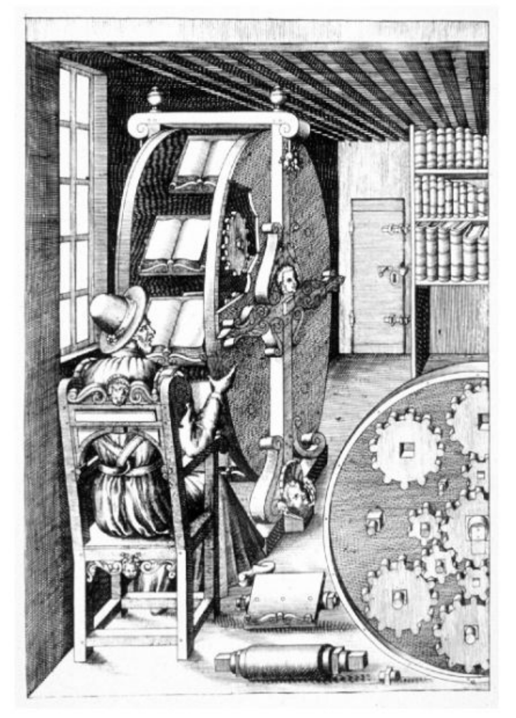
SpiderMonkey
Byte-sized Architectures

Daniel Minor
Staff SpiderMonkey Engineer / Mozilla
https://github.com/dminor

What is a Spider Monkey?

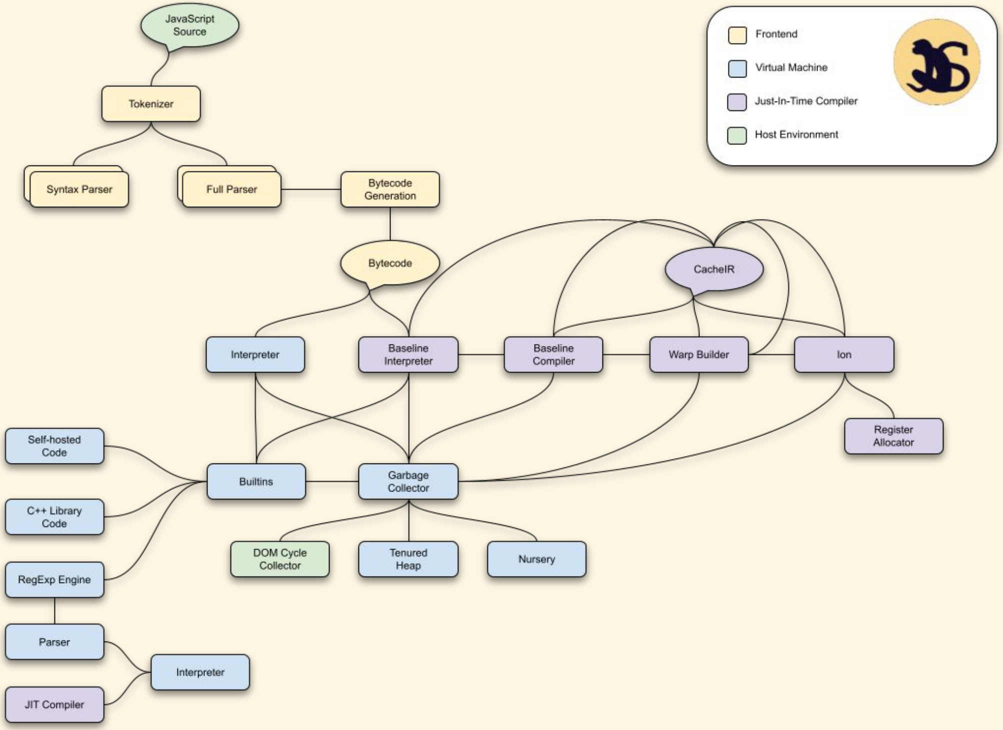
- Genus *Ateles*
- Most intelligent New World Monkey
- Lack thumbs, but have very long limbs and tails

What is SpiderMonkey?

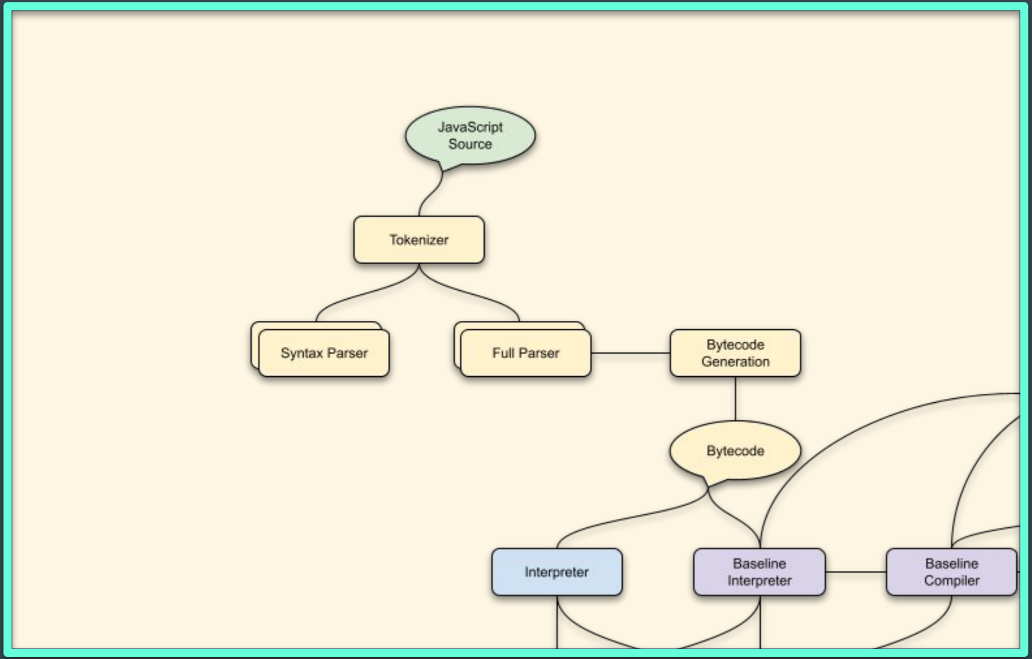


- The JavaScript engine in Firefox
- An engine is the part of the browser which runs JavaScript code
- An engine consists of:
    - Parsers
    - Interpreters
    - Just-in-time compilers
    - Garbage collection
    - A library of useful functions

Simplified SpiderMonkey Architecture

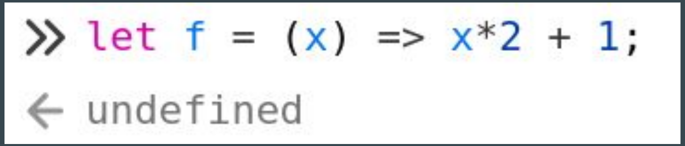


Frontend


JavaScript Source
Tokenizer
Syntax Parser
Full Parser
Bytecode Generation
Bytecode
Interpreter
Baseline Interpreter
Baseline Compiler

What is the Frontend?

Takes JavaScript source provided by the host environment



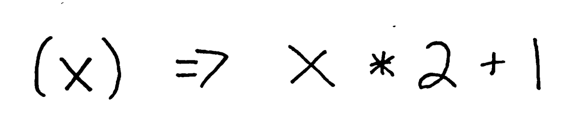
```
>> let f = (x) => x*2 + 1;
<- undefined
```

And transforms it into a format usable by the rest of the engine.

Tokenizer

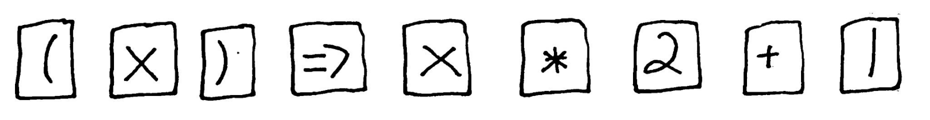
A ‘token’ is an indivisible unit in the input to the parser:

The ‘tokenizer’ divides the input source



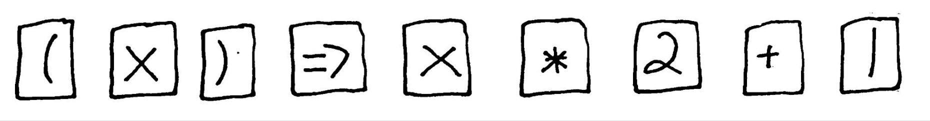
```
(x) => x * 2 + 1
```

into a list of tokens

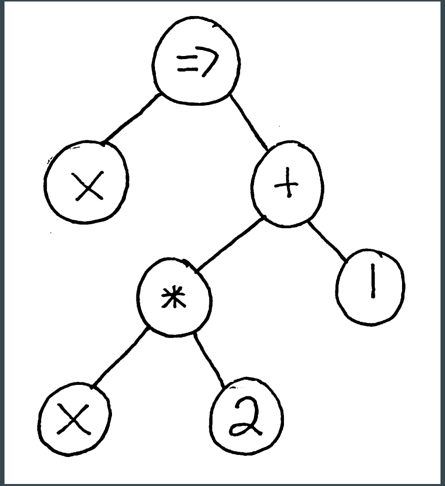


Parser

The parser uses the list of tokens and the grammar of JavaScript



to construct an Abstract Syntax Tree (AST)



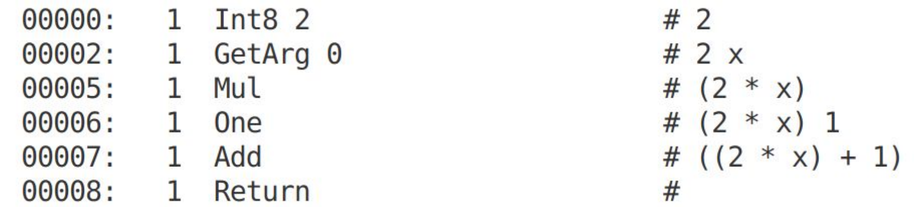
Parsers in SpiderMonkey

- SpiderMonkey has four parsers!
- Syntax Parser
  - Runs quickly
  - Avoids allocating memory when possible
  - Only checks for syntax errors
- Full Parser
  - Runs more slowly
  - Builds the AST
- And there’s a version of each parser for UTF-8 and UTF-16 input.

Virtual Machines and Bytecode

A virtual machine is an abstract definition of a computer

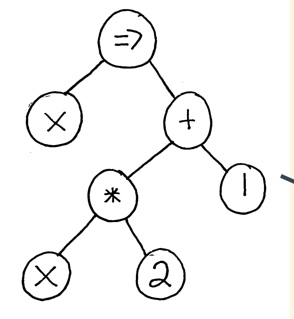
- Defined at a higher level than the instruction set of an actual CPU
- **Bytecodes** are the instructions understood by the virtual machine
    - e.g. bytecode from SpiderMonkey:

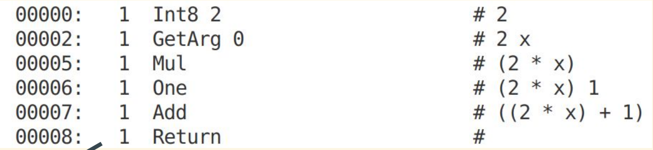


```
00000:    1  Int8 2                           # 2
00002:    1  GetArg 0                         # 2 x
00005:    1  Mul                              # (2 * x)
00006:    1  One                              # (2 * x) 1
00007:    1  Add                              # ((2 * x) + 1)
00008:    1  Return                           #
```

Bytecode Generation

AST (input)





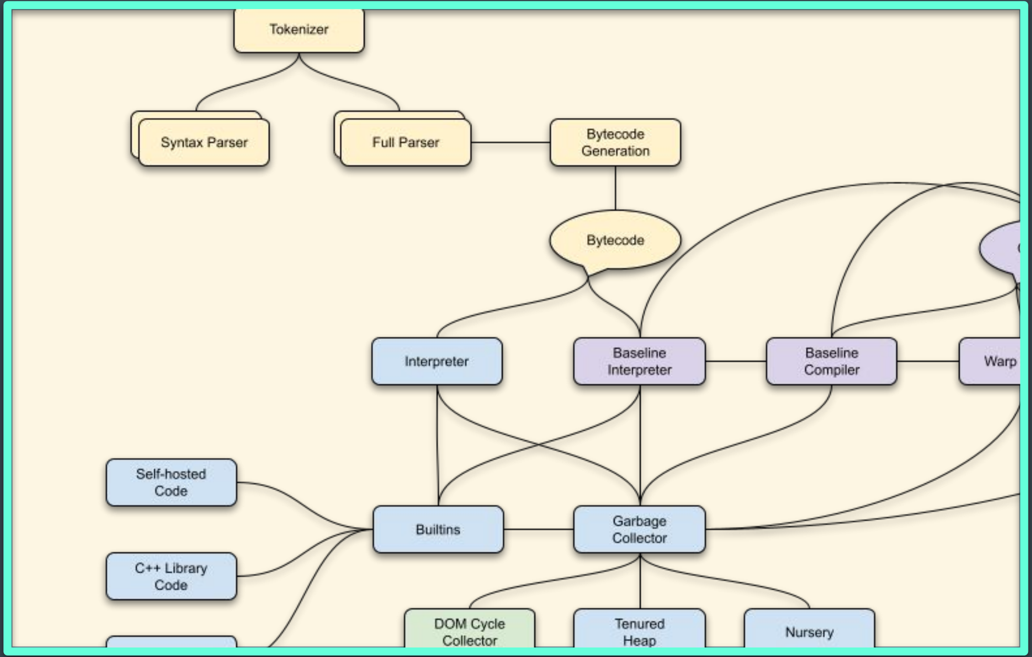
```
00000:    1  Int8 2                          # 2
00002:    1  GetArg 0                        # 2 x
00005:    1  Mul                             # (2 * x)
00006:    1  One                             # (2 * x) 1
00007:    1  Add                             # ((2 * x) + 1)
00008:    1  Return                          #
```

Bytecode (output)

Bytecode Generation

Interpreter (consumer)

Interpreters



What is an interpreter?

Basically a program which runs the instructions for another program :)

Can work by walking the Abstract Syntax Tree directly

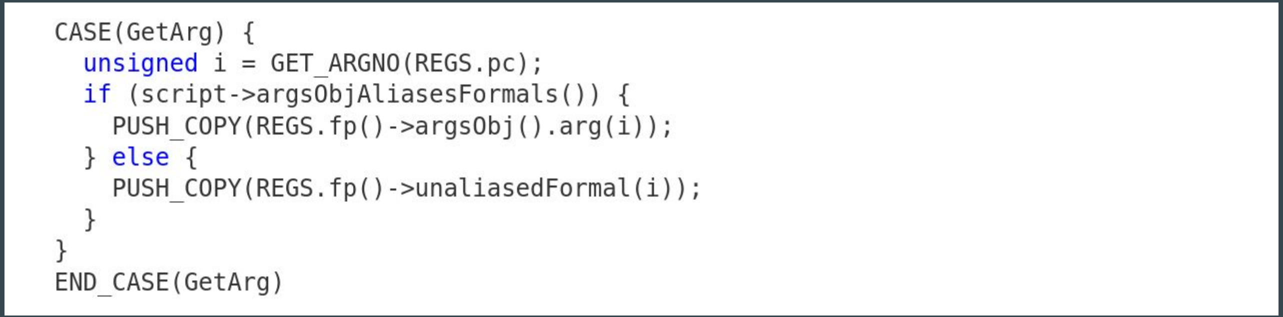
- Early versions of Ruby worked this way
- Simpler, but slower

Or by implementing a virtual machine:

- Interpreter runs in a loop, executing each bytecode one by one
- Implemented as a giant case or goto statement
    - Each label corresponds to a bytecode
- Faster

The “C++” Interpreter

- Simplest interpreter in SpiderMonkey
- Implemented as a giant goto statement inside a loop
- Each label corresponds to a bytecode
    - e.g. GetArg:



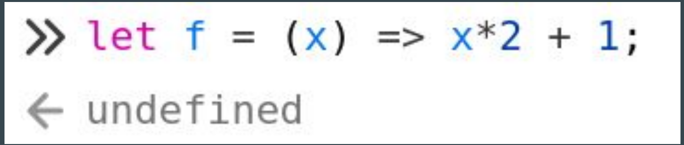
```
CASE(GetArg) {
  unsigned i = GET_ARGNO(REGS.pc);
  if (script->argsObjAliasesFormals()) {
    PUSH_COPY(REGS.fp()->argsObj().arg(i));
  } else {
    PUSH_COPY(REGS.fp()->unaliasedFormal(i));
  }
}
END_CASE(GetArg)
```

Baseline Interpreter

Similar to the C++ interpreter...

But it also collects type information and other metadata as it runs!

- e.g, consider this function again:

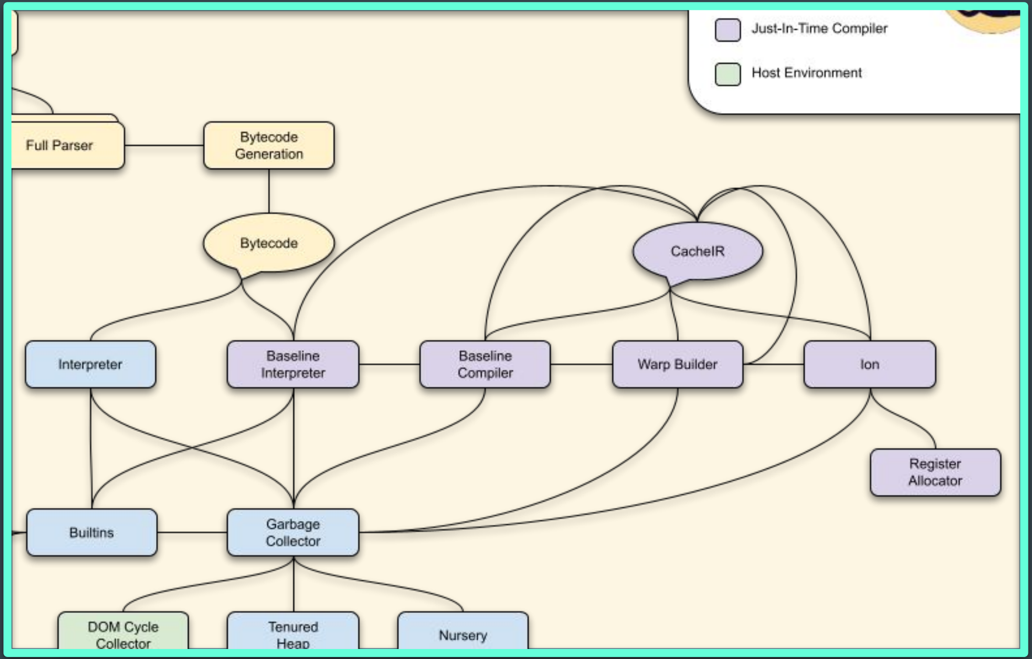


```
>> let f = (x) => x*2 + 1;
<- undefined
```

- The parameter **x** is likely a number, probably an integer

How can we use this information?

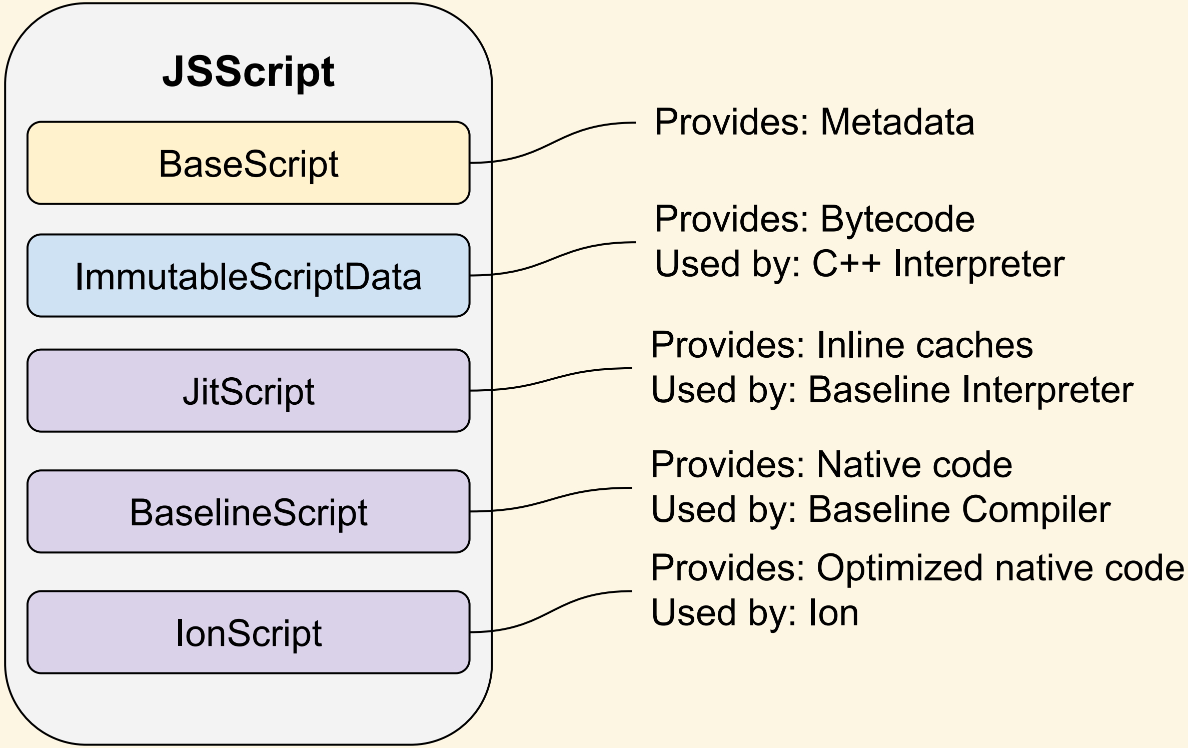
Just-in-time Compilers



Just-in-time Compilers

- Use data gathered while interpreting code to choose what to compile
- Most JavaScript code is only executed once!
- Trade off between compile time and execution time:
    - Baseline interpreter: no compile time at all, but runs slowly
    - Baseline compiler: faster compile time, generated code is not optimized
    - Ion: slower compile time, generated code is optimized

C++ Representation of JavaScript Scripts



Provides: Metadata

Provides: Bytecode
Used by: C++ Interpreter

Provides: Inline caches
Used by: Baseline Interpreter

Provides: Native code
Used by: Baseline Compiler

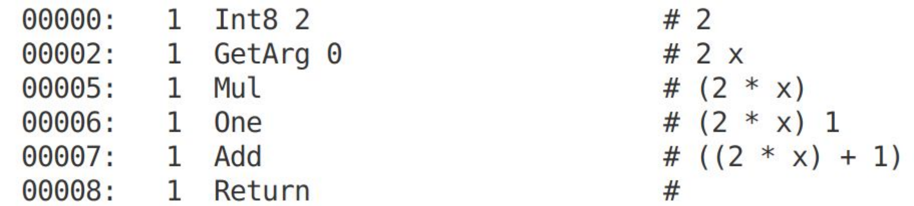
Provides: Optimized native code
Used by: Ion

Inline Caches

- A dynamic dispatch is when we branch based upon type information
  - e.g. whether the function is applied to strings or integers
- An inline cache stores the result of previous dynamic dispatches
  - Avoids cost of branching
  - Code is specialized for the particular type
  - Stored **inline** with the function itself
- This also gives us metadata:
  - Type information
  - Which parts of a function are actually used

Baseline Interpreter

- Basic idea: replace opcodes with **stubs** in inline caches



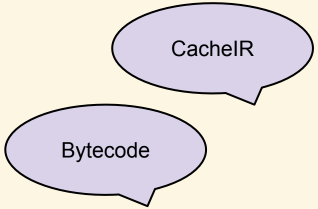
```
00000:   1  Int8 2                          # 2
00002:   1  GetArg 0                        # 2 x
00005:   1  Mul                             # (2 * x)
00006:   1  One                             # (2 * x) 1
00007:   1  Add                             # ((2 * x) + 1)
00008:   1  Return                          #
```

- The baseline interpreter will create stubs in the inline cache for:
  - BinaryArith.Int32Mul
  - BinaryArith.Int32Add
- Executing this code will be much faster next time (maybe 6x faster)
- Stores data in an intermediate format called **CacheIR**

Baseline compiler

- Generates native code for each bytecode in the script
- Avoids overhead of interpreter switch / dispatch loop
    - Still uses Inline Caches to handle different types
- Is maybe 2x faster than Baseline Interpreter

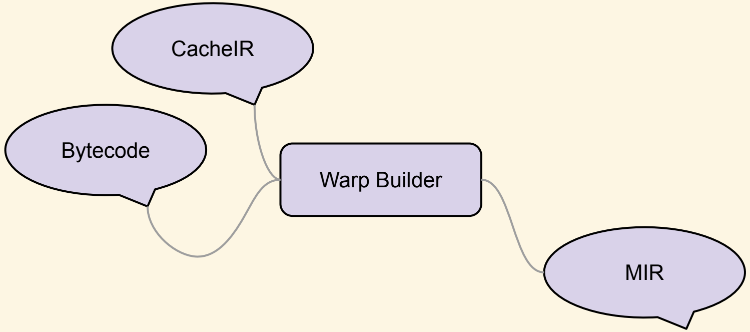
Ion



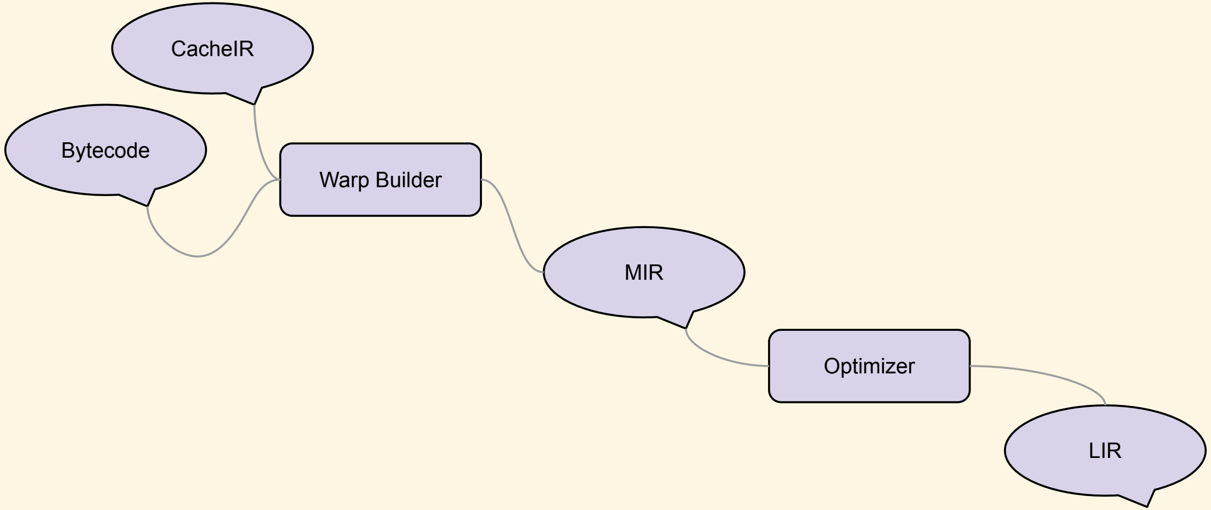
CacheIR

Bytecode

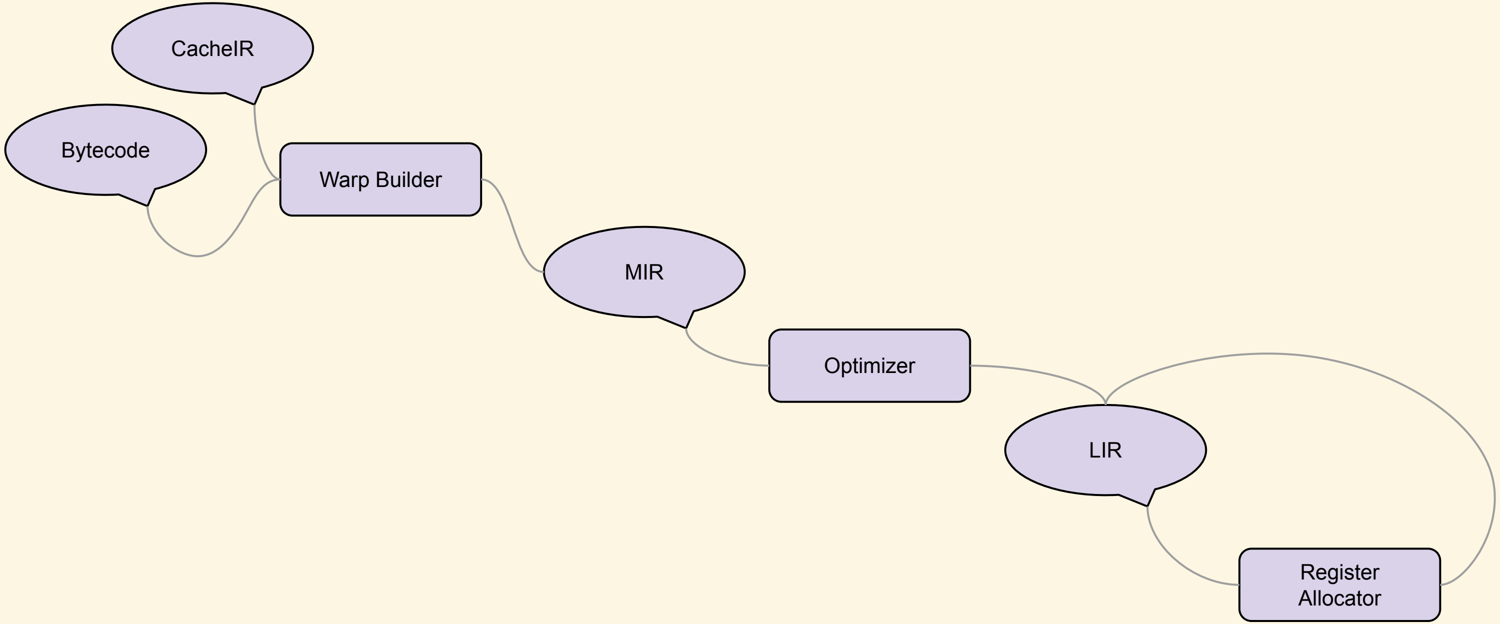
Ion



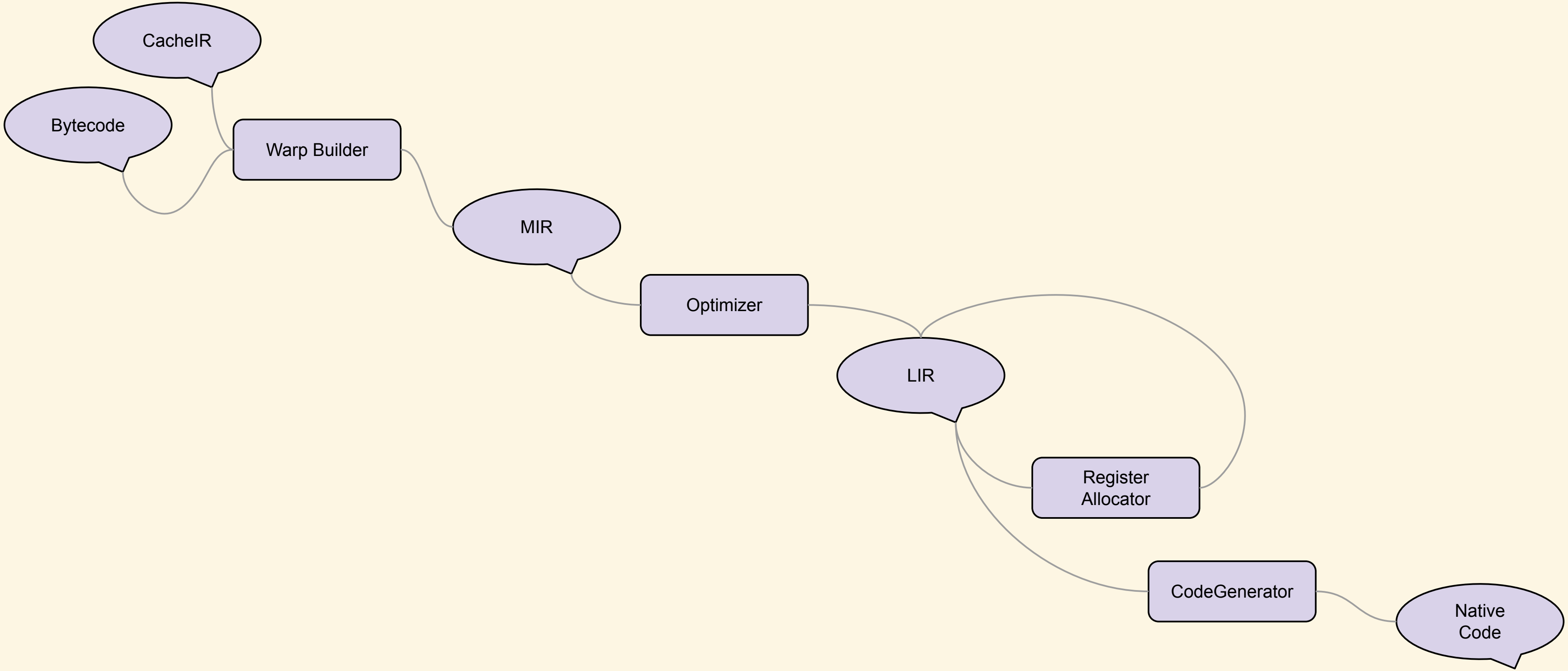
Ion



Ion

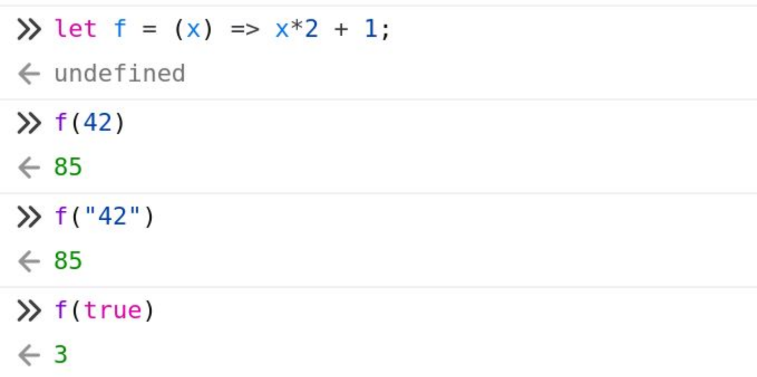

CacheIR
Bytecode
Warp Builder
MIR
Optimizer
LIR
Register Allocator

Ion


CacheIR
Bytecode
Warp Builder
MIR
Optimizer
LIR
Register Allocator
CodeGenerator
Native Code

Bailouts

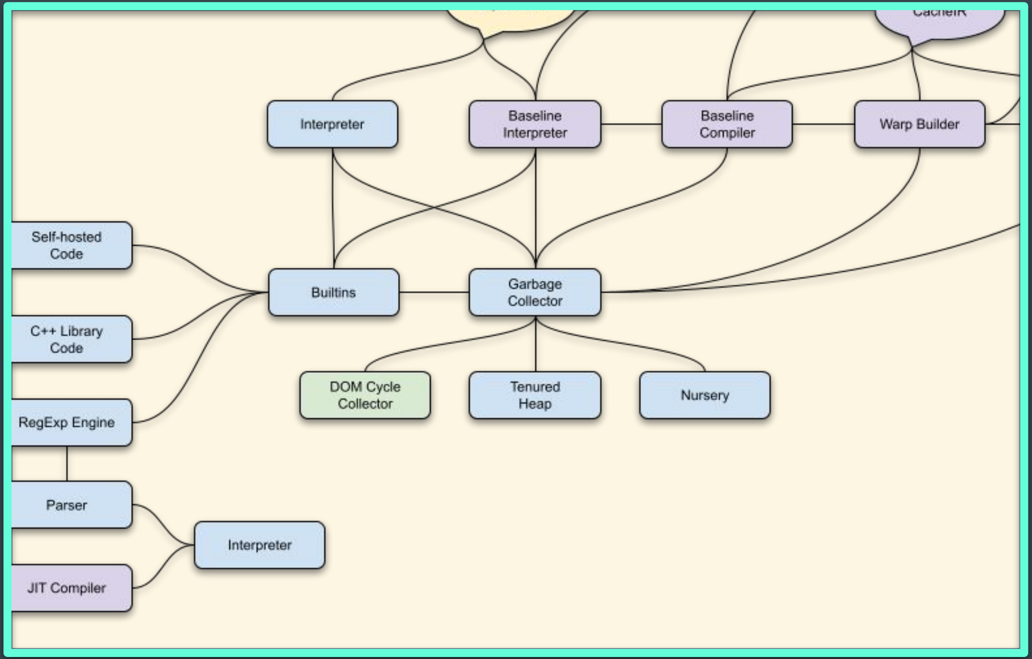
- Happen when our assumptions about types are wrong :(
- JavaScript is a flexible language:



```
>> let f = (x) => x*2 + 1;
<- undefined
>> f(42)
<- 85
>> f("42")
<- 85
>> f(true)
<- 3
```

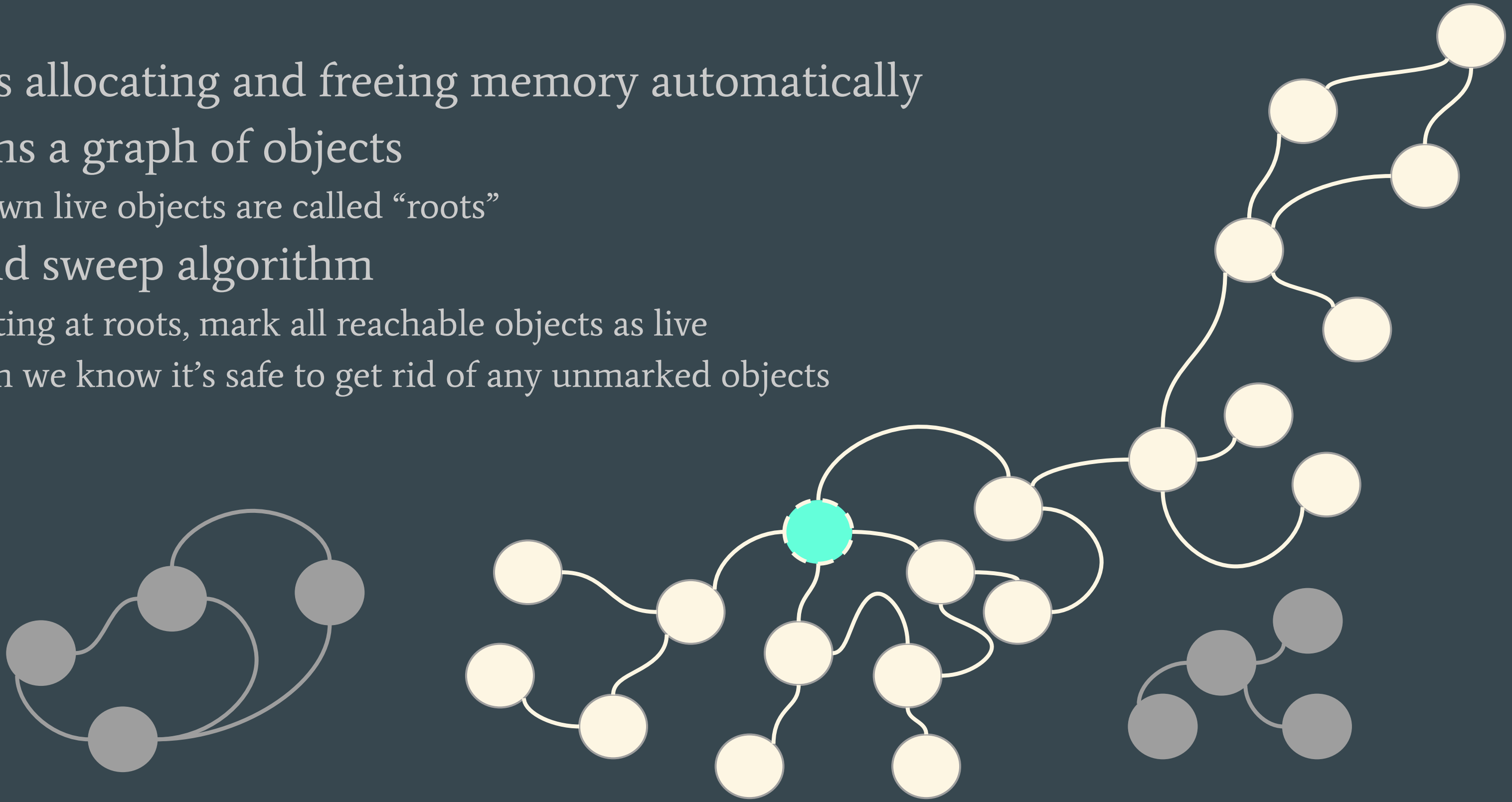
- Forces us to drop back to baseline interpreter
  - But we can attach a new stub to handle the different type
  - If this happens often enough, ion compiled code is invalidated

Garbage Collection



What’s a garbage collector?

- Manages allocating and freeing memory automatically
- Maintains a graph of objects
  - Known live objects are called “roots”
- Mark and sweep algorithm
  - Starting at roots, mark all reachable objects as live
  - Then we know it’s safe to get rid of any unmarked objects

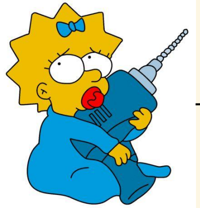


Generational Garbage Collection

Newly Allocated Object

Generational Garbage Collection

Newly Allocated Object



Nursery

Generational Garbage Collection

Newly Allocated Object

Nursery

Minor GC

Generational Garbage Collection

AMUSING TOMBSTONES
THANK YO
FOR NOT
USSING
THE
UTSIDE
WORLD

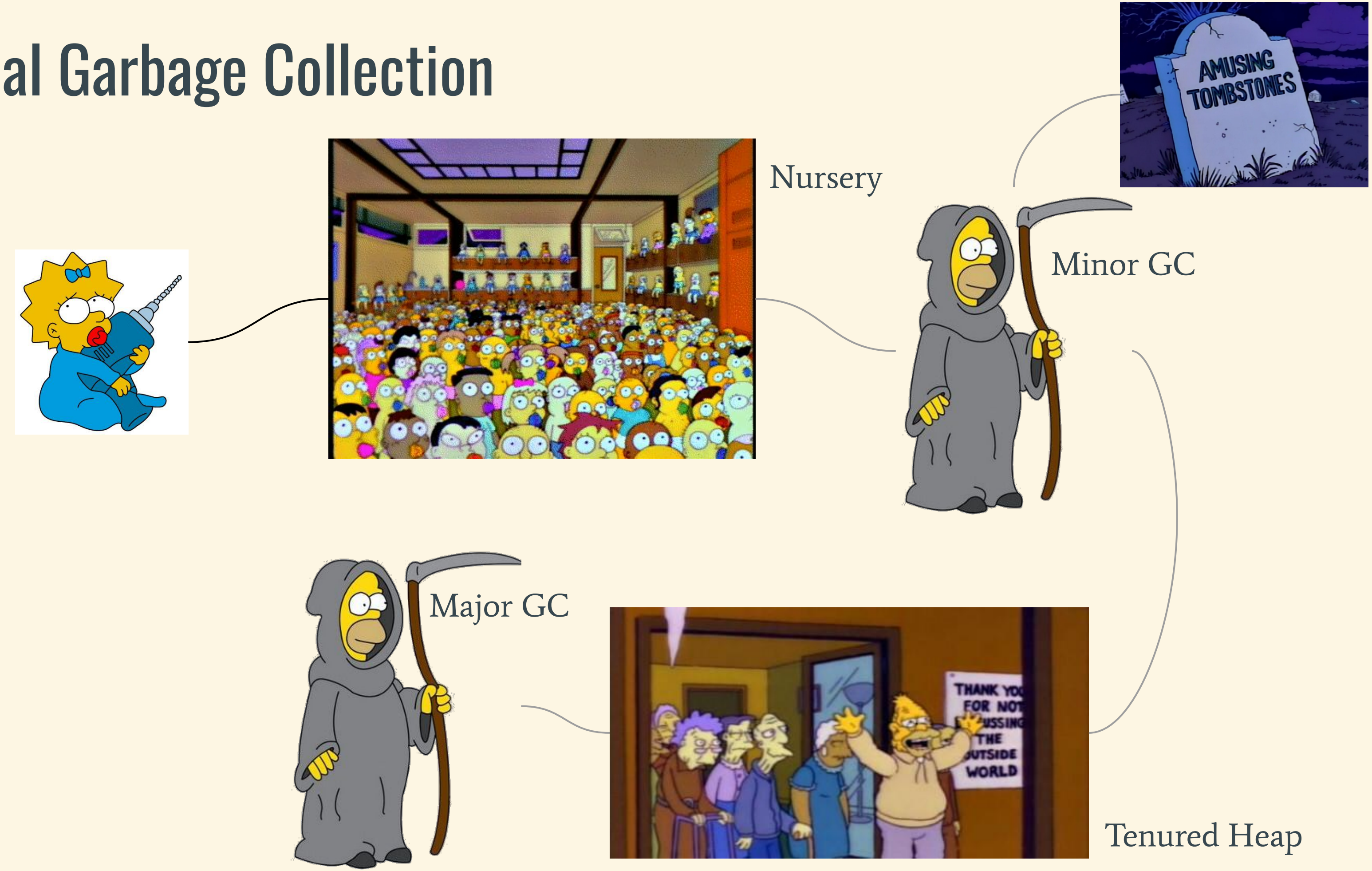
Newly Allocated Object

Nursery

Minor GC

Tenured Heap

Generational Garbage Collection



Newly Allocated Object

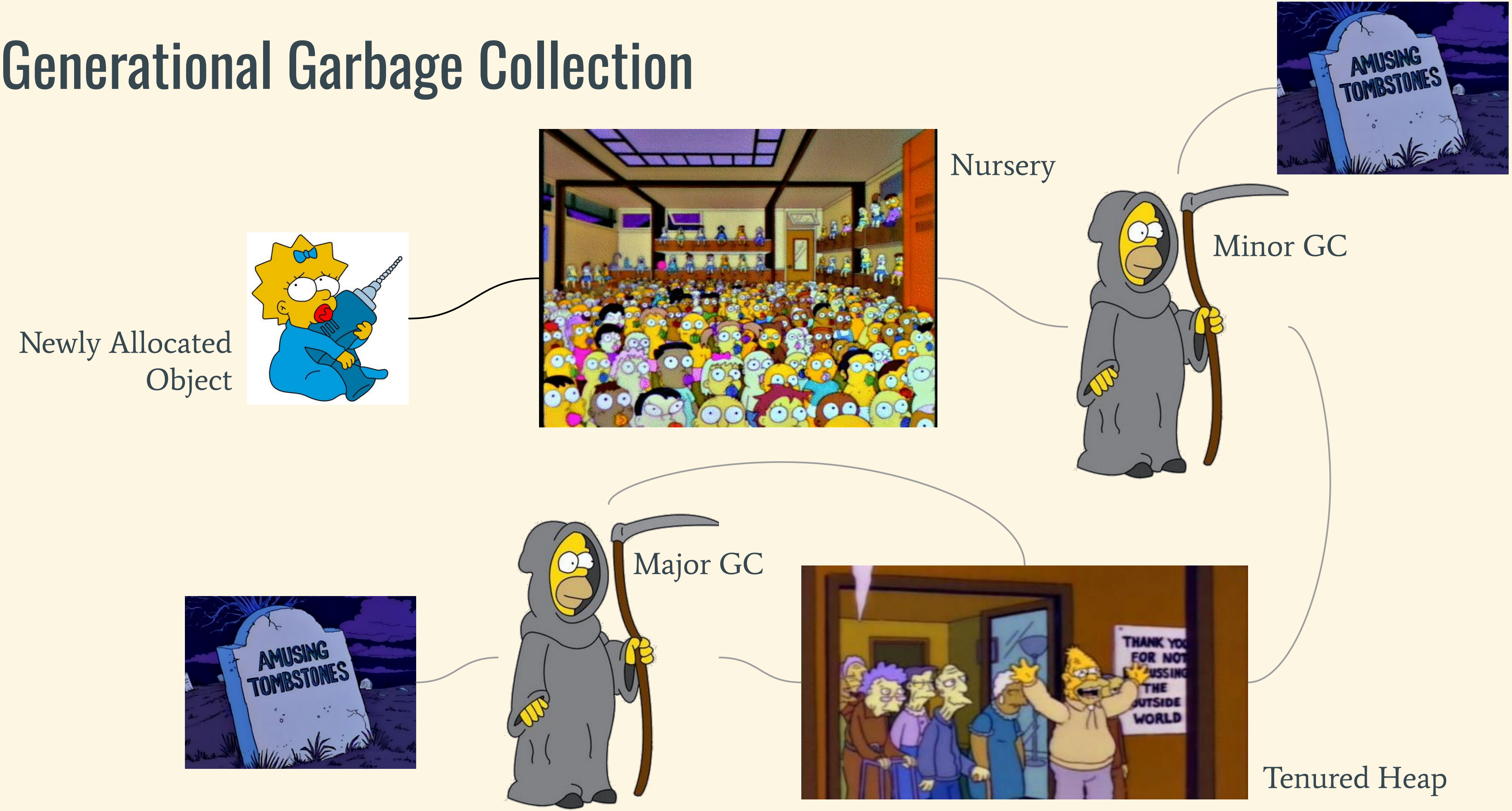
Nursery

Minor GC

Tenured Heap

Major GC

Generational Garbage Collection



Newly Allocated Object

Nursery

Minor GC

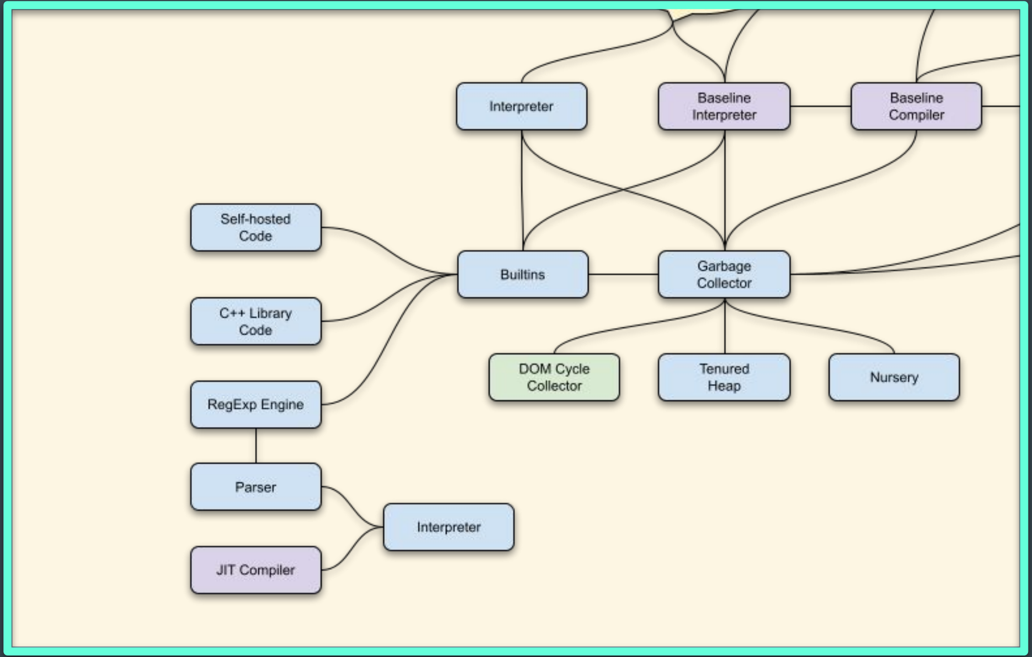
Major GC

Tenured Heap

Cycle-collection

- Cycles between the host environment and JavaScript
    - e.g. a C++ DOM object holds a reference to a JavaScript callback, which holds a reference to the DOM object
- The Garbage Collector can’t see these cycles
- In Firefox, DOM objects are ref-counted and allocated separately
    - Need careful cooperation between the DOM cycle collector and the garbage collector

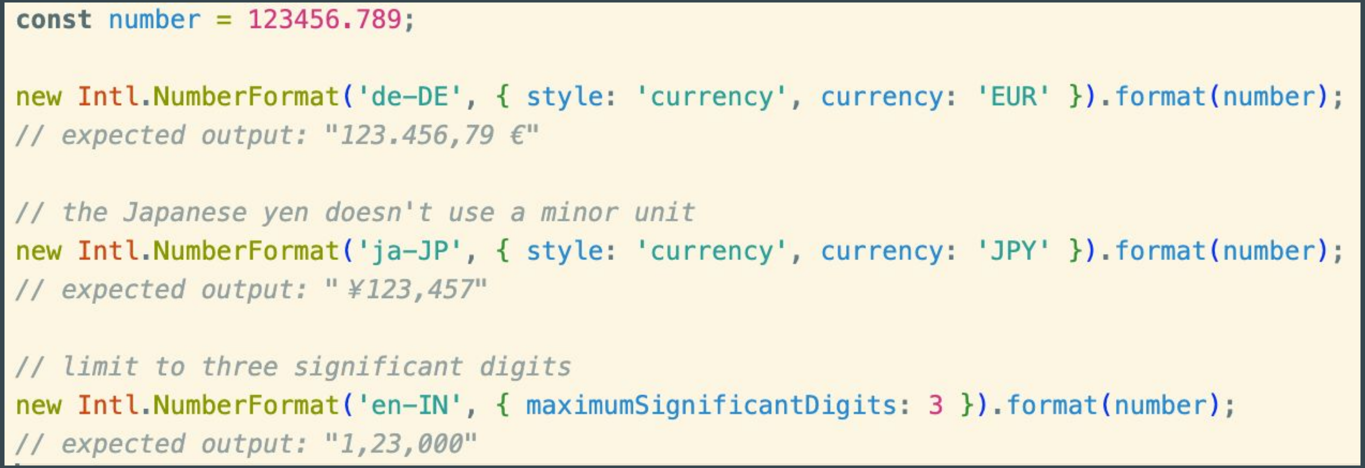
Builtins


Interpreter
Baseline Interpreter
Baseline Compiler
Self-hosted Code
C++ Library Code
RegExp Engine
Builtins
Garbage Collector
DOM Cycle Collector
Tenured Heap
Nursery
Parser
Interpreter
JIT Compiler

Builtins

- Provide “standard library” functionality:
  - e.g. Math, String, RegExp, Intl, Map, Set etc.
- Implemented in a mix of “Self-hosted” JavaScript and C++
  - JavaScript where we can :)
  - C++ when necessary:
    - Sometimes performance
    - Sometimes to be able to use third-party libraries like ICU4C
- Self-hosted JavaScript is a subset of JavaScript
  - Restricted to avoid security problems, e.g. with prototype pollution

Example: Internationalization

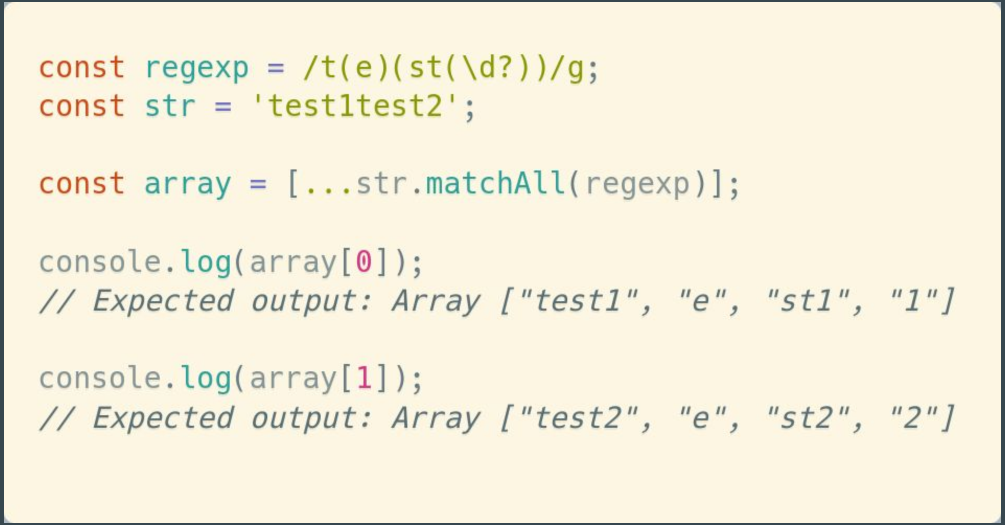


```
const number = 123456.789;

new Intl.NumberFormat('de-DE', { style: 'currency', currency: 'EUR' }).format(number);
// expected output: "123.456,79 €"

// the Japanese yen doesn't use a minor unit
new Intl.NumberFormat('ja-JP', { style: 'currency', currency: 'JPY' }).format(number);
// expected output: "￥123,457"

// limit to three significant digits
new Intl.NumberFormat('en-IN', { maximumSignificantDigits: 3 }).format(number);
// expected output: "1,23,000"
```

Example: Internationalization

- Self-hosted code
  - Provides API exposed to JavaScript
- C++ code
  - Implementation of objects like **Intl.NumberFormat**
  - Handle integration with internationalization libraries
- ICU4C and CLDR
  - Standard internationalization library used by browsers and operating systems
  - CLDR provides underlying data for each language and locale supported

Example: Regular Expressions



```
const regexp = /t(e)(st(\d?))/g;
const str = 'test1test2';

const array = [...str.matchAll(regexp)];

console.log(array[0]);
// Expected output: Array ["test1", "e", "st1", "1"]

console.log(array[1]);
// Expected output: Array ["test2", "e", "st2", "2"]
```

Example: Regular Expressions

- Self-hosted code
  - Provides API exposed to JavaScript
- C++ code
  - Handle integration with underlying regular expression library
- Irregexp library
  - Irregexp is like a microcosm of the engine as a whole, it has its own:
    - Parser
    - Bytecode
    - Interpreter
    - Just-in-time compiler
  - Originally written for V8, we have shim layer which emulates V8's memory management

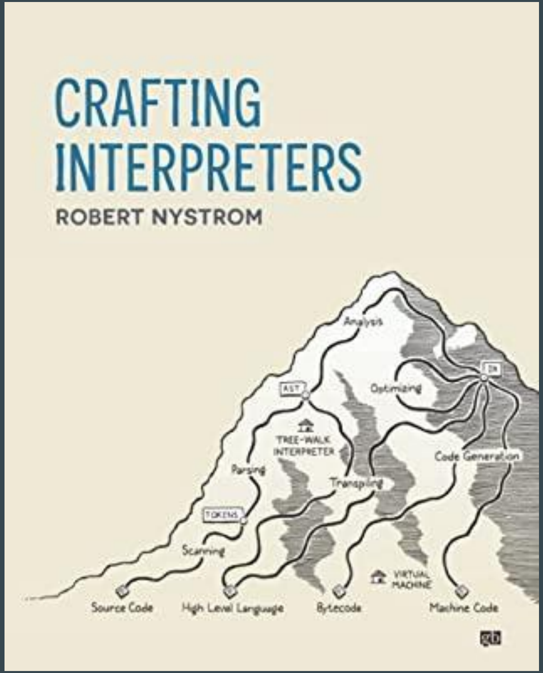
Interested in contributing to SpiderMonkey?

Many TC39 proposals can be implemented in JavaScript using self-hosting code

You don’t have to be an expert :)

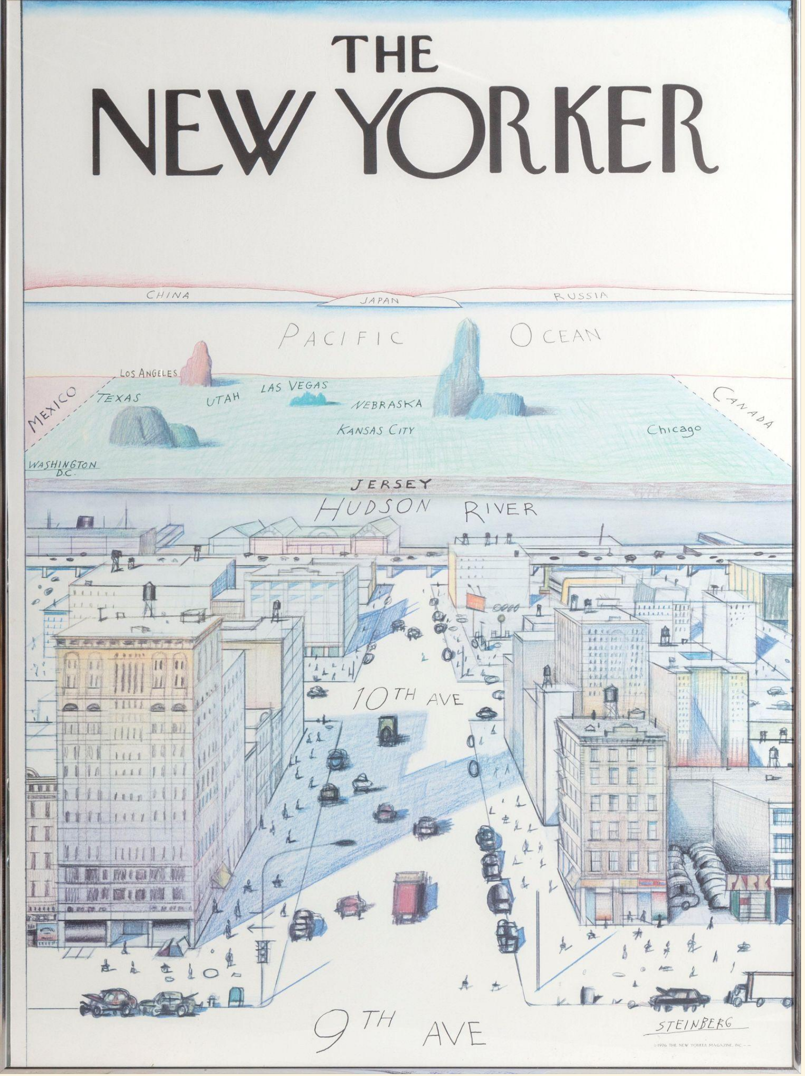
Contact us:

- Matrix: #spidermonkey:mozilla.org
- Bugzilla: https://bugzilla.mozilla.org/show_bug.cgi?id=1435811



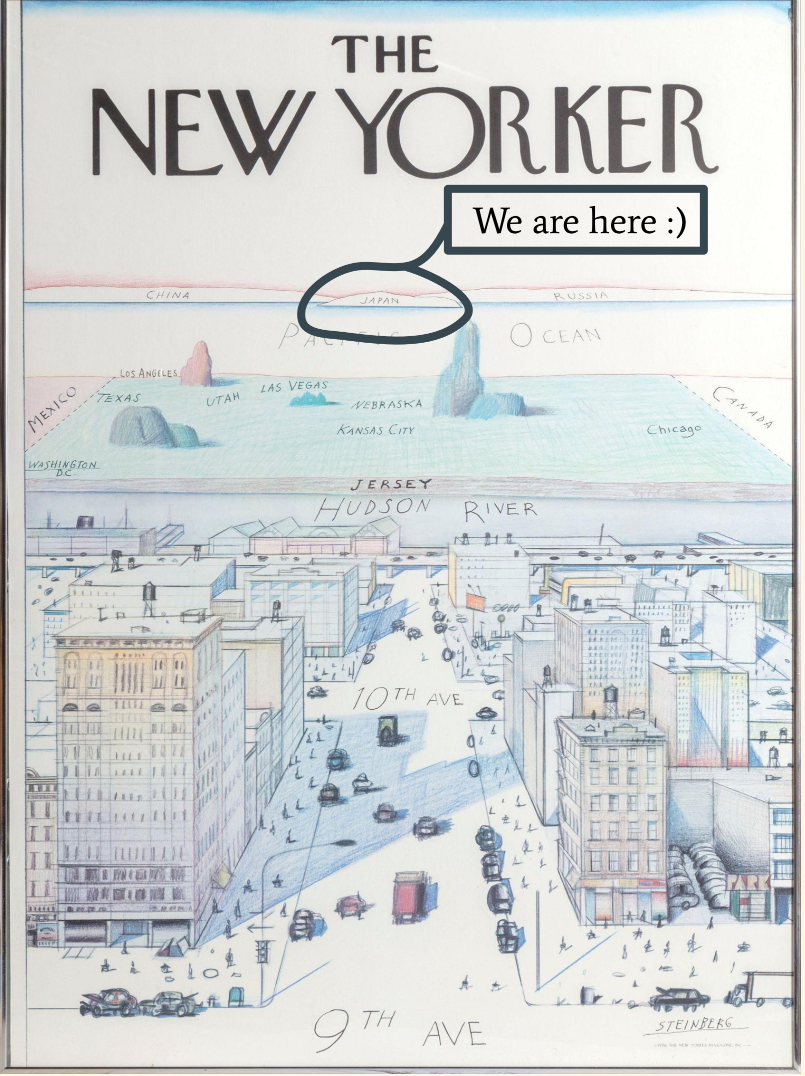
https://craftinginterpreters.com/

Byte-sized architectures



The view from 9th avenue

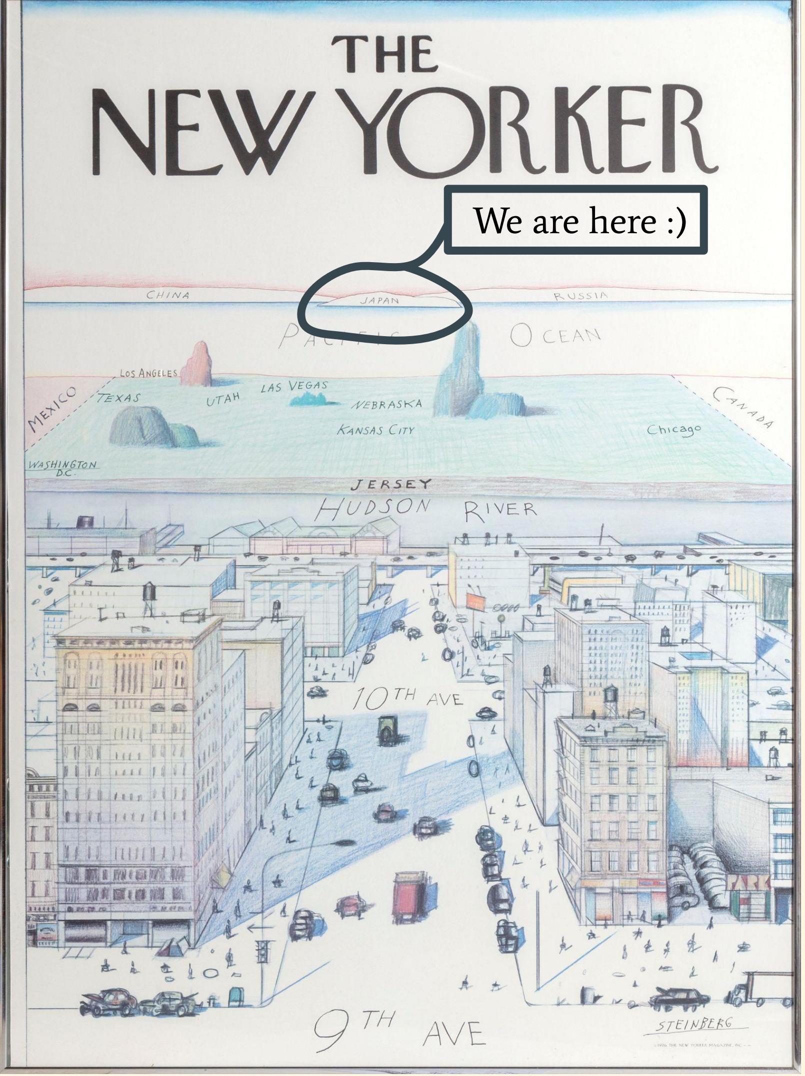
- We all have our own point of view on the world...



The view from 9th avenue

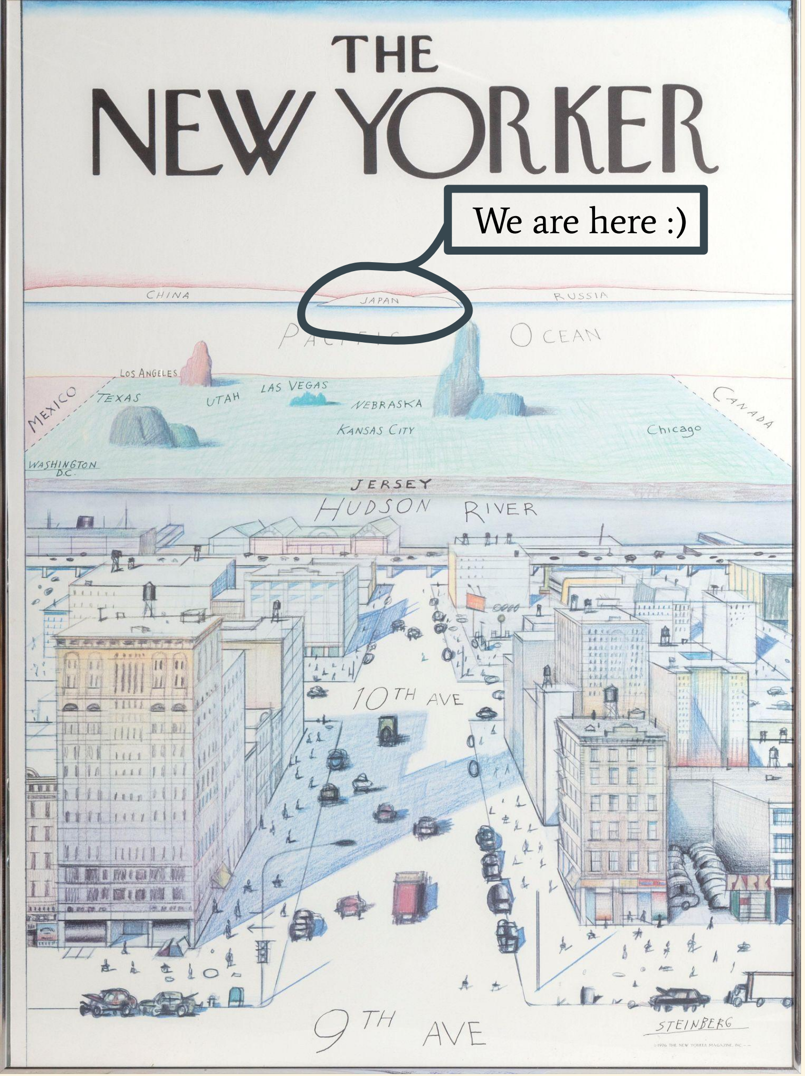
- We all have our own point of view on the world...

The view from 9th avenue



We are here :)

- We all have our own point of view on the world...
- That’s not a bad thing!



The view from 9th avenue

- We all have our own point of view on the world...
- That’s not a bad thing!
- Everyone has something to teach us :)

How does this relate to software?

Every member of the team has a different point of view of the system

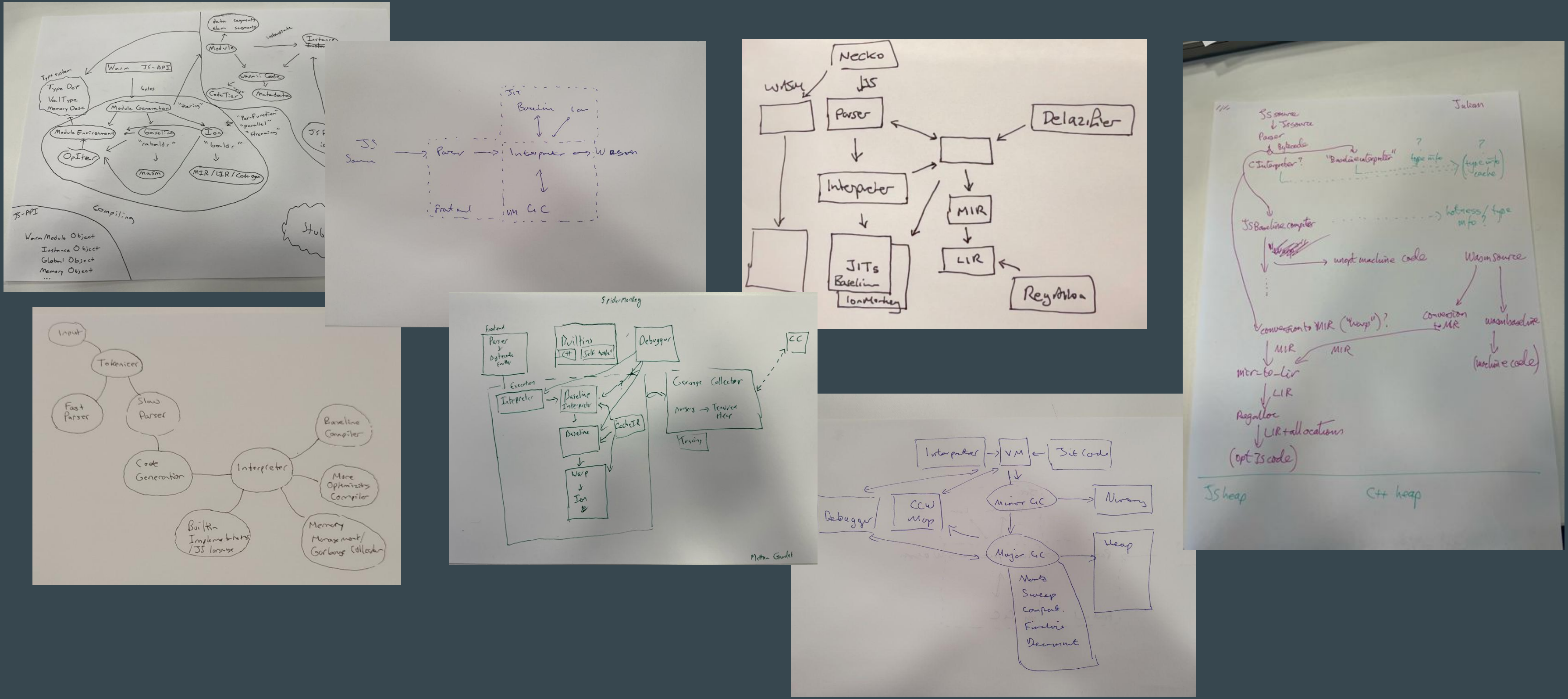
For a really complicated system, no one can understand the whole thing

So how can we build a shared understanding?

Byte-sized Architectures

- The team gets together for about an hour
- For the first five minutes everyone draws an architecture diagram
- Then we take turns showing our drawings and discussing them
- Not our idea :)
  - Comes from Andrea Magnorsky's bytesize architectures (https://www.roundcrisis.com/2021/09/28/bytesize-architecture-sessions/, https://bytesizearchitecturesessions.com/)
  - Her version is a bit more complicated

A collection of our byte-sized architectures



The Benefits

- Quiet time together as a team, working on our drawings
    - Our team is completely remote, so this is important for us
- The drawings are definitely useful
    - I used ours to help prepare this presentation :)
- But more important: the questions and conversations

Psychological Safety

Feel safe to raise ideas, voice opinions, ask questions and admit mistakes without fearing the consequences.

Byte-sized architectures help build psychological safety in the team:

- Admit when we don’t know something
- Ask questions

ありがとうございました

thank you!!