

JSSpecVis

A JavaScript Language Specification
Visualization Tool

Minseok Choe*, Kyungho Song*, Hyunjoon Kim, and Jihyeok Park

Programming Language Research Group @ Korea University

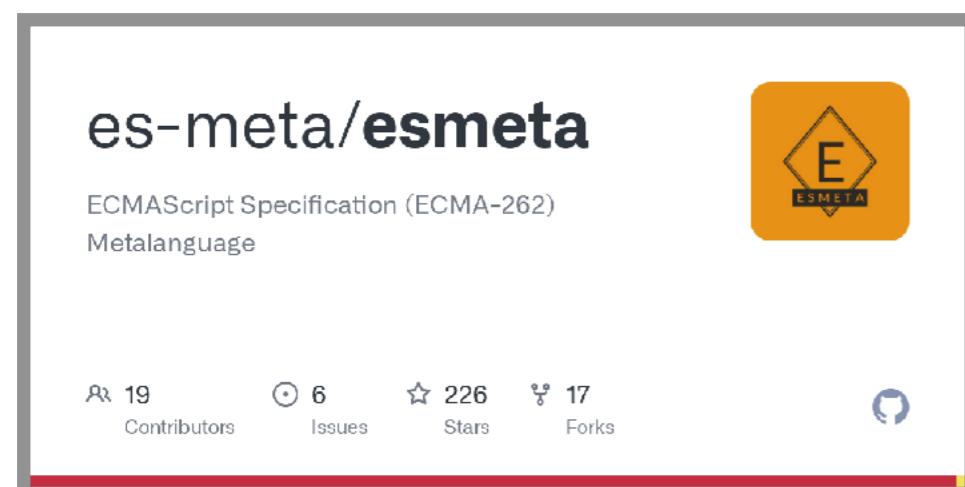
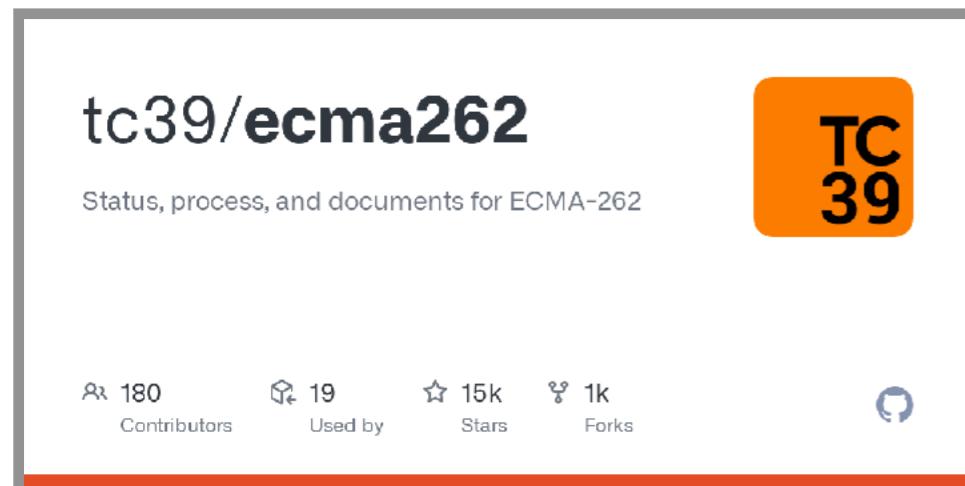
* Both authors contributed equally to this research.



JavaScript Language Specification

ECMA-262

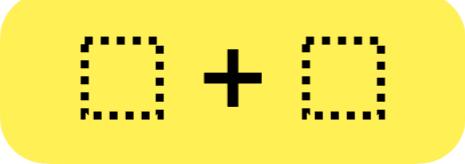
- is written in english prose, somewhat like **pseudocode**
- which make **automatic processing and tooling** possible



ApplyStringOrNumericBinaryOperator (*lVal*, *opText*, *rVal*)
It performs the following steps when called:

1. If *opText* is +, then
 - a. Let *lPrim* be ? ToPrimitive(*lVal*).
 - b. Let *rPrim* be ? ToPrimitive(*rVal*).
 - c. If *lPrim* is a String or *rPrim* is a String, then
 - i. Let *lStr* be ? ToString(*lPrim*).
 - ii. Let *rStr* be ? ToString(*rPrim*).
 - iii. Return the string-concatenation of *lStr* and *rStr*.
 - d. Set *lVal* to *lPrim*.
 - e. Set *rVal* to *rPrim*.
2. NOTE: At this point, it must be a numeric operation.
3. Let *lNum* be ? ToNumeric(*lVal*).
4. Let *rNum* be ? ToNumeric(*rVal*).
5. ...

JavaScript Language Specification is complex

- Evaluation of an addition expression:  **includes 120+ specification steps**

Let's say you want to know: why ` !0 + 1n ` throws **TypeError**?

Syntax

*AdditiveExpression*_[Yield, Await] :

*MultiplicativeExpression*_[?Yield, ?Await]

*AdditiveExpression*_[?Yield, ?Await] + *MultiplicativeExpression*_[?Yield, ?Await]

*AdditiveExpression*_[?Yield, ?Await] - *MultiplicativeExpression*_[?Yield, ?Await]

Semantic

Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Return ? *EvaluateStringOrNumericBinaryExpression*(*AdditiveExpression*, +, *MultiplicativeExpression*).

Runtime Semantics: Evaluation

AdditiveExpression : AdditiveExpression + MultiplicativeExpression

1. Return ? EvaluateStringOrNumericBinaryExpr !0 Add + Expr 1n MultiplicativeExpression).

EvaluateStringOrNumericBinaryExpression (*leftOperand*, *opText*, *rightOperand*)

1. Let *lref* be ? Evaluation of *leftOperand*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be ? Evaluation of *rightOperand*.
4. Let *rval* be ? GetValue(*rref*).
5. Return ? ApplyStringOrNumericBinaryOperator (*lval*, *opText*, *rval*).

Evaluate Left (**lval** = Boolean true)

Evaluate Right (**rval** = BigInt 1n)

ApplyStringOrNumericBinaryOperator (*lval*, *opText*, *rval*)

- What if we can do this in a more interactive way?**
1. If *opText* is '+' then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
 2. NOTE: At this point, it must be a numeric.
 3. Let *lnum* be ? ToNumeric(*lval*).
 4. Let *rnum* be ? ToNumeric(*rval*).
 5. If Type(*lnum*) is not Type(*rnum*), throw a **TypeError** exception.
 6. If *lnum* is a BigInt, then

Convert to Primitive

(**lprim** = Boolean true, **rprim** = BigInt 1n)

Convert to Numeric

(**lnum** = Number 1, **rnum** = BigInt 1n)

...

Solution 1. Double Debugger

The screenshot shows a browser-based debugger interface for ECMAScript. The top bar includes buttons for RUN, QUIT, SPEC STEP, OVER, OUT, CONTINUE, and SPEC BACK. Below the bar are additional buttons for BACK OVER, BACK OUT, REWIND, JS STEP, AST STEP, OVER, and OUT. The main area has tabs for JavaScript Editor and ECMAScript Specification. The Specification Environment pane shows the following code and state:

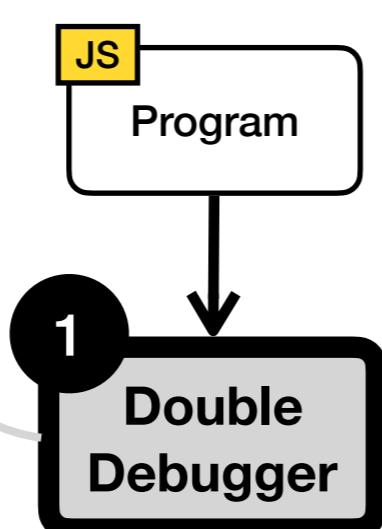
```
function add(left, right) {
  left + right;
}

add(!0, 1n);
```

ApplyStringOrNumericBinaryOperator(lval, opText, rval) &

1. If `opText` is `+`, then

- Let `lprim` be ?
ToPrimitive(`lval`).
b. Let `rprim` be ?
ToPrimitive(`rval`).
c. If `lprim` is a
String or `rprim` is a
String, then
 - Let `lstr` be ?
ToString(`lprim`). - Let `rstr` be ?
ToString(`rprim`). - iii. Return the
string-concate-
nation of `lstr`



The screenshot shows the ESMeta Double Debugger Playground interface. The top bar includes a title bar with the name of the tab, a back/forward navigation bar, a search bar with the URL "es-meta.github.io/playground/", and a toolbar with various icons for file operations and settings. A status bar at the bottom indicates "READY" and "ES2024".

The main area is divided into three main sections:

- JS. Editor**: On the left, it contains a code editor with the following JavaScript code:

```
1 function add(l, r) {  
2   l + r;  
3 }  
4  
5 add(!0, 1n);
```
- Spec. Viewer**: In the center, it displays the ECMAScript Specification for the expression `l + r;`. It shows the **Evaluation** steps:
 - Let `exprRef` be ? Evaluation of Expression.
 - Return ? GetValue(`exprRef`).
- JS. Environment**: On the right, it shows the state of the specification environment. It includes tabs for State, Env, Heap, Breaks, and Callstack. The Env tab is active, showing the following objects:
 - this**: `|ExpressionStatement|[FF]<0>`
 - JavaScript Environment**:
 - this**: `undefined`
 - l**: `Record[MutableBinding]`
• `BoundValue`: `true`
• `mutable`: `true`
• `initialized`: `true`
• `strict`: `false`
 - r**: `Record[MutableBinding]`
• `BoundValue`: `1n`
• `mutable`: `true`
• `initialized`: `true`
• `strict`: `false`
 - arguments**: `Record[ImmutableBinding]`

The screenshot shows a browser window titled "ECMAScript Double Debugger" with the URL "es-meta.github.io/playground/". The interface includes a toolbar with various icons, a menu bar, and several control buttons like RUN, QUIT, and SPEC STEP.

The main area is divided into three sections:

- JavaScript Editor**: Contains the following code:

```
1 var x = 1;
2 var y = 2;
3 var z = x + y;
4 var w = z + x;
5
6 function f () {
7   let a = 42;
8   g(a);
9   return 0;
10 }
11
12 function g(a) {
13   a = 1;
14   a = 1;
15   a = 1;
16   a = 1;
17   a = 1;
18   a = 1;
19   a = 1;
```
- ECMAScript Specification**: Displays the message "Context Not Found".
- Toolbars**: Includes State (highlighted), Env, Heap, Breaks, and Callstack.

A large yellow box highlights the "JavaScript Editor" section. A yellow banner at the bottom of the editor area contains the text "1. Edit JavaScript Program".

The screenshot shows a browser window titled "ECMAScript Double Debugger" at "es-meta.github.io/playground/". The interface is divided into two main sections by a yellow border.

Section 1: Edit JavaScript Program

JavaScript Editor:

```
1 function add(l, r) {  
2   l + r;  
3 }  
4  
5 add(!0, 1n);
```

ECMAScript Specification:

Context Not Found

Callstack:

Disabled. Start debugger to use.

Section 2: Run double debugger

Run button:

SPEC BACK | BACK OVER | BACK OUT | REWIND | JS STEP

AST STEP | OVER | OUT

State Env Heap Breaks Callstack

ECMAScript Double Debugger x +

es-meta.github.io/playground/

ESMeta / Double Debugger Playground

2. Run double debugger

SPEC BACK BACK OVER BACK OUT REWIND JS STEP

State Env Heap Breaks Callstack

No environment variables.

No environment variables.

Breakpoints

JavaScript Editor

```
1 function add(l, r) {  
2   l + r;  
3 }  
4  
5 add(!0, 1n);
```

ECMAScript Specification

RunJobs()

1. Perform ? InitializeHostDefinedRealm().
2. Let *scriptEvaluationJob* be a new Abstract Closure with no parameters that captures nothing and performs the following steps when called:
 - a. Let *sourceText* be the source code of the current Realm Record, empty).
 - c. Perform ? ScriptEvaluation(*script*).
 - d. Return **undefined**.
3. Perform HostEnqueuePromiseJob(*scriptEvaluationJob*, the current Realm Record).

Execution reached the front

The screenshot shows a browser window titled "ECMAScript Double Debugger" at "es-meta.github.io/playground/". The interface includes a toolbar with various icons, a navigation bar, and a main content area divided into sections.

JavaScript Editor: Contains the following code:

```
1 function add(l, r) {  
2   l + r;  
3 }  
4  
5 add(!0, 1n);
```

ECMAScript Specification: Shows the "RunJobs()" specification with the following steps:

1. Perform ? InitializeHostDefinedRealm().
2. Let *scriptEvaluationJob* be a new Abstract Closure with no parameters that captures nothing and performs the following steps when called:
 - a. Let *sourceText* be the source code of

Breakpoints: A table with the following columns: Step, Name, Enable, Remove. It displays the message: "No breakpoints. Add Breakpoint by clicking on steps in spec viewer or by searching name".

Section Headers:

- 2. Run double debugger** (highlighted in yellow)
- Breakpoints** (highlighted in yellow)
- 1. Edit JavaScript Program** (highlighted in yellow)

ECMAScript Double Debugger x +

es-meta.github.io/playground/

ESMeta / Double Debugger Playground

RUN QUIT SPEC STEP OVER OUT CONTINUE SPEC BACK BACK OVER BACK OUT REWIND JS STEP

AST STEP OVER OUT

Breakpoints

State Env Heap Breaks Callstack

Breakpoints

apply^s

ApplyStringOrNumericBinaryOperator

ValidateAndApplyPropertyDescriptor

PropertyDestructuringAssignmentEvaluation

AssignmentPropertyList : AssignmentPropertyList , AssignmentProperty

JavaScript Editor

```
1 function add(l, r) {  
2   l + r;  
3 }  
4  
5 add(!0, 1n);
```

ECMAScript Specification

RunJobs()

1. Perform ? InitializeHostDefinedRealm().

2. Let *scriptEvaluationJob* be a new Abstract Closure with no parameters that captures nothing and performs the following steps when called:

- Let *sourceText* be the source code of a script.
- Let *script* be ParseScript(*sourceText*, the current Realm Record, empty).
- Perform ? ScriptEvaluation(*script*).
- Return **undefined**.

3. Perform HostEnqueuePromiseJob(*scriptEvaluationJob*, the current Realm Record).

ECMAScript Double Debugger x +

es-meta.github.io/playground/ ⌂

Continue

RUN QUIT SPEC STEP OVER OUT CONTINUE SPEC BACK BACK OVER BACK OUT REWIND JS STEP AST STEP OVER OUT

ESMeta / Double Debugger Playground

JavaScript Editor

```
1 function add(l, r) {  
2   l + r;  
3 }  
4  
5 add(!0, 1n);
```

ECMAScript Specification

RunJobs()

1. Perform ? InitializeHostDefinedRealm().

2. Let *scriptEvaluationJob* be a new Abstract Closure with no parameters that captures nothing and performs the following steps when called:

- Let *sourceText* be the source code of a script.
- Let *script* be ParseScript(*sourceText*, the current Realm Record, empty).
- Perform ? ScriptEvaluation(*script*).
- Return **undefined**.

3. Perform HostEnqueuePromiseJob(*scriptEvaluationJob*, the current Realm Record).

State Env Heap Breaks Callstack

Breakpoints (Using Breakpoints)

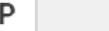
search by name

Step	Name
1	ApplyStringOrNumericBinaryOperation

ECMAScript Double Debugger x +

es-meta.github.io/playground/    

ESMeta / Double Debugger Playground      READY  ES2024

 RUN  QUIT  SPEC STEP  OVER  OUT  CONTINUE  SPEC BACK  BACK OVER  BACK OUT  REWIND  JS STEP

 AST STEP  OVER  OUT

JavaScript Editor ECMAScript Specification State Env Heap Breaks Callstack

Breakpoints  USING BREAKPOINTS

search by name

Step	Name
1	ApplyStringOrNumericBinaryOperator(lval, opText, rval)

Execution stopped at breakpoint 

```
function add(l, r) {
  l + r;
}
add(!0, 1n);
```

ApplyStringOrNumericBinaryOperator(lval, opText, rval)

- 1. If *opText* is **+**, then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
- 2. NOTE: At this point, it must be a numeric operation.
- 3. Let *lnum* be ? ToNumeric(*lval*)

ECMAScript Double Debugger x +

es-meta.github.io/playground/

ESMeta / Double Debugger Playground READY ES2024

RUN QUIT SPEC STEP OVER OUT CONTINUE SPEC BACK BACK OVER BACK OUT REWIND JS STEP

AST STEP OVER OUT

JavaScript Editor ECMAScript Specification State Env Heap Breaks Callstack

Specification Environment

- lval : true
- opText : "+"
- rval : 1n

JavaScript Environment

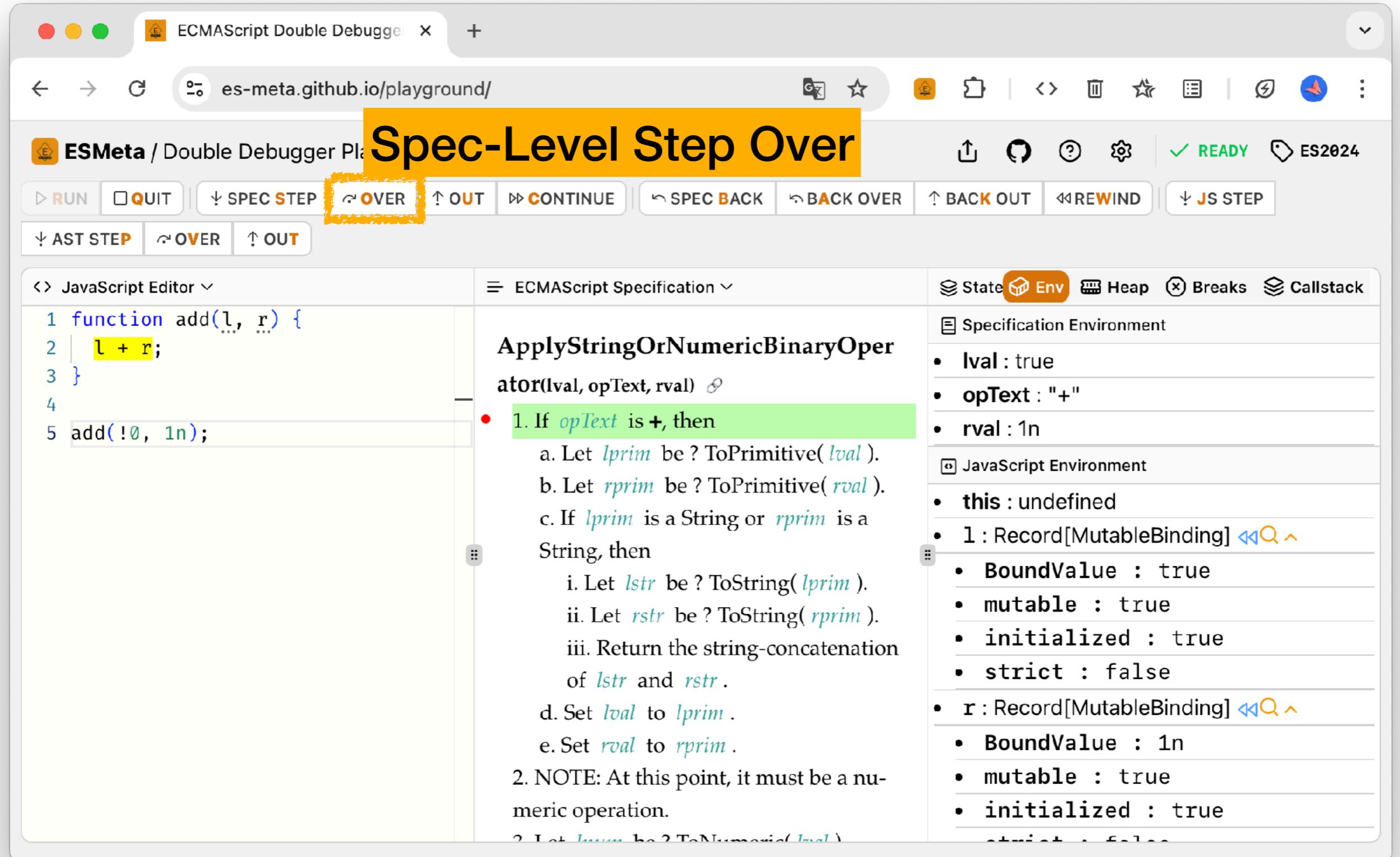
- this : undefined
- l : Record[MutableBinding]
- BoundValue : true
- mutable : true
- initialized : true
- strict : false
- r : Record[MutableBinding]
- BoundValue : 1n
- mutable : true
- initialized : true

Spec

ApplyStringOrNumericBinary Operator(lval, opText, rval)

- 1. If *opText* is `+`, then
 - a. Let *lprim* be ? ToPrimitive(*lval*)
 - b. Let *rprim* be ? ToPrimitive(*rval*)
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
- 2. NOTE: At this point, it must be a numeric operation.

2. Let *lnum* be ? ToNumeric(*lval*)



ECMAScript Double Debugger x +

es-meta.github.io/playground/           

Spec-Level Step Over

JavaScript Editor ECMAScript Specification State Env Heap Breaks Callstack

Specification Environment

- lval : true
- opText : "+"
- rval : 1n

JavaScript Environment

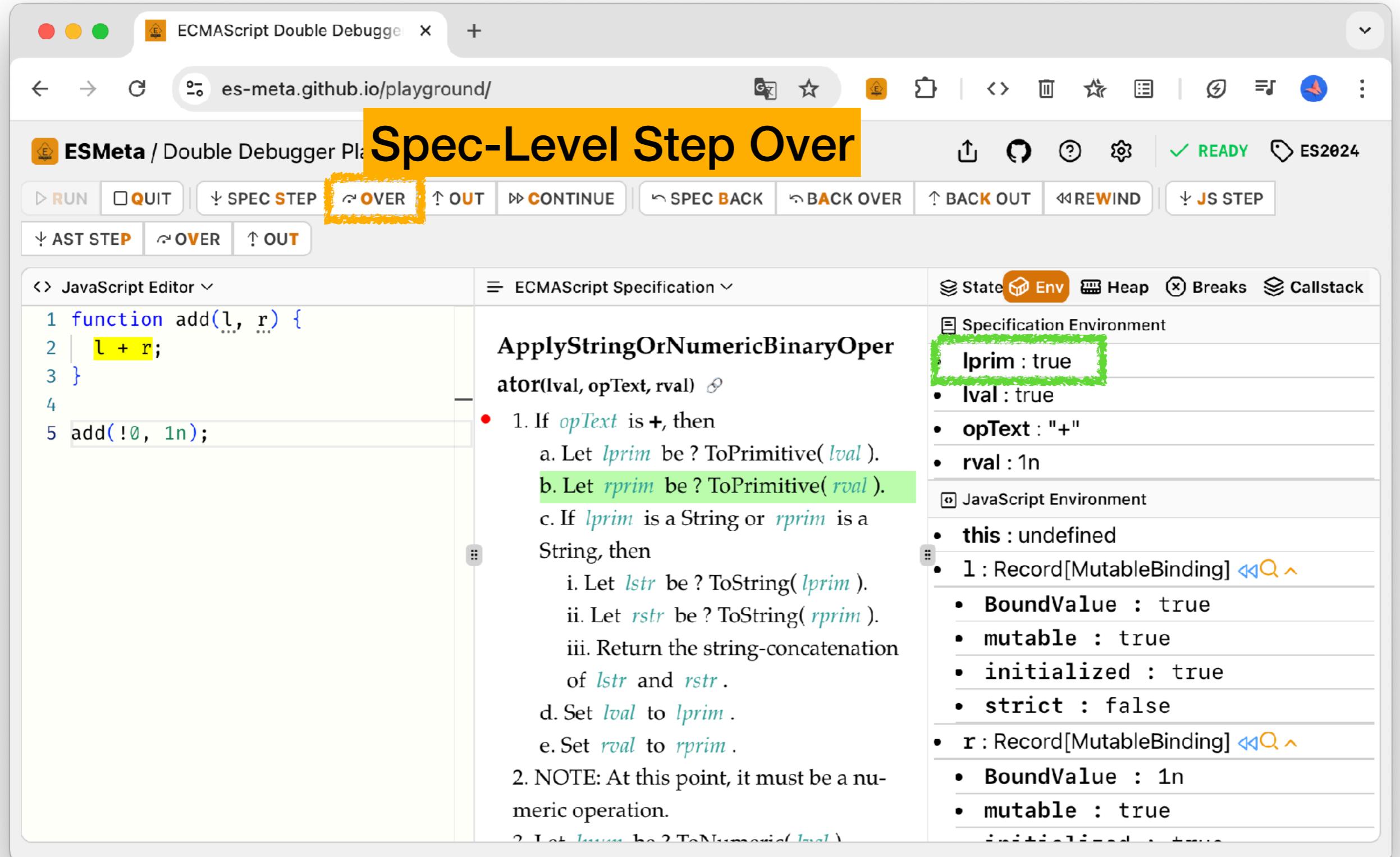
- this : undefined
- l : Record[MutableBinding]  
- BoundValue : true
- mutable : true
- initialized : true
- strict : false
- r : Record[MutableBinding]  
- BoundValue : 1n
- mutable : true
- initialized : true

1 function add(l, r) {
2 | l + r;
3 }
4
5 add(!0, 1n);

ApplyStringOrNumericBinaryOperator(lval, opText, rval)

- 1. If *opText* is **+**, then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
- 2. NOTE: At this point, it must be a numeric operation.

3. Let *lnum* be ? ToNumeric(*lval*)



ECMAScript Double Debugger x +

es-meta.github.io/playground/ ESMeta / Double Debugger Plat Spec-Level Step Over READY ES2024

RUN QUIT SPEC STEP ⌘ OVER ⌘ OUT CONTINUE SPEC BACK ⌘ BACK OVER ⌘ BACK OUT ⌘ REWIND JS STEP

AST STEP ⌘ OVER ⌘ OUT

JavaScript Editor ECMAScript Specification State Env Heap Breaks Callstack

Specification Environment

- lprim : true
- lval : true
- opText : "+"
- rprim : 1n
- rval : 1n

JavaScript Environment

- this : undefined
- l : Record[MutableBinding] ↪ ↵
 - BoundValue : true
 - mutable : true
 - initialized : true
 - strict : false
- r : Record[MutableBinding] ↪ ↵
 - BoundValue : 1n

1 function add(l, r) {
2 | l + r;
3 }
4
5 add(!0, 1n);

ApplyStringOrNumericBinaryOperator(lval, opText, rval)

- 1. If *opText* is **+**, then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
- 2. NOTE: At this point, it must be a numeric operation.

Let *lnum* be ? ToNumeric(*lval*)

ECMAScript Double Debugger x +

es-meta.github.io/playground/

Spec-Level Step Over

(highlighted with a yellow box)

JavaScript Editor ▾ ECMAScript Specification ▾ State Env Heap Breaks Callstack

Specification Environment

- lprim : true
- lval : true
- opText : "+" (highlighted with a green box)
- rprim : 1n
- rval : 1n

JavaScript Environment

- this : undefined
- l : Record[MutableBinding] ↪
 - BoundValue : true
 - mutable : true
 - initialized : true
 - strict : false
- r : Record[MutableBinding] ↪
 - BoundValue : 1n

1 function add(l, r) {
2 | l + r;
3 }
4
5 add(!0, 1n);

ApplyStringOrNumericBinaryOperator(lval, opText, rval) ↪

- 1. If *opText* is **+**, then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
- 2. NOTE: At this point, it must be a numeric operation.
- 3. Let *lnum* be ? ToNumeric(*lval*)

ECMAScript Double Debugger x +

es-meta.github.io/playground/         

Spec-Level Step Over

RUN QUIT SPEC STEP ↵ OVER ↑ OUT CONTINUE SPEC BACK BACK OVER BACK OUT REWIND JS STEP AST STEP ↵ OVER ↑ OUT

JavaScript Editor ECMAScript Specification State Env Heap Breaks Callstack

Specification Environment

- lprim : true
- lval : true
- opText : "+"
- rprim : 1n
- rval : 1n

JavaScript Environment

- this : undefined
- l : Record[MutableBinding] ↪
 - BoundValue : true
 - mutable : true
 - initialized : true
 - strict : false
- r : Record[MutableBinding] ↪
 - BoundValue : 1n

function add(l, r) {
 l + r;
}
add(!0, 1n);

ApplyStringOrNumericBinaryOperator(lval, opText, rval)

- 1. If *opText* is **+**, then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
- 2. NOTE: At this point, it must be a numeric operation.

Let *lnum* be ? ToNumeric(*lval*)

ECMAScript Double Debugger x +

es-meta.github.io/playground/        

Spec-Level Step Over

JavaScript Editor ▾ ECMAScript Specification ▾ State Env Heap Breaks Callstack

Specification Environment

- lprim : true
- lval : true
- opText : "+"
- rprim : 1n
- rval : 1n

JavaScript Environment

- this : undefined
- l : Record[MutableBinding] ↪ Q ↵
 - BoundValue : true
 - mutable : true
 - initialized : true
 - strict : false
- r : Record[MutableBinding] ↪ Q ↵
 - BoundValue : 1n

1 function add(l, r) {
2 | l + r;
3 }
4
5 add(!0, 1n);

ApplyStringOrNumericBinaryOperator(lval, opText, rval) ↪

- 1. If *opText* is **+**, then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
- 2. NOTE: At this point, it must be a numeric operation.

2 Let *lnum* be ? ToNumeric(*lval*)

The screenshot shows a browser-based debugger interface for ECMAScript. The title bar reads "ECMAScript Double Debugger" and the address bar shows "es-meta.github.io/playground/". The main title "Spec-Level Step Over" is displayed prominently in a yellow box. The toolbar includes buttons for RUN, QUIT, SPEC STEP, OVER (highlighted with a yellow box), OUT, CONTINUE, SPEC BACK, BACK OVER, BACK OUT, REWIND, and JS STEP. Below the toolbar are AST STEP, OVER, and OUT buttons. The left panel is a "JavaScript Editor" showing the code:

```
1 function add(l, r) {  
2     l + r;  
3 }  
4  
5 add(10, 1n);
```

The middle panel displays the "ECMAScript Specification" for the `StringConcatenationAlgorithm`. It details the steps for concatenating two values, including handling numeric and string inputs and setting up environment records. The current step, "Let lnum be ? ToNumeric(lval).", is highlighted with a green box.

The right panel shows the "Specification Environment" and "JavaScript Environment" with their respective variable states. The "lprim : true" entry in the specification environment and the "rprim : 1n" entry in the JavaScript environment are both highlighted with green boxes.

ECMAScript Double Debugger x +

es-meta.github.io/playground/         

Spec-Level Step Over

JavaScript Editor ▾

```
1 function add(l, r) {  
2   l + r;  
3 }  
4  
5 add(!0, 1n);
```

ECMAScript Specification ▾

ator(lval, opText, rval) ↗

- 1. If *opText* is **+**, then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
- 2. NOTE: At this point, it must be a numeric operation.
- 3. Let *lnum* be ? ToNumeric(*lval*).
- 4. Let *rnum* be ? ToNumeric(*rval*).

State Env Heap Breaks Callstack

Specification Environment

- *lnum* : 1
- *lprim* : true
- *lval* : true
- *opText* : "+"
- *rprim* : 1n
- *rval* : 1n

JavaScript Environment

- *this* : undefined
- *l* : Record[MutableBinding] ↪ ↩ ↩
- *BoundValue* : true
- *mutable* : true
- *initialized* : true
- *strict* : false
- *r* : Record[MutableBinding] ↪ ↩ ↩

ECMAScript Double Debugger x +

es-meta.github.io/playground/

Spec-Level Step Over

(highlighted with a yellow box)

JavaScript Editor ▾ ECMAScript Specification ▾ State Env Heap Breaks Callstack

Specification Environment

- lnum : 1
- lprim : true
- lval : true
- opText : "+"
- rnum : 1n
- rprim : 1n
- rval : 1n

JavaScript Environment

- this : undefined
- l : Record[MutableBinding] ↪ ↩ ↩
 - BoundValue : true
 - mutable : true
 - initialized : true
 - strict : false

ECMAScript Specification

```
function add(l, r) {
  l + r;
}
add(!0, 1n);
```

b. Let *rprim* be ? ToPrimitive(*rval*).
c. If *lprim* is a String or *rprim* is a String, then

- i. Let *lstr* be ? ToString(*lprim*).
- ii. Let *rstr* be ? ToString(*rprim*).
- iii. Return the string-concatenation of *lstr* and *rstr*.

d. Set *lval* to *lprim*.
e. Set *rval* to *rprim*.
2. NOTE: At this point, it must be a numeric operation.
3. Let *lnum* be ? ToNumeric(*lval*).
4. Let *rnum* be ? ToNumeric(*rval*).
5. If Type(*lnum*) is not Type(*rnum*), throw a **TypeError** exception.
6. If *lnum* is a BigInt, then

ECMAScript Double Debugger x +

es-meta.github.io/playground/

Spec-Level Step Over

(highlighted with a yellow box)

JavaScript Editor ▾ ECMAScript Specification ▾ State Env Heap Breaks Callstack

Specification Environment

- RETURN : Record[CompletionRecord]
- lnum : 1
- lprim : true
- lval : true
- opText : "+"
- rnum : 1n
- rprim : 1n
- rval : 1n

JavaScript Environment

- this : undefined
- l : Record[MutableBinding]
 - BoundValue : true
 - mutable : true
 - initialized : true

JavaScript Editor:

```
1 function add(l, r) {  
2   l + r;  
3 }  
4  
5 add(!0, 1n);
```

ECMAScript Specification:

- Let *rprim* be ? ToPrimitive(*rval*).
- If *lprim* is a String or *rprim* is a String, then
 - Let *lstr* be ? ToString(*lprim*).
 - Let *rstr* be ? ToString(*rprim*).
 - Return the string-concatenation of *lstr* and *rstr*.
- Set *lval* to *lprim*.
- Set *rval* to *rprim*.

NOTE: At this point, it must be a numeric operation.

- Let *lnum* be ? ToNumeric(*lval*).
- Let *rnum* be ? ToNumeric(*rval*).
- If Type(*lnum*) is not Type(*rnum*), throw a **TypeError** exception.
- If *lnum* is a BigInt, then

The screenshot shows the ESMeta Double Debugger Playground interface. The top bar includes tabs for "ECMAScript Double Debugge" and "es-meta.github.io/playground/". The main area has tabs for "JavaScript Editor", "ECMAScript Specification", and "Specification Environment".

JavaScript Editor:

```
function add(l, r) {  
    l + r;  
}  
add(!0, 1n);
```

ECMAScript Specification:

1. Let *lprim* be ? ToPrimitive(*rval*).
2. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
ii. Let *rstr* be ? ToString(*rprim*).
iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
2. NOTE: At this point, it must be a numeric operation.
3. Let *lnum* be ? ToNumeric(*lval*).
4. Let *rnum* be ? ToNumeric(*rval*).
5. If Type(*lnum*) is not Type(*rnum*), throw a **TypeError** exception.
6. If *lnum* is a BigInt, then

Specification Environment:

- **RETURN** : Record[CompletionRecord]
- **Target** : ~empty~
- **Type** : ~throw~
- **Value** : Record[Object]
- **lnum** : 1
- **lprim** : true
- **lval** : true
- **opText** : "+"
- **rnum** : 1n
- **rprim** : 1n
- **rval** : 1n

JavaScript Environment:

- **this** : undefined

ESMeta / Double Debugger Playground

JavaScript Editor

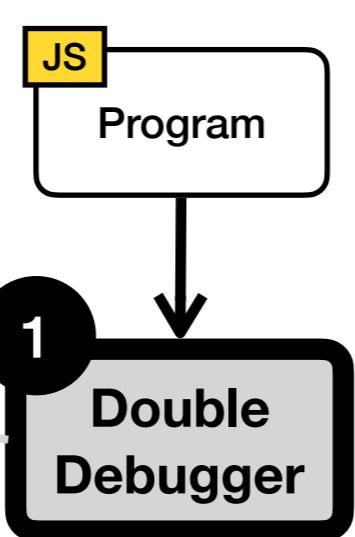
```
function add(left, right) {
  left + right;
}
add(1, 1);
```

ECMAScript Specification

ApplyStringOrNumericBinaryOperator(lval, opText, rval)

1. If `opText` is `+`, then

- Let `lprim` be ?
ToPrimitive(`lval`).
b. Let `rprim` be ?
ToPrimitive(`rval`).
c. If `lprim` is a
String or `rprim` is a
String, then
 - Let `lstr` be ?
ToString(`lprim`). - Let `rstr` be ?
ToString(`rprim`). - iii. Return the
string-concate-
nation of `lstr`



Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Return ? EvaluateStringOrNumericBinaryExpression(*AdditiveExpression*, +, *MultiplicativeExpression*).

EvaluateStringOrNumericBinaryExpression (*leftOperand*, *opText*, *rightOperand*)

1. Let *lref* be ? Evaluation of *leftOperand*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be ? Evaluation of *rightOperand*.
4. Let *rval* be ? GetValue(*rref*).
5. Return ? ApplyStringOrNumericBinaryOperator(*lval*, *opText*, *rval*).

ApplyStringOrNumericBinaryOperator (*lval*, *opText*, *rval*)

1. If *opText* is +, then
a. Let *lprim* be ? ToPrimitive(*lval*).
b. Let *rprim* be ? ToPrimitive(*rval*).
c. If *lprim* is a String or *rprim* is a String, then
 i. Let *lstr* be ? ToString(*lprim*).
 ii. Let *rstr* be ? ToString(*rprim*).
 iii. Return the string-concatenation of *lstr* and *rstr*.
d. Set *lval* to *lprim*.
e. Set *rval* to *rprim*.
syntactic sugar of:
 return { [[Type]] : throw, [[Value]] : ... , ... }
2. NOTE: At this point, it must be a number.
3. Let *lnum* be ? ToNumeric(*lval*).
4. Let *rnum* be ? ToNumeric(*rval*).
5. If Type(*lnum*) is not Type(*rnum*), throw a TypeError exception.
6. If *lnum* is a BigInt, then

...

Runtime Semantics: Evaluation

AdditiveExpression : AdditiveExpression + MultiplicativeExpression

1. Return $\text{EvaluateStringOrNumericBinaryExpression}(\text{AdditiveExpression}, +, \text{MultiplicativeExpression})$.

What if the spec came with example JavaScript programs?

EvaluateStringOrNumericBinaryExpression (leftOperand, opText, rightOperand)

1. Let $lref$ be $\text{Evaluation of leftOperand}$.
2. Let $lval$ be $\text{GetValue}(lref)$.
3. Let $rref$ be $\text{Evaluation of rightOperand}$.
4. Let $rval$ be $\text{GetValue}(rref)$.
5. Return $\text{ApplyStringOrNumericBinaryOperator}(lval, opText, rval)$.

ApplyStringOrNumericBinaryOperator (lval, opText, rval)

1. If $opText$ is $+$,
a. Let $lprim$ be $\text{ToPrimitive}(lval)$.
b. Let $rprim$ be $\text{ToPrimitive}(rval)$.
c. If $lprim$ is a String or $rprim$ is a String, then
i. Let $lstr$ be $\text{ToString}(lprim)$.
ii. Let $rstr$ be $\text{ToString}(rprim)$.
iii. Return the string-concatenation of $lstr$ and $rstr$.
d. Set $lval$ to $lprim$.
e. Set $rval$ to $rprim$.
2. NOTE: At this point, it must be a numeric operation.
3. Let $lnum$ be $\text{ToNumeric}(lval)$.
4. Let $rnum$ be $\text{ToNumeric}(rval)$.
5. If $\text{Type}(lnum)$ is not $\text{Type}(rnum)$, throw a **TypeError** exception.
6. If $lnum$ is a **BigInt**, then

...

1 + 1

$(\{ \text{Symbol.toPrimitive} : 0 \}) + 1$

$1 + (\{ \text{Symbol.toPrimitive} : 0 \})$

[] + 0

$\{ \text{Symbol.toPrimitive} : 0 \} * 0$

$0 * \{ \text{Symbol.toPrimitive} : 0 \}$

0 + 1n

Solution 2. Program Visualizer

ESMeta / Double Debugger Playground

JavaScript Editor

```
function add(left, right) {
    left + right;
}

add(10, 1n);
```

ECMAScript Specification

- **Specification Environment**
 - **lval** : true
 - **opText** : "+"
 - **rval** : 1n
- **JavaScript Environment**
 - **this** : undefined
 - **left** : Record[MutableBinding]
 - **BoundValue** : true
 - **mutable** : true
 - **initialized** : true
 - **strict** : false
 - **right** : Record[MutableBinding]
 - **BoundValue** : 1n
 - **mutable** : true

Program Collector

Test

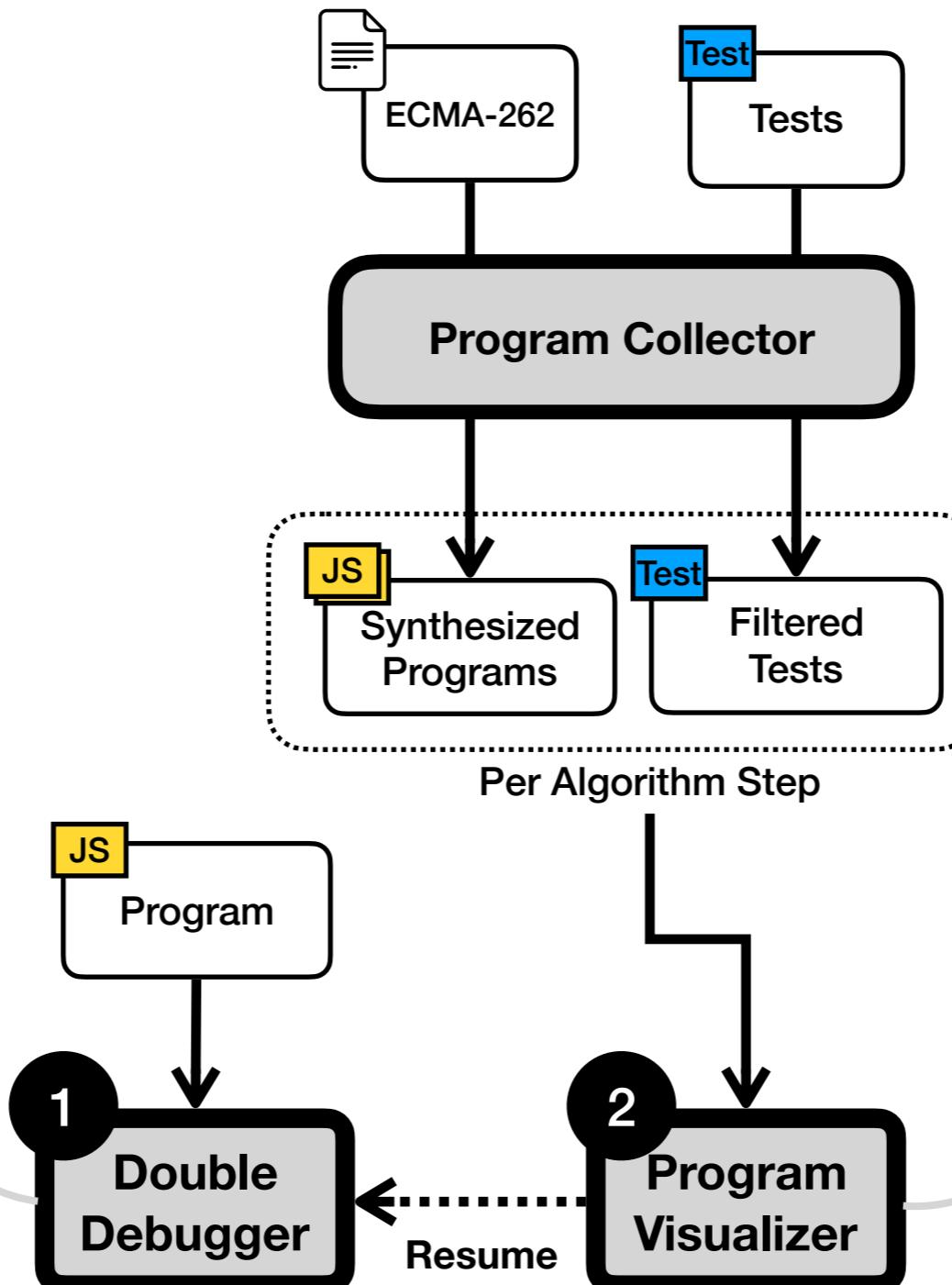
Program Visualizer

Double Debugger

```

graph TD
    ECMA[ECMA-262] --> PC[Program Collector]
    Tests[Tests] --> PC
    PC --> SP[Synthesized Programs]
    PC --> TT[Filtered Tests]
    SP --- PA[Per Algorithm Step]
    TT --- PA
    subgraph PA [Per Algorithm Step]
        direction TB
        P[Program] --> DD[1 Double Debugger]
        P --> PV[2 Program Visualizer]
        DD -.-> PV
        PV -.-> DD
    end

```



5.3 ApplyStringOrNumericBinaryOperator (*lval*, *opText*, *rval*)

The abstract operation **ApplyStringOrNumericBinaryOperator** takes arguments *lval* (an ECMAScript language value), *opText* (**, *, /, %, +, -, <<, >>, >>>, &, ^, or |), and *rval* (an ECMAScript language value) and returns either a **normal completion** containing either a String, a BigInt, or a Number, or a **throw completion**. It performs the following steps when called:

1. If *opText* is +, then
 - a. Let *lprim* be ? **ToPrimitive**(*lval*).
b. Let *rprim* be ? **ToPrimitive**(*rval*).
c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? **ToString**(*lprim*).
ii. Let *rstr* be ? **ToString**(*rprim*).
iii. Return the string-concatenation of *lstr*





13.15.3 ApplyStringOrNumberBinaryOperator (*lval*, *opText*, *rval*)

The abstract operation `ApplyStringOrNumberBinaryOperator` takes arguments *lval* (an ECMAScript language value), *opText* (one of the binary operators `**`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, or `|`), and *rval* (an ECMAScript language value) and returns either a `normal completion` containing either a String, a BigInt, or a Number, or a `throw completion`. It performs the following steps when called:

1. If *opText* is `+`, then
 - a. Let *lprim* be `? ToPrimitive(lval)`.
 - b. Let *rprim* be `? ToPrimitive(rval)`.
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be `? ToString(lprim)`.
 - ii. Let *rstr* be `? ToString(rprim)`.
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
2. NOTE: At this point, it must be a numeric operation.
3. Let *lnum* be `? ToNumeric(lval)`.
4. Let *rnum* be `? ToNumeric(rval)`.

The abstract operation `ApplyStringOrNumericBinaryOperator` takes arguments `lval` (an ECMAScript language value), `opText` (`**`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, or `|`), and `rval` (an ECMAScript language value) and returns either a `normal completion containing` either a String, a BigInt, or a Number, or a `throw completion`. It performs the following steps when called:

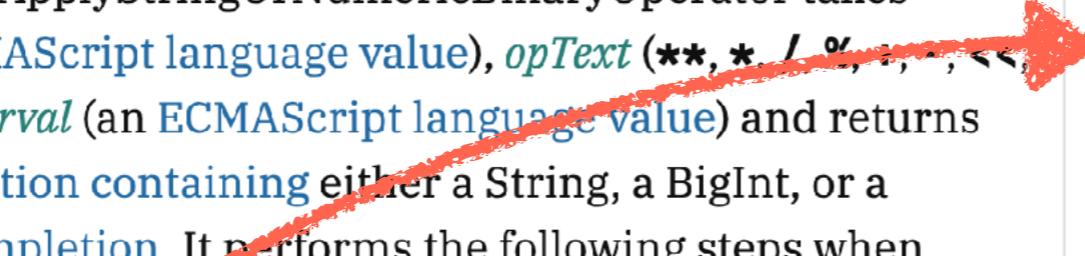
1. If `opText` is `+`, then
 - a. Let `lprim` be `? ToPrimitive(lval)`.
 - b. Let `rprim` be `? ToPrimitive(rval)`.
 - c. If `lprim` is a String or `rprim` is a String, then
 - i. Let `lstr` be `? ToString(lprim)`.
 - ii. Let `rstr` be `? ToString(rprim)`.
 - iii. Return the string-concatenation of `lstr` and `rstr`.
 - d. Set `lval` to `lprim`.
 - e. Set `rval` to `rprim`.
2. NOTE: At this point, it must be a numeric operation.
3. Let `lnum` be `? ToNumeric(lval)`.

The screenshot shows a browser window with the URL `tc39.es/ecma262/2024/multipage/ecmascript-lan...`. A yellow box highlights the right-hand sidebar of the browser, which contains three tabs:

- ECMA Visualizer**: Shows a placeholder message: "Start by pressing a production with Option (Alt) + Left Click".
- ESMeta ECMA Visualizer**: Shows a placeholder message: "Start by pressing a production with Option (Alt) + Left Click".
- CallStack**: Shows a placeholder message: "Start by pressing a link with Left Click".

The abstract operation `ApplyStringOrNumericBinaryOperator` takes arguments `lval` (an ECMAScript language value), `opText` (`**`, `*`, `/`, `%`, `+`, `-`, `<`, `<<`, `>`, `>>`, `&`, `^`, or `|`), and `rval` (an ECMAScript language value) and returns either a `normal completion` containing either a String, a `BigInt`, or a `Number`, or a `throw completion`. It performs the following steps when called:

1. If `opText` is `+`, then
 - a. Let `lprim` be `? ToString(lval)`.
 - b. Let `rprim` be `? ToString(rval)`.
 - c. If `lprim` is a String and `rprim` is a String, then
 - i. Let `lstr` be `? ToString(lprim)`.
 - ii. Let `rstr` be `? ToString(rprim)`.
 - iii. Return the string-concatenation of `lstr` and `rstr`.
 - d. Set `lval` to `lprim`.
 - e. Set `rval` to `rprim`.
2. NOTE: At this point, it must be a numeric operation.
3. Let `lnum` be `? ToNumeric(lval)`.



Option (alt) + Click

ECMA Visualizer

ESMeta ECMA Visualizer

Program Run on Double Debugger

1 1 + 1 ;

Test262 8660 found Download All

built-ins/Array/S15.4_A1.1_T10.js

built-ins/Array/from/calling-from-valid-1-onlyStrict.js

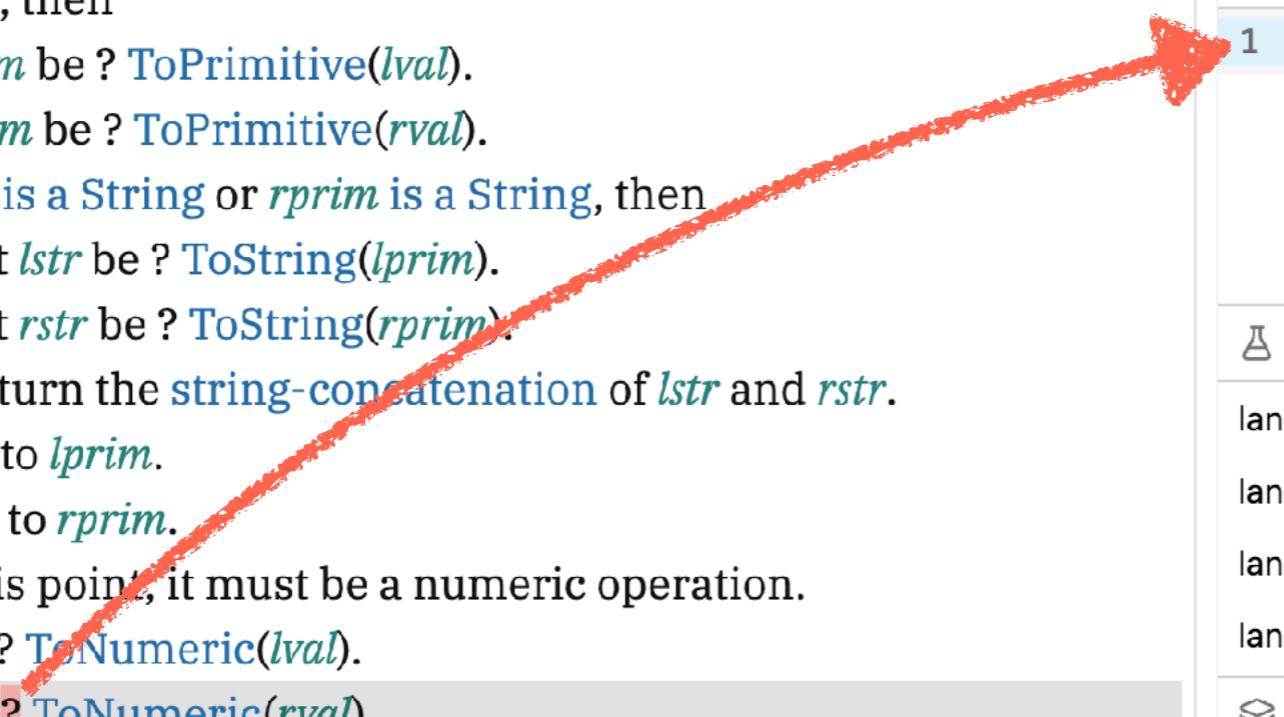
built-ins/Array/from/calling-from-valid-2.js

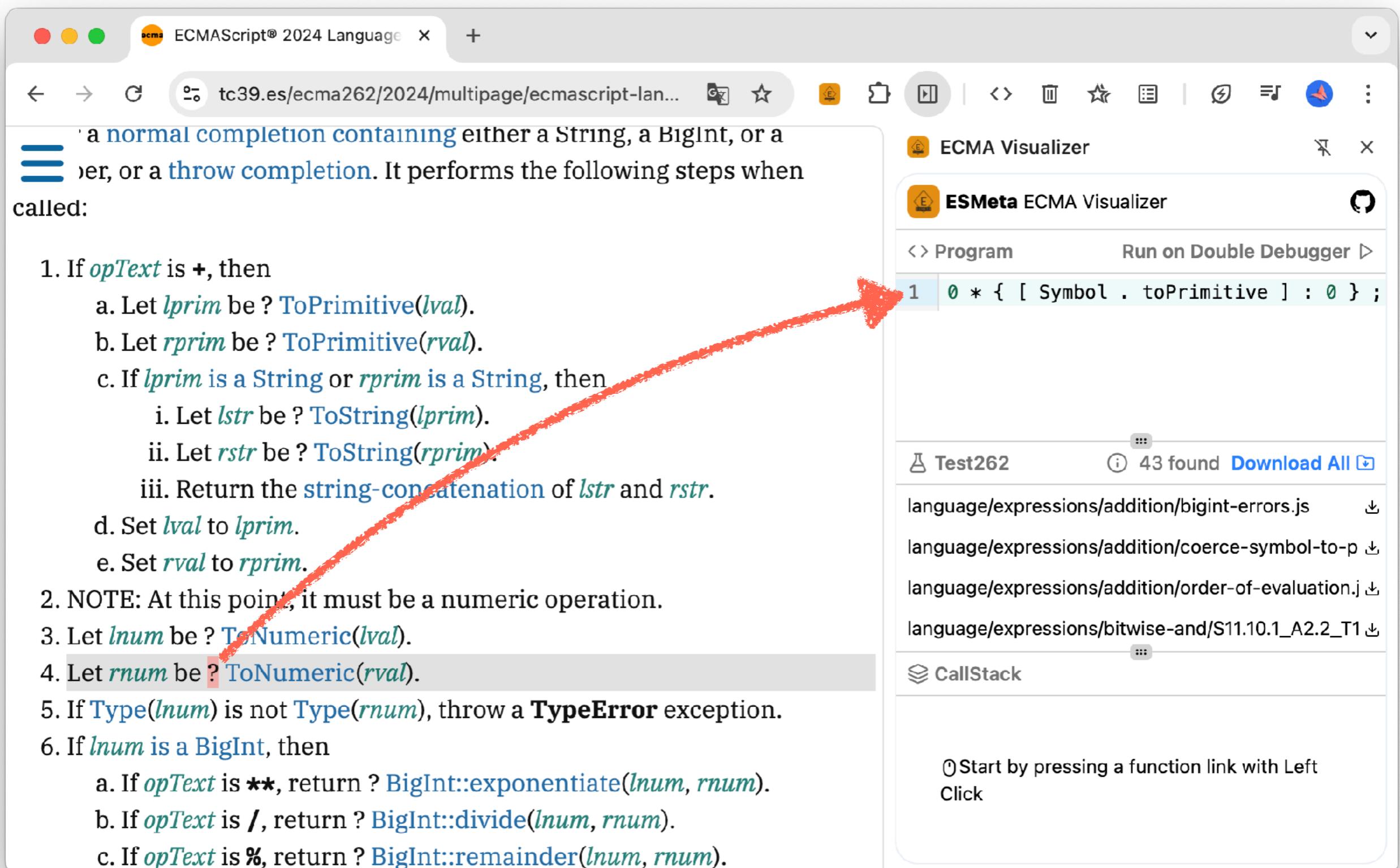
built-ins/Array/from/elements-added-after.js

CallStack

Start by pressing a function link with Left Click

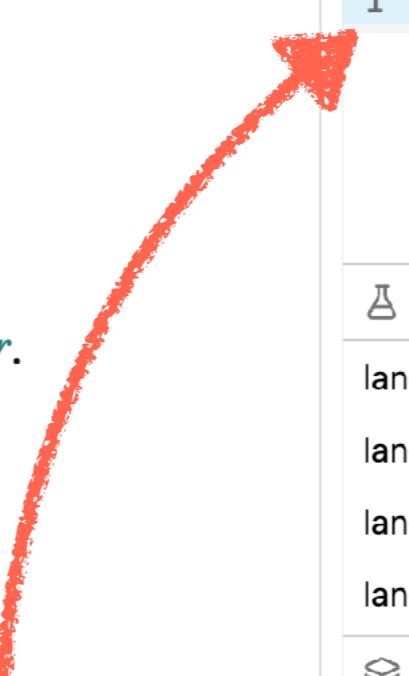
a normal completion containing either a String, a BigInt, or a Number, or a throw completion. It performs the following steps when called:

1. If *opText* is **+**, then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
2. NOTE: At this point, it must be a numeric operation.
3. Let *lnum* be ? ToNumeric(*lval*).
4. Let *rnum* be ? ToNumeric(*rval*). 
5. If Type(*lnum*) is not Type(*rnum*), throw a **TypeError** exception.
6. If *lnum* is a BigInt, then
 - a. If *opText* is ******, return ? BigInt::exponentiate(*lnum*, *rnum*).
 - b. If *opText* is **/**, return ? BigInt::divide(*lnum*, *rnum*).
 - c. If *opText* is **%**, return ? BigInt::remainder(*lnum*, *rnum*).



a normal completion containing either a String, a BigInt, or a Number, or a throw completion. It performs the following steps when called:

1. If *opText* is **+**, then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
2. NOTE: At this point, it must be a numeric operation.
3. Let *lnum* be ? ToNumeric(*lval*).
4. Let *rnum* be ? ToNumeric(*rval*).
5. If Type(*lnum*) is not Type(*rnum*), throw a **TypeError** exception.
6. If *lnum* is a BigInt, then
 - a. If *opText* is ******, return ? BigInt::exponentiate(*lnum*, *rnum*).
 - b. If *opText* is **/**, return ? BigInt::divide(*lnum*, *rnum*).
 - c. If *opText* is **%**, return ? BigInt::remainder(*lnum*, *rnum*).



The screenshot shows the ECMAScript 2024 Language browser extension. The main window displays the ECMA Visualizer interface, which includes a 'Program' tab containing the expression '1 + 0n ;'. Below the visualizer are several test links: 'language/expressions/addition/bigint-and-number.js', 'language/expressions/bitwise-and/bigint-and-number.js', 'language/expressions/bitwise-or/bigint-and-number.js', and 'language/expressions/bitwise-xor/bigint-and-number.js'. A call stack section is also visible. A red arrow has been drawn from the explanatory text in the main window to the 'Program' tab of the visualizer.

ECMAScript® 2024 Language

tc39.es/ecma262/2024/multipage/ecmascript-lan...

a normal completion containing either a String, a BigInt, or a Number, or a throw completion. It performs the following steps when called:

1. If *opText* is **+**, then
 - a. Let *lprim* be ? ToPrimitive(*lval*).
 - b. Let *rprim* be ? ToPrimitive(*rval*).
 - c. If *lprim* is a String or *rprim* is a String, then
 - i. Let *lstr* be ? ToString(*lprim*).
 - ii. Let *rstr* be ? ToString(*rprim*).
 - iii. Return the string-concatenation of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
2. NOTE: At this point, it must be a numeric operation.
3. Let *lnum* be ? ToNumeric(*lval*).
4. Let *rnum* be ? ToNumeric(*rval*).
5. If Type(*lnum*) is not Type(*rnum*), throw a **TypeError** exception.
6. If *lnum* is a BigInt, then
 - a. If *opText* is ******, return ? BigInt::exponentiate(*lnum*, *rnum*).
 - b. If *opText* is **/**, return ? BigInt::divide(*lnum*, *rnum*).
 - c. If *opText* is **%**, return ? BigInt::remainder(*lnum*, *rnum*).

ECMA Visualizer

ESMeta ECMA Visualizer

Program

Run on Double Debugger

Resume

Test262

language/expressions/addition/bigint-and-number.js

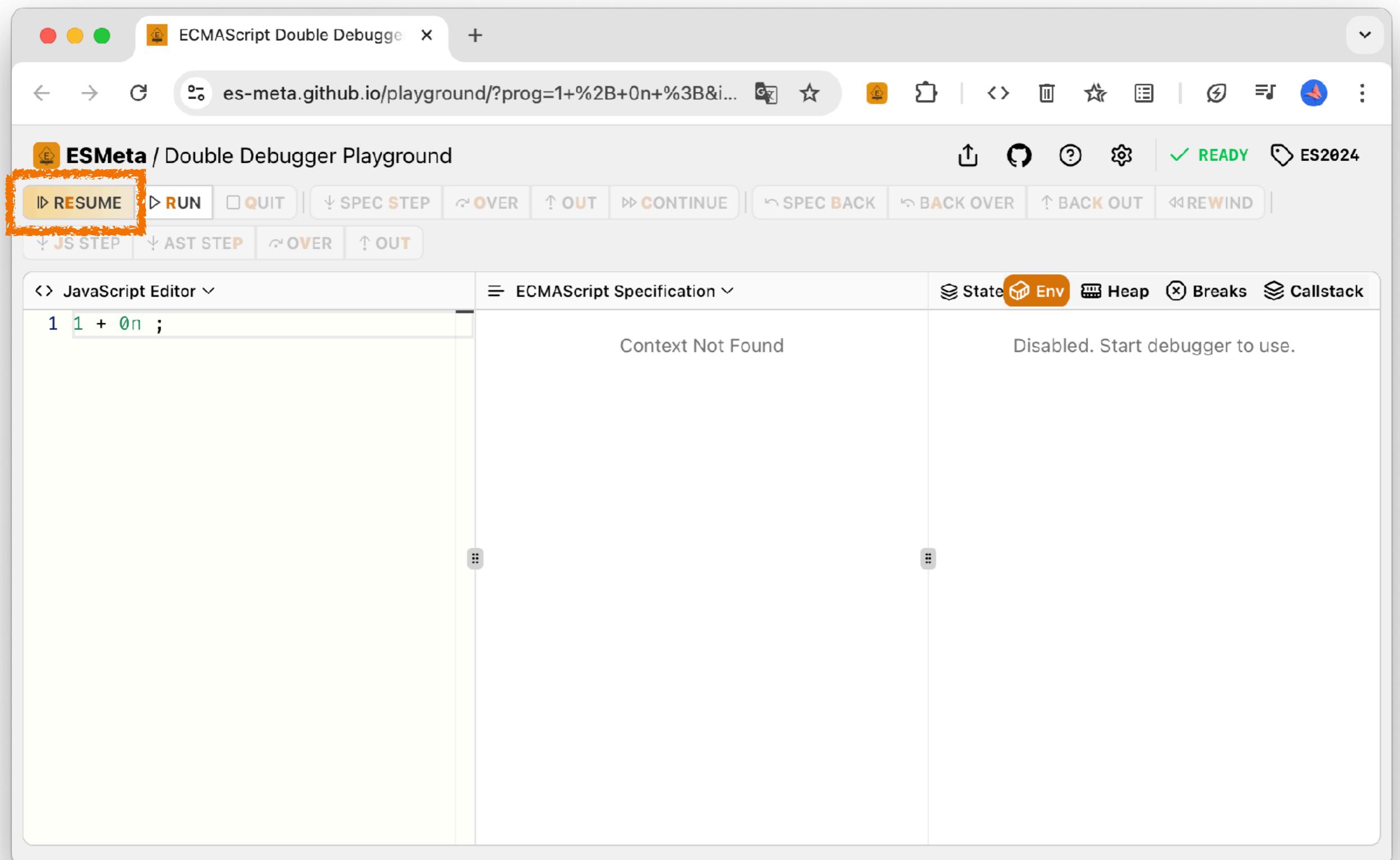
language/expressions/bitwise-and/bigint-and-number.js

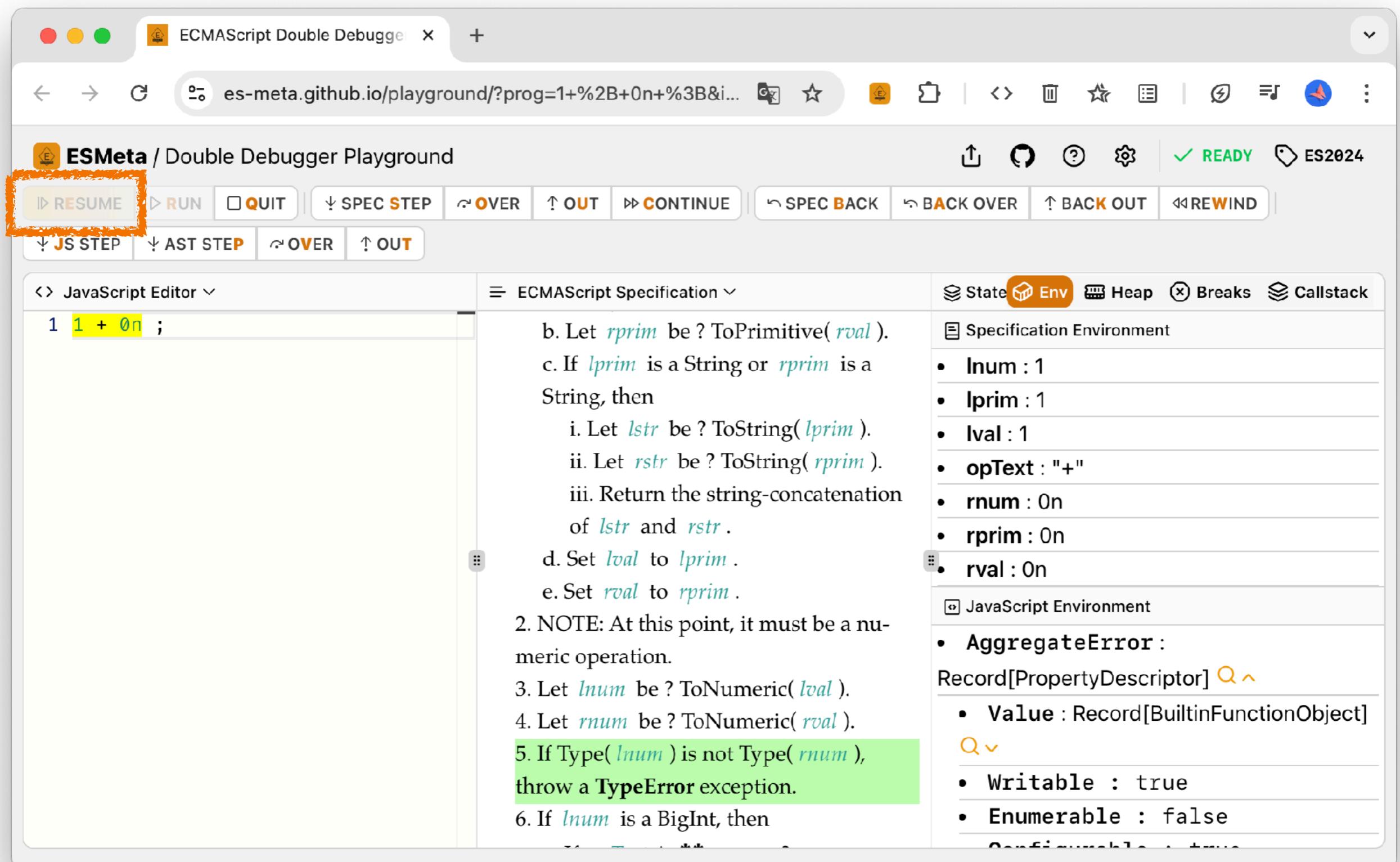
language/expressions/bitwise-or/bigint-and-number.js

language/expressions/bitwise-xor/bigint-and-number.js

CallStack

Start by pressing a function link with Left Click





ESMeta / Double Debugger Playground

JavaScript Editor

```
function add(left, right) {
    left + right;
}

add(10, 1n);
```

ECMAScript Specification

Program Collector

Specification Environment

- lval : true
- opText : "+"
- rval : 1n

JavaScript Environment

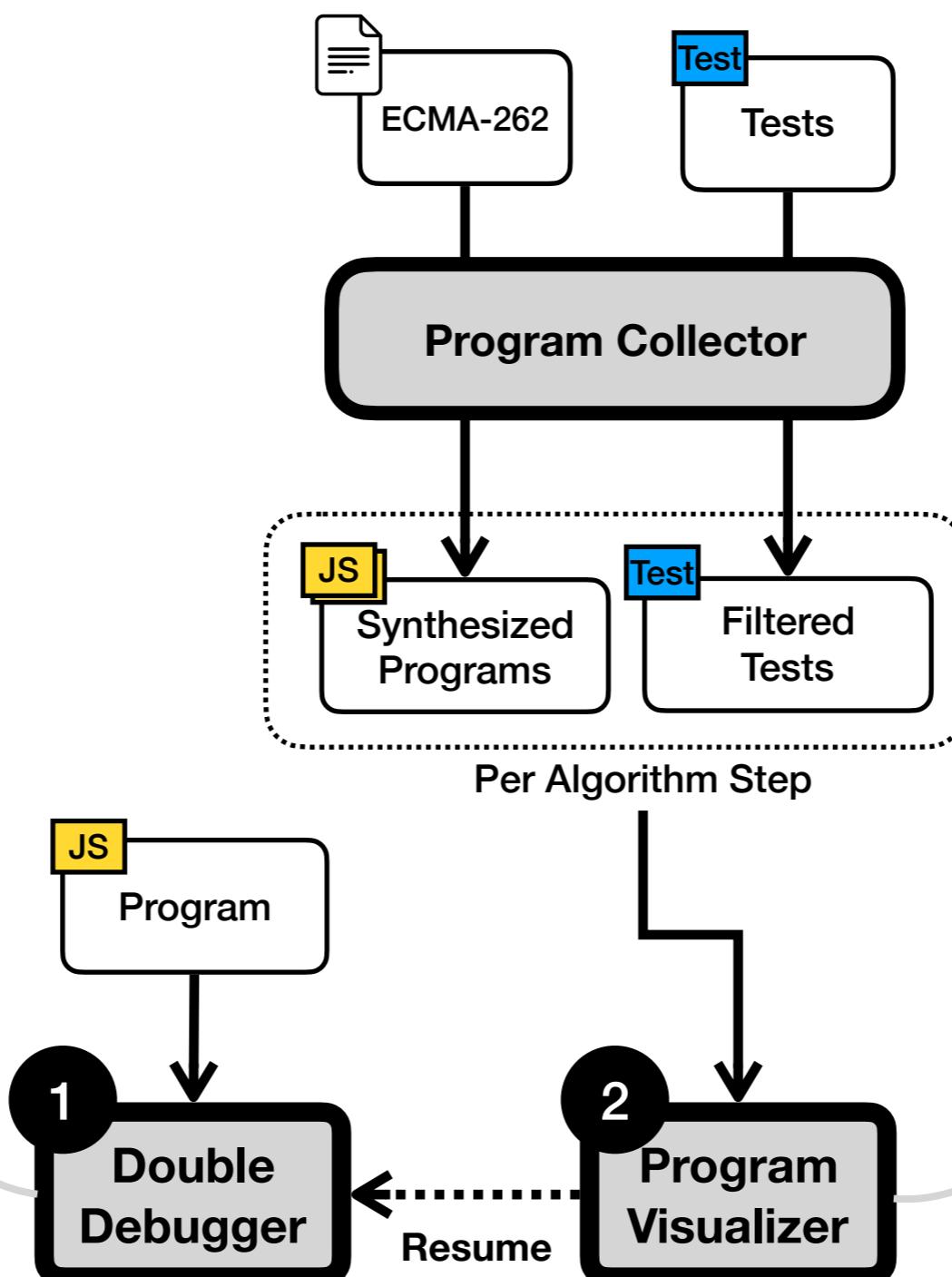
- this : undefined
- left : Record[MutableBinding]
 - BoundValue : true
 - mutable : true
 - initialized : true
 - strict : false
 - right : Record[MutableBinding]
 - BoundValue : 1n
 - mutable : true

Synthesized Programs

Filtered Tests

The abstract operation `ApplyStringOrNumericBinaryOperator(lval, opText, rval)` takes arguments `lval` (an ECMAScript language value), `opText` (`**`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `^`, or `|`), and `rval` (an ECMAScript language value) and returns either a normal completion containing either a String, a BigInt, or a Number, or a throw completion. It performs the following steps when called:

1. If `opText` is `+`, then
 - a. Let `lprim` be ?`ToPrimitive(lval)`.
 - b. Let `rprim` be ?`ToPrimitive(rval)`.
 - c. If `lprim` is a String or `rprim` is a String, then
 - i. Let `lstr` be ?`ToString(lprim)`.
 - ii. Let `rstr` be ?`ToString(rprim)`.
 - iii. Return the string-concatenation of `lstr`



5.3 ApplyStringOrNumericBinaryOperator (`lval, opText, rval`)

The abstract operation `ApplyStringOrNumericBinaryOperator` takes arguments `lval` (an ECMAScript language value), `opText` (`**`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `^`, or `|`), and `rval` (an ECMAScript language value) and returns either a normal completion containing either a String, a BigInt, or a Number, or a throw completion. It performs the following steps when called:

1. If `opText` is `+`, then
 - a. Let `lprim` be ?`ToPrimitive(lval)`.
 - b. Let `rprim` be ?`ToPrimitive(rval)`.
 - c. If `lprim` is a String or `rprim` is a String, then
 - i. Let `lstr` be ?`ToString(lprim)`.
 - ii. Let `rstr` be ?`ToString(rprim)`.

