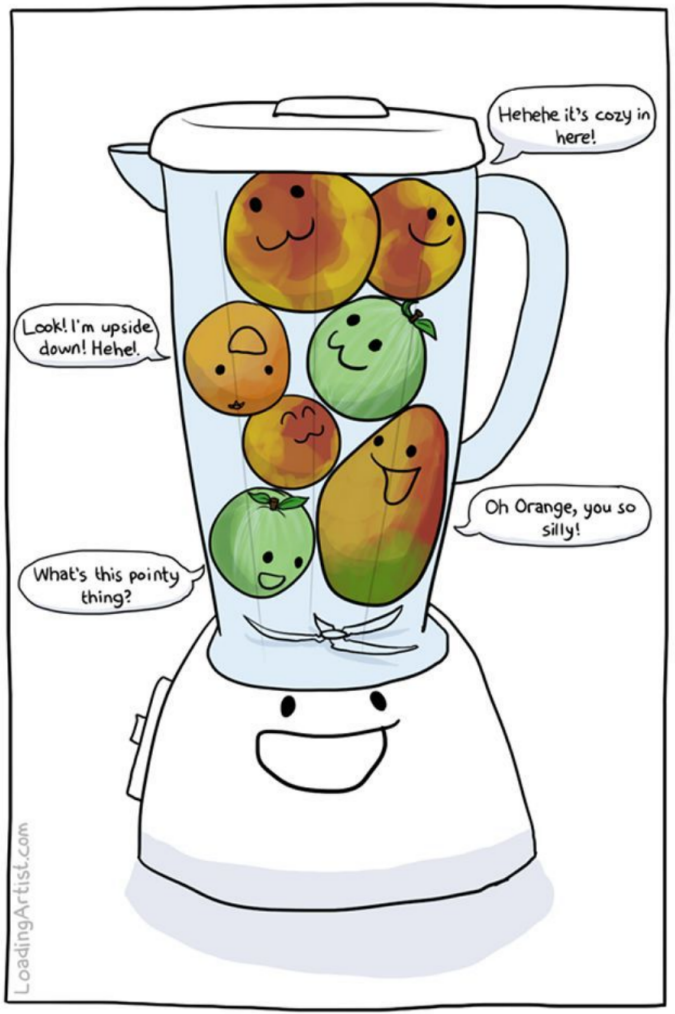
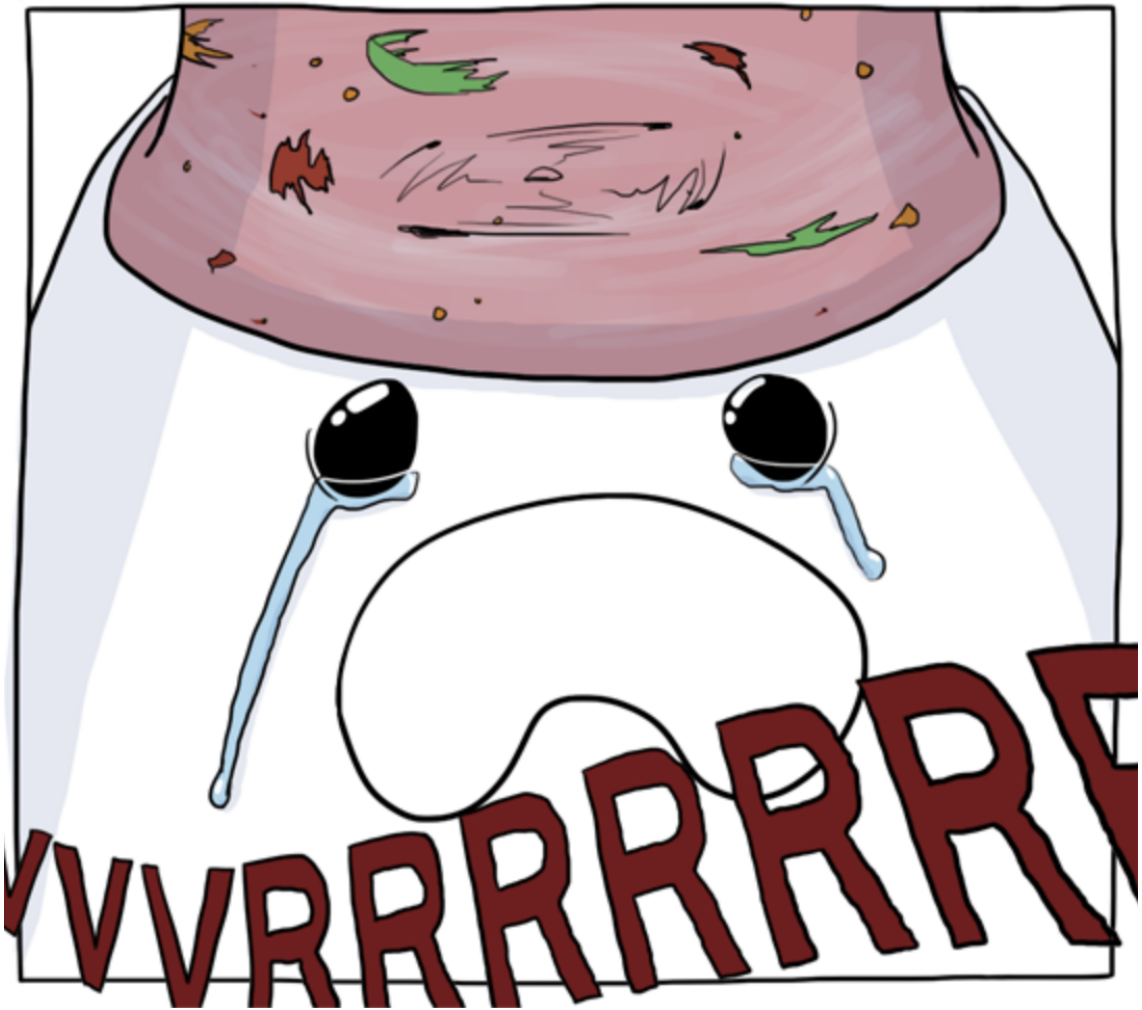
# From Specification to Implementation

Holyjs Moscow 2019

Yulia Startsev I @ioctaptceb

```
class ChildNetworkResponseLoader {
  constructor(context, requestId) {
    this.context = context;
    this.requestId = requestId;
  }

  api() {
    const {context, requestId} = this;
    return {
      getContent(callback) {
        return context.childManager.callParentAsyncFunction(
          "devtools.network.Request.getContent",
          [requestId],
          callback);
      },
    };
  }
}
```


Hehehe it's cozy in here!
Look! I'm upside down! Hehe!
Oh Orange, you so silly!
What's this pointy thing?
LoadingArtist.com


VVVRRRRRRR

Yulia Startsev
@ioctaptceb
Mozilla SpiderMonkey Team

Image of the programmer at work

# Overview

- Very brief Introduction to TC39
- Stage 0 - 3: Nullish Coalescing
  - Reading the spec
  - Issues that shaped the proposal
- Implementation details
  - Implementing the parsing
  - Implementing the byte code
- FREEDOM!

# TC39 and its structure

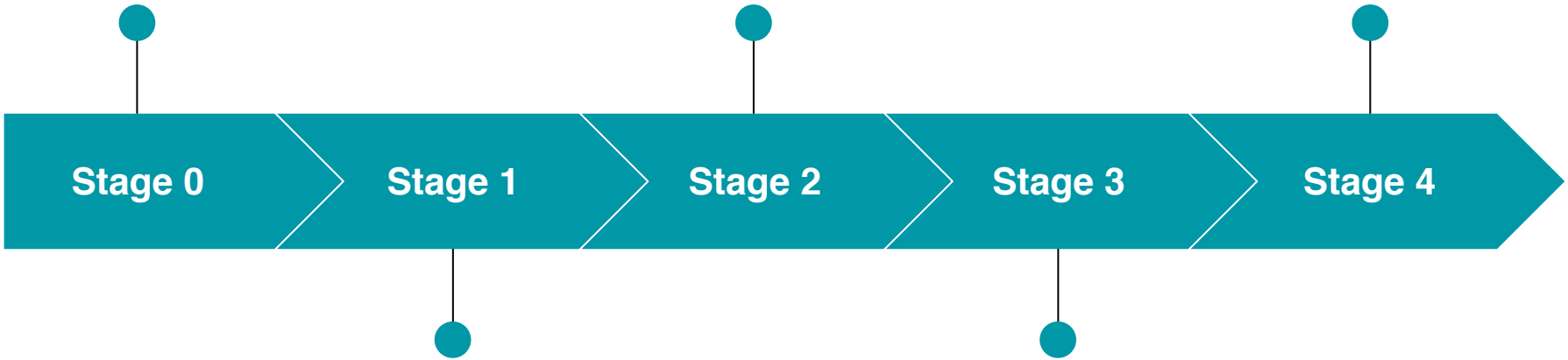
- Part of Ecma International
- Technical Committee 39 of Ecma International
- Takes care of several standards aside from JavaScript, including ECMA-402, ECMA-404, ECMA-414

- Operates via “consensus”

Someone has an idea and they write it up

Committee discusses if this feature “should be in the language”

Proposal is included in the specification




The idea is presented to the committee, committee makes comments

Polyfill and browser implementations, final form of the proposal takes shape


Nullish Coalescing
??

# Stage 0

- Allow input into the specification

# Stage 1

- Make the case for the addition
- Describe the shape of a solution
- Identify potential challenges

**Requirements**

- Identified “champion” who will advance the addition
- Prose outlining the problem or need and the general shape of a solution
- Illustrative examples of usage
- High-level API
- Discussion of key algorithms, abstractions and semantics
- Identification of potential “cross-cutting” concerns and implementation challenges/complexity

README.md

# Nullary Coalescing for JavaScript

## Status

Current Stage:

- Stage 0

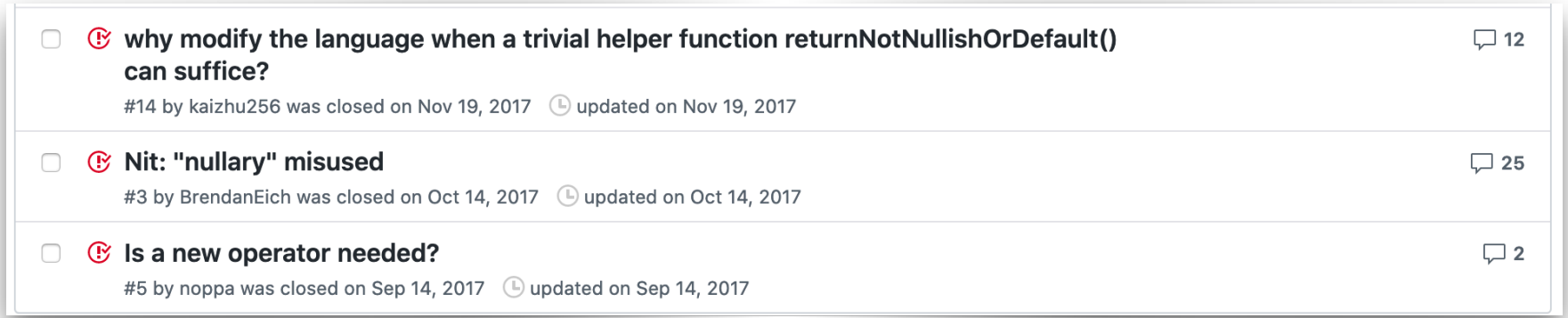
## Authors

- Gabriel Isenberg (github, twitter)

## Overview and motivation

When performing optional property access in a nested structure in conjunction with the optional chaining operator, it is often desired to provide a default value if the result of that property access is `null` or `undefined`. At present, a typical way to express this intent in JavaScript is by using the `||` operator.

```
const response = {
  settings: {
    nullValue: null,
    height: 400,
    animationDuration: 0,
    headerText: '',
    showSplashScreen: false
  }
};
```


why modify the language when a trivial helper function returnNotNullishOrDefault() can suffice?
12
#14 by kaizhu256 was closed on Nov 19, 2017
updated on Nov 19, 2017
Nit: "nullary" misused
25
#3 by BrendanEich was closed on Oct 14, 2017
updated on Oct 14, 2017
Is a new operator needed?
2
#5 by noppa was closed on Sep 14, 2017
updated on Sep 14, 2017

# Stage 2

Precisely describe the syntax and semantics using formal spec language

*The committee expects the feature to be developed and eventually included in the standard*

**Requirements**

- Initial spec text
- all *major* semantics, syntax and API are covered, but TODOs, placeholders and editorial issues are expected

Stage 2 Draft

# Nullish Coalescing Operator

## Introduction

This document specifies the nullish coalescing operator `??`. See the explainer for an introduction.

The main design decisions made in this specification are:

1. The right argument of `??` is evaluated only if needed ("short circuiting").
2. `??` has the same precedence than `||`.
3. The right argument is selected if the left argument is `null` or `undefined`.

## 1 Binary Logical Operators

### Syntax

*LogicalORExpression* `[In, Yield, Await]` :
*LogicalANDExpression* `[?In, ?Yield, ?Await]`

**Syntax**

Semantics

**Structure**

Meaning

# Syntax

*LogicalORExpression* `[In, Yield, Await]` :

*LogicalANDExpression* `[?In, ?Yield, ?Await]`

*LogicalORExpression* `[?In, ?Yield, ?Await]` || *LogicalANDExpression* `[?In, ?Yield, ?Await]`

*LogicalORExpression* `[?In, ?Yield, ?Await]` ?? *LogicalANDExpression* `[?In, ?Yield, ?Await]`

*StatementList* `[Return]` :

*ReturnStatement*

*ExpressionStatement*

*StatementList* :

*ReturnStatement*

*ExpressionStatement*

*StatementList_Return* :

*ReturnStatement*

*ExpressionStatement*

# Issues that shaped the specification

?? has poor associativity/precedence, leading to counterintuitive behavior #26

Edit | New issue

# Please simplify the grammar #44

Edit | New issue

**Closed** waldemarhorwat opened this issue on Jul 24 · 13 comments · Fixed by #45

# Rename LogicalExpression to ShortCircuitExpression #50

Edit

**Merged** DanielRosenwas... merged 1 commit into `tc39:master` from `rkirsling:patch-1` on Aug 22

Conversation 3 | Commits 1 | Checks 0 | Files changed 1 | +8 −8

**rkirsling** commented on Aug 8 — Member

A nitpick to be sure, but we do want to set a good example for the community at large. 😄

The word "logical" means "truth(iness)-oriented", so (unless we mean ternary logic now 😛) I think it's important not to bucket our nullish-oriented `??` with `||`, `&&`, and `!` in this way. The thing that `??` truly has in common with `||` and `&&` is short-circuiting, so it's probably best to say just that.

👍 2

Rename LogicalExpression to ShortCircuitExpression — Verified ✔ 9ac9a2c

**ljharb** approved these changes on Aug 8 — View changes

Reviewers: ljharb ✓

Assignees: No one—assign yourself

Labels: None yet

Projects: None yet

Milestone: No milestone

# ?? has poor associativity/precedence, leading to counterintuitive behavior #26

Edit | New issue

Open **waldemarhorwat** opened this issue on Mar 15, 2018 · 21 comments

**waldemarhorwat** commented on Mar 15, 2018 — Member

When faced with a long short-circuiting expression such as

```
a || b || c || d || e
```

I usually think of it as selecting the first one of `a`, `b`, `c`, `d`, `e` that's not falsy and not evaluating the rest. In effect, I'm viewing the expression as though it were right-associative:

```
a || (b || (c || (d || e)))
```

That works fine. The spec grammar happens to actually specify it as left-associative:

```
(((a || b) || c) || d) || e
```

However, that associativity is merely a spec-writing artifact, invisible to users. The spec could have associated the other way without any visible changes to existing user code behavior.

Unfortunately, introducing `??` at the same precedence level as `||` makes the spec associativity visible in the language and leads to problematic consequences:

**Assignees**
No one—assign yourself

**Labels**
None yet

**Projects**
None yet

**Milestone**
No milestone

**Notifications** Customize

Subscribe

You're not receiving notifications from this thread.

**10 participants**

```
24  24      <emu-grammar>
25      -       LogicalORExpression[In, Yield, Await] :
26      -         LogicalANDExpression[?In, ?Yield, ?Await]
27      -         LogicalORExpression[?In, ?Yield, ?Await] `||` LogicalANDExpression[?In, ?Yield, ?Await]
28      -         <ins>LogicalORExpression[?In, ?Yield, ?Await] `??` LogicalANDExpression[?In, ?Yield, ?Await]</ins>
    25  +       ConditionalExpression[In, Yield, Await] :
    26  +         <del>LogicalORExpression[?In, ?Yield, ?Await]</del>
    27  +         <del>LogicalORExpression[?In, ?Yield, ?Await] `?` AssignmentExpression[+In, ?Yield, ?Await] `:` AssignmentExpr
    28  +         <ins>NullishExpression[?In, ?Yield, ?Await, ~Nullish]</ins>
    29  +         <ins>NullishExpression[?In, ?Yield, ?Await, ~Nullish] `?` AssignmentExpression[+In, ?Yield, ?Await] `:` Assign
    30  +
    31  +       <ins>
    32  +       NullishExpression[In, Yield, Await, Nullish] :
    33  +         LogicalORExpression[?In, ?Yield, ?Await, ?Nullish]
    34  +         LogicalORExpression[?In, ?Yield, ?Await, +Nullish] `??` NullishExpression[?In, ?Yield, ?Await, +Nullish]
    35  +       </ins>
    36  +
    37  +       LogicalORExpression[In, Yield, Await, <ins>Nullish</ins>] :
    38  +         LogicalANDExpression[?In, ?Yield, ?Await, <ins>?Nullish</ins>]
    39  +         <del>LogicalORExpression[?In, ?Yield, ?Await] `||` LogicalANDExpression[?In, ?Yield, ?Await]</del>
    40  +         <ins>[~Nullish] LogicalORExpression[?In, ?Yield, ?Await, ~Nullish] `||` LogicalANDExpression[?In, ?Yield, ?Awa
    41  +
    42  +       LogicalANDExpression[In, Yield, Await, <ins>Nullish</ins>] :
    43  +         BitwiseORExpression[?In, ?Yield, ?Await]
    44  +         <del>LogicalANDExpression[?In, ?Yield, ?Await] `&&` BitwiseORExpression[?In, ?Yield, ?Await]</del>
    45  +         <ins>[~Nullish] LogicalANDExpression[?In, ?Yield, ?Await] `&&` BitwiseORExpression[?In, ?Yield, ?Await]</ins>
29  46      </emu-grammar>
30  47    </emu-clause>
31  48
```

*StatementList* `[Return]` :

[+Return] *ReturnStatement*

*ExpressionStatement*

*StatementList* :

*ExpressionStatement*

*StatementList_Return* :

*ReturnStatement*

*ExpressionStatement*

*StatementList* `[Return]` :

[~Return] *ReturnStatement*

*ExpressionStatement*

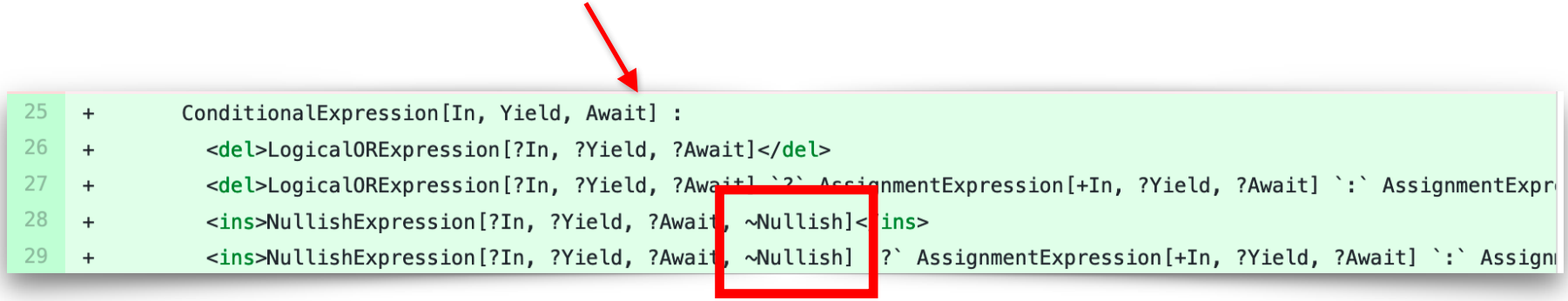
*StatementList* :

*ReturnStatement*

*ExpressionStatement*

*StatementList_Return* :

*ExpressionStatement*


25 + ConditionalExpression[In, Yield, Await] :
26 + <del>LogicalORExpression[?In, ?Yield, ?Await]</del>
27 + <del>LogicalORExpression[?In, ?Yield, ?Await] `?` AssignmentExpression[+In, ?Yield, ?Await] `:` AssignmentExpr
28 + <ins>NullishExpression[?In, ?Yield, ?Await, ~Nullish]</ins>
29 + <ins>NullishExpression[?In, ?Yield, ?Await, ~Nullish] ?` AssignmentExpression[+In, ?Yield, ?Await] `:` Assign

# Please simplify the grammar #44

Edit | New issue

Closed **waldemarhorwat** opened this issue on Jul 24 · 13 comments · Fixed by #45

**waldemarhorwat** commented on Jul 24 — Member

The Coalesced grammar parameter is unnecessary. There are a variety of ways to get rid of it. Here is one possible simpler way to encode the grammar:

Modify *ConditionalExpression* to become:
*ConditionalExpression* :
  *LogicalExpression*
  *LogicalExpression* ? *ConditionalExpression* : *ConditionalExpression*

Write three new productions:
*LogicalExpression* :
  *LogicalORExpression*
  *CoalesceExpression*

*CoalesceExpression* :
  *BitwiseORExpression* ?? *CoalesceExpressionRest*

*CoalesceExpressionRest* :
  *BitwiseORExpression*
  *CoalesceExpression*

👍 2

**Assignees**
No one—assign yourself

**Labels**
None yet

**Projects**
None yet

**Milestone**
No milestone

**Notifications** Customize

Subscribe

You're not receiving notifications from this thread.

**4 participants**

# Rename LogicalExpression to ShortCircuitExpression #50

Edit

Merged **DanielRosenwas...** merged 1 commit into `tc39:master` from `rkirsling:patch-1` on Aug 22

Conversation 3 | Commits 1 | Checks 0 | Files changed 1 | +8 −8

**rkirsling** commented on Aug 8 — Member

A nitpick to be sure, but we do want to set a good example for the community at large. 😄

The word "logical" means "truth(iness)-oriented", so (unless we mean ternary logic now 😛) I think it's important not to bucket our nullish-oriented `??` with `||`, `&&`, and `!` in this way. The thing that `??` truly has in common with `||` and `&&` is short-circuiting, so it's probably best to say just that.

👍 2

Rename LogicalExpression to ShortCircuitExpression — Verified ✓ 9ac9a2c

**ljharb** approved these changes on Aug 8 — View changes

**ljharb** left a comment — Member

I typically refer to pipes and amps as “value selection operators”, but perhaps this is more accurate.

**Reviewers**

ljharb ✓

**Assignees**

No one—assign yourself

**Labels**

None yet

**Projects**

None yet

**Milestone**

No milestone

**Notifications** Customize

Subscribe

You’re not receiving notifications from

Stage 2 Draft / August 22, 2019

# Nullish Coalescing Operator

## Introduction

This document specifies the nullish coalescing operator `??`. See the explainer for an introduction.

The main design decisions made in this specification are:

1. The right argument of `??` is evaluated only if needed ("short circuiting").
2. `??` has lower precedence than `||`.
3. `??` cannot immediately contain, or be contained within, an `&&` or `||` operation.
4. The right argument is selected if the left argument is `null` or `undefined`.

## 1 Binary Logical Operators

### Syntax

*ShortCircuitExpression*`[In, Yield, Await]` :
    *LogicalORExpression*`[?In, ?Yield, ?Await]`
    *CoalesceExpression*`[?In, ?Yield, ?Await]`

# Stage 3

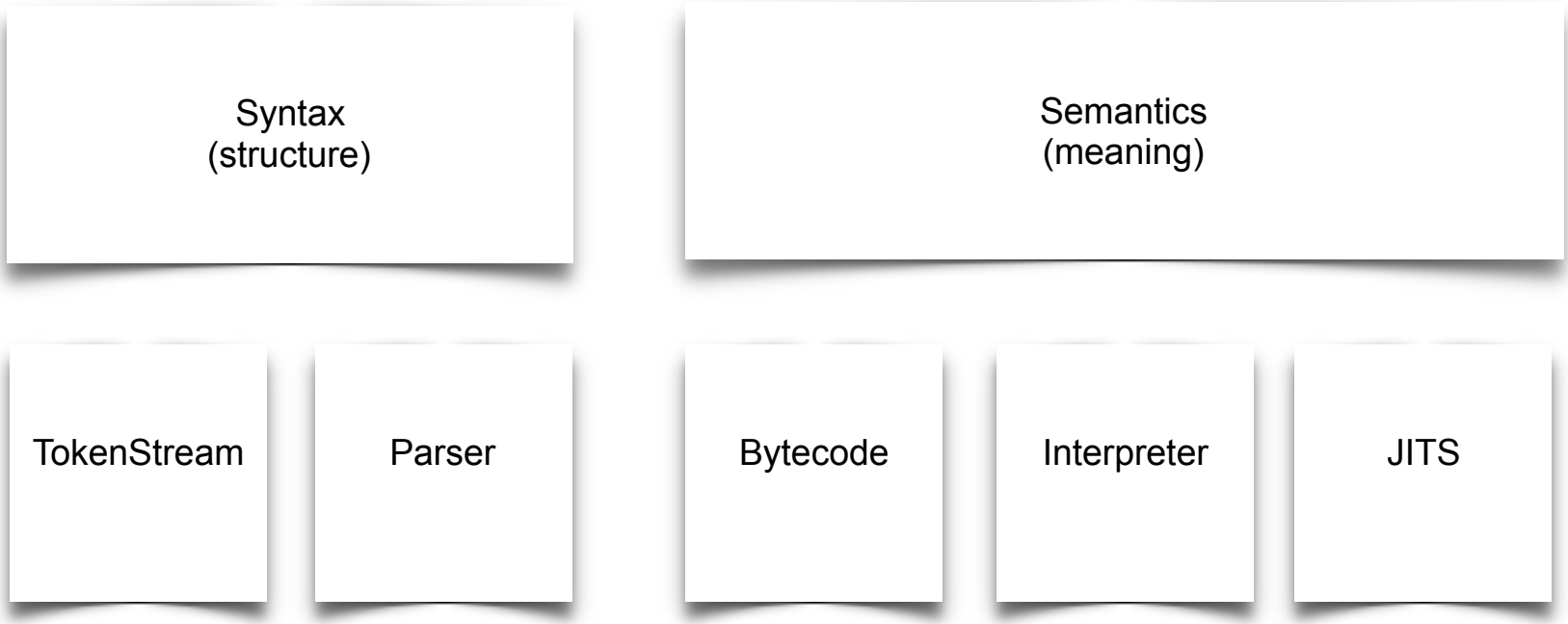
Indicate that further refinement will require feedback from implementations and users

*The solution is complete and no further work is possible without implementation experience, significant usage and external feedback.*

**Requirements**

- Complete spec text
- Designated reviewers have signed off on the current spec text
- All ECMAScript editors have signed off on the current spec text
- All semantics, syntax and API are completed described

# How do we start?


Syntax
(structure)
Semantics
(meaning)
TokenStream
Parser
Bytecode
Interpreter
JITS

Syntax
(structure)

Semantics
(meaning)

TokenStream

Parser

Bytecode

Interpreter

JITS

# Implementing Syntax

# Syntax

*ShortCircuitExpression*`[In, Yield, Await]` :

*LogicalORExpression*`[?In, ?Yield, ?Await]`

*CoalesceExpression*`[?In, ?Yield, ?Await]`

*CoalesceExpression*`[In, Yield, Await]` :

*CoalesceExpressionHead*`[?In, ?Yield, ?Await]` **??**

*BitwiseORExpression*`[?In, ?Yield, ?Await]`

*CoalesceExpressionHead*`[In, Yield, Await]` :

*CoalesceExpression*`[?In, ?Yield, ?Await]`

*BitwiseORExpression*`[?In, ?Yield, ?Await]`

# Given a Minimal program ....

```
function coalesce(a, b) {
  return a ?? b;
}
```

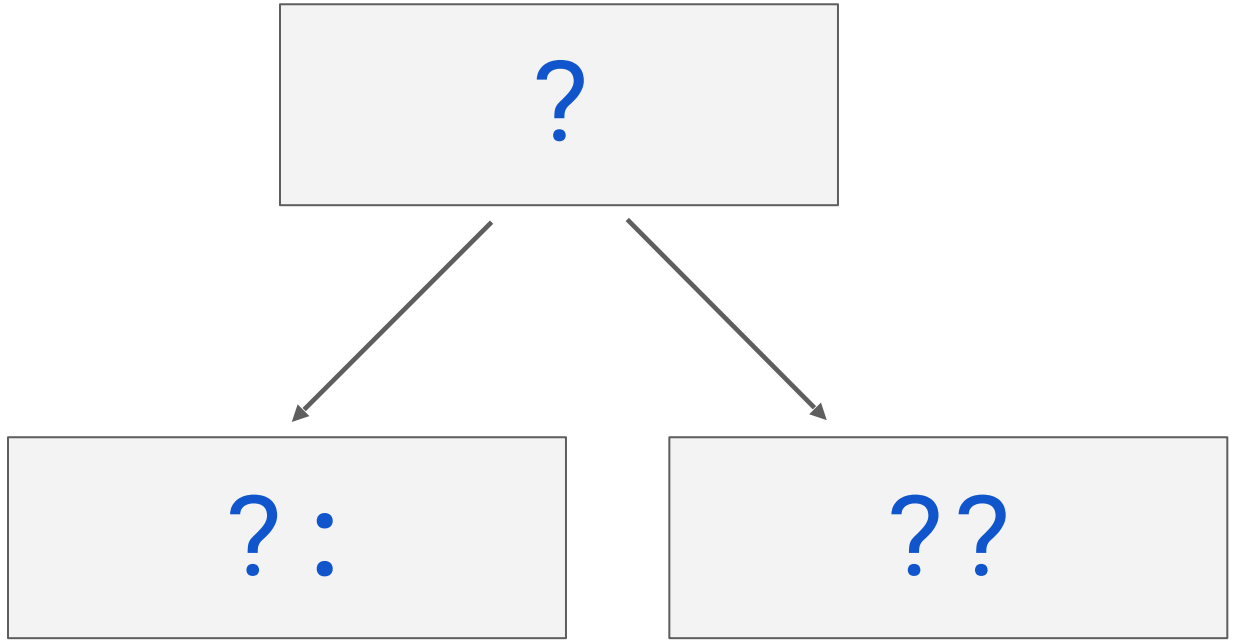
Text

```
function coalesce(a, b) {
  return a ?? b;
}
```

Tokenize

Tokens

```
[Function]
[Iden="coalesce"][LeftParen][Iden="a"][Comma][Iden="b"]
[RightParen][LeftCurly]
[Return][Iden="a"][Coalesce][Iden="b"][Semi]
[RightCurly]
```




Ternary

Nullish Coalesce

# Update TokenStream.cpp

```
2325
2326    LastCharKind = Other
2327  };
2328
2329  // OneChar: 40,  41,  44,  58,  59,  63,  91,  93,  123, 125, 126:
2330  //          '(', ')', ',', ':', ';', '?', '[', ']', '{', '}', '~'
2331  // Ident:   36, 65..90, 95, 97..122: '$', 'A'..'Z', '_', 'a'..'z'
2332  // Dot:     46: '.'
2333  // Equals:  61: '='
2334  // String:  34, 39, 96: '"', '\'', '`'
```

# Update TokenStream.cpp

```
2344 #define T_LP size_t(TokenKind::LeftParen)
2345 #define T_RP size_t(TokenKind::RightParen)
2346 #define T_SEMI size_t(TokenKind::Semi)
2347 #define T_HOOK size_t(TokenKind::Hook)
2348 #define T_LB size_t(TokenKind::LeftBracket)
```

```
2361 /*  50+ */     Dec,     Dec,     Dec,     Dec,     Dec,     Dec,     Dec,     Dec, T_COLON,
T_SEMI,
2362 /*  60+ */ _______, _______, _______, T_HOOK, _______,   Ident,   Ident,   Ident,   Ident,
Ident,
2363 /*  70+ */   Ident,   Ident,   Ident,   Ident,   Ident,   Ident,   Ident,   Ident,   Ident,
Ident,
2364 /*  80+ */   Ident,   Ident,   Ident,   Ident,   Ident,   Ident,   Ident,   Ident,   Ident,
```

```
2376 #undef T_SEMI
2377 #undef T_HOOK
2378 #undef T_LB
2379 #undef T_RB
```

# Update TokenKind.h List

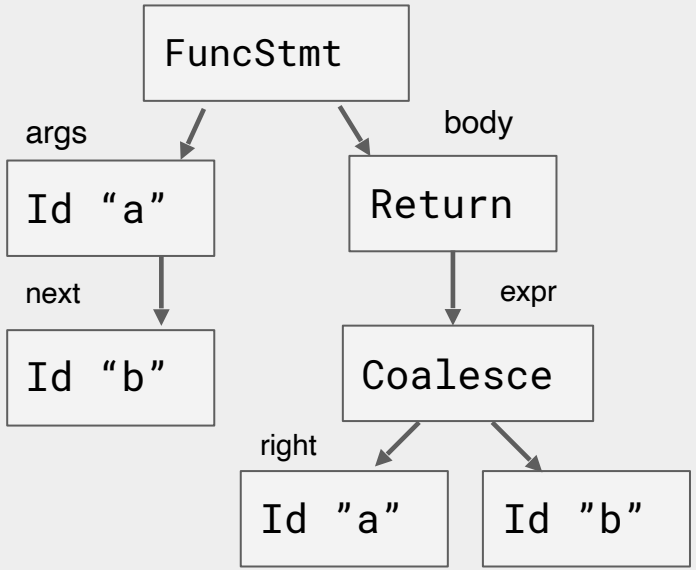
```
159   /*                                                                  \
160    * Binary operators tokens, Or thru Pow. These must be in the same  \
161    * order as F(Or) and friends in FOR_EACH_PARSE_NODE_KIND in ParseNode.h. \
162    */                                                                 \
163   MACRO(Pipeline, "'|>'")                                             \
164   RANGE(BinOpFirst, Pipeline)                                         \
165   MACRO(Coalesce, "'??'")                                             \
166   MACRO(Or, "'||'")    /* logical or */                               \
167   MACRO(And, "'&&'")   /* logical and */                              \
168   MACRO(BitOr, "'|'")  /* bitwise-or */                               \
169   MACRO(BitXor, "'^'") /* bitwise-xor */                              \
170   MACRO(BitAnd, "'&'") /* bitwise-and */                              \
171                                                                       \
```

# Update TokenStream::getTokenInternal

```
2733  template <typename Unit, class AnyCharsAccess>
2734  MOZ_MUST_USE bool TokenStreamSpecific<Unit, AnyCharsAccess>::getTokenInternal(
2735      TokenKind* const ttp, const Modifier modifier) {
```

```
3142                matchCodeUnit('=') ? TokenKind::BitAndAssign : TokenKind::BitAnd;
3143          }
3144          break;
3145
3146        case '?':
3147          simpleKind = matchCodeUnit('?') ? TokenKind::Coalesce : TokenKind::Hook;
3148          break;
3149
3150        case '!':
3151          if (matchCodeUnit('=')) {
3152            simpleKind = matchCodeUnit('=') ? TokenKind::StrictNe : TokenKind::Ne;
3153          } else {
3154            simpleKind = TokenKind::Not;
```

# Building the AST

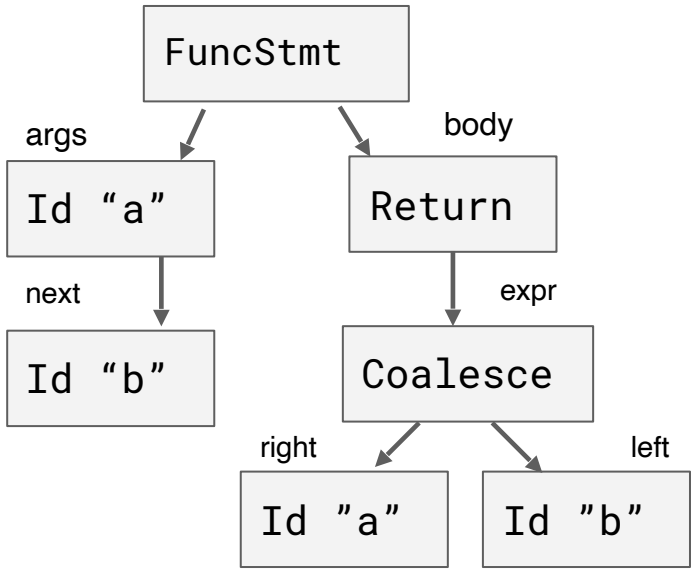


Tokens

```
[Function]
[Iden="coalesce"][LeftParen][Iden="a"][Comma][Iden="b"]
[RightParen][LeftCurly]
[Return][Iden="a"][Coalesce][Iden="b"][Semi]
[RightCurly]
```

Parse

Abstract Syntax Tree (AST)



# ParseNode.h

```
162   /*                                                                  \
163    * Binary operators.                                                \
164    * These must be in the same order as TOK_OR and friends in TokenStream.h. \
165    */                                                                 \
166   F(PipelineExpr, ListNode)                                           \
167   F(CoalesceExpr, ListNode)                                                \
168   F(OrExpr, ListNode)                                                 \
169   F(AndExpr, ListNode)                                                \
170   F(BitOrExpr, ListNode)                                              \
171   F(BitXorExpr, ListNode)                                             \
172   F(BitAndExpr, ListNode)                                             \
173   F(StrictEqExpr, ListNode)                                           \
```

# Parser.cpp

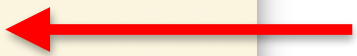
```
8325  // This list must be kept in the same order in several places:
8326  //   - The binary operators in ParseNode.h ,
8327  //   - the binary operators in TokenKind.h
8328  //   - the JSOp code list in BytecodeEmitter.cpp
8329  static const int PrecedenceTable[] = {
8330      1,  /* ParseNodeKind::PipeLine */
8331      2,  /* ParseNodeKind::Coalesce */
8332      3,  /* ParseNodeKind::Or */
8333      4,  /* ParseNodeKind::And */
8334      5,  /* ParseNodeKind::BitOr */
8335      6,  /* ParseNodeKind::BitXor */
8336      7,  /* ParseNodeKind::BitAnd */
8337      8,  /* ParseNodeKind::StrictEq */
8338      8,  /* ParseNodeKind::Eq */
8339      8,  /* ParseNodeKind::StrictNe */
```

# BytecodeEmitter.cpp

```
7513  // This list must be kept in the same order in several places:
7514  //   - The binary operators in ParseNode.h ,
7515  //   - the binary operators in TokenKind.h
7516  //   - the precedence list in Parser.cpp
7517  static const JSOp ParseNodeKindToJSOp[] = {
7518      // JSOP_NOP is for pipeline operator which does not emit its own JSOp
7519      // but has highest precedence in binary operators
7520      JSOP_NOP,      JSOP_NOP,     JSOP_OR,        JSOP_AND, JSOP_BITOR,
7521      JSOP_BITXOR,   JSOP_BITAND, JSOP_STRICTEQ, JSOP_EQ,  JSOP_STRICTNE,
7522      JSOP_NE,       JSOP_LT,     JSOP_LE,        JSOP_GT,  JSOP_GE,
7523      JSOP_INSTANCEOF, JSOP_IN,   JSOP_LSH,       JSOP_RSH, JSOP_URSH,
7524      JSOP_ADD,      JSOP_SUB,    JSOP_MUL,       JSOP_DIV, JSOP_MOD,
7525      JSOP_POW};
7526
```

# If we don’t keep the lists in sync, this happens

```
shouldBe(1 & null ?? 3, 0); // true
shouldBe(3 == null ?? 3, false); // false
shouldBe(3 ≠ null ?? 3, true); // false
shouldBe(3 === null ?? 3, false); // true
shouldBe(3 ≠= null ?? 3, true); // true
shouldBe(1 < null ?? 3, false); // true
shouldBe(1 > null ?? 3, true); // true
shouldBe(1 ⩽ null ?? 3, false); // false
shouldBe(1 ⩾ null ?? 3, true); // TypeError: invalid
shouldBe(1 << null ?? 3, 1); // true
shouldBe(1 >> null ?? 3, 1); // true
shouldBe(1 >>> null ?? 3, 1); // true
```



# Parser.cpp

```
8375  template <class ParseHandler, typename Unit>
8376  MOZ_ALWAYS_INLINE typename ParseHandler::Node
8377  GeneralParser<ParseHandler, Unit>::orExpr(
8378      InHandling inHandling, YieldHandling yieldHandling,
8379      TripledotHandling tripledotHandling, PossibleError* possibleError,
8380      InvokedPrediction invoked /* = PredictUninvoked */) {
```

```
8391    for (;;) {
8392      pn = unaryExpr(yieldHandling, tripledotHandling, possibleError, invoked);
8393      if (!pn) {
8394        return null();
8395      }
8396
8397      // If a binary operator follows, consume it and compute the
8398      // corresponding operator.
8399      TokenKind tok;
8400      if (!tokenStream.getToken(&tok)) {
8401        return null();
8402      }
```

```
8415        pnk = BinaryOpTokenKindToParseNodeKind(tok);
8416      } else {
8417        tok = TokenKind::Eof;
8418        pnk = ParseNodeKind::Limit;
8419      }
```

# Pretending to be `||`

# Tiny bit of byte code

# BytecodeEmitter.cpp

```
9230      case ParseNodeKind::CoalesceExpr:
9231      case ParseNodeKind::OrExpr:
9232      case ParseNodeKind::AndExpr:
9233        if (!emitLogical(&pn->as<ListNode>())) {
9234          return false;
9235        }
9236        break;
```

# BytecodeEmitter.cpp

```
7558  bool BytecodeEmitter::emitLogical(ListNode* node) {
7559    MOZ_ASSERT(node->isKind(ParseNodeKind::OrExpr) ||
7560               node->isKind(ParseNodeKind::CoalesceExpr) ||
7561               node->isKind(ParseNodeKind::AndExpr));
```

```
7575    /* Left-associative operator chain: avoid too much recursion. */
7576    ParseNode* expr = node->head();
7577    if (!emitTree(expr)) {
7578      return false;
7579    }
7580    JSOp op = (node->isKind(ParseNodeKind::OrExpr) || node->isKind(ParseNodeKind::Co
        alesceExpr)) ? JSOP_OR : JSOP_AND;
7581    JumpList jump;
```

# Let’s run it

# ?? has poor associativity/precedence, leading to counterintuitive behavior #26

Edit | New issue

Open **waldemarhorwat** opened this issue on Mar 15, 2018 · 21 comments

**waldemarhorwat** commented on Mar 15, 2018 — Member

When faced with a long short-circuiting expression such as

```
a || b || c || d || e
```

I usually think of it as selecting the first one of `a`, `b`, `c`, `d`, `e` that's not falsy and not evaluating the rest. In effect, I'm viewing the expression as though it were right-associative:

```
a || (b || (c || (d || e)))
```

That works fine. The spec grammar happens to actually specify it as left-associative:

```
(((a || b) || c) || d) || e
```

However, that associativity is merely a spec-writing artifact, invisible to users. The spec could have associated the other way without any visible changes to existing user code behavior.

Unfortunately, introducing `??` at the same precedence level as `||` makes the spec associativity visible in the language and leads to problematic consequences:

**Assignees**
No one—assign yourself

**Labels**
None yet

**Projects**
None yet

**Milestone**
No milestone

**Notifications** Customize

Subscribe

You're not receiving notifications from this thread.

**10 participants**

# Parser.cpp

```
8325  // This list must be kept in the same order in several places:
8326  //   - The binary operators in ParseNode.h ,
8327  //   - the binary operators in TokenKind.h
8328  //   - the JSOp code list in BytecodeEmitter.cpp
8329  static const int PrecedenceTable[] = {
8330      1,  /* ParseNodeKind::PipeLine */
8331      2,  /* ParseNodeKind::Coalesce */
8332      3,  /* ParseNodeKind::Or */
8333      4,  /* ParseNodeKind::And */
8334      5,  /* ParseNodeKind::BitOr */
8335      6,  /* ParseNodeKind::BitXor */
8336      7,  /* ParseNodeKind::BitAnd */
8337      8,  /* ParseNodeKind::StrictEq */
8338      8,  /* ParseNodeKind::Eq */
8339      8,  /* ParseNodeKind::StrictNe */
```

# Parser.cpp

```
8373  enum class EnforcedParentheses : uint8_t { CoalesceExpr, AndOrExpr, None };
8374
8375  template <class ParseHandler, typename Unit>
8376  MOZ_ALWAYS_INLINE typename ParseHandler::Node
8377  GeneralParser<ParseHandler, Unit>::orExpr(
8378      InHandling inHandling, YieldHandling yieldHandling,
8379      TripledotHandling tripledotHandling, PossibleError* possibleError,
8380      InvokedPrediction invoked /* = PredictUninvoked */) {
8381    // Shift-reduce parser for the binary operator part of the JS expression
8382    // syntax.
```

```
8413        switch (tok) {
8422          case TokenKind::Or:
8423          case TokenKind::And:
8427            if (unparenthesizedExpression == EnforcedParentheses::CoalesceExpr) {
8428              error(JSMSG_BAD_COALESCE_MIXING);
8429              return null();
8430            }
8431            // If we have not detected a mixing error at this point, record that
8432            // we have an unparenthesized expression, in case we have one later.
8433            unparenthesizedExpression = EnforcedParentheses::AndOrExpr;
```

```
8436          case TokenKind::Coalesce:
8437            if (unparenthesizedExpression == EnforcedParentheses::AndOrExpr) {
8438              error(JSMSG_BAD_COALESCE_MIXING);
8439              return null();
8440            }
8441            // If we have not detected a mixing error at this point, record that
8442            // we have an unparenthesized expression, in case we have one later.
8443            unparenthesizedExpression = EnforcedParentheses::CoalesceExpr;
8444            break;
8445
```

# Let’s run it

# Syntax

*ShortCircuitExpression*`[In, Yield, Await]` :

*LogicalORExpression*`[?In, ?Yield, ?Await]`

*CoalesceExpression*`[?In, ?Yield, ?Await]`

*CoalesceExpression*`[In, Yield, Await]` :

*CoalesceExpressionHead*`[?In, ?Yield, ?Await]` **??**

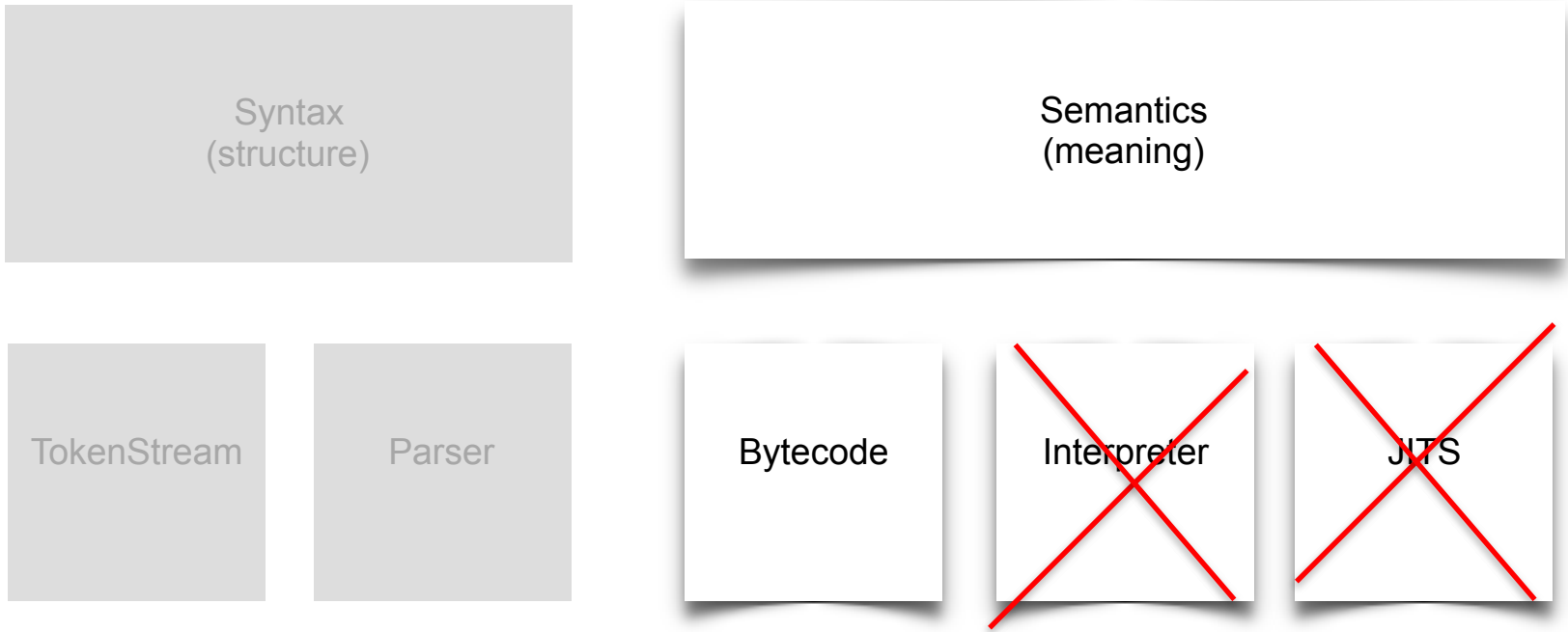
*BitwiseORExpression*`[?In, ?Yield, ?Await]`

*CoalesceExpressionHead*`[In, Yield, Await]` :

*CoalesceExpression*`[?In, ?Yield, ?Await]`

*BitwiseORExpression*`[?In, ?Yield, ?Await]`

# Syntax Covered.


Syntax
(structure)
Semantics
(meaning)
TokenStream
Parser
Bytecode
Interpreter
JITS

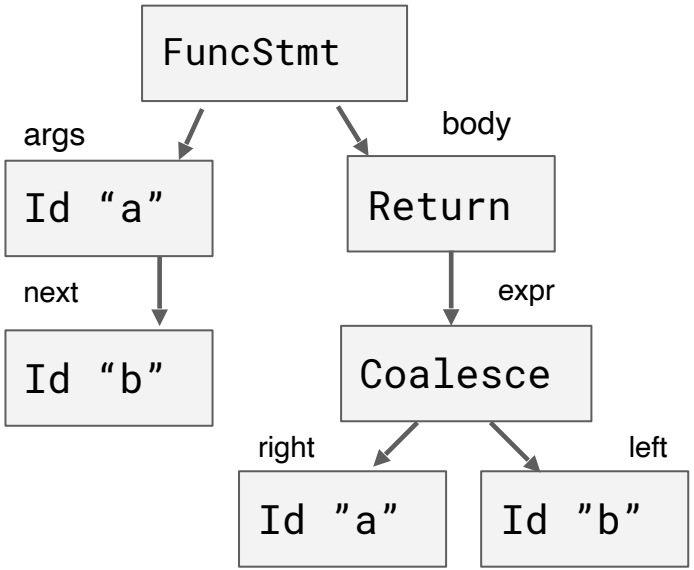
# Implementing Semantics

# 1.3 Runtime Semantics: Evaluation

*CoalesceExpression* : *CoalesceExpressionHead* **??** *BitwiseORExpression*

1. Let *lref* be the result of evaluating *CoalesceExpressionHead*.
2. Let *lval* be ? GetValue(*lref*).
3. If *lval* is **undefined** or **null**,
    a. Let *rref* be the result of evaluating *BitwiseORExpression*.
    b. Return ? GetValue(*rref*).
4. Otherwise, return *lval*.

(AST)




Bytecode-
Compile

Bytecode

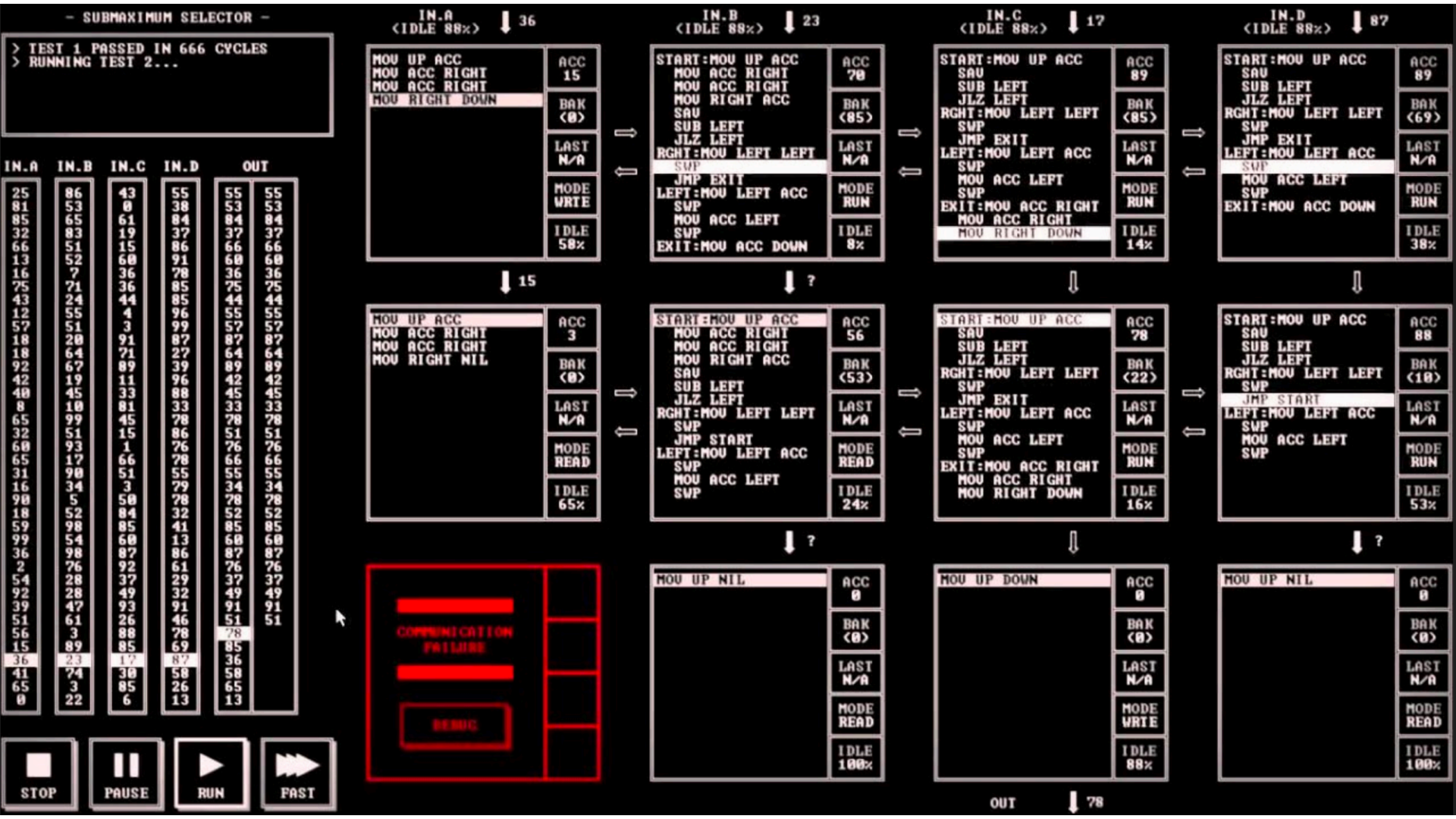
JSScript

```
getarg 0
coalesce
jumptarget 0
pop
getarg 1
jumptarget 0
return
```

```
C01E 8D F0      INHEX   BSR     INCH        GET A CHAR
C020 81 30              CMP A   #'0         ZERO
C022 2B 11              BMI     HEXERR      NOT HEX
C024 81 39              CMP A   #'9         NINE
C026 2F 0A              BLE     HEXRTS      GOOD HEX
C028 81 41              CMP A   #'A
```

# How do you feel about assembly?

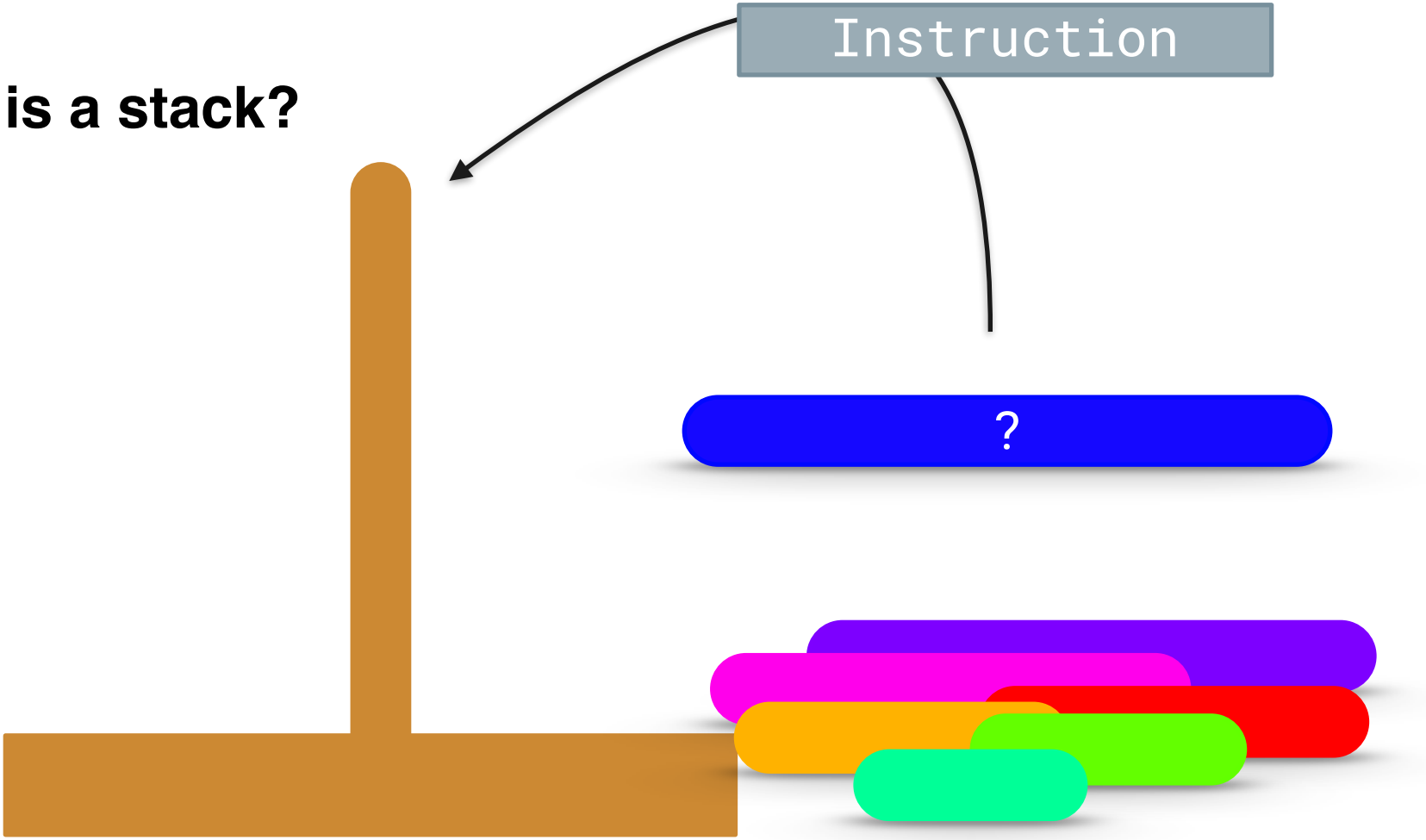
```
C030 80 07              SUB A   #7          FIX A-F
C032 84 0F      HEXRTS  AND A   #$0F        CONVERT ASCII TO DIGIT
C034 39                 RTS

C035 7E C0 AF   HEXERR  JMP     CTRL        RETURN TO CONTROL LOOP
```


- SUBMAXIMUM SELECTOR -
> TEST 1 PASSED IN 666 CYCLES
> RUNNING TEST 2...
IN.A
IN.B
IN.C
IN.D
OUT
STOP
PAUSE
RUN
FAST
IN.A (IDLE 88%) 36
IN.B (IDLE 88%) 23
IN.C (IDLE 88%) 17
IN.D (IDLE 88%) 87
MOV UP ACC
MOV ACC RIGHT
MOV ACC RIGHT
MOV RIGHT DOWN
ACC 15
BAK (0)
LAST N/A
MODE WRTE
IDLE 58%
START:MOV UP ACC
SAV
MOV RIGHT ACC
SUB LEFT
JLZ LEFT
RGHT:MOV LEFT LEFT
SWP
JMP EXIT
LEFT:MOV LEFT ACC
MOV ACC LEFT
EXIT:MOV ACC DOWN
ACC 70
BAK (85)
LAST N/A
MODE RUN
IDLE 8%
START:MOV UP ACC
SAV
SUB LEFT
JLZ LEFT
RGHT:MOV LEFT LEFT
SWP
JMP EXIT
LEFT:MOV LEFT ACC
SWP
MOV ACC LEFT
SWP
EXIT:MOV ACC RIGHT
MOV ACC RIGHT
MOV RIGHT DOWN
ACC 89
BAK (85)
LAST N/A
MODE RUN
IDLE 14%
START:MOV UP ACC
SAV
SUB LEFT
JLZ LEFT
RGHT:MOV LEFT LEFT
SWP
JMP EXIT
LEFT:MOV LEFT ACC
SWP
MOV ACC LEFT
SWP
EXIT:MOV ACC DOWN
ACC 89
BAK (69)
LAST N/A
MODE RUN
IDLE 38%
15
?
MOV UP ACC
MOV ACC RIGHT
MOV ACC RIGHT
MOV RIGHT NIL
ACC 3
BAK (0)
LAST N/A
MODE READ
IDLE 65%
START:MOV UP ACC
MOV ACC RIGHT
MOV ACC RIGHT
MOV RIGHT ACC
SAV
SUB LEFT
JLZ LEFT
RGHT:MOV LEFT LEFT
SWP
JMP START
LEFT:MOV LEFT ACC
SWP
MOV ACC LEFT
SWP
ACC 56
BAK (53)
LAST N/A
MODE READ
IDLE 24%
START:MOV UP ACC
SAV
SUB LEFT
JLZ LEFT
RGHT:MOV LEFT LEFT
SWP
JMP EXIT
LEFT:MOV LEFT ACC
SWP
MOV ACC LEFT
SWP
EXIT:MOV ACC RIGHT
MOV ACC RIGHT
MOV RIGHT DOWN
ACC 78
BAK (22)
LAST N/A
MODE RUN
IDLE 16%
START:MOV UP ACC
SAV
SUB LEFT
JLZ LEFT
RGHT:MOV LEFT LEFT
SWP
JMP START
LEFT:MOV LEFT ACC
SWP
MOV ACC LEFT
SWP
ACC 88
BAK (10)
LAST N/A
MODE RUN
IDLE 53%
COMMUNICATION FAILURE
REBUG
?
?
MOV UP NIL
ACC 0
BAK (0)
LAST N/A
MODE READ
IDLE 100%
MOV UP DOWN
ACC 0
BAK (0)
LAST N/A
MODE WRTE
IDLE 88%
MOV UP NIL
ACC 0
BAK (0)
LAST N/A
MODE READ
IDLE 100%
OUT 78

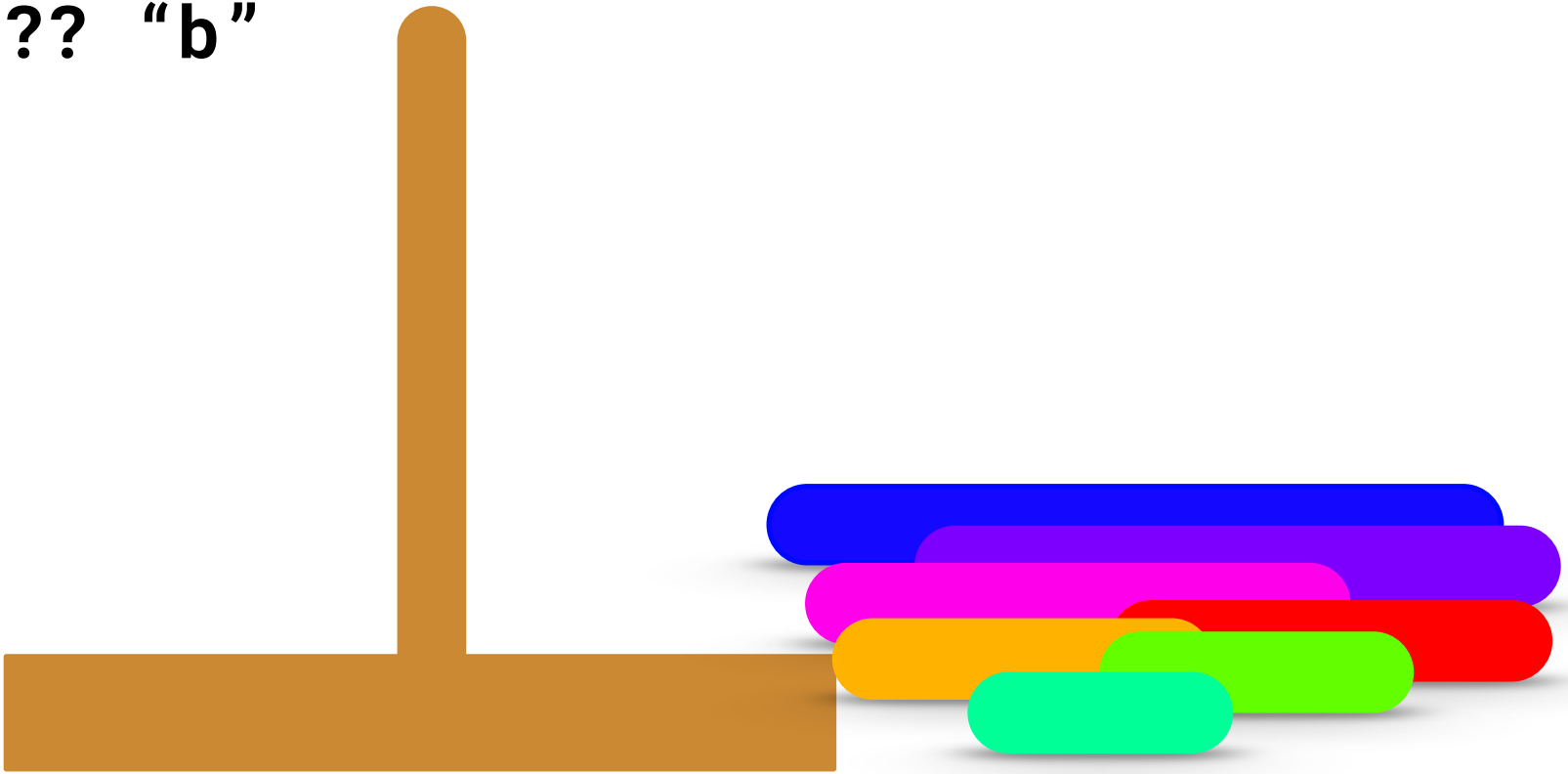
**“the assembly writing game you never knew you wanted”**

# What is a stack?

# What is a stack?

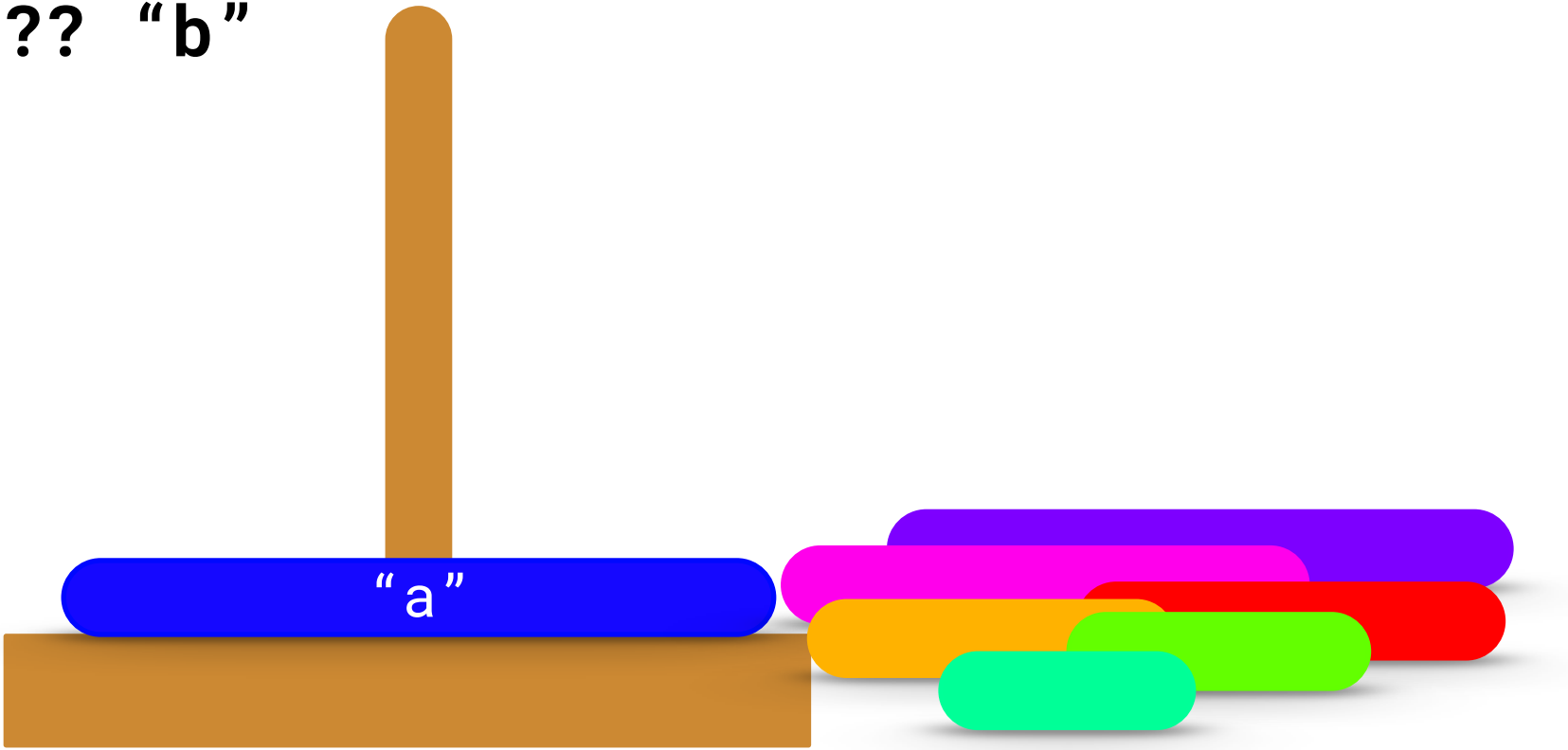


input:
“a” ?? “b”



get 0

**input:**
**“a” ?? “b”**



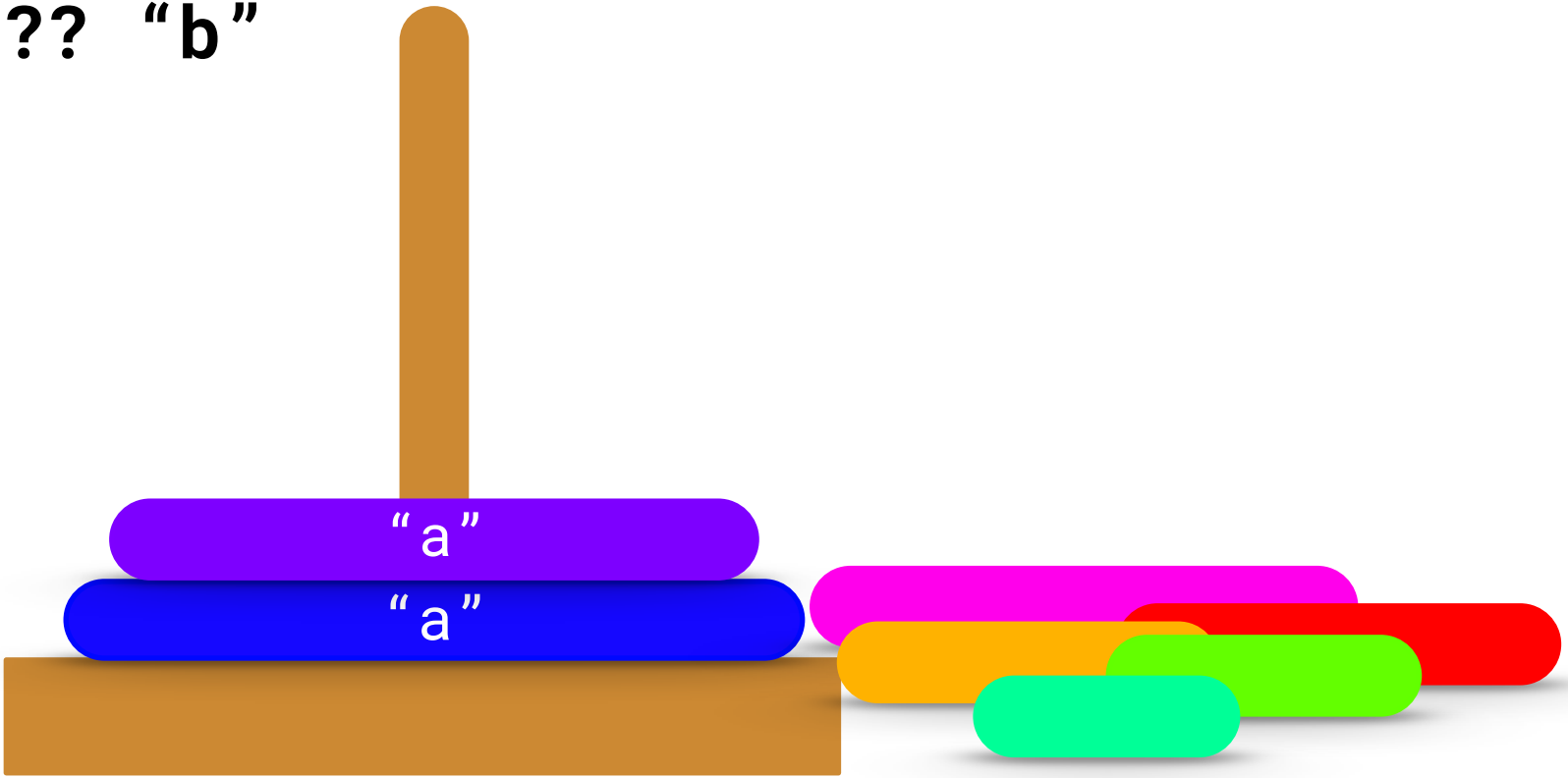
## JSOP_DUP [-1, +2]

| | |
|---|---|
| **Value** | `12 (0x0c)` |
| **Operands** | |
| **Length** | `1` |
| **Stack Uses** | `v` |
| **Stack Defs** | `v, v` |

Pushes a copy of the top value on the stack.

```
get 0
JSOP_DUP
```

**input:**
**“a” ?? “b”**



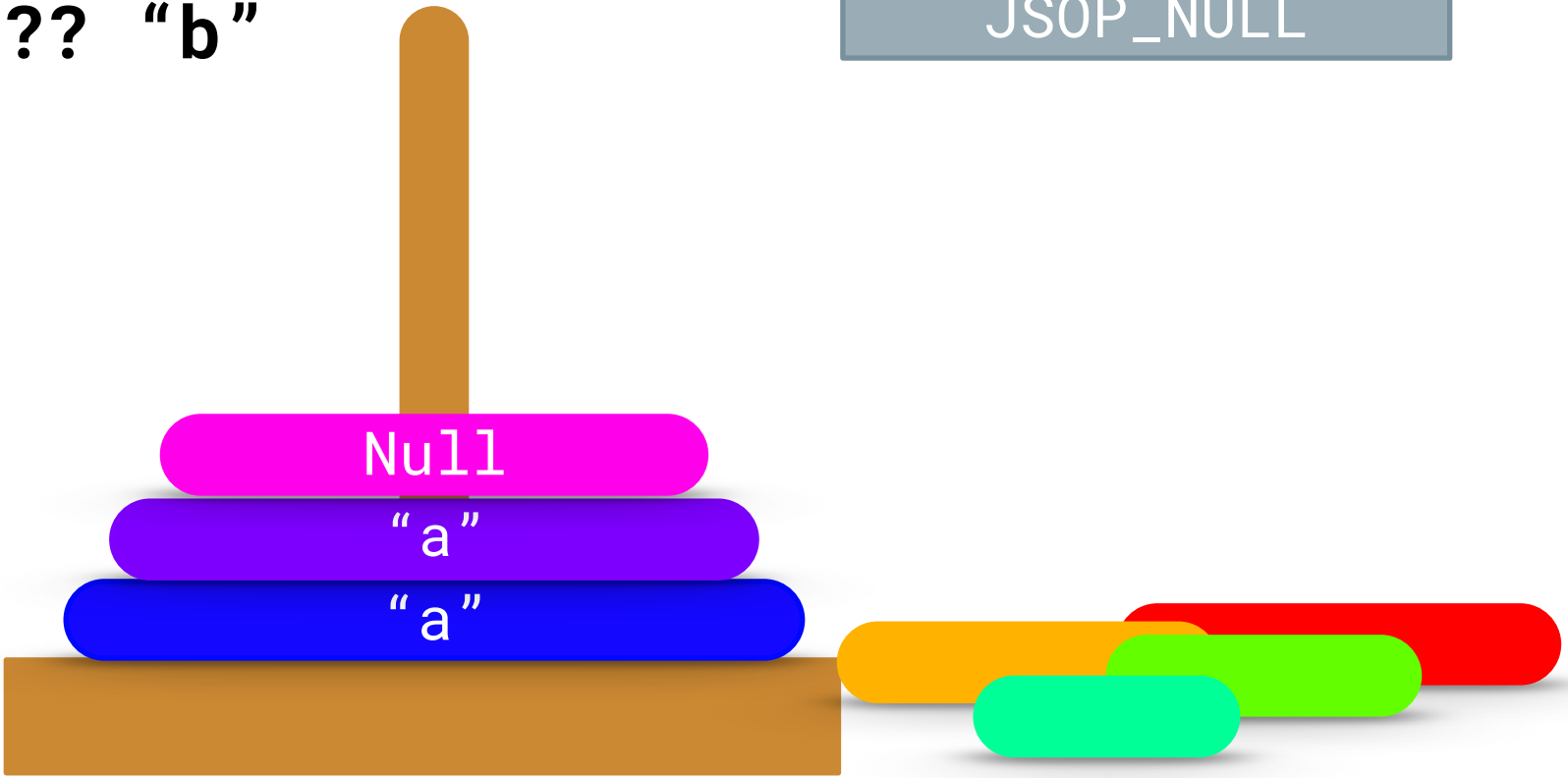
## JSOP_NULL [-0, +1]

| | |
|---|---|
| **Value** | `64 (0x40)` |
| **Operands** | |
| **Length** | `1` |
| **Stack Uses** | |
| **Stack Defs** | `null` |

Pushes `null` onto the stack.

**input:**
**“a” ?? “b”**

```
get 0
JSOP_DUP
JSOP_NULL
```



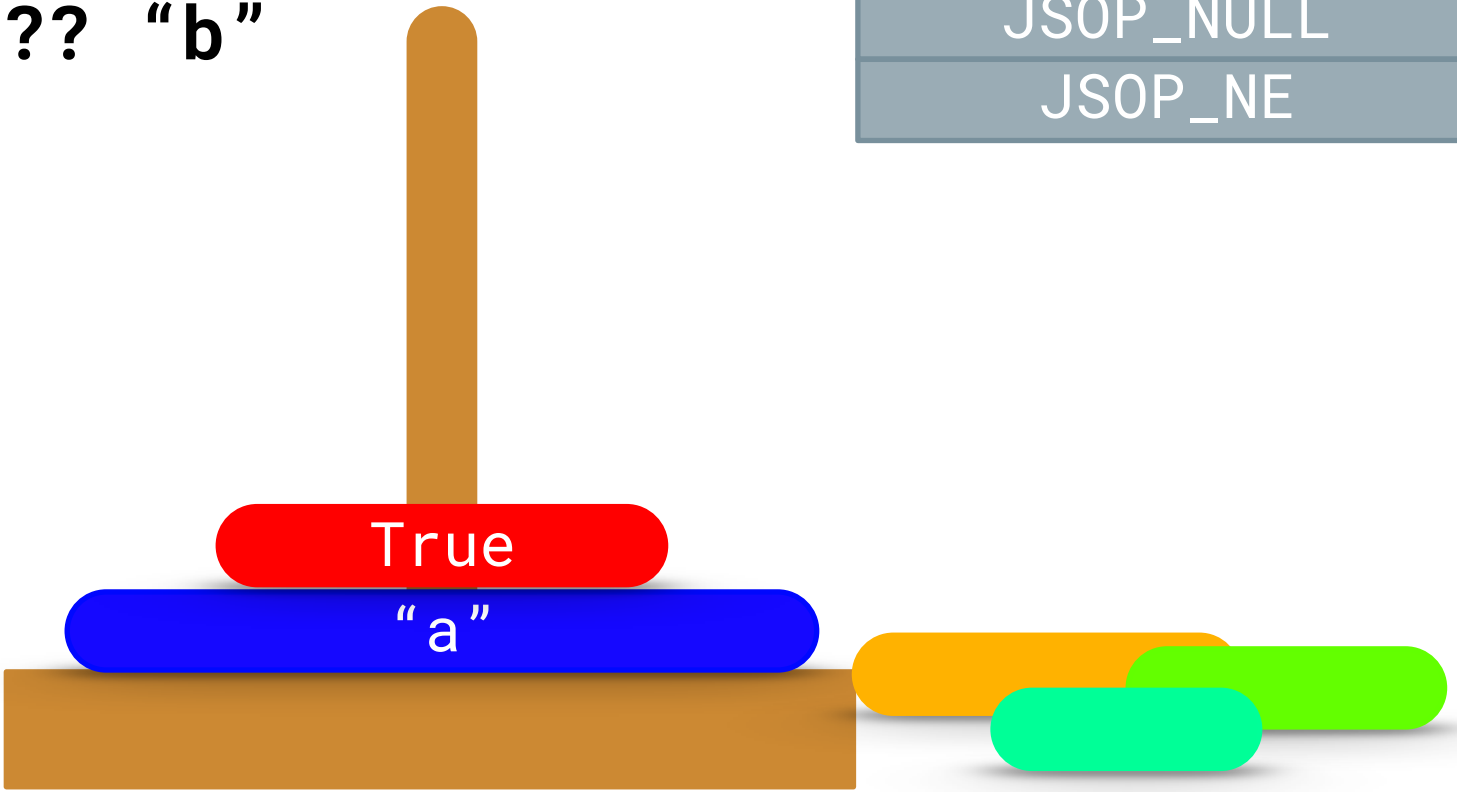
## JSOP_NE [-2, +1] (DETECTING, IC)

| | |
|---|---|
| **Value** | `JSOP_NE: 19 (0x13)` |
| **Operands** | |
| **Length** | `1` |
| **Stack Uses** | `lval, rval` |
| **Stack Defs** | `(lval OP rval)` |

Pops the top two values from the stack and pushes the result of comparing them.

**input:**
**“a” ?? “b”**

```
get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
```



## JSOP_IFNE [-1, +0] (JUMP, IC)

| | |
|---|---|
| **Value** | `8 (0x08)` |
| **Operands** | `int32_t offset` |
| **Length** | `5` |
| **Stack Uses** | `cond` |
| **Stack Defs** | |

Pops the top of stack value, converts it into a boolean, if the result is `true`, jumps to a 32-bit offset from the current bytecode.

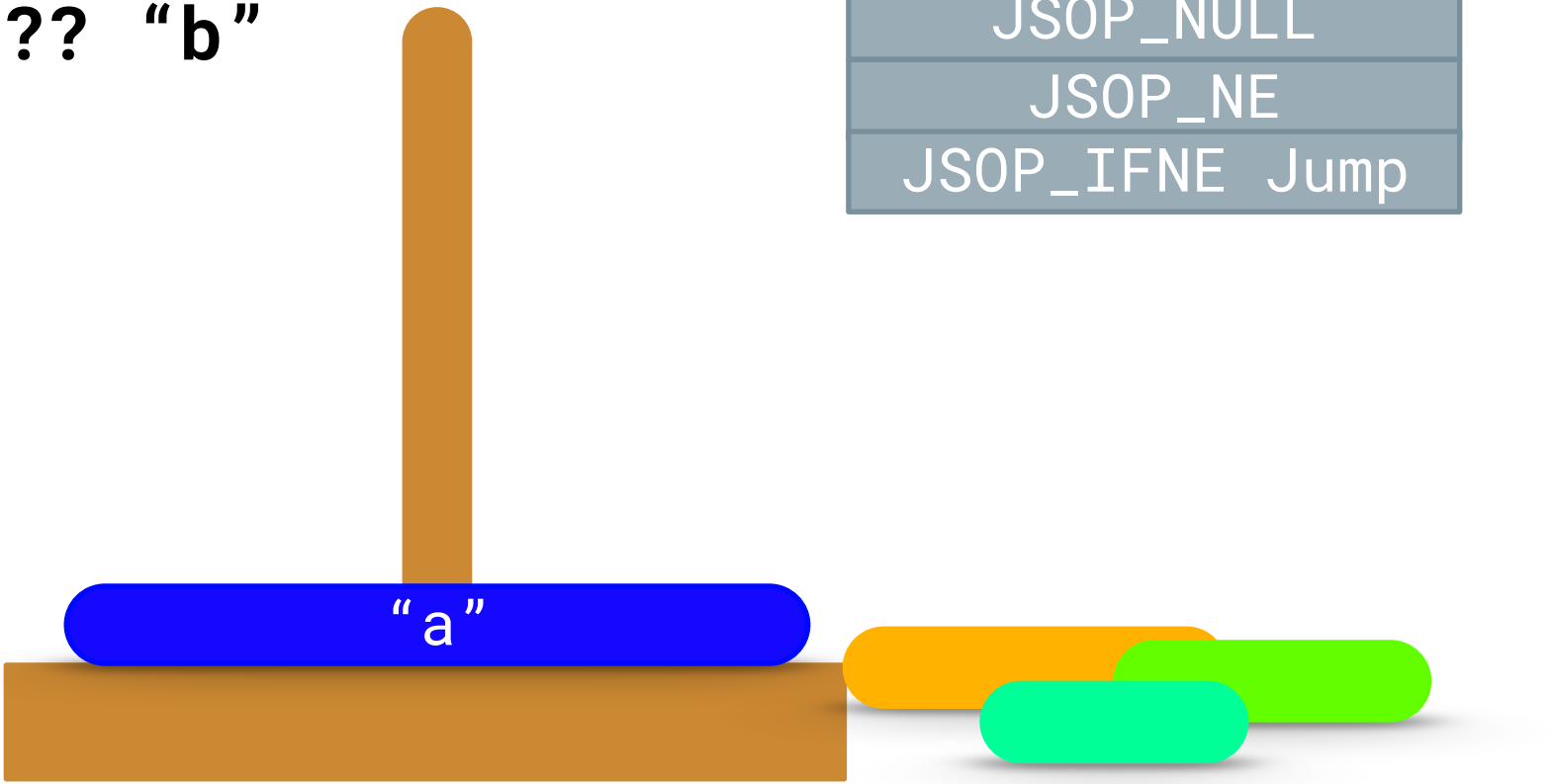
### JSOP_IFEQ [-1, +0] (JUMP, DETECTING, IC)

| | |
|---|---|
| **Value** | `7 (0x07)` |
| **Operands** | `int32_t offset` |
| **Length** | `5` |
| **Stack Uses** | `cond` |
| **Stack Defs** | |

Pops the top of stack value, converts it into a boolean, if the result is `false`, jumps to a 32-bit offset from the current bytecode.

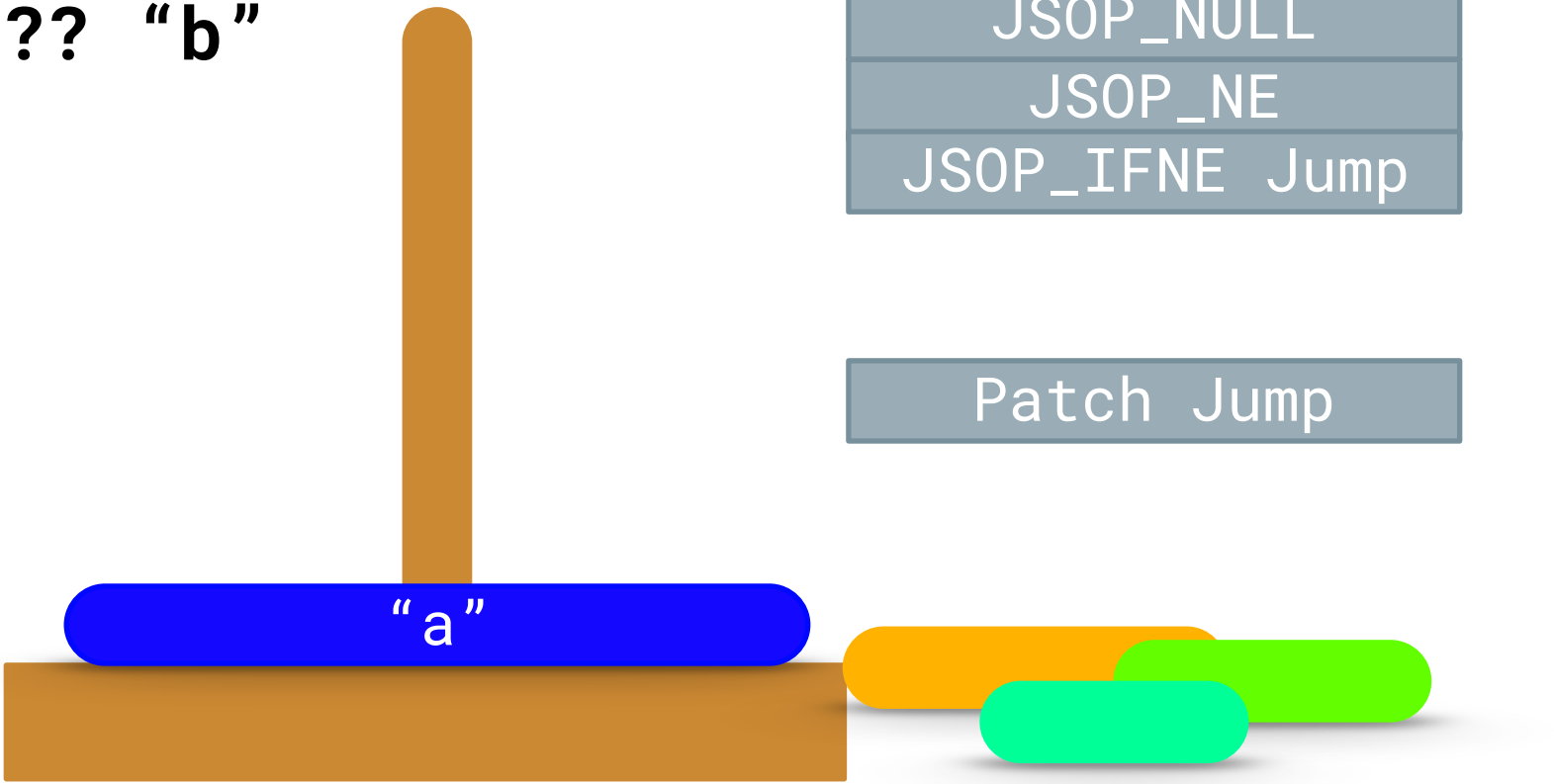
The idea is that a sequence like `JSOP_ZERO`; `JSOP_ZERO`; `JSOP_EQ`; `JSOP_IFEQ`; `JSOP_RETURN`; reads like a nice linear sequence that will execute the return.

**input:**
**“a” ?? “b”**




```
get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
JSOP_IFNE Jump
```

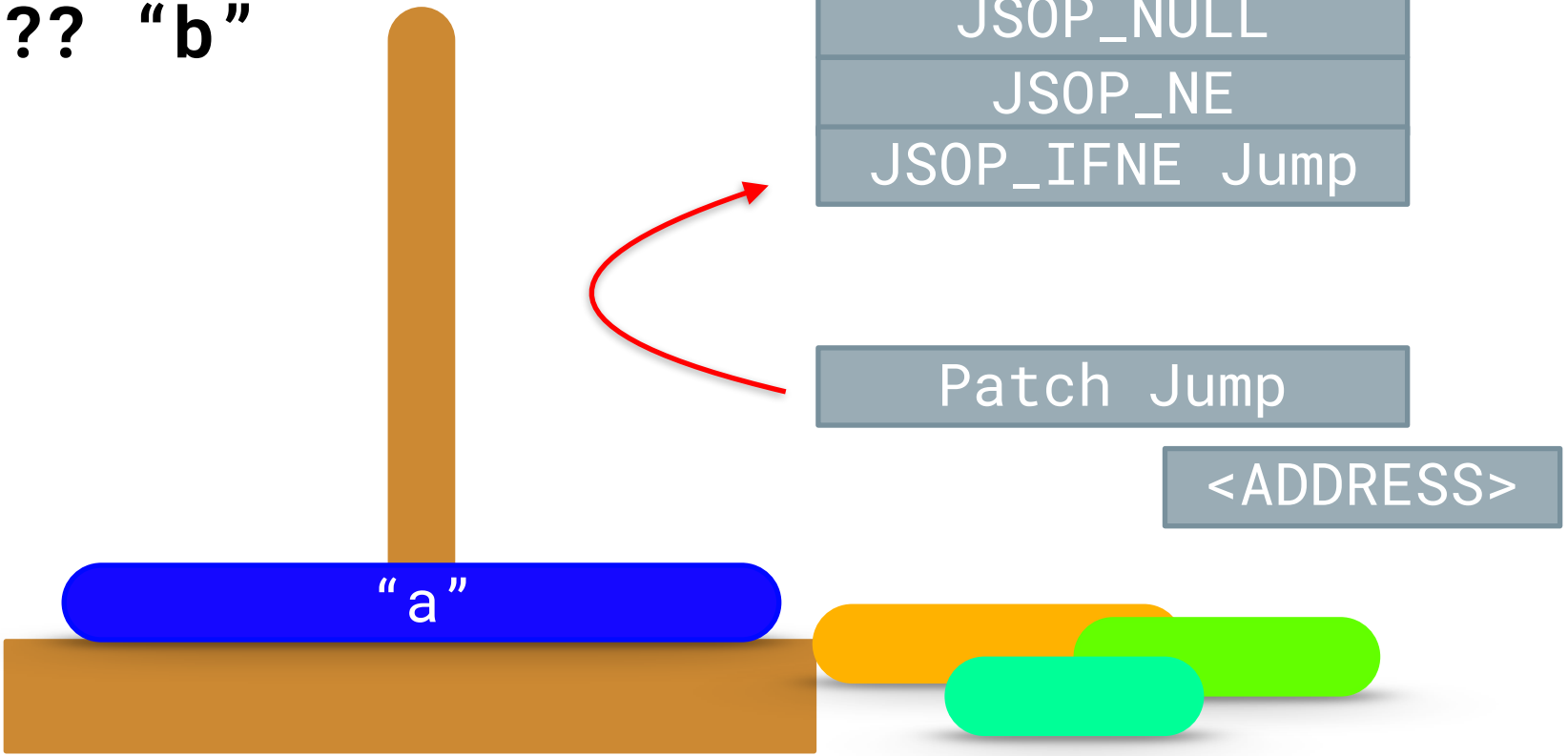
**input:**
**“a” ?? “b”**



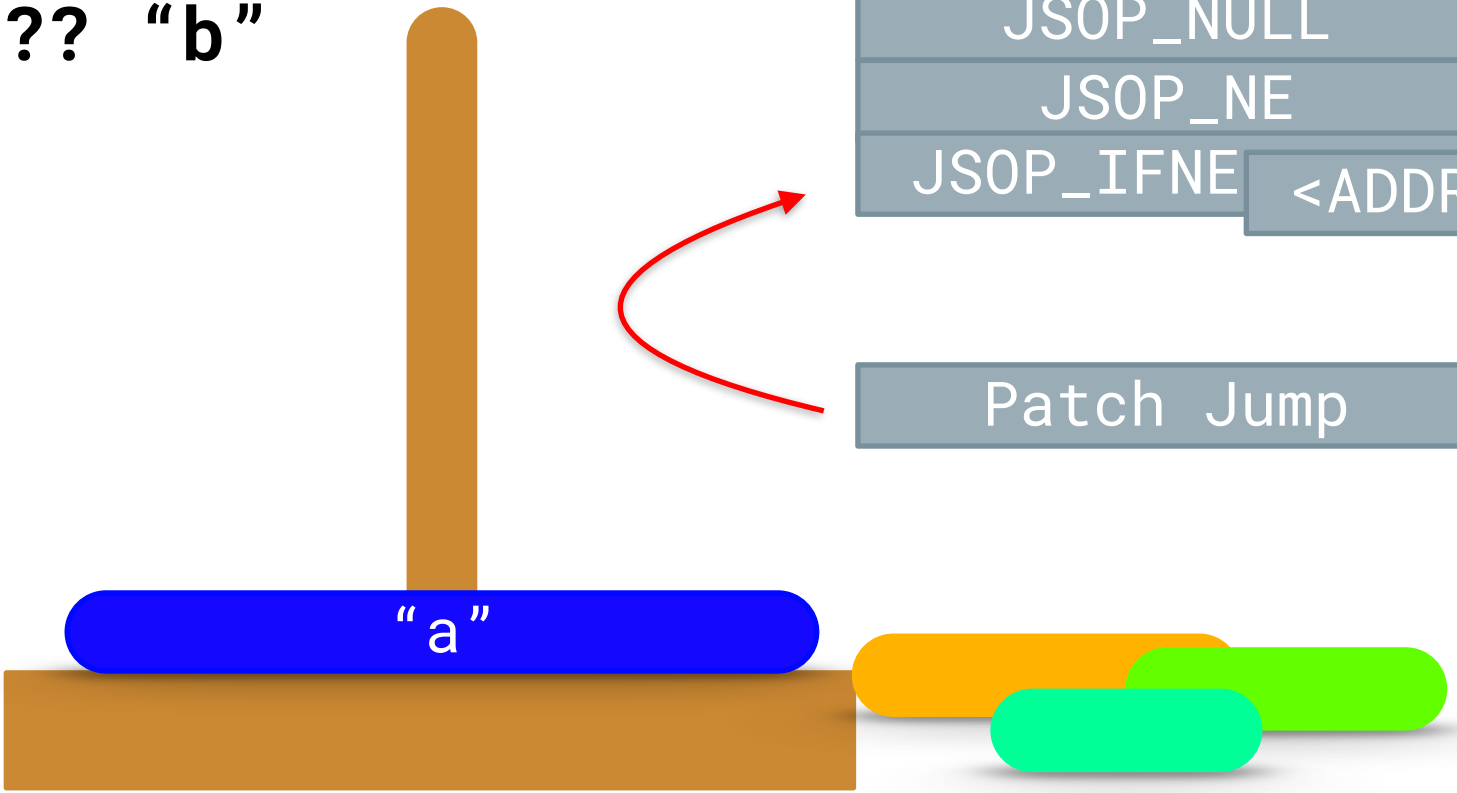

```
get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
JSOP_IFNE Jump
```

```
Patch Jump
```

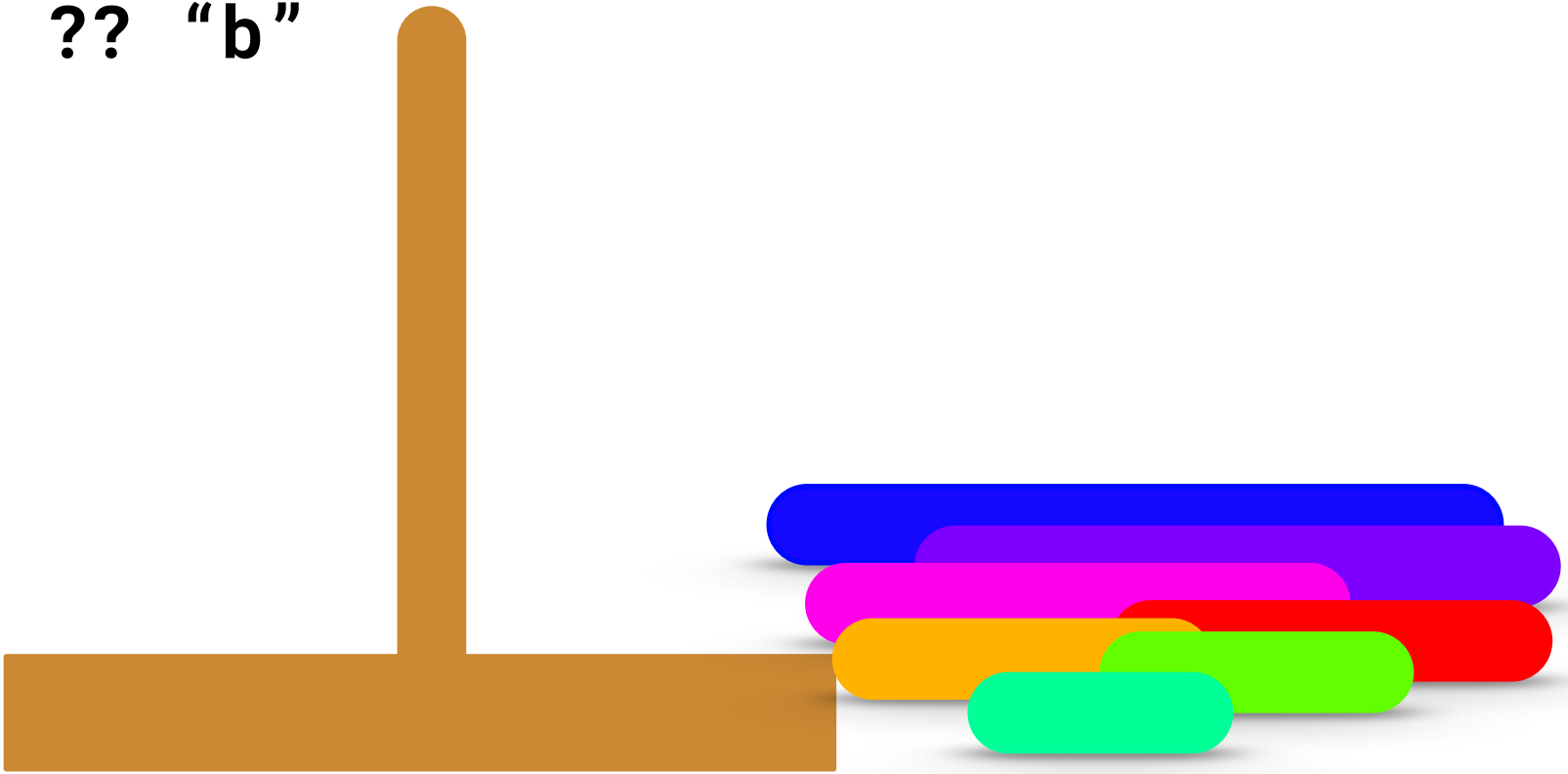
**input:**
**“a” ?? “b”**


get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
JSOP_IFNE Jump
Patch Jump
<ADDRESS>
“a”

**input:**
**“a” ?? “b”**

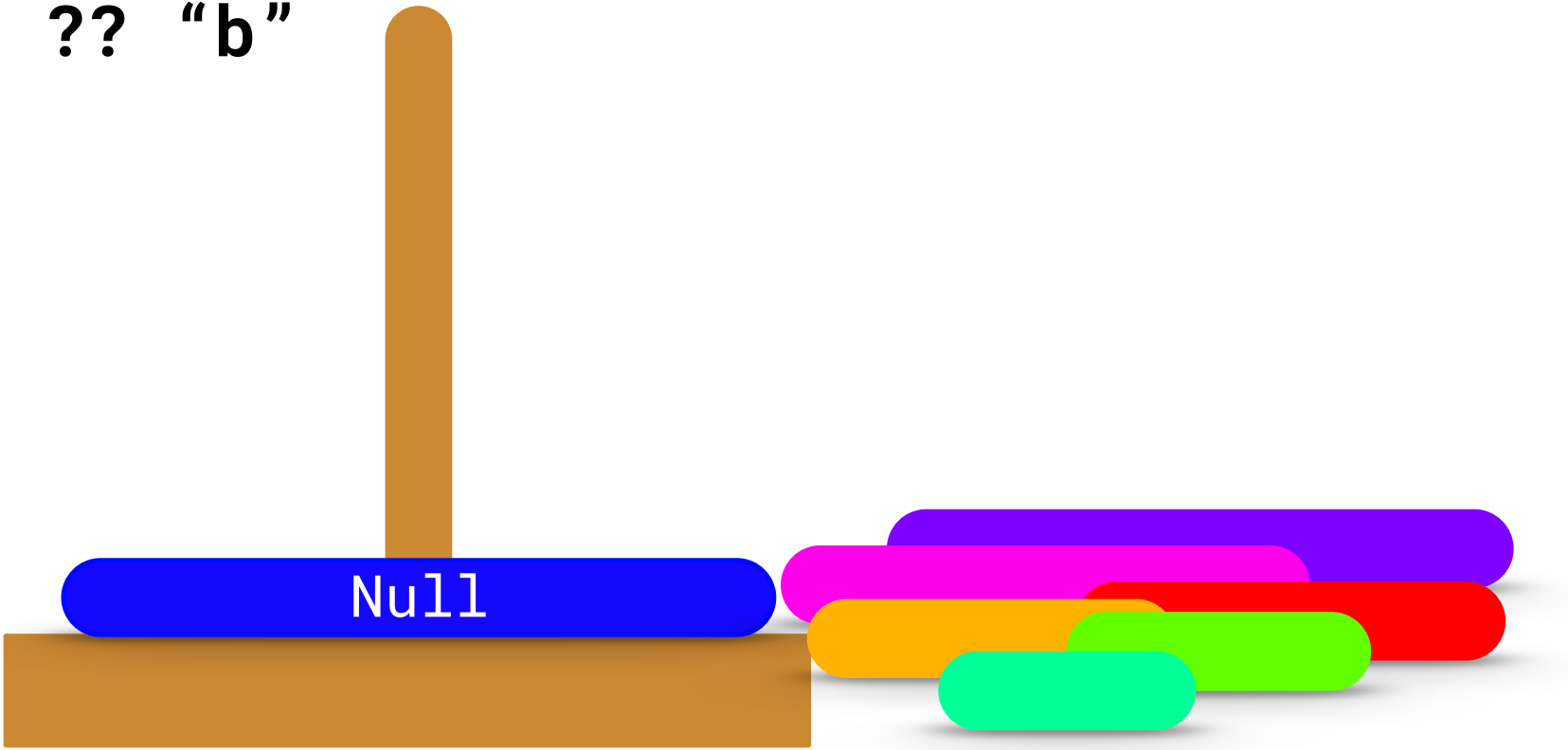

get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
JSOP_IFNE
<ADDRESS>
Patch Jump
“a”

```
input:
null ?? “b”
```



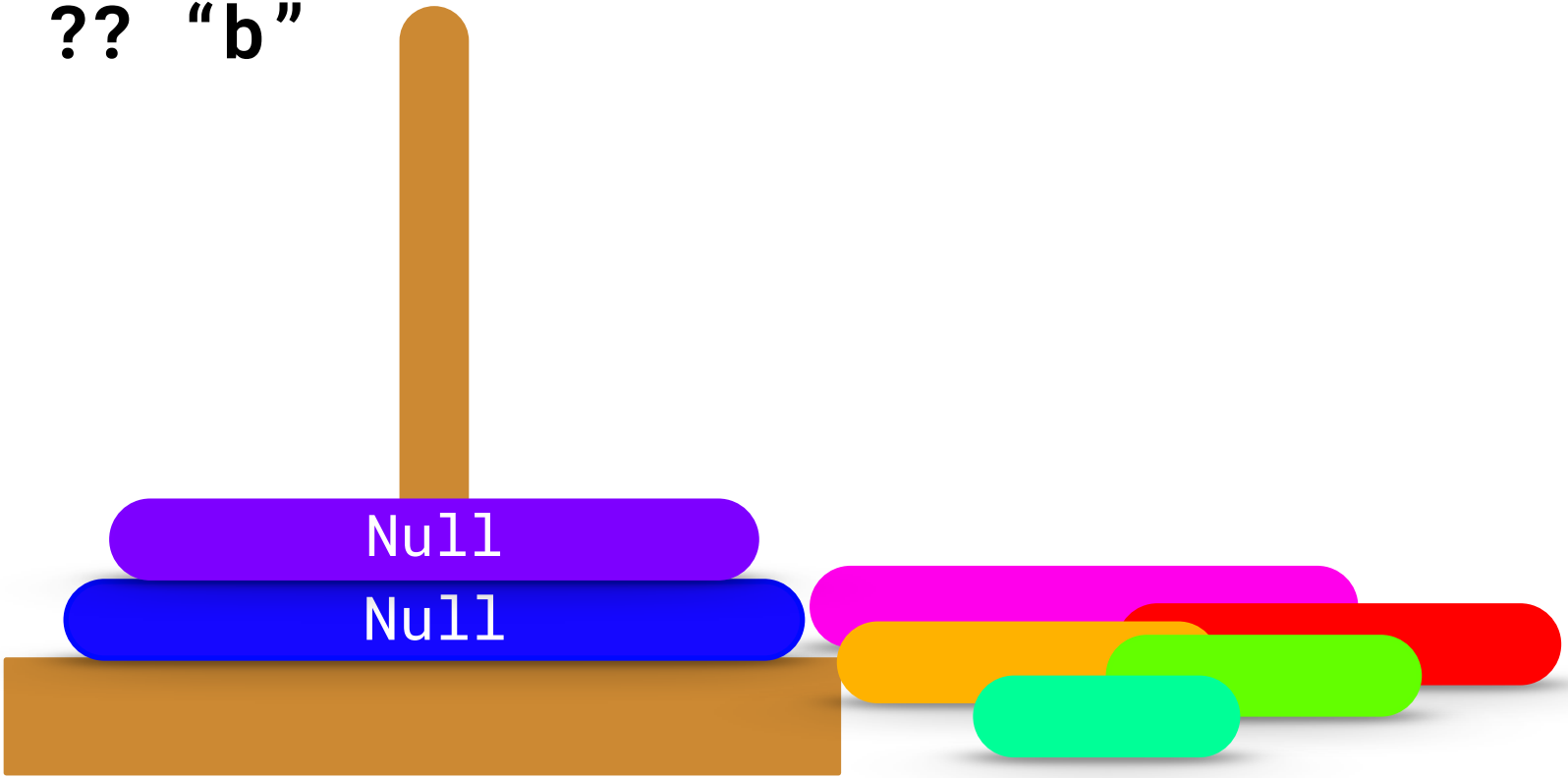
get 0

**input:**
**null ?? “b”**



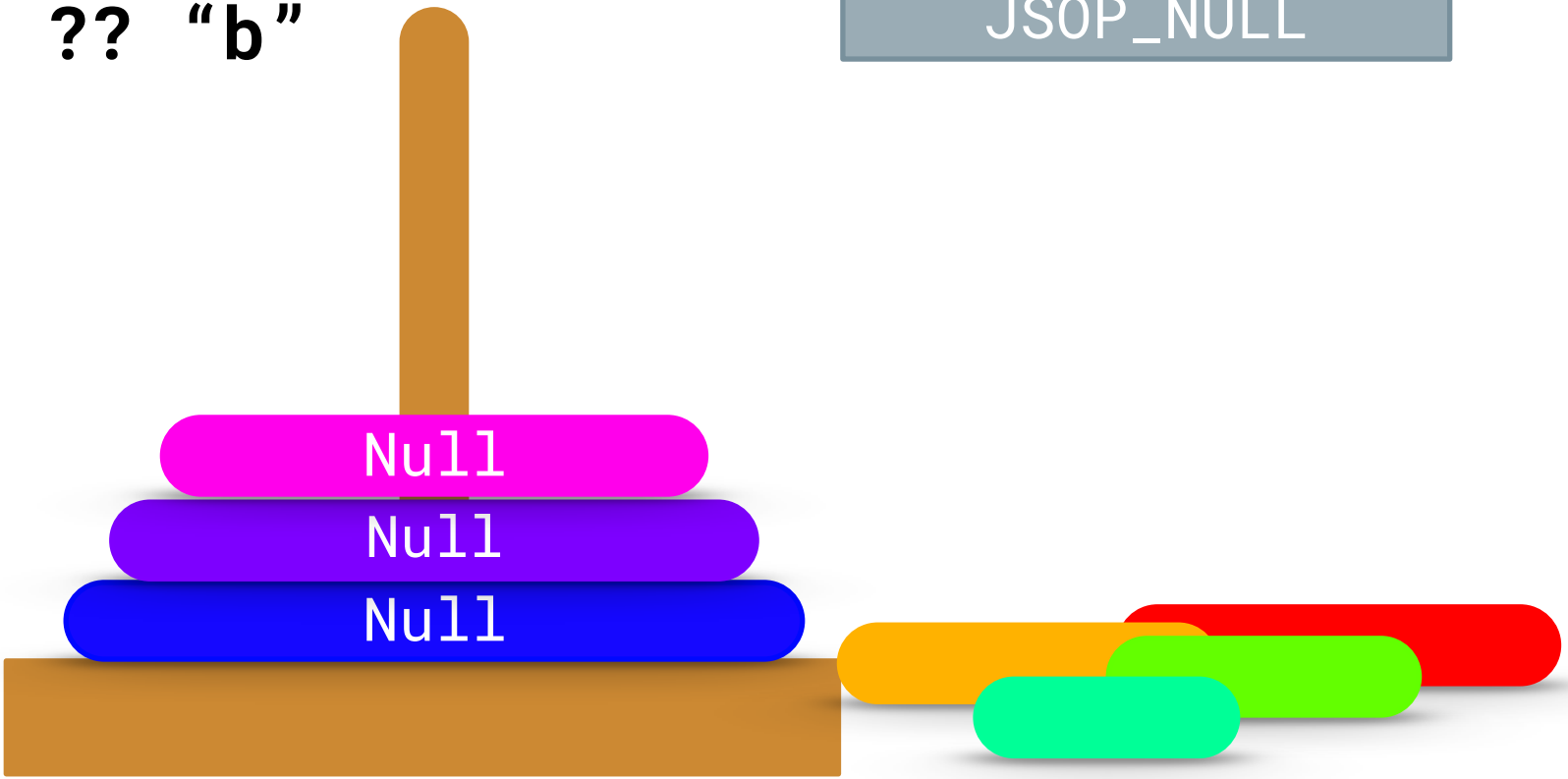
**input:**
**null ?? “b”**

```
get 0
JSOP_DUP
```



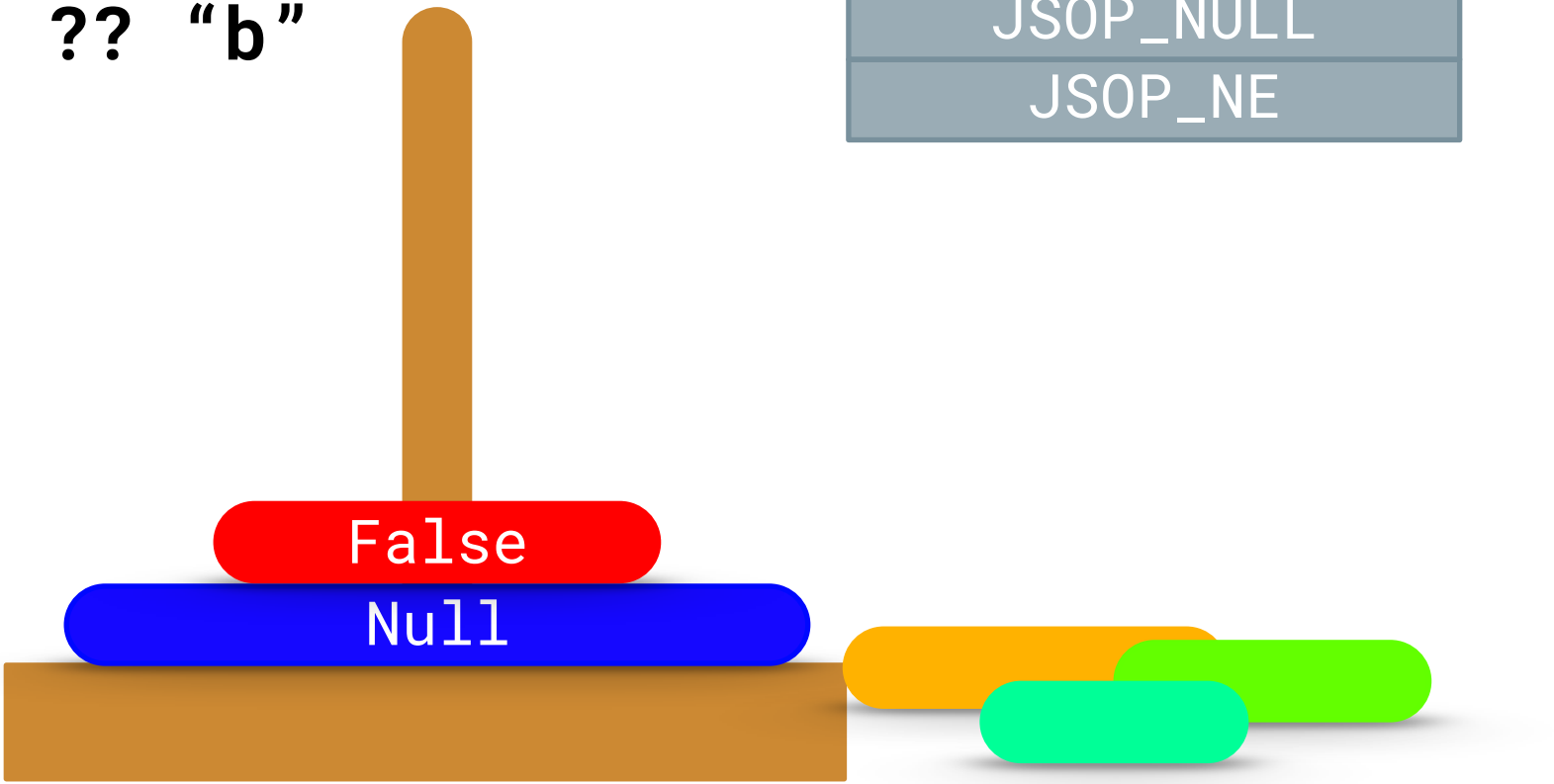
**input:**
**null ?? “b”**

```
get 0
JSOP_DUP
JSOP_NULL
```



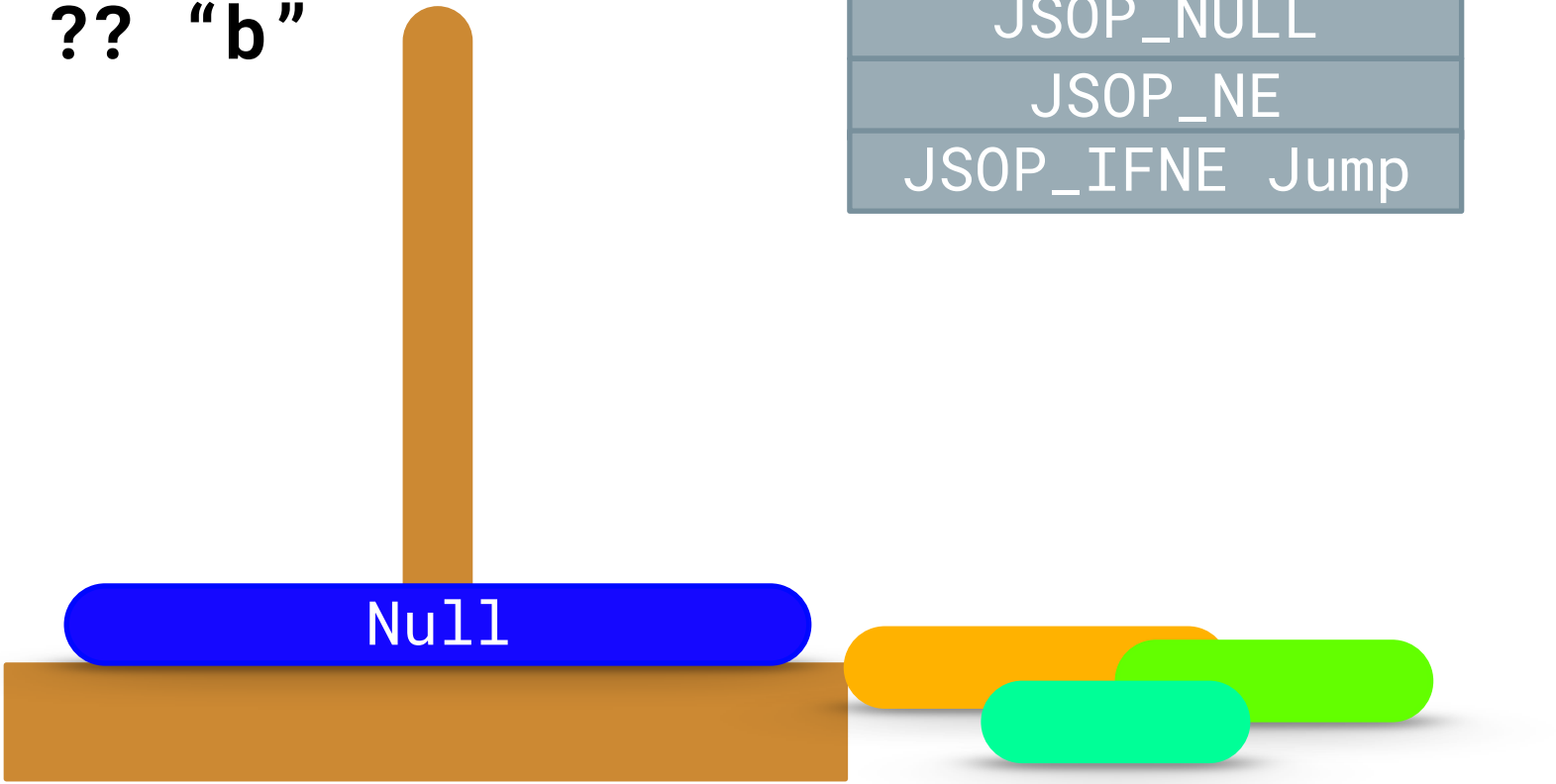
**input:**
**null ?? “b”**

```
get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
```



**input:**
**null ?? “b”**

```
get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
JSOP_IFNE Jump
```



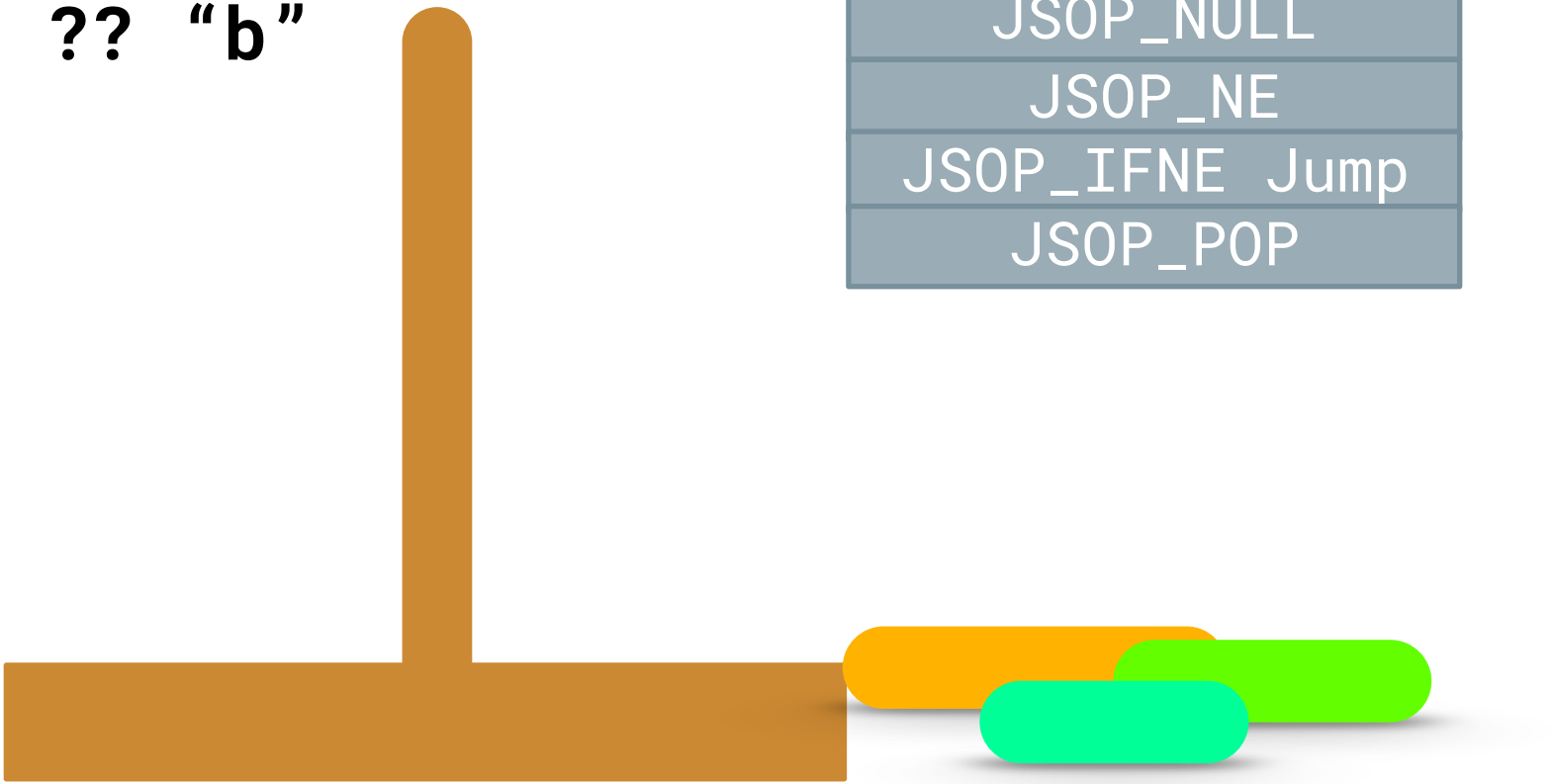
## JSOP_POP [-1, +0]

| | |
|---|---|
| **Value** | `81 (0x51)` |
| **Operands** | |
| **Length** | `1` |
| **Stack Uses** | `v` |
| **Stack Defs** | |

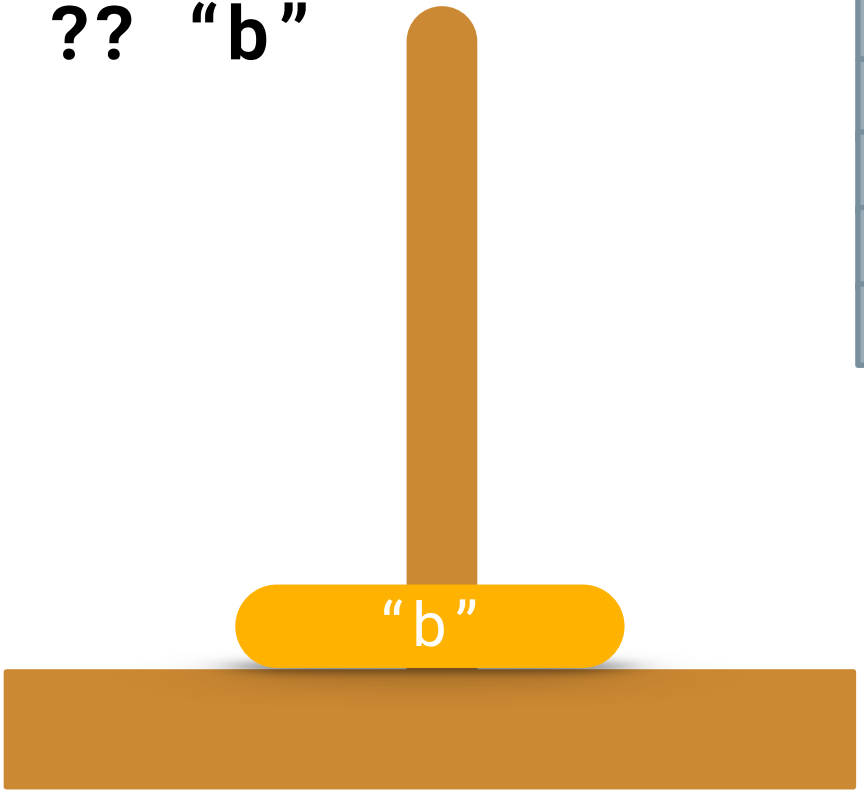
Pops the top value off the stack.

**input:**
**null ?? “b”**

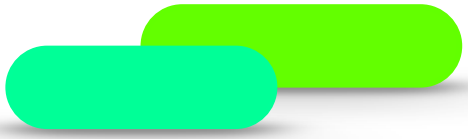
```
get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
JSOP_IFNE Jump
JSOP_POP
```



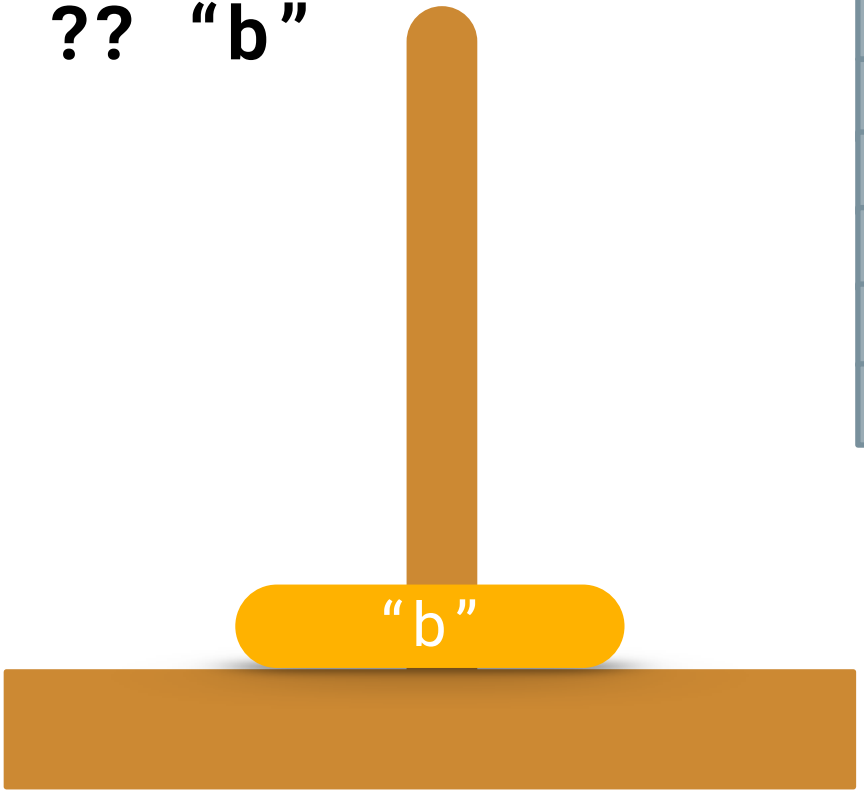
**input:**
**null ?? “b”**



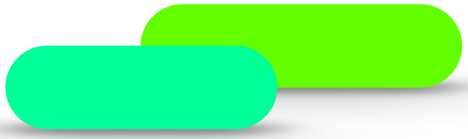

```
get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
JSOP_IFNE Jump
JSOP_POP
get 1
```



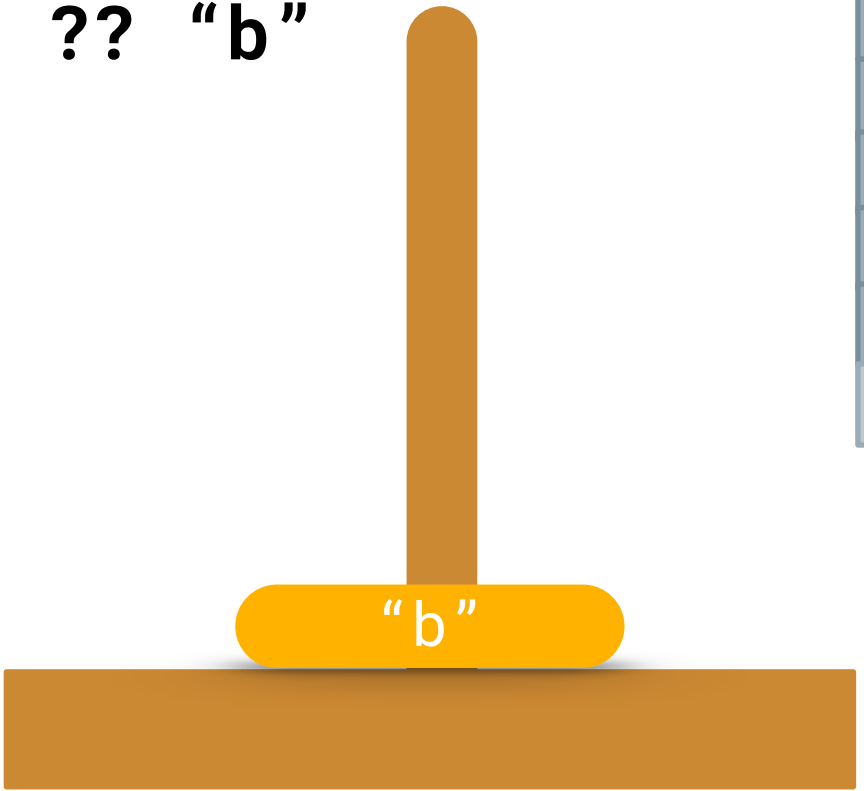
**input:**
**null ?? “b”**



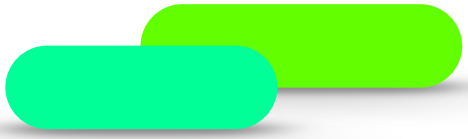

```
get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
JSOP_IFNE Jump
JSOP_POP
get 1
Patch Jump
```



**input:**
**null ?? “b”**




```
get 0
JSOP_DUP
JSOP_NULL
JSOP_NE
JSOP_IFNE Jump
JSOP_POP
get 1
Patch Jump
```



# The implementation

```
7576 bool BytecodeEmitter::emitNullCoalesce(ListNode* node) {
7577   MOZ_ASSERT(node->isKind(ParseNodeKind::CoalesceExpr));
7578   TDZCheckCache tdzCache(this);
7579   JumpList jump;
7580   for (ParseNode* expr = node->head();; expr = expr->pn_next) {
7581     // Loop
7582   }
7583
7584   // end
7585
7586   return true;
7587 }
```

## JSOP_IFNE [-1, +0] (JUMP, IC)

| | |
|---|---|
| **Value** | `8 (0x08)` |
| **Operands** | `int32_t offset` |
| **Length** | `5` |
| **Stack Uses** | `cond` |
| **Stack Defs** | |

Pops the top of stack value, converts it into a boolean, if the result is `true`, jumps to a 32-bit offset from the current bytecode.

### JSOP_IFEQ [-1, +0] (JUMP, DETECTING, IC)

| | |
|---|---|
| **Value** | `7 (0x07)` |
| **Operands** | `int32_t offset` |
| **Length** | `5` |
| **Stack Uses** | `cond` |
| **Stack Defs** | |

Pops the top of stack value, converts it into a boolean, if the result is `false`, jumps to a 32-bit offset from the current bytecode.

The idea is that a sequence like `JSOP_ZERO`; `JSOP_ZERO`; `JSOP_EQ`; `JSOP_IFEQ`; `JSOP_RETURN`; reads like a nice linear sequence that will execute the return.

# The implementation

```
7576 bool BytecodeEmitter::emitNullCoalesce(ListNode* node) {
7577   MOZ_ASSERT(node->isKind(ParseNodeKind::CoalesceExpr));
7578   TDZCheckCache tdzCache(this);
7579   JumpList jump;
7580   for (ParseNode* expr = node->head();; expr = expr->pn_next) {
7581     // Loop
7582   }
7583
7584   // end
7585
7586   return true;
7587 }
```

# Full Parse

Text

```
function coalesce(a, b) {
  return a ?? b;
}
```

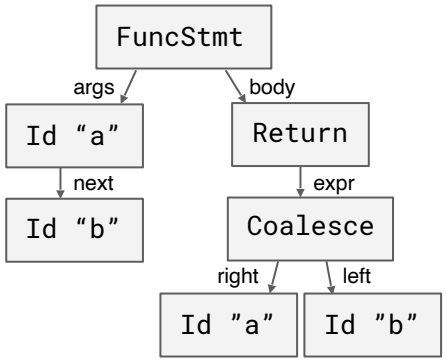
Tokenize

Tokens

```
[Function][Iden="coalesce"][LeftParen][Iden="a"][Comma][Iden="b"][RightParen][LeftCurly]
[Return][Iden="a"][Coalesce][Iden="b"][Semi][RightCurly]
```

Parse

Abstract Syntax Tree (AST)

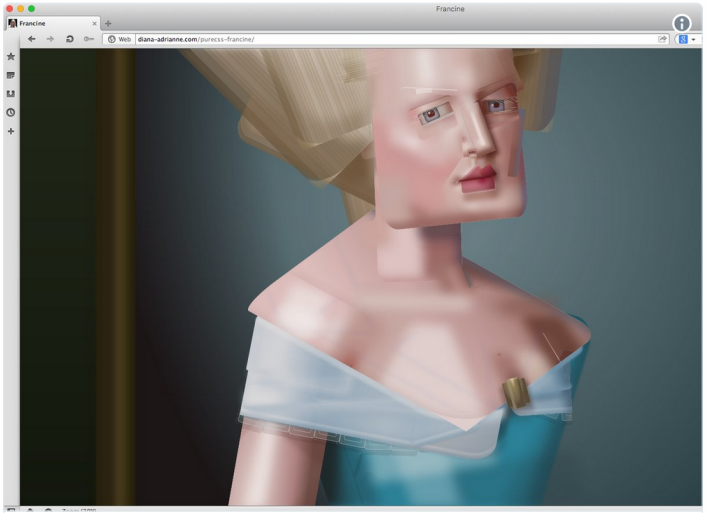



Bytecode-Compile

Bytecode

```
JSScript
getarg 0
coalesce
Jumptarget 0
pop
getarg 1
Jumptarget 0
return
```

# <aside>




**David Zhou** ✓
@dz

i tried @cyanharlow's incredible pure css portrait in an old version of opera and well, the disclaimer wasn't lying: "so the live preview will most likely look laughable in anything other than chrome"github.com/cyanharlow/pur…

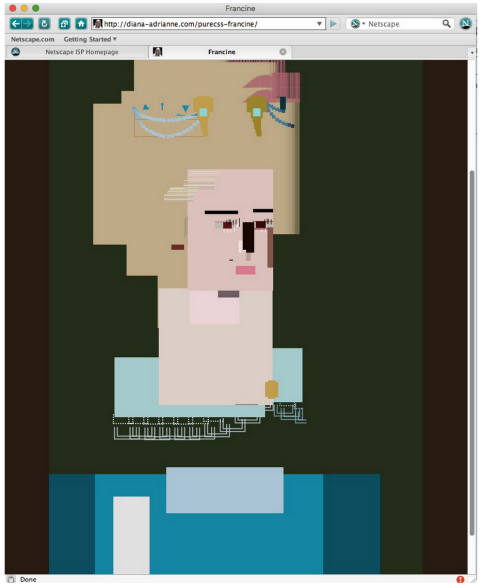
7:17 PM - May 1, 2018

♡ 1,269 💬 424 people are talking about this

**Mayowa Tomori**
@mdotslash

And Netscape Navigator for the true romantics amongst you.
pic.twitter.com/hO12KvVoJg

4:50 AM - May 2, 2018




♡ 238 💬 54 people are talking about this

[Pure CSS Francine ](#) by Diana Smith

# </aside>


» document.all

Image of the programmer at work


» document.all

Image of the programmer at work

## JSOP_STRICTEQ [-2, +1] (DETECTING, IC)

| | |
|---|---|
| **Value** | `JSOP_STRICTEQ: 72 (0x48)` |
| **Operands** | |
| **Length** | `1` |
| **Stack Uses** | `lval, rval` |
| **Stack Defs** | `(lval OP rval)` |

Pops the top two values from the stack, then pushes the result of applying the operator to the two values.

```
7576 bool BytecodeEmitter::emitNullCoalesce(ListNode* node) {
7577   MOZ_ASSERT(node->isKind(ParseNodeKind::CoalesceExpr));
7578   TDZCheckCache tdzCache(this);
7579   JumpList jump;
7580   for (ParseNode* expr = node->head();; expr = expr->pn_next) {
7581     if (!emitTree(expr)) { return false; }
7582
7583     if (!expr->pn_next) { break; }
7584
7585     if (!emitPushNotUndefinedOrNull()) { return false; }
7586
7587     if (!emit1(JSOP_NOT)) { return false; }
7588
7589     if(!this->newSrcNote(SRC_IF)) { return false; }
7590
7591     if (!emitJump(JSOP_IFEQ, &jump)) { return false; }
7592
7593     if (!emit1(JSOP_POP)) { return false; }
7594   }
7595
7596   if (!emitJumpTargetAndPatch(jump)) { return false; }
7597
7598   return true;
7599 }
```

# Inspecting the byte code

```
js> function foo(a, b) { a ?? b }
js> dis(foo)
flags: CONSTRUCTOR
loc     op
-----   --
main:
00000:  getarg 0                        # a
00003:  dup                             # a a
00004:  undefined                       # a a undefined
00005:  strictne                        # a (a !== undefined)
00006:  and 20 (+14)                    # a (a !== undefined)
00011:  jumptarget (ic: 5)              # a (a !== undefined)
00016:  pop                             # a
00017:  dup                             # a a
00018:  null                            # a a null
00019:  strictne                        # a (a !== null)

# from and @ 00006
00020:  jumptarget (ic: 6)              # a merged<(a !== undefined)>
00025:  ifeq 40 (+15)                   # a
00030:  jumptarget (ic: 7)              # a
00035:  goto 49 (+14)                   # a

# from ifeq @ 00025
00040:  jumptarget (ic: 7)              # a
00045:  pop                             #
00046:  getarg 1                        # b

# from goto @ 00035
00049:  jumptarget (ic: 7)              # merged<a>
00054:  pop                             #
00055:  retrval                         #
```

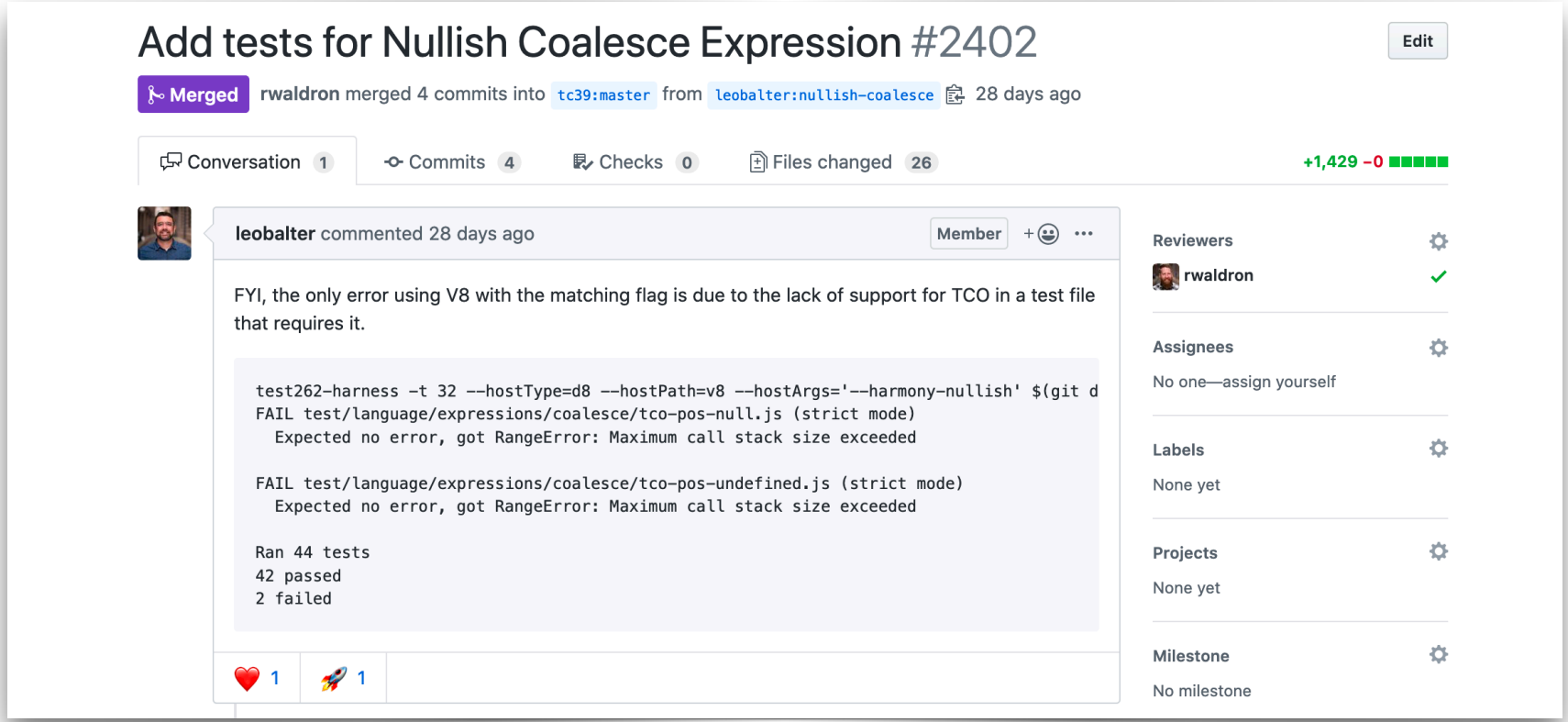
Image of the programmer at work

# JSOP_COALESCE

# JSOP_COALESCE bytecode

```
js> dis(foo)
flags: CONSTRUCTOR
loc     op
-----   --
main:
00000:  getarg 0                        # a
00003:  coalesce 17 (+14)               # <unknown>
00008:  jumptarget (ic: 3)              # <unknown>
00013:  pop                             #
00014:  getarg 1                        # b

# from coalesce @ 00003
00017:  jumptarget (ic: 3)              # merged<<unknown>>
00022:  return                          #
00023:  retrval                         # !!! UNREACHABLE !!!
```

# Pulling in tests



# 1.3 Runtime Semantics: Evaluation

*CoalesceExpression* : *CoalesceExpressionHead* **??** *BitwiseORExpression*

1. Let *lref* be the result of evaluating *CoalesceExpressionHead*.
2. Let *lval* be ? GetValue(*lref*).
3. If *lval* is **undefined** or **null**,
    a. Let *rref* be the result of evaluating *BitwiseORExpression*.
    b. Return ? GetValue(*rref*).
4. Otherwise, return *lval*.

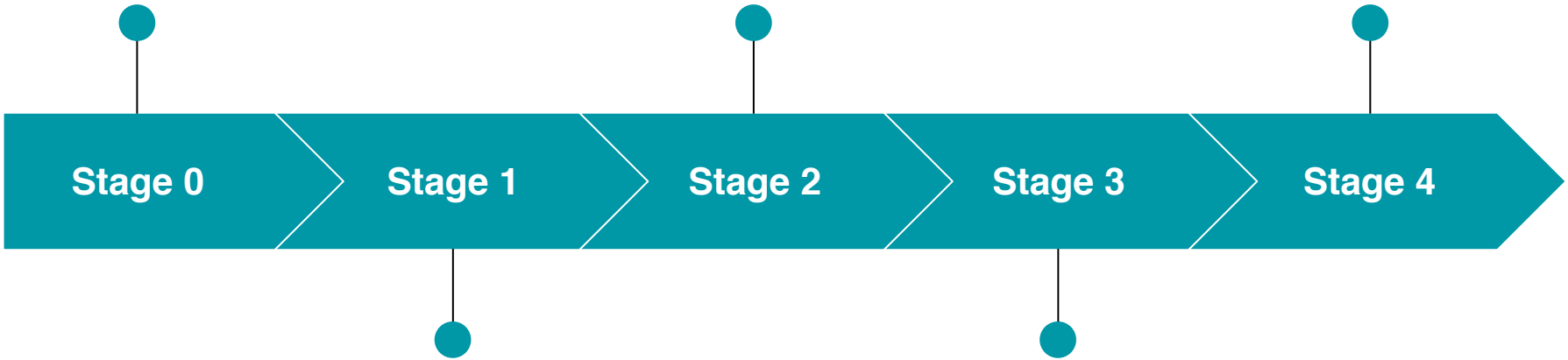
# Semantics Covered.

# Review

Someone has an idea and they write it up

Committee discusses if this feature “should be in the language”

Proposal is included in the specification




The idea is presented to the committee, committee makes comments

Polyfill and browser implementations, final form of the proposal takes shape

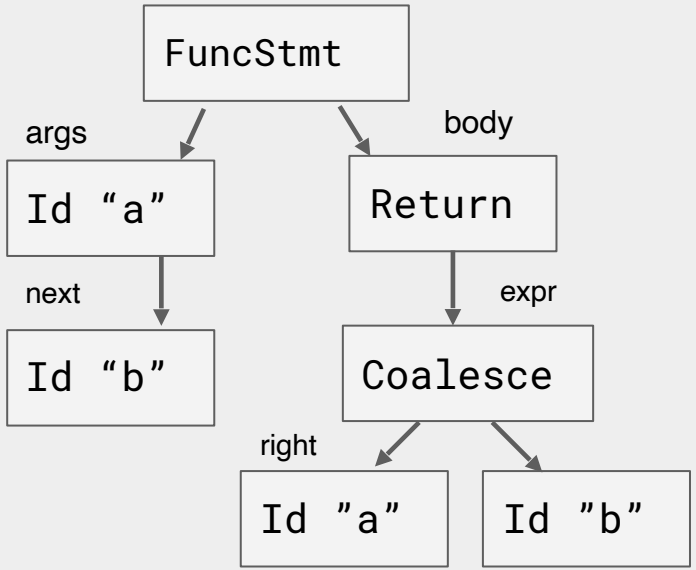
**Syntax**

Semantics

# Given a Minimal program ....

```
function coalesce(a, b) {
  return a ?? b;
}
```

# Building the AST



# Emit the Bytecode

JSScript

```
getarg 0
coalesce
jumptarget 0
pop
getarg 1
jumptarget 0
return
```

# Full Parse

Text

```
function coalesce(a, b) {
  return a ?? b;
}
```

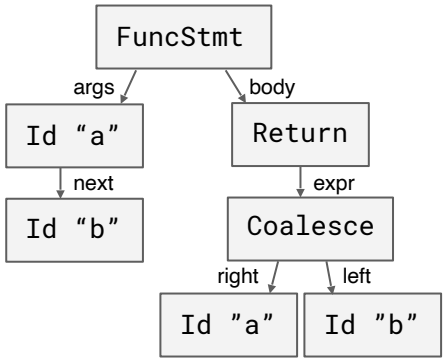
Tokenize

Tokens

```
[Function][Iden="coalesce"][LeftParen][Iden="a"][Comma][Iden="b"][RightParen][LeftCurly]
[Return][Iden="a"][Coalesce][Iden="b"][Semi][RightCurly]
```

Parse

Abstract Syntax Tree (AST)




Bytecode-Compile

Bytecode

```
JSScript
getarg 0
coalesce
Jumptarget 0
pop
getarg 1
Jumptarget 0
return
```

# Are we done?

# Stage 4

Indicate that the addition is ready for inclusion in the formal ECMAScript standard

**Requirements**

- Test262 acceptance tests have been written for mainline usage scenarios, and merged
- Two compatible implementations which pass the acceptance tests
- Significant in-the-field experience with shipping implementations, such as that provided by two independent VMs
- A pull request has been sent to tc39/ecma262 with the integrated spec text
- All ECMAScript editors have signed off on the pull request

# December 2019?


(this is an Aguaje)
Thanks!

# Still here?

# Time to do it again!