BUG JOURNAL

#1286997

https://www.twitch.tv/codehag

@ioctaptceb

Test 262. report

test name: Prefix-increment / S11.4.4_A6_T2.js

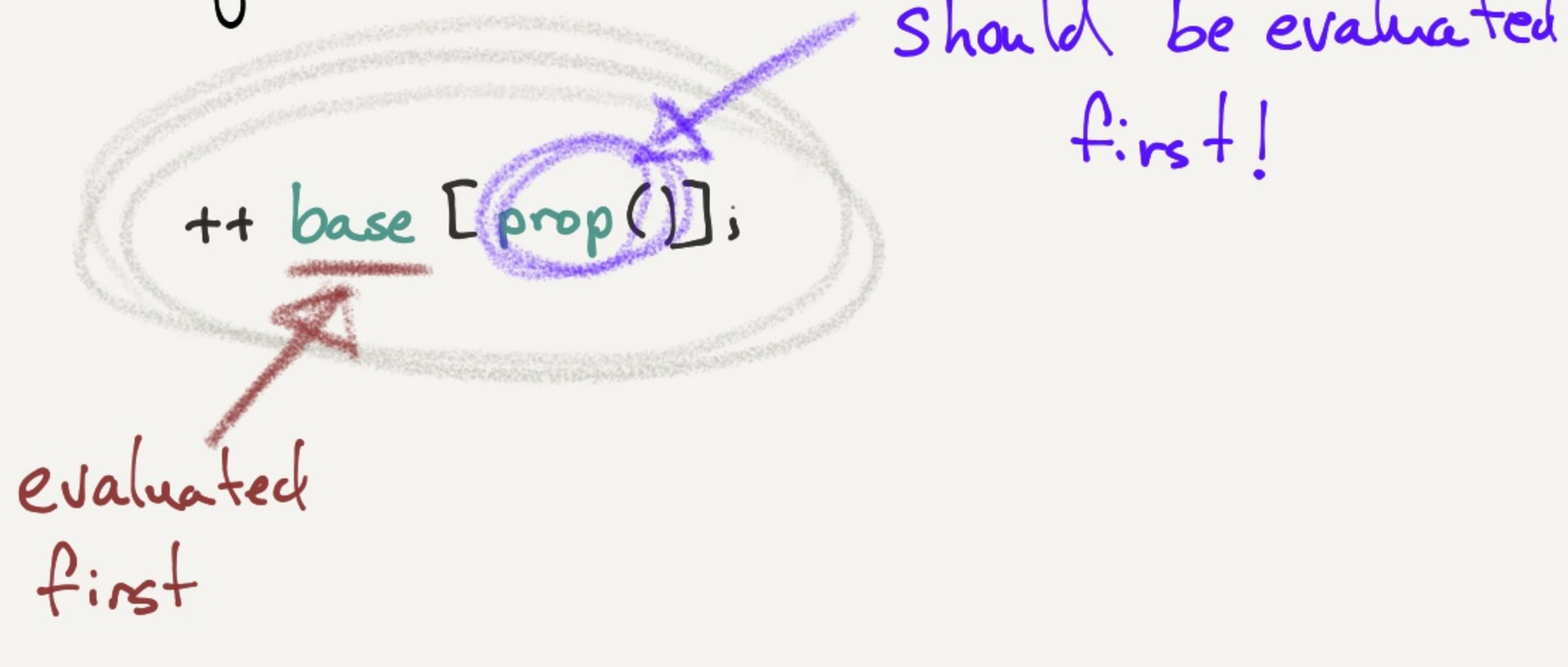
test body:

```
function DummyError() {}
assert.throws(DummyError, function() {
    var base = undefined
    var prop = function() {
        throw new DummyError();
    }
    ++base[prop()];
});
```

Expected: DummyError

Actual outcome: "Undefined has no properties"

Problem summary:



The problem ...

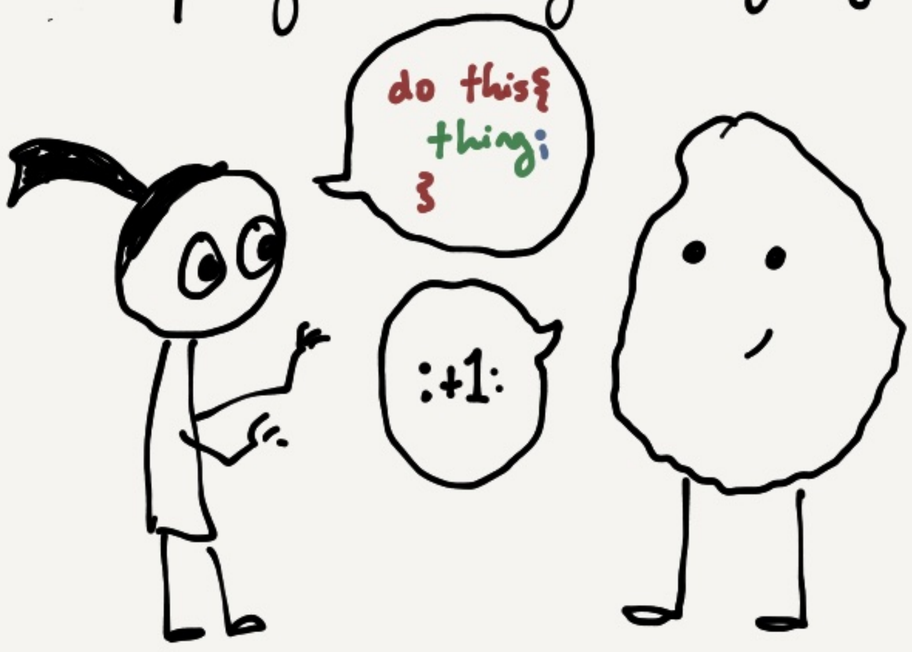
```
0000x:   Check Obj Coercibe   ← wrong spot!
000xx:   Get GName "x"
xxxxx:   ...
xxxxx:   To Property key
```

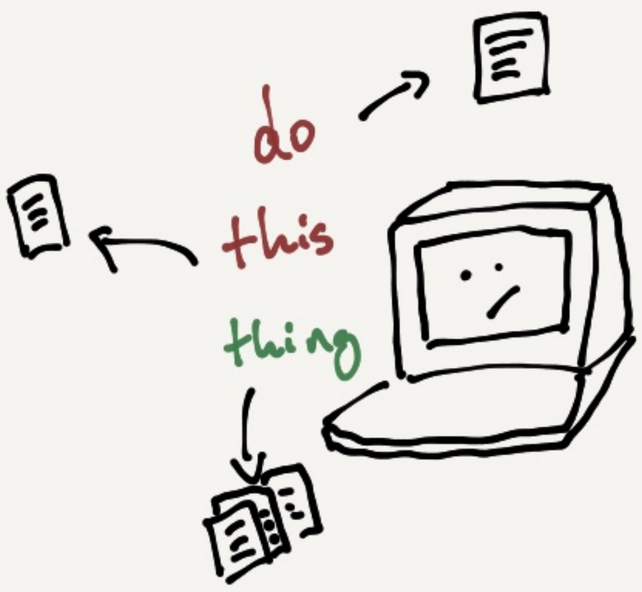
Our Bytecode was in the wrong order...

```
000xx:   Get GName "x"
xxxxx:   ...
xxxxx:   Check Obj Coercible
xxxxx:   To Property key
```

What is Bytecode?

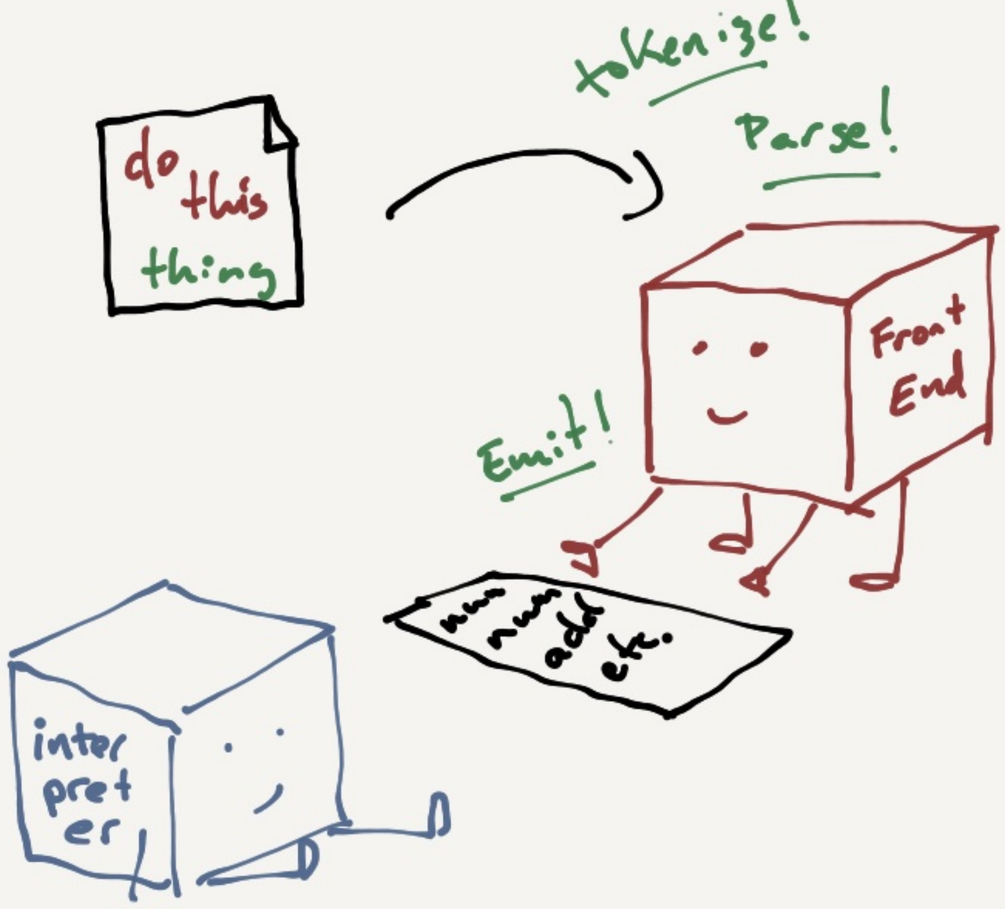
High level programming languages (like JavaScript) help programmers communicate the Intent of a set of instructions to other programmers.





But! these instructions are not something a computer can work with efficiently

So, interpreters and compilers translate sourcecode into an easier-to-process form. BYTECODE!



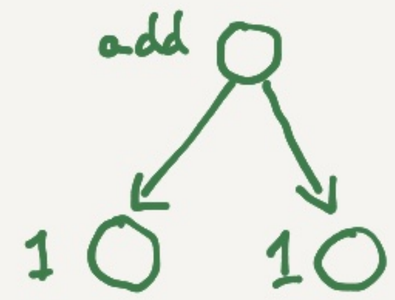
A concrete example from SpiderMonkey

Source code: add.js ⟶ Abstract Syntax Tree

add.js

```
1 d:s(  // helper disassembly function
2   (a, b) => a + b
3 )
```

run with dist/bin/js add.js (see how to build)



or

Ast.json-ish

```
{ node: add
  Left: {
    ...
    value: 1,
  },
  Right: {
    ...
    value: 1,
  }
}
```

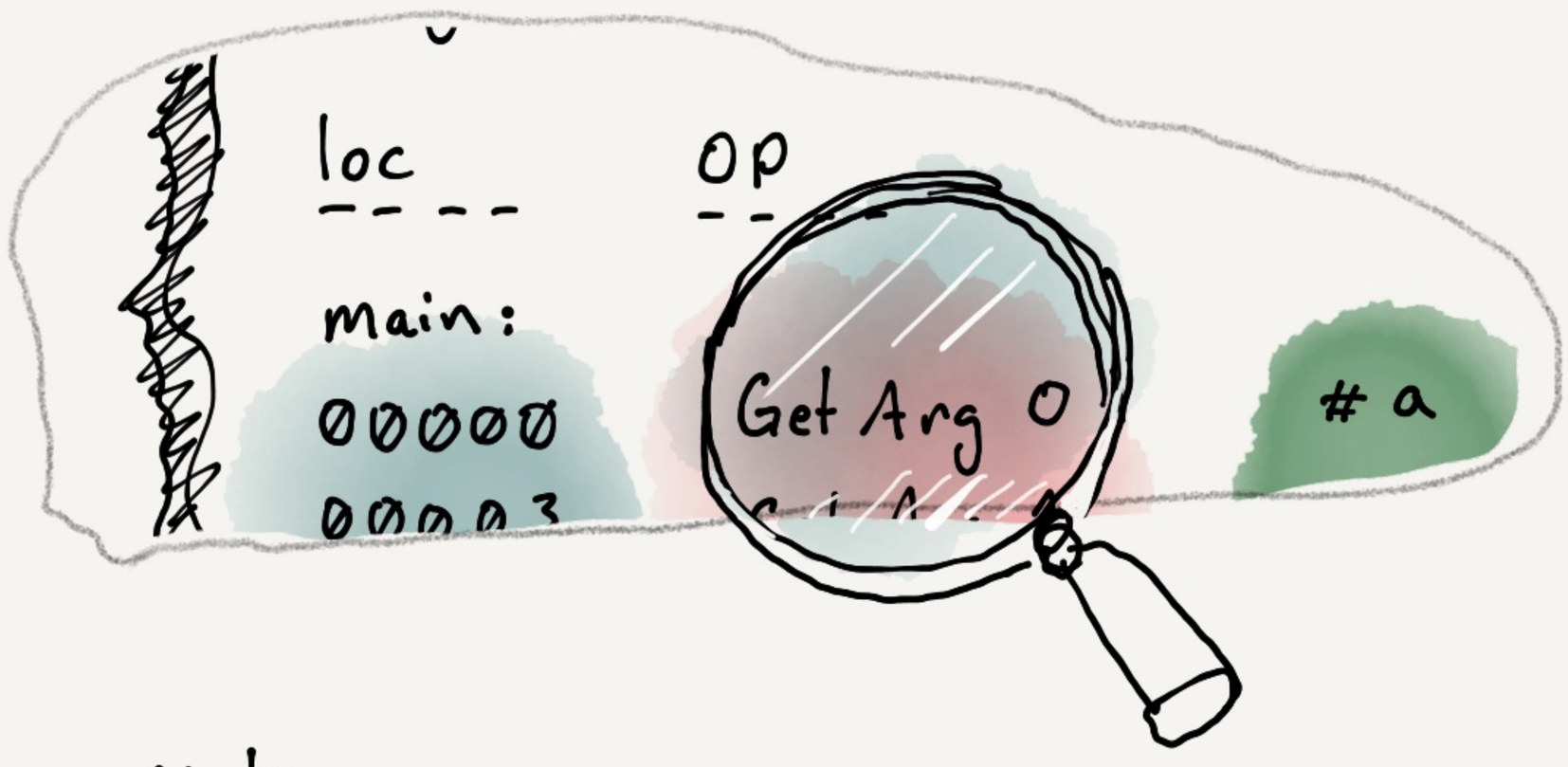
flags: LAMDA ARROW

loc	op	
main:		
00000	Get Arg 0	# a
00003	Get Arg 1	# a b
00006	Add	# (a + b)
00007	Return	# blank
00008	RetRval	# !! unreachable !!

offsets

bytecode operator name

The value stack



Zooming in on Get Arg

Note:

0 0 0 0 0
0 0 0 0 3 ← offset is the size!

Byte code stack

use x/1x <address> in gdb or LLDB to get this

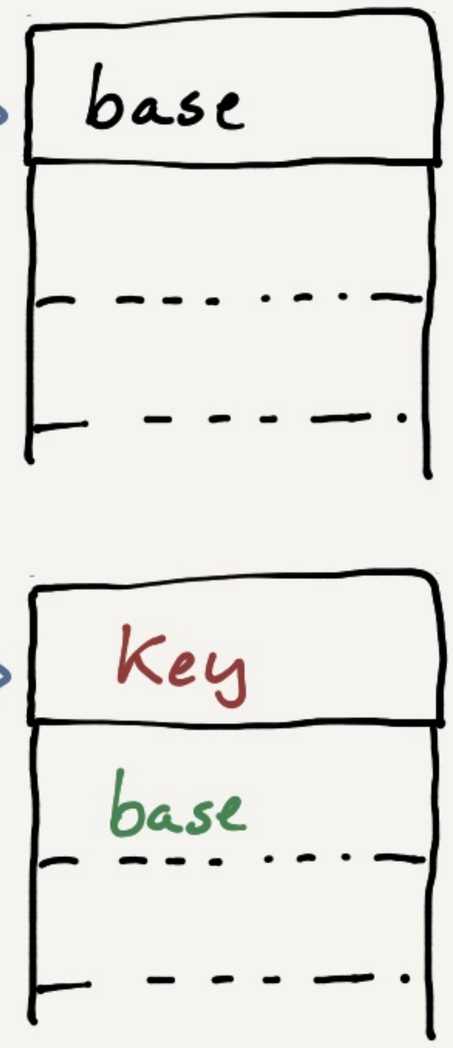
offset	Address (PC → program counter)	value	
0:	0x0000000010092af7e1	0xba	Get Arg
1:	" " 10092af7e2	0x00	
2:	" " 10092af7e3	0x00	
3:	" " 10092af7e4	0xba	Get Arg
4:	" " 10092af7e5	0x01	
5:	" " 10092af7e6	0x00	

0xba → representation of Get Arg operator

0x00 + 0x00 → u16 "0" represented as u8

First fix, steps:

① push base →



② push Key →

we have to swap the values

because Check Obj Coercible uses the value at the top of the stack

④ Check Obj Coercible →

we have to swap again

⑤ Swap

⑥ To Property Key →

③ Swap

Ⓧ! To Property Key →

THINKING OF SOLUTIONS...

```
000xx:   Get GName "x"
xxxxx:   ...
xxxxx:   Check Obj Coercible
xxxxx:   To Property key
```

← Moving the byte code? doesn't work

```
000xx:   Get GName "x"
xxxxx:   ...
xxxxx:   Swap
xxxxx:   Check Obj Coercible
xxxxy:   Swap
xxxxx:   To Property key
```

} 3 byte codes?
Works, but not Ideal.

```
000xx:   Get GName "x"
xxxxx:   ...
xxxxx:   * Prepare Set Elem *
```

← New Byte Code?

The Difference between

Interpreters

and

Compilers

Big loop with a lot of case statements

Interpreter

```
for code in bytecode
    match
        "A": do instruction ✡
        "B": do instruction △
        "C": do instruction □
```

Bytecode

"A"
"A"
"A"
"C"
"B"

Compiler (it uses the same instructions)

Also uses a loop with case statements

BUT outputs a compiled script.

```
// compled script for "AAACB"

do instruction ✡
do instruction ✡
do instruction ✡
do instruction △
do instruction □
```

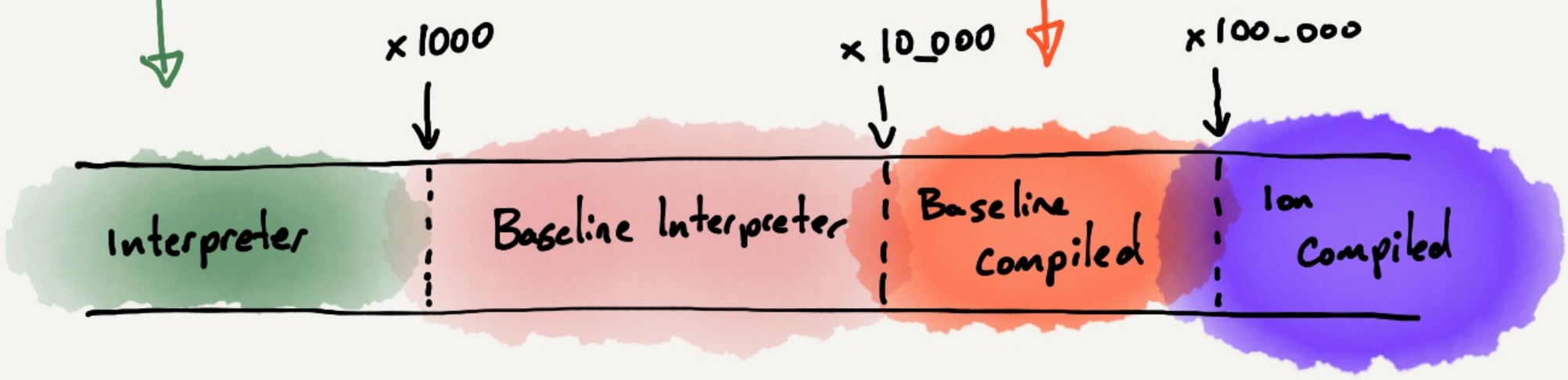
SpiderMonkey

Compiler phases

loops over bytecode and executes "NATIVE" functions

C++

- Uses inline caches
- compiles code to machine language using the Macro assembler.
- Does not have registers!



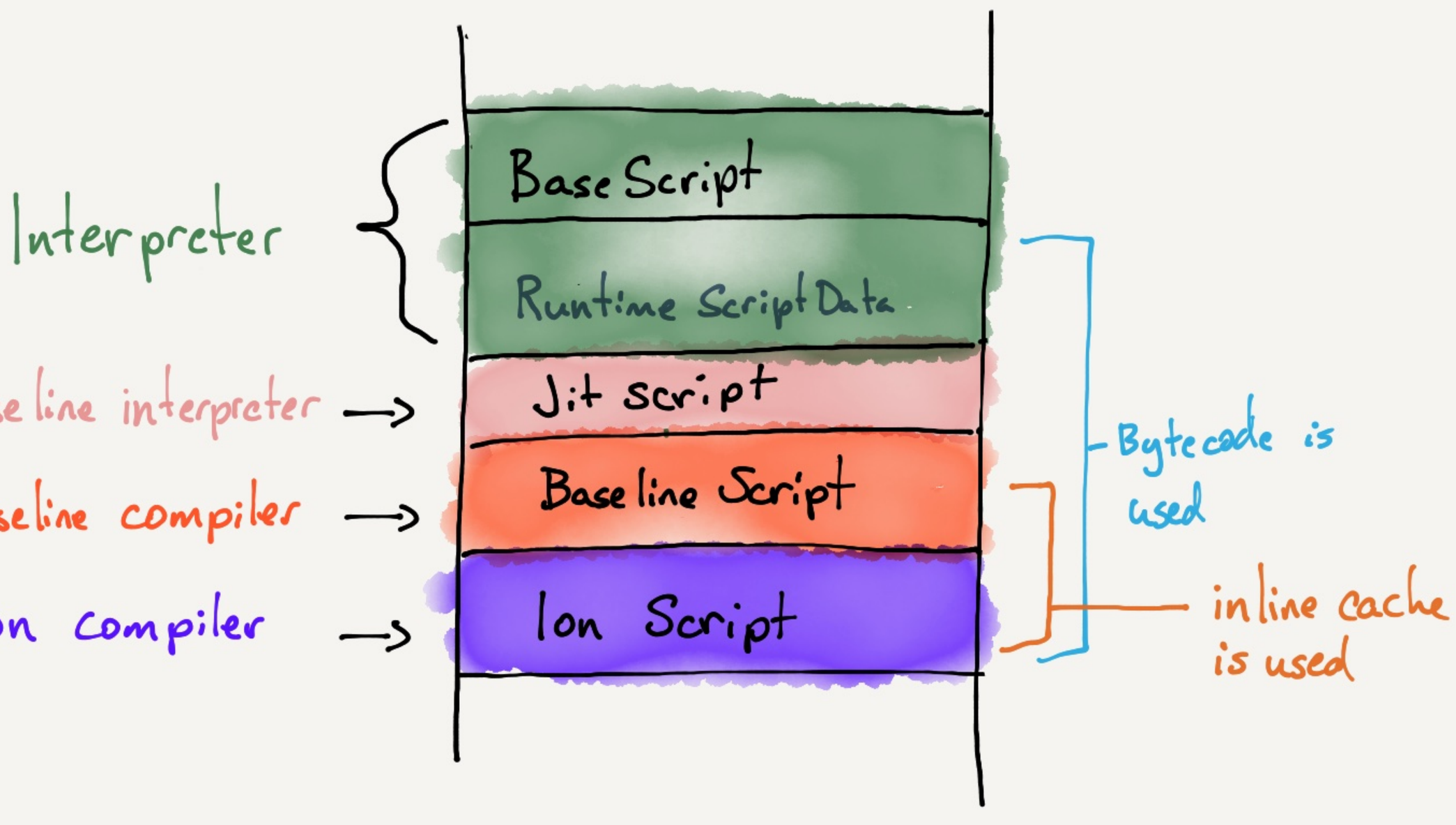
Similar to Interpreter but gathers information about types and creates something called an "inline cache"

- uses inline caches
- compiles code to machine language using the Macro Assembler
- Has register allocation

SpiderMonkey

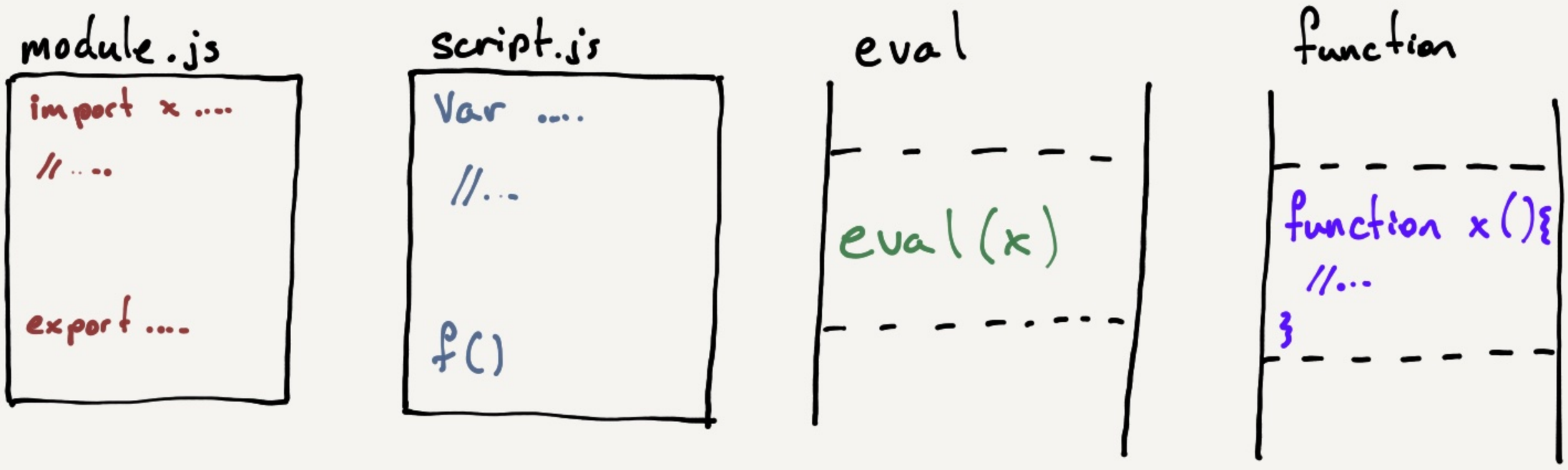
Script Representation

Scripts are represented in SpiderMonkey as a data structure called "JSScript". It has a few slots where it stores compiler related information.

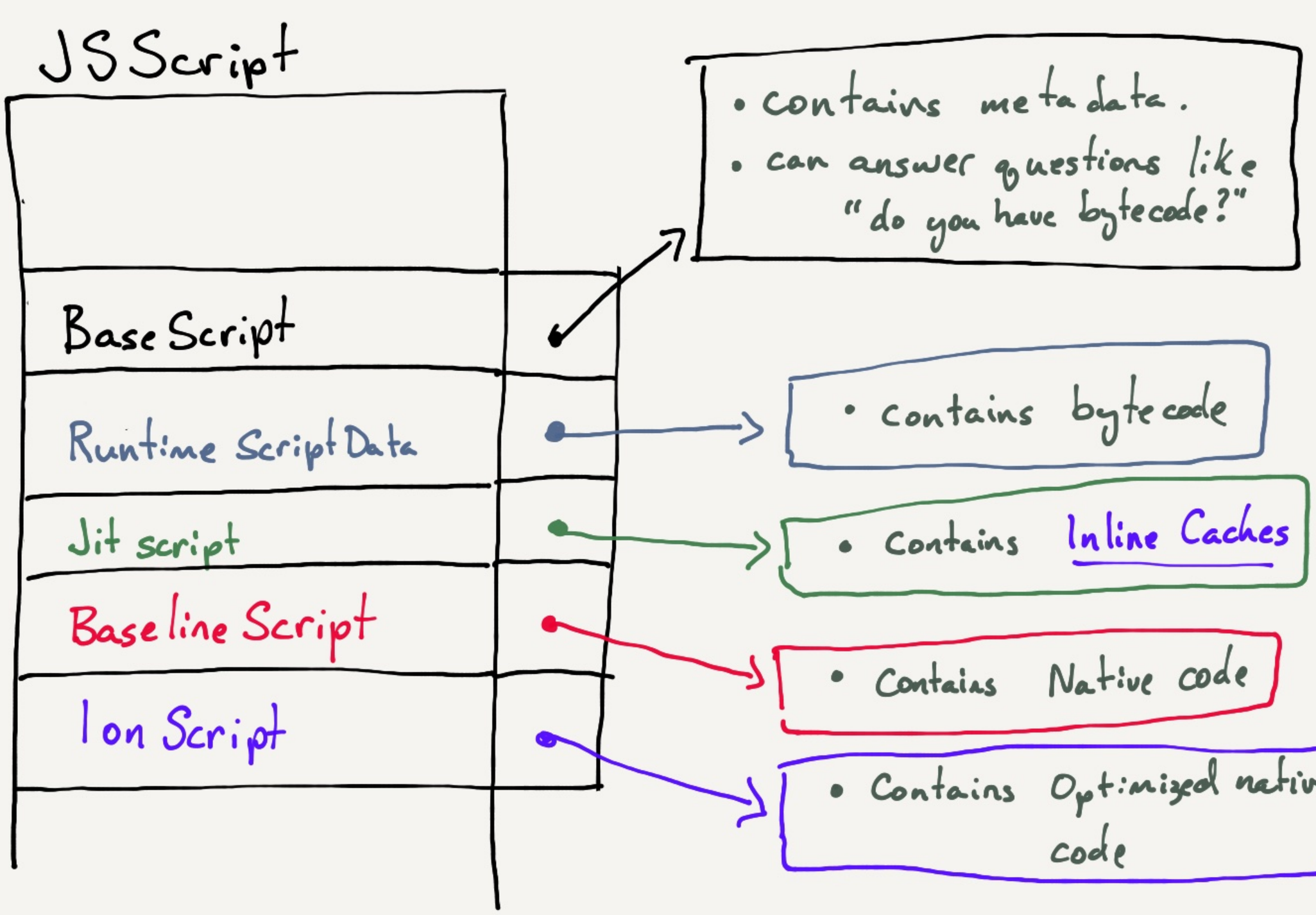


Inline caches are generic stubs of code that can be used to generate native code

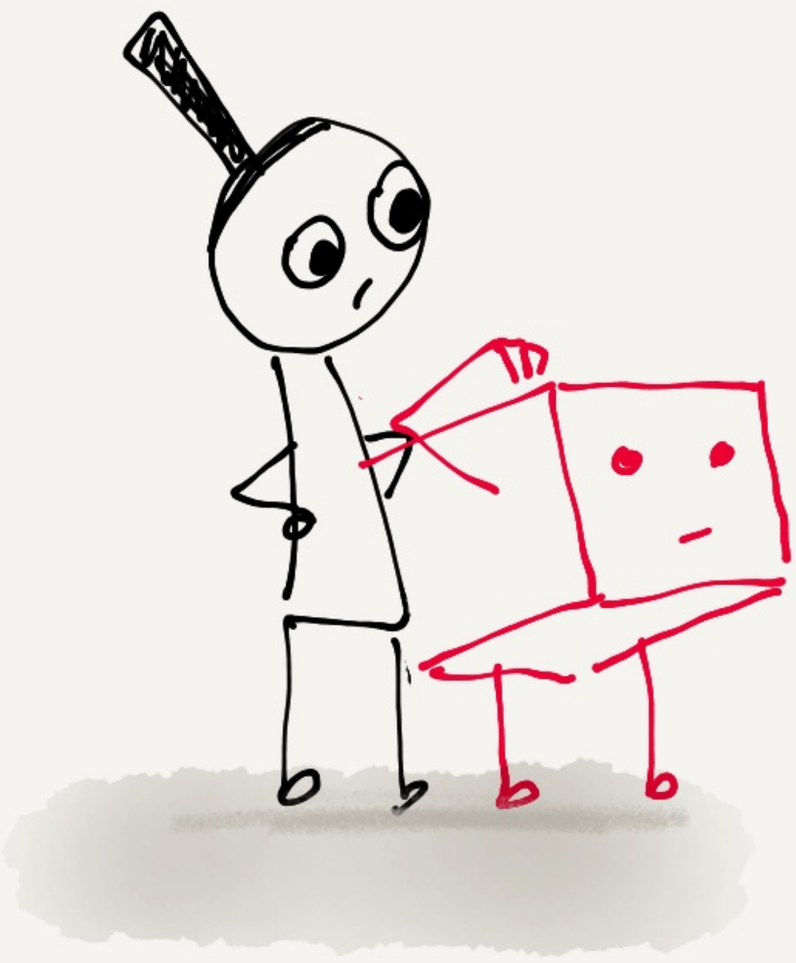
What is a "Script" anyway?



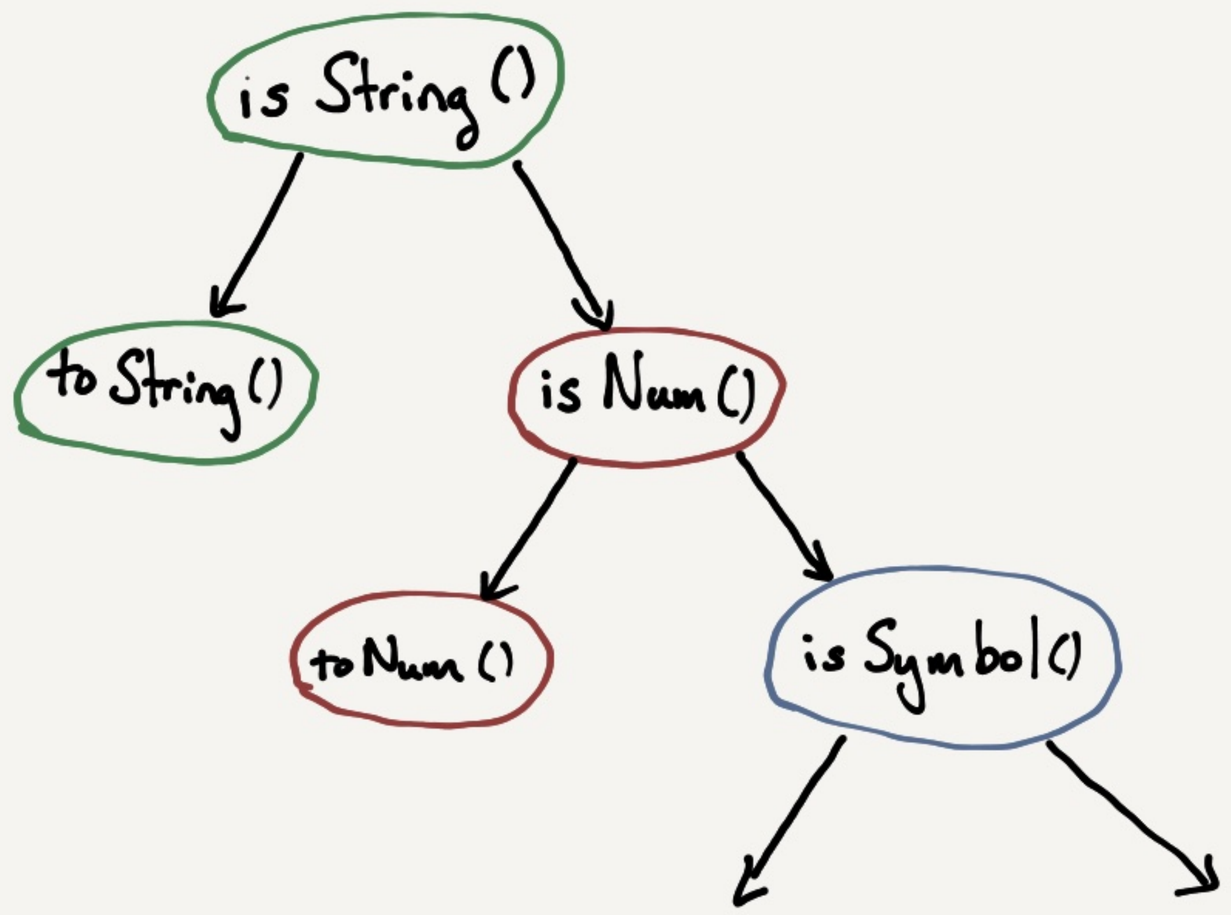
How are Scripts Represented?



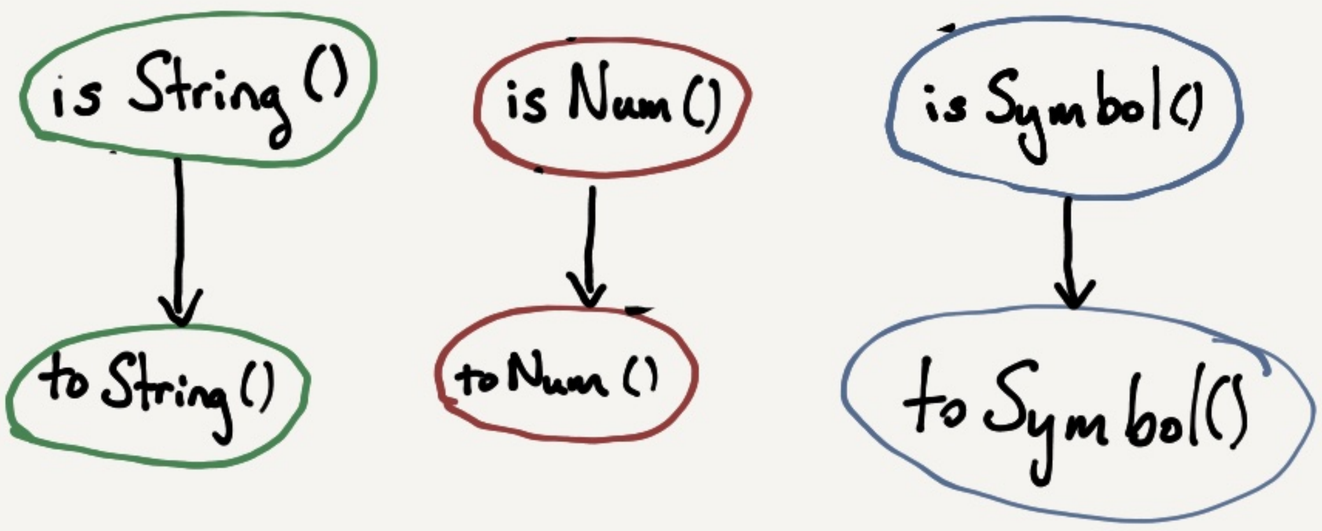
How do we make This fast?



f [<something>]

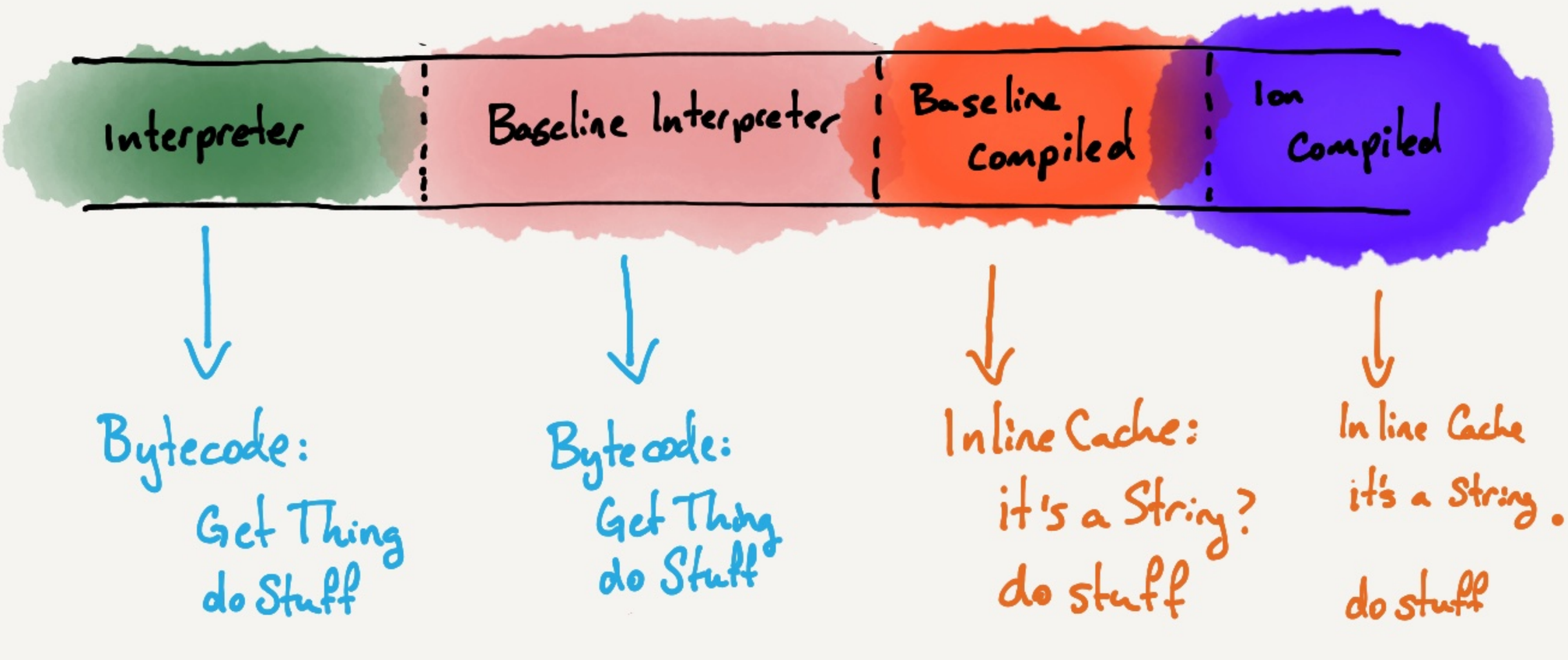


It is easier to go fast if we know what we are working with!

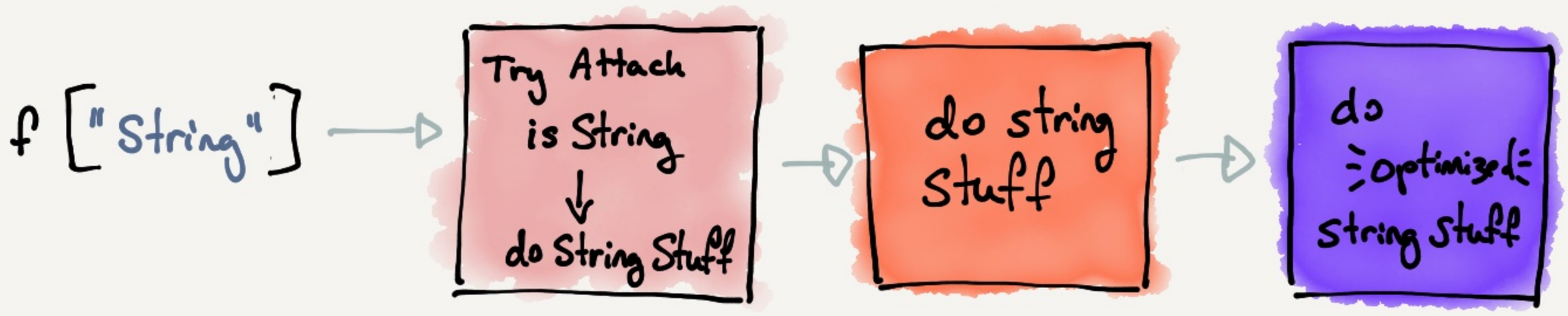


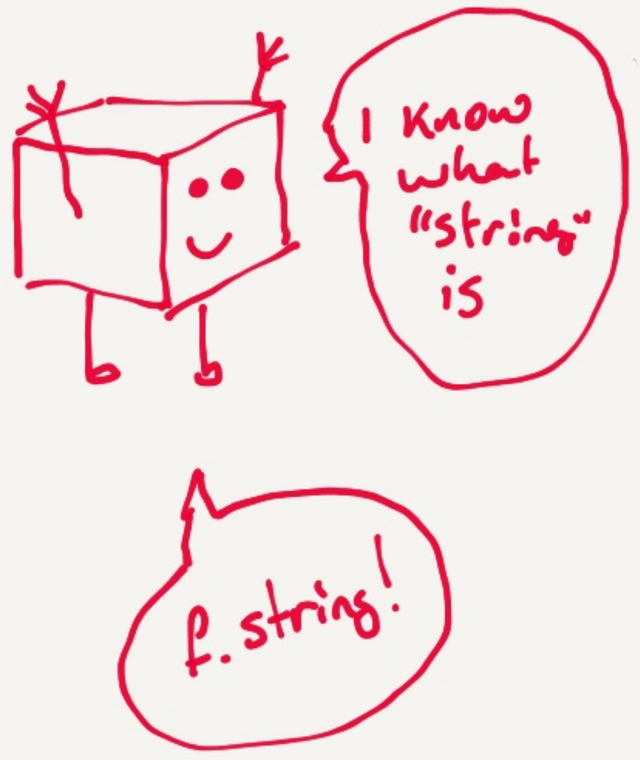
Inline caches: a way to prune branches

```
for (var i = 0; i < 1_000_000; i++) { f["string"] }
```



Inline Caches tell us, locally what to do. The instructions are "in-line" and we don't have to go to get them from somewhere.





But ...

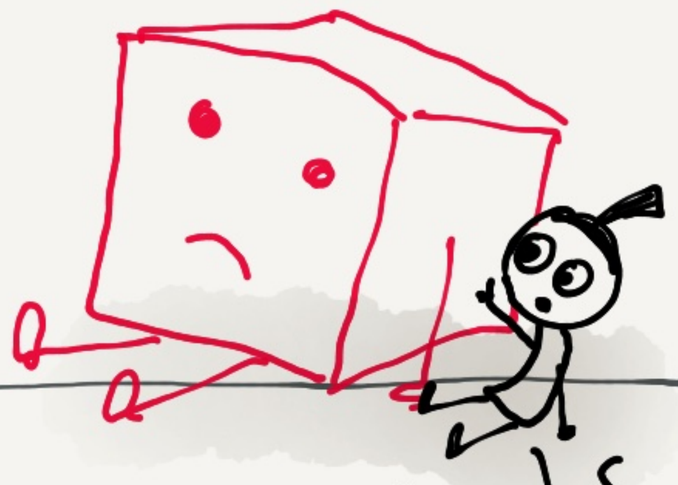
This depends on an assumption.

That we always work with strings.

... JavaScript is quite dynamic...

Bailouts: invalidating our assumptions

- In order to correct our assumptions, we have guards which check the type.



it will be ok. We all have our assumptions challenged sometimes

```
for (var i = 0; i < 1_000_000; i++) {
    if (i < 100_001) {
        f["string"] i
    }
    f[2]; ← now f takes a number
}
```

guard: is String?

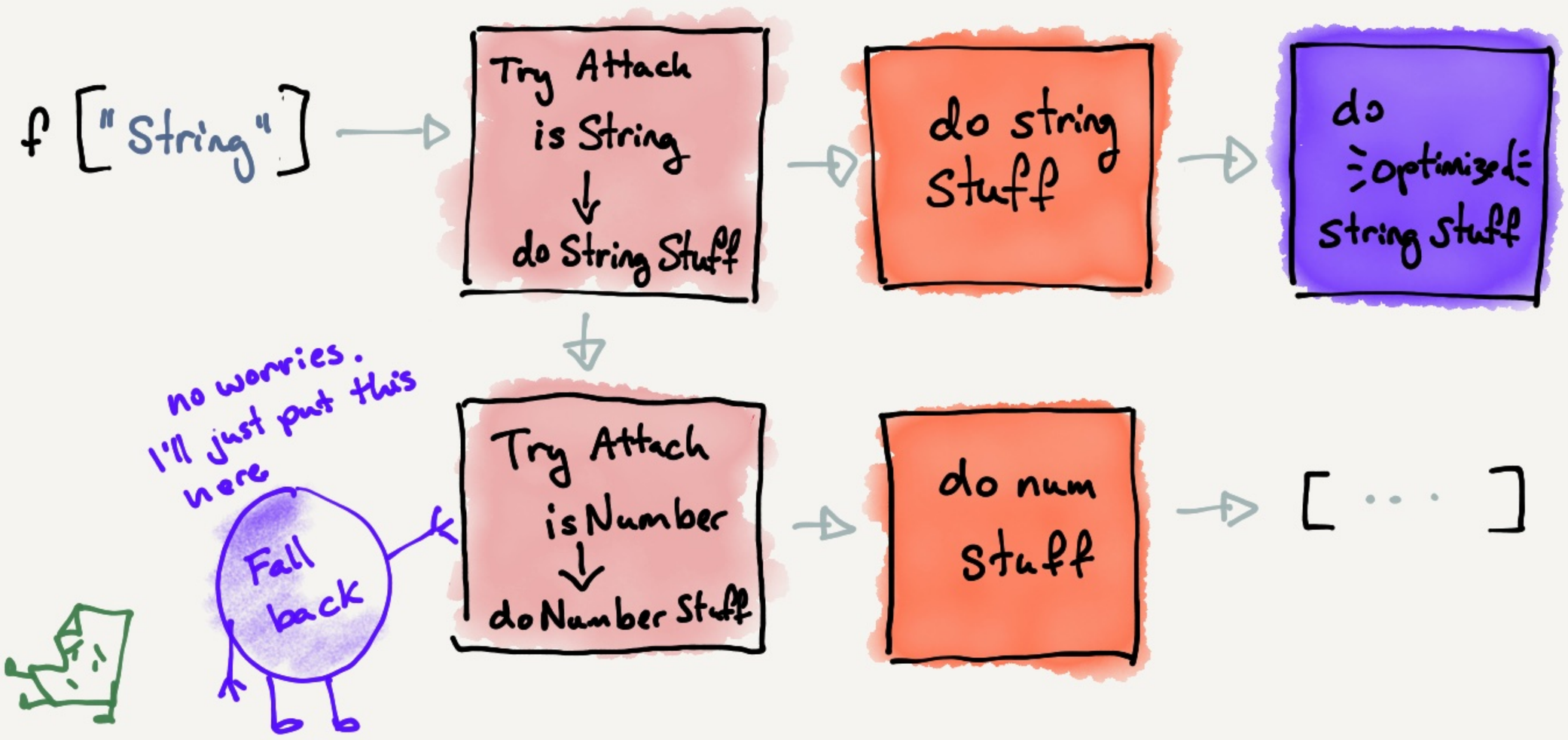
yes

guard: is String?

uh... no.? am number

Go straight to Home (Fallback)

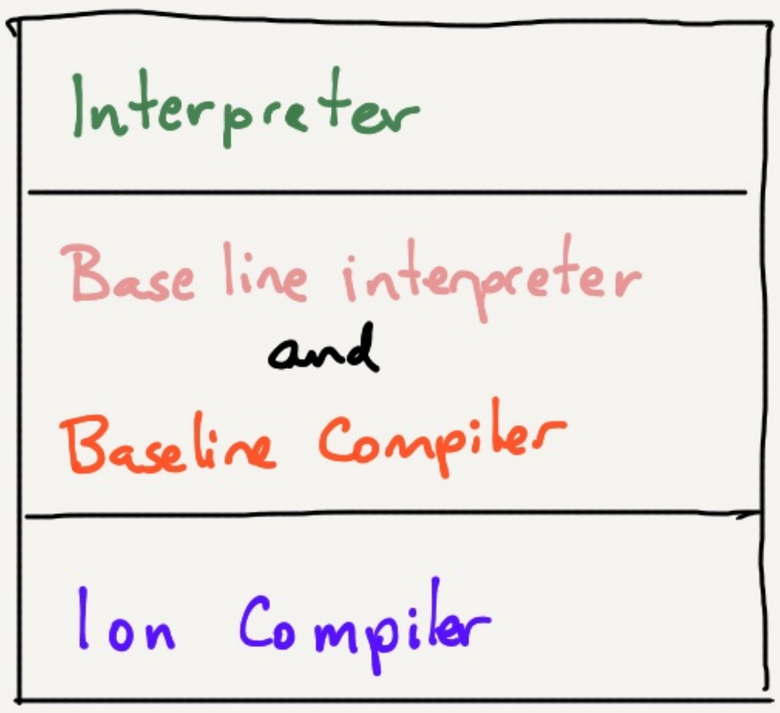
→ The Fallback path will attach a new stub.



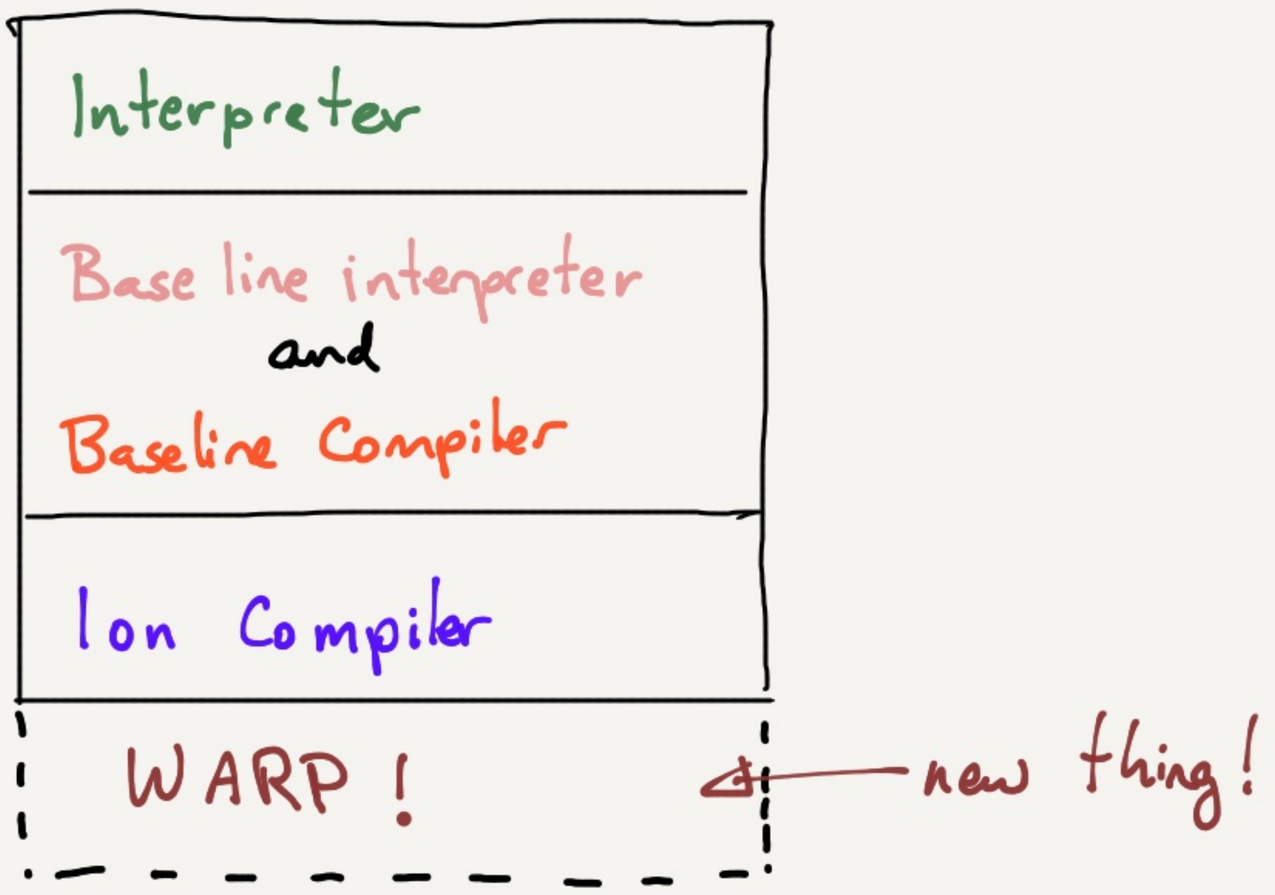
Finishing the bug ...

What we did:

- Introduced a new bytecode PrepareSetElem
- Emitted it from ElemOpEmitter.cpp
- added handling for the new opcode in



- added new CacheIR for PrepareSetElem
- Implemented CacheIR for



- Send to :review:!

#CompilerCompiler

- Every other Friday
  @ 5:00 pm CEST
- https://www.twitch.tv/codehag
- Twitter: @ioctaptceb
- Github playground:
  https://www.github.com/codehag/compiler-compiler-dev

Feel free to join and ask questions!