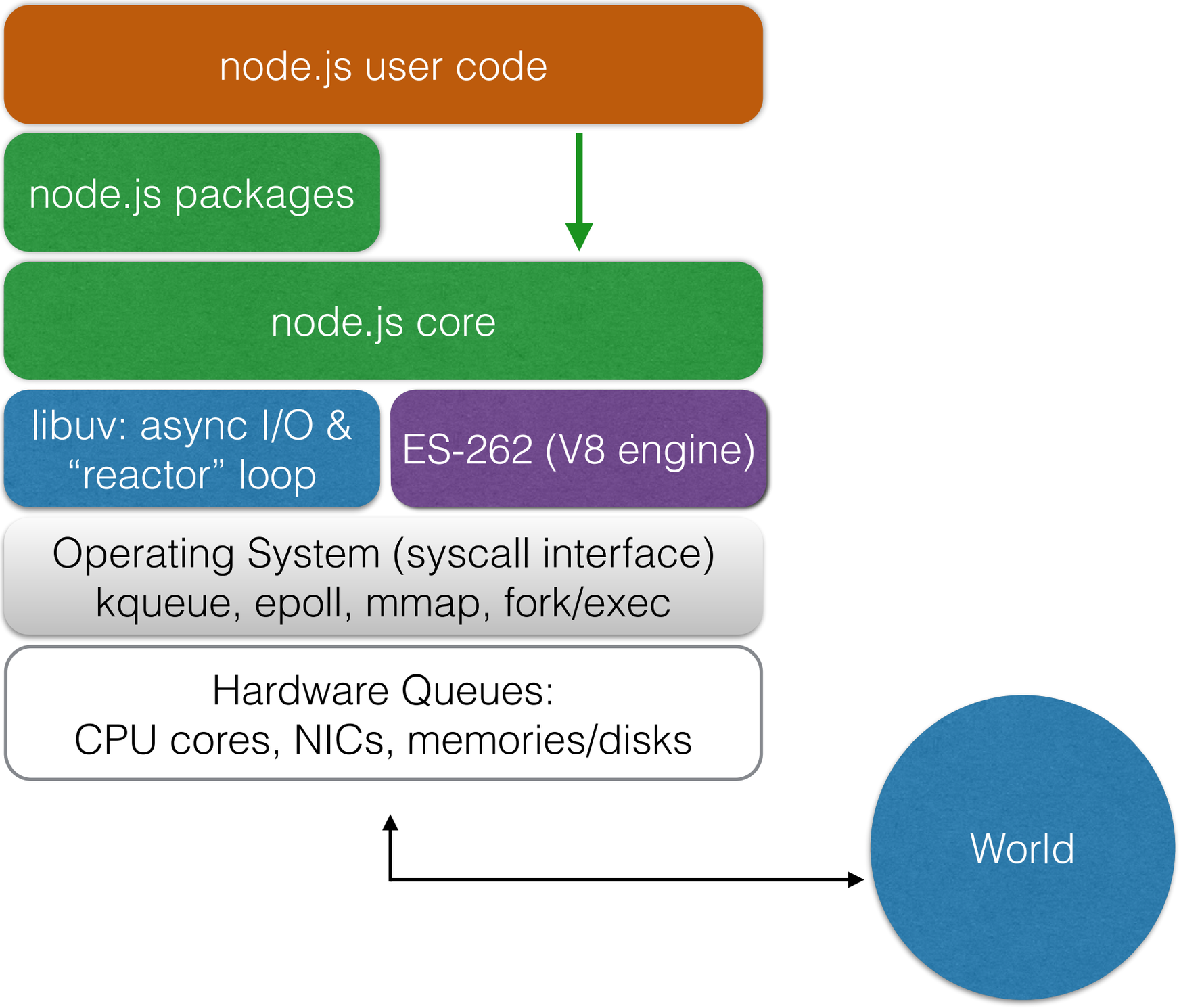
Distributed Smalltalk cross-node GC (1987)
Erlang/BEAM VM process links to remote `pids`

Runtime mismatched to use case?

It can be taught

In many different ways…

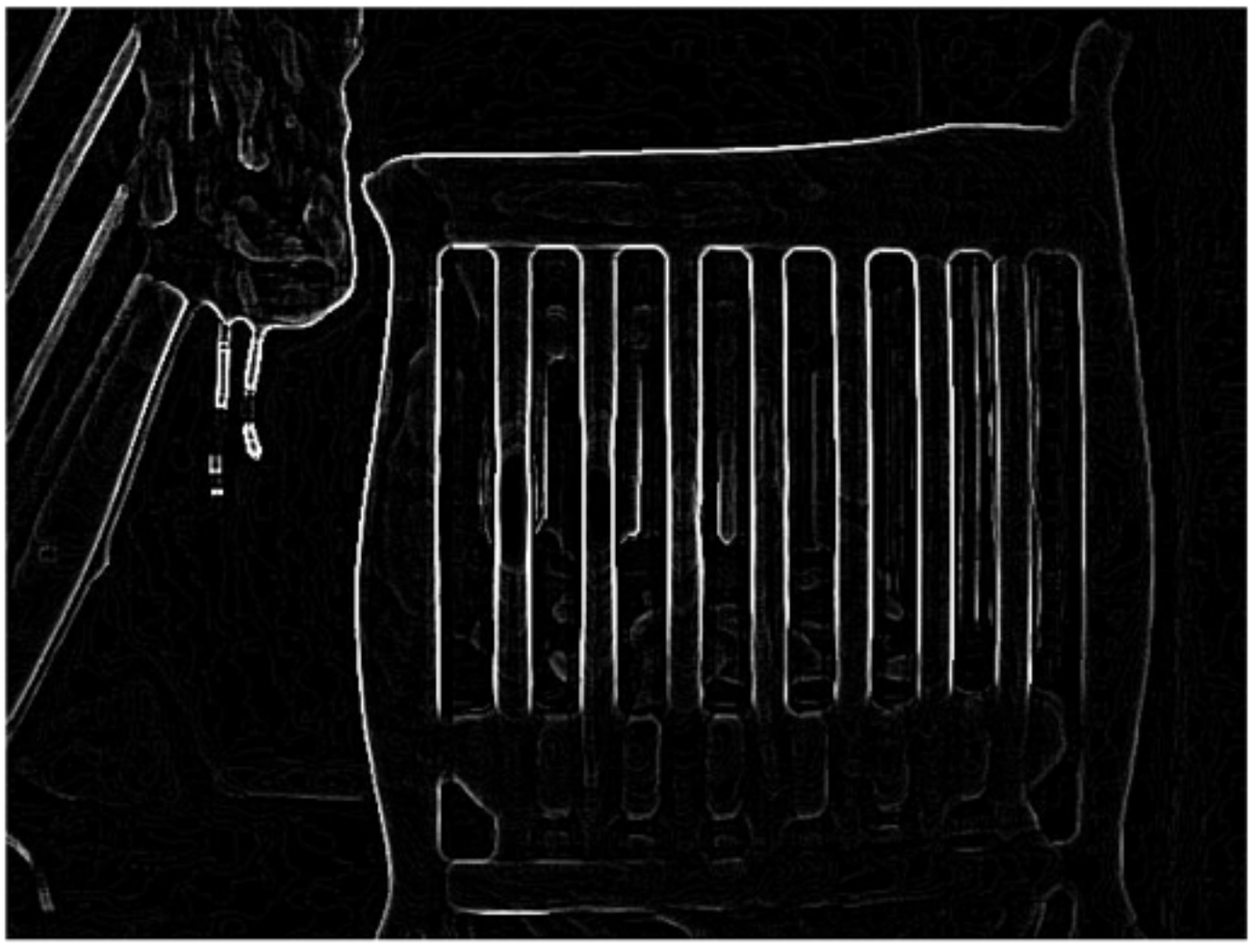
At all the different levels.


node.js user code
node.js packages
node.js core
libuv: async I/O &
“reactor” loop
ES-262 (V8 engine)
Operating System (syscall interface)
kqueue, epoll, mmap, fork/exec
Hardware Queues:
CPU cores, NICs, memories/disks
World

Technique	Approach
Modules	extend lightly
RPCs	bridge to other platforms
DSLs	constrict, specialize, raise
“Declarative Sandboxes”	constrict w/full language expressiveness
Macros/Desugaring (sweet.js)	embrace, extend
Binary Extensions Hosted FFI Bindings	extend deeply, control tradeoffs
Hosted Natives (JIT Hinting)	symbiotic optimization
Transpilers	reface, change emphasis

Backup

Thought I was going to talk mostly about **maxing perf**



Could only pull **4FPS** from endpoint for “my toddler sleep-motion project”, so **DSLs** it was :-)

JS runtime gives you:

Aggressive inlining (closures too)

Constant propagation

Not: value specialization

Not: opt across JS/C boundaries

Teaching the runtime **more** about performance-critical pieces: **“hosted natives”**

```
// ECMA-262 section 15.5.4.7
function StringIndexOf(pattern /* position */) {  // length == 1
  CHECK_OBJECT_COERCIBLE(this, "String.prototype.indexOf");

  var subject = TO_STRING_INLINE(this);
  pattern = TO_STRING_INLINE(pattern);
  var index = 0;
  if (%_ArgumentsLength() > 1) {
    index = %_Arguments(1);  // position
    index = TO_INTEGER(index);
    if (index < 0) index = 0;
    if (index > subject.length) index = subject.length;
  }
  return %StringIndexOf(subject, pattern, index);
}
```

runtime implementation is a balancing act between

build **it in**

build **in it**

jsctypes/node-**ffi based “bolting”**

```
var uv = ctypes.open(“libuv.so”);
var loop = uv.uv_loop_new();
var retcode = uv.uv_run(loop);
```

The dream of Mozilla's **privileged JS runtime** is alive in node (+ ES6)

	node	Moz privileged JS
FFI	node-ffi	jsctypes
Parser Exposure	esprima	Reflect.parse
Message Bus	seneca	XPCOM*
Transpilers?	coffeescript, sweet.js	privileged JS extensions, XUL*

* Disclaimer: I never worked on Gecko, so this may be completely off.
In any case, these elements feel familiar.