Filling the Gap Between the JavaScript Language Specification and Tools Using the JISET Family

PLDI'22 Tutorial

Sukyoung Ryu[1], Jihyeok Park[2], **Seungmin An**[1]

[1] KAIST, South Korea
[2] Oracle Labs, Australia

June 13, 2022

PLRG
Programming Language
Research Group

Installation Guide

ESMeta & Double Debugger

- https://github.com/es-meta/esmeta
- https://github.com/es-meta/esmeta-debugger-client

Introduction to Double Debugger

JavaScript

JavaScript

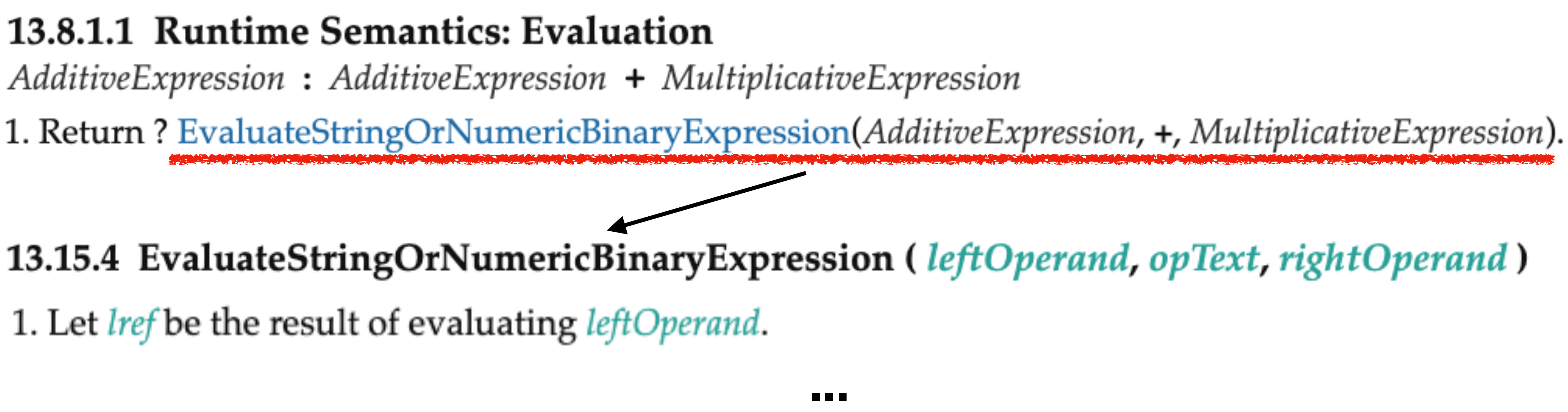
```
var x = ""; var y = ({valueOf: () => { return x = 3 }) + x;
```

Q. What are the values of x and y?

Language Specification

ECMA-262

TC 39



13.8.1.1 Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **+** *MultiplicativeExpression*

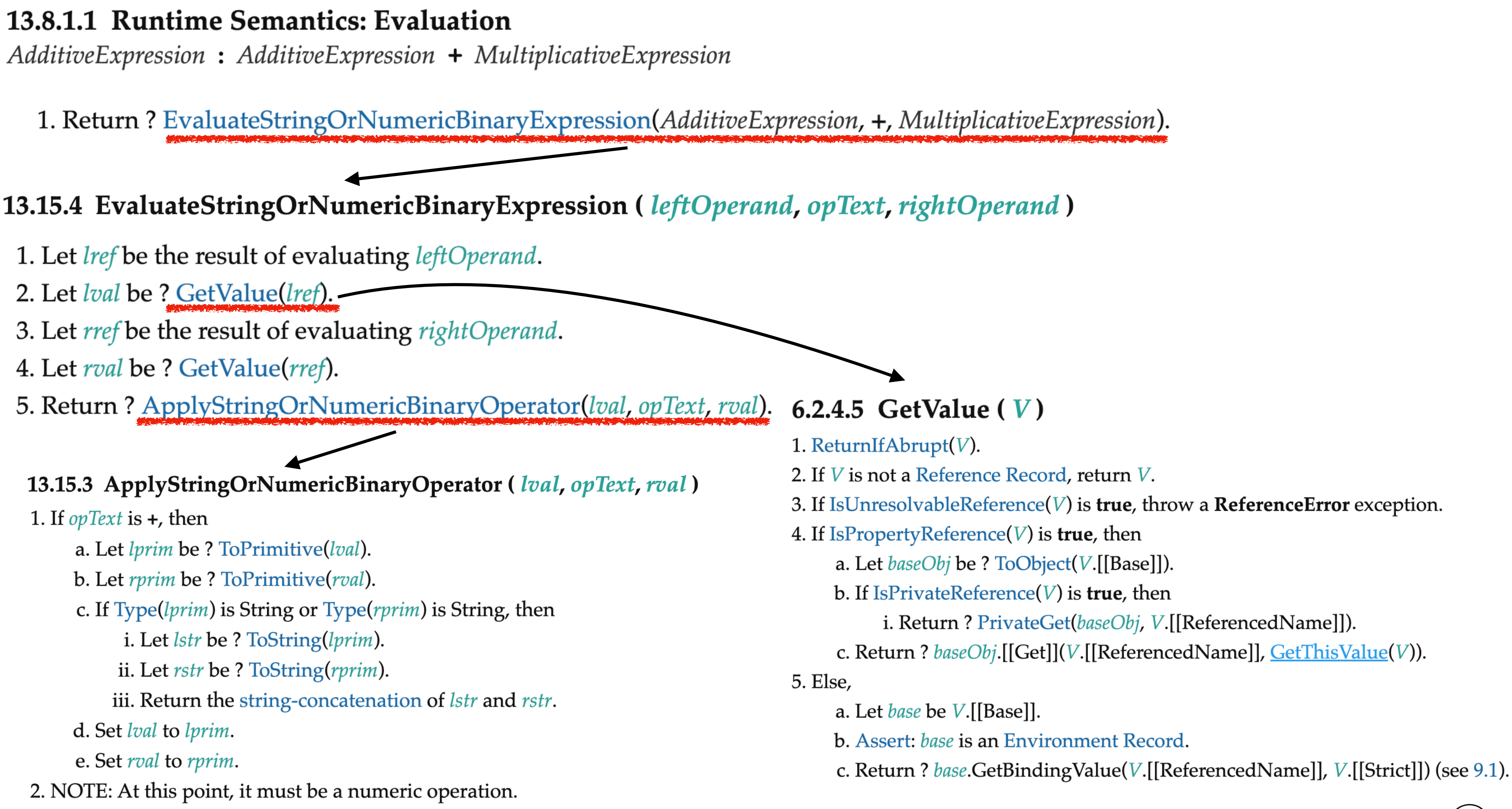
1. Return ? EvaluateStringOrNumericBinaryExpression(*AdditiveExpression*, +, *MultiplicativeExpression*).

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

1. Let *lref* be the result of evaluating *leftOperand*.

...

ECMA-262 Is Hard to Understand and Write.



13.8.1.1 Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **+** *MultiplicativeExpression*

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

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

```
1. Let lref be the result of evaluating leftOperand.
2. Let lval be ? GetValue(lref).
3. Let rref be the result of evaluating rightOperand.
4. Let rval be ? GetValue(rref).
5. Return ? ApplyStringOrNumericBinaryOperator(lval, opText, rval).
```

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

```
1. If opText is +, then
    a. Let lprim be ? ToPrimitive(lval).
    b. Let rprim be ? ToPrimitive(rval).
    c. If Type(lprim) is String or Type(rprim) is 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.
```

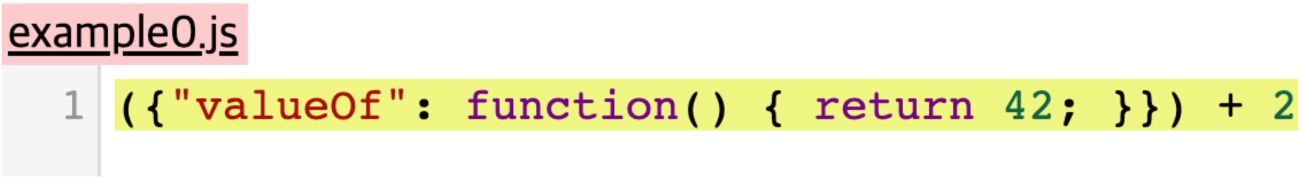
6.2.4.5 GetValue (*V*)

```
1. ReturnIfAbrupt(V).
2. If V is not a Reference Record, return V.
3. If IsUnresolvableReference(V) is true, throw a ReferenceError exception.
4. If IsPropertyReference(V) is true, then
    a. Let baseObj be ? ToObject(V.[[Base]]).
    b. If IsPrivateReference(V) is true, then
        i. Return ? PrivateGet(baseObj, V.[[ReferencedName]]).
    c. Return ? baseObj.[[Get]](V.[[ReferencedName]], GetThisValue(V)).
5. Else,
    a. Let base be V.[[Base]].
    b. Assert: base is an Environment Record.
    c. Return ? base.GetBindingValue(V.[[ReferencedName]], V.[[Strict]]) (see 9.1).
```

PLRG
Programming Language Research Group

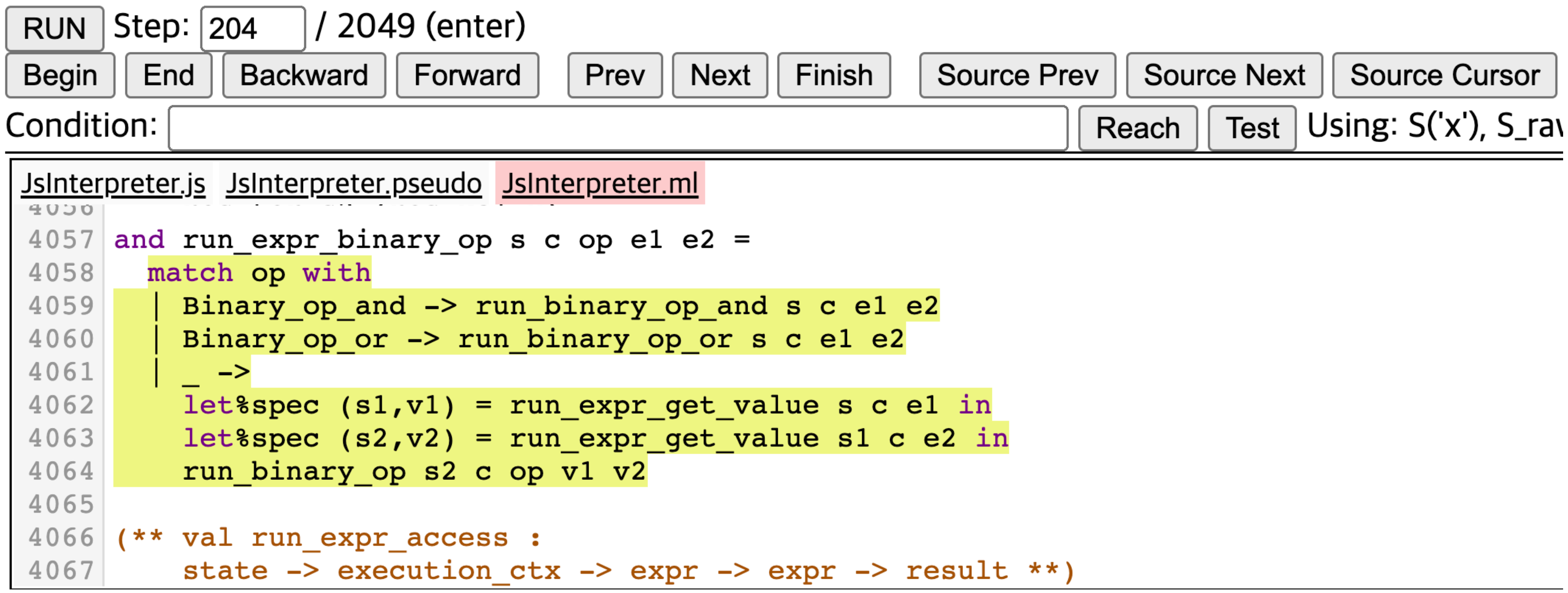
Double Debugger

JavaScript



```
({"valueOf": function() { return 42; }}) + 2
```

JavaScript Interpreter



Step: 204 / 2049 (enter)

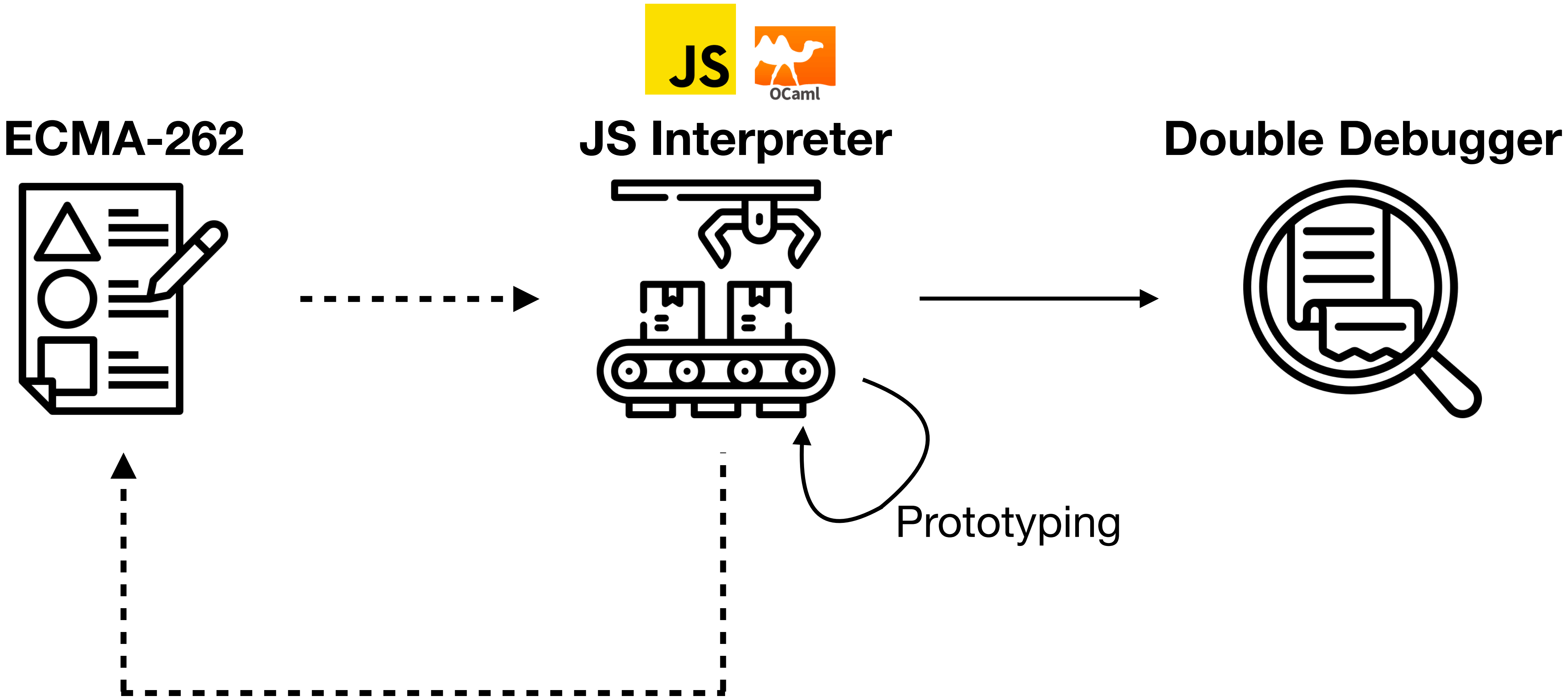
Condition:

Using: S('x'), S_ra

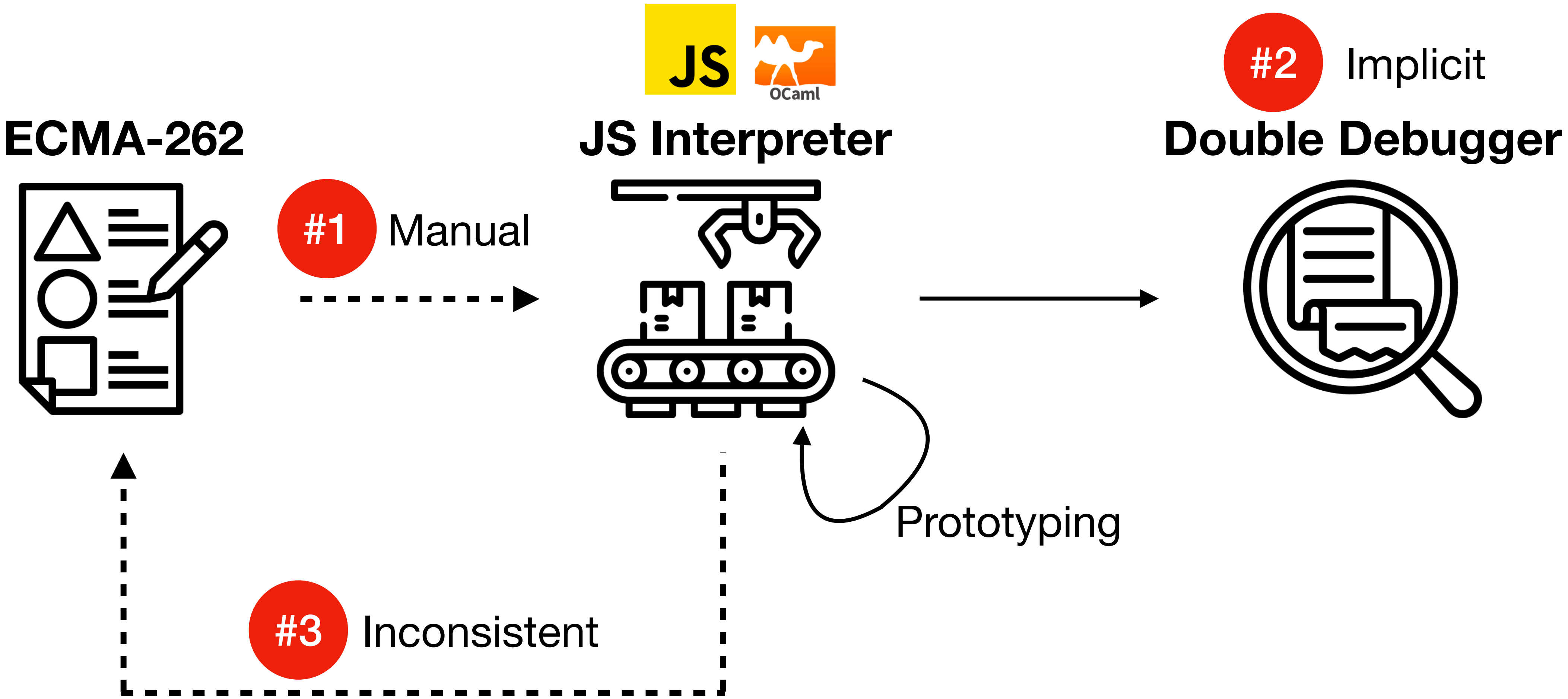
```
and run_expr_binary_op s c op e1 e2 =
  match op with
  | Binary_op_and -> run_binary_op_and s c e1 e2
  | Binary_op_or -> run_binary_op_or s c e1 e2
  | _ ->
    let%spec (s1,v1) = run_expr_get_value s c e1 in
    let%spec (s2,v2) = run_expr_get_value s1 c e2 in
    run_binary_op s2 c op v1 v2

(** val run_expr_access :
    state -> execution_ctx -> expr -> expr -> result **)
```

PLRG
Programming Language
Research Group

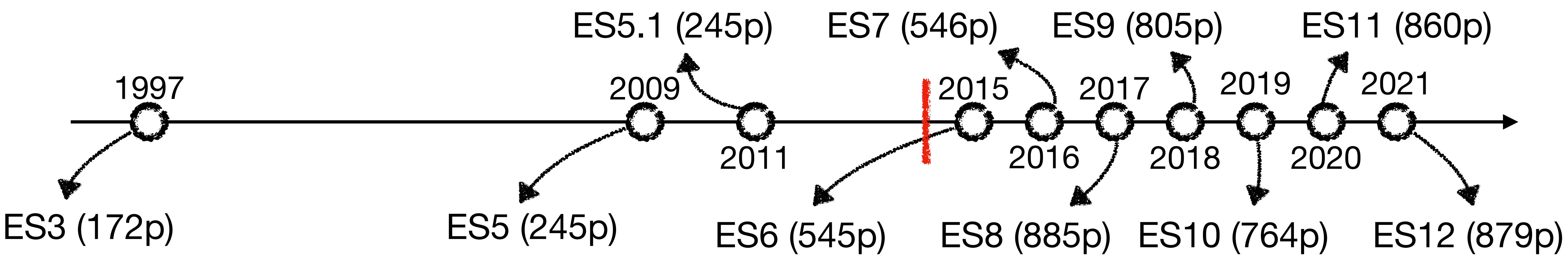
Current Solution



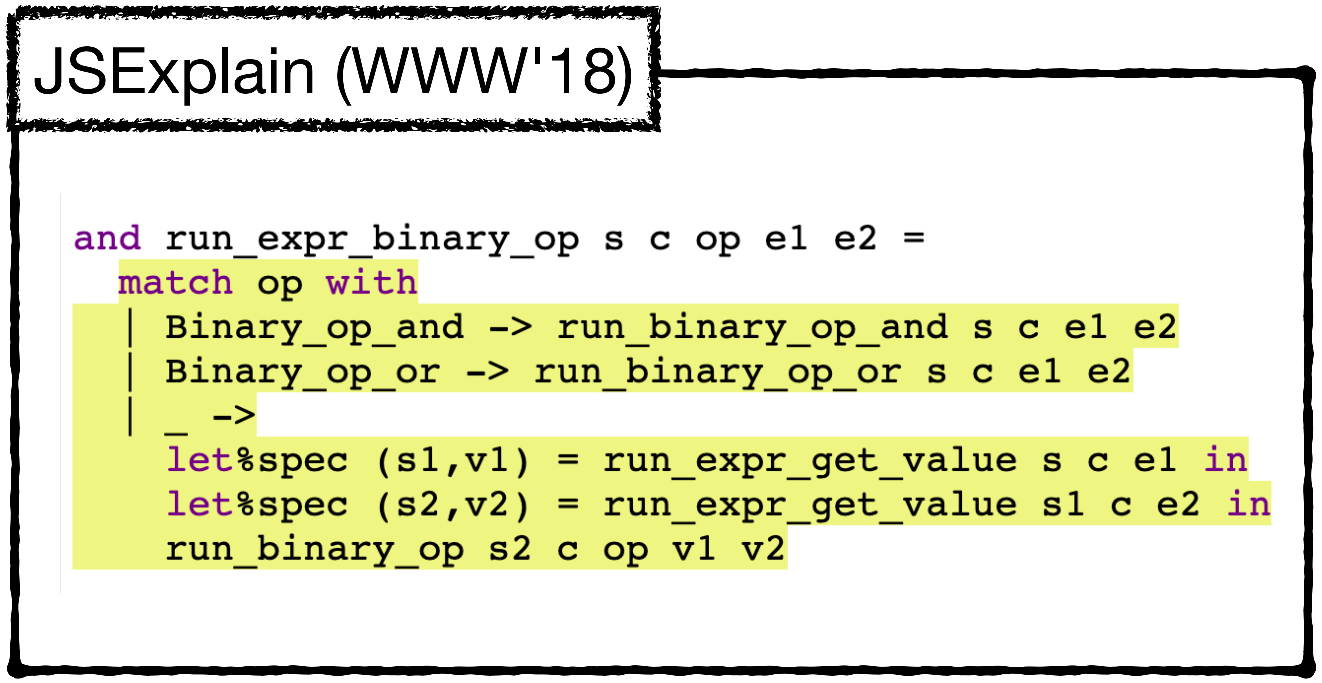
Current Solution



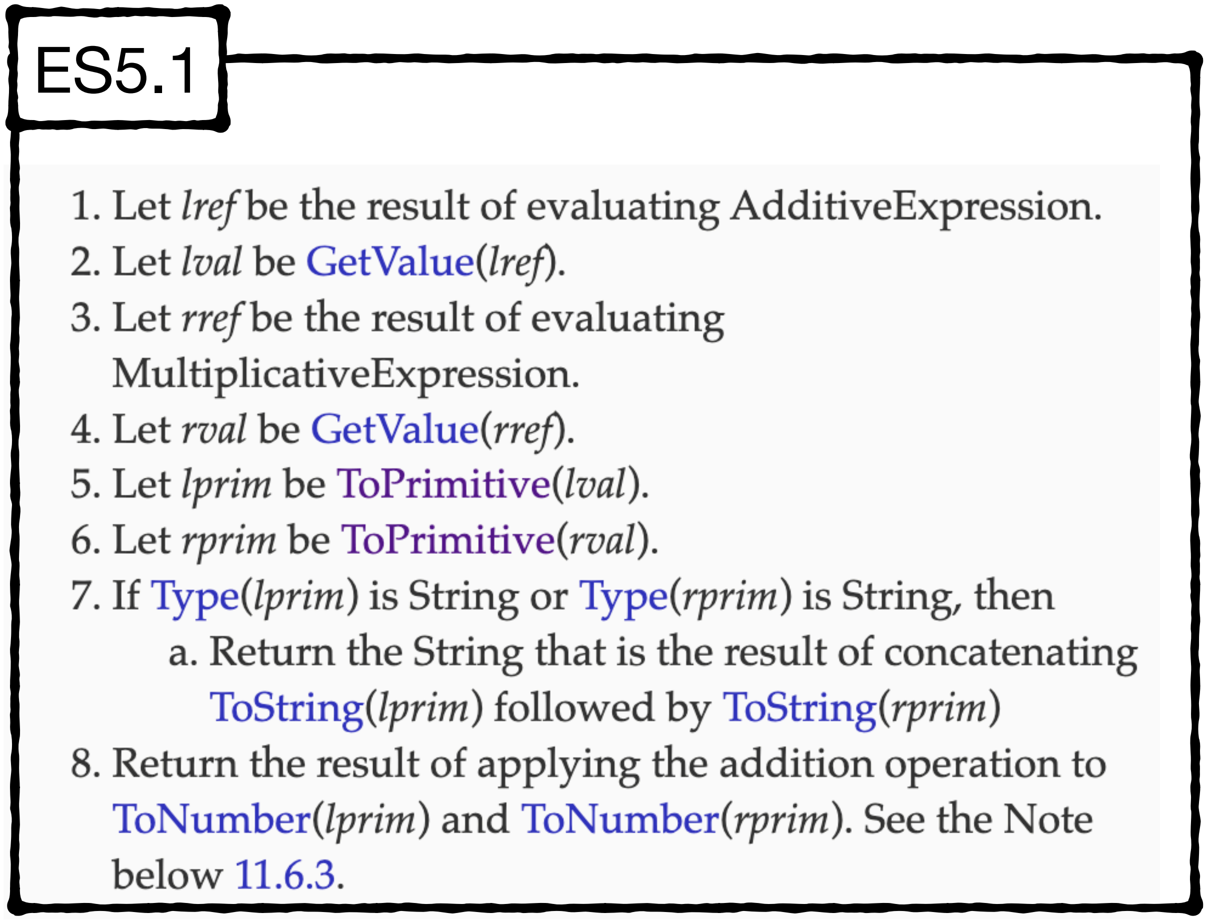
#1: JavaScript interpreter is manually implemented.


1997
ES3 (172p)
2009
ES5 (245p)
2011
ES5.1 (245p)
2015
ES6 (545p)
2016
ES7 (546p)
2017
ES8 (885p)
2018
ES9 (805p)
2019
ES10 (764p)
2020
ES11 (860p)
2021
ES12 (879p)

#2: ECMA-262 is not explicitly displayed.

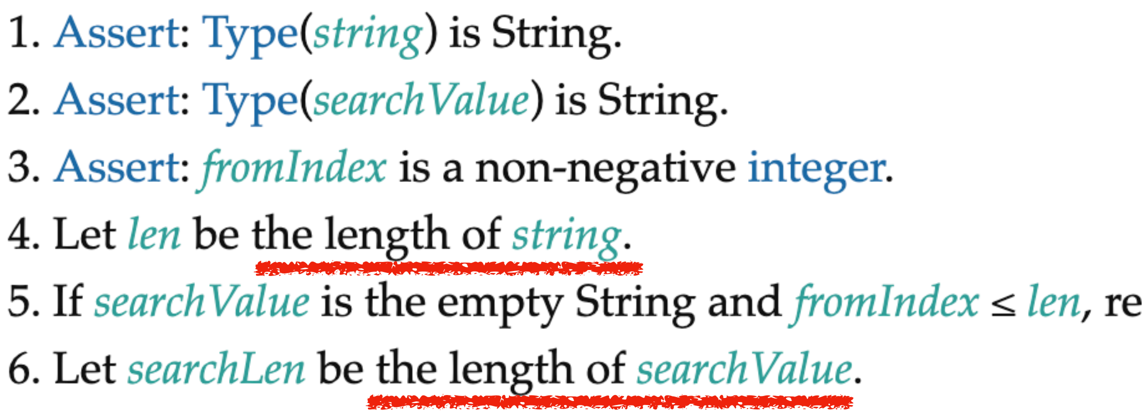

JSExplain (WWW'18)

```
and run_expr_binary_op s c op e1 e2 =
  match op with
  | Binary_op_and -> run_binary_op_and s c e1 e2
  | Binary_op_or -> run_binary_op_or s c e1 e2
  | _ ->
    let%spec (s1,v1) = run_expr_get_value s c e1 in
    let%spec (s2,v2) = run_expr_get_value s1 c e2 in
    run_binary_op s2 c op v1 v2
```

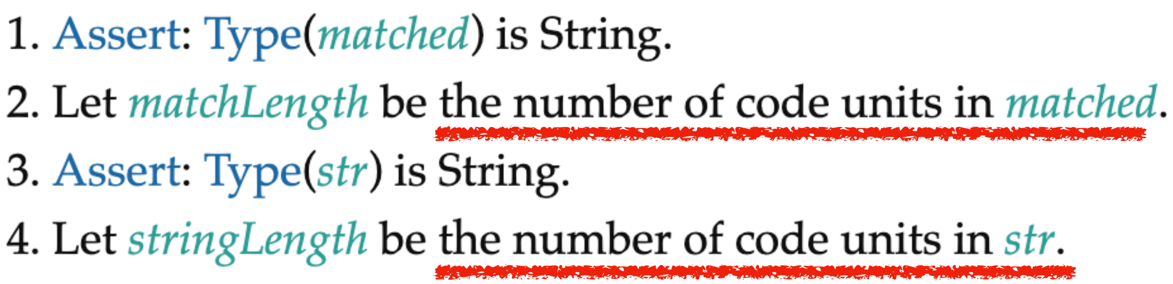

ES5.1

1. Let *lref* be the result of evaluating AdditiveExpression.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating MultiplicativeExpression.
4. Let *rval* be GetValue(*rref*).
5. Let *lprim* be ToPrimitive(*lval*).
6. Let *rprim* be ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Return the String that is the result of concatenating ToString(*lprim*) followed by ToString(*rprim*)
8. Return the result of applying the addition operation to ToNumber(*lprim*) and ToNumber(*rprim*). See the Note below 11.6.3.

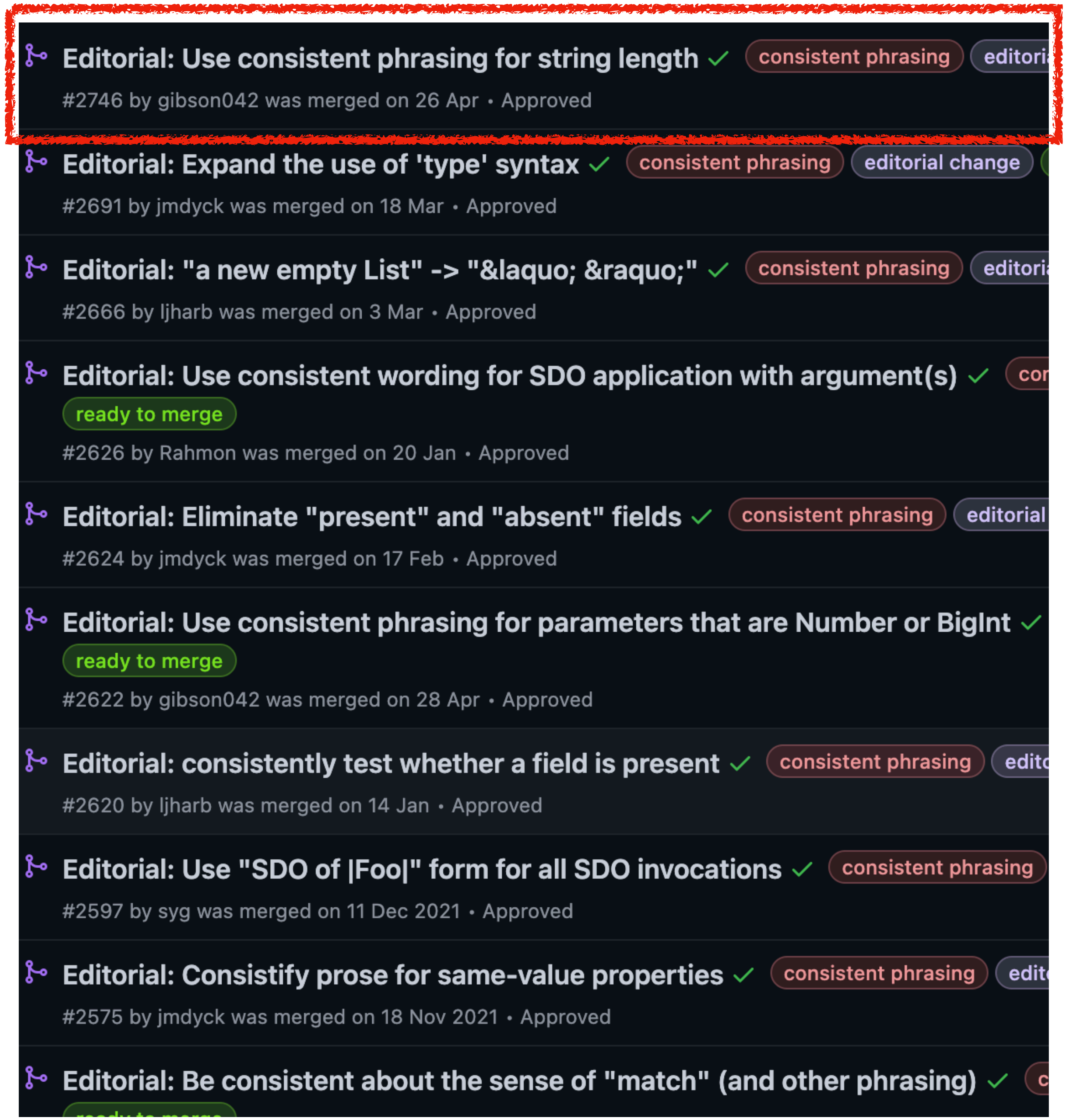
#3: English phrases are inconsistent.



1. Assert: Type(*string*) is String.
2. Assert: Type(*searchValue*) is String.
3. Assert: *fromIndex* is a non-negative integer.
4. Let *len* be the length of *string*.
5. If *searchValue* is the empty String and *fromIndex* ≤ *len*, re
6. Let *searchLen* be the length of *searchValue*.



1. Assert: Type(*matched*) is String.
2. Let *matchLength* be the number of code units in *matched*.
3. Assert: Type(*str*) is String.
4. Let *stringLength* be the number of code units in *str*.



Editorial: Use consistent phrasing for string length ✓ consistent phrasing editori
#2746 by gibson042 was merged on 26 Apr • Approved

Editorial: Expand the use of 'type' syntax ✓ consistent phrasing editorial change
#2691 by jmdyck was merged on 18 Mar • Approved

Editorial: "a new empty List" -> "&laquo; »" ✓ consistent phrasing editori
#2666 by ljharb was merged on 3 Mar • Approved

Editorial: Use consistent wording for SDO application with argument(s) ✓ cor
ready to merge
#2626 by Rahmon was merged on 20 Jan • Approved

Editorial: Eliminate "present" and "absent" fields ✓ consistent phrasing editorial
#2624 by jmdyck was merged on 17 Feb • Approved

Editorial: Use consistent phrasing for parameters that are Number or BigInt ✓
ready to merge
#2622 by gibson042 was merged on 28 Apr • Approved

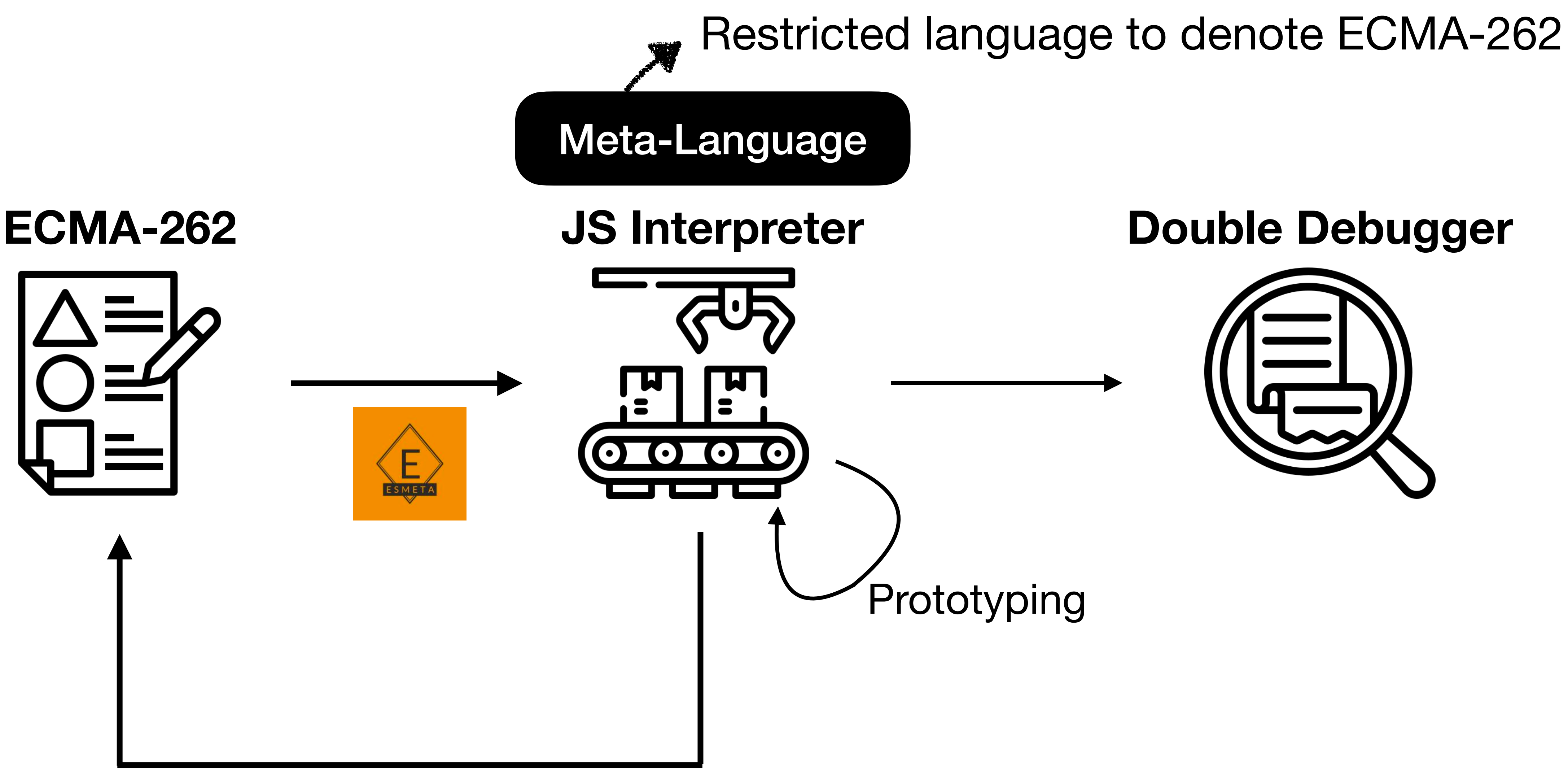
Editorial: consistently test whether a field is present ✓ consistent phrasing edito
#2620 by ljharb was merged on 14 Jan • Approved

Editorial: Use "SDO of |Foo|" form for all SDO invocations ✓ consistent phrasing
#2597 by syg was merged on 11 Dec 2021 • Approved

Editorial: Consistify prose for same-value properties ✓ consistent phrasing edit
#2575 by jmdyck was merged on 18 Nov 2021 • Approved

Editorial: Be consistent about the sense of "match" (and other phrasing) ✓ c

Our Solution: ESMeta



Restricted language to denote ECMA-262

Meta-Language

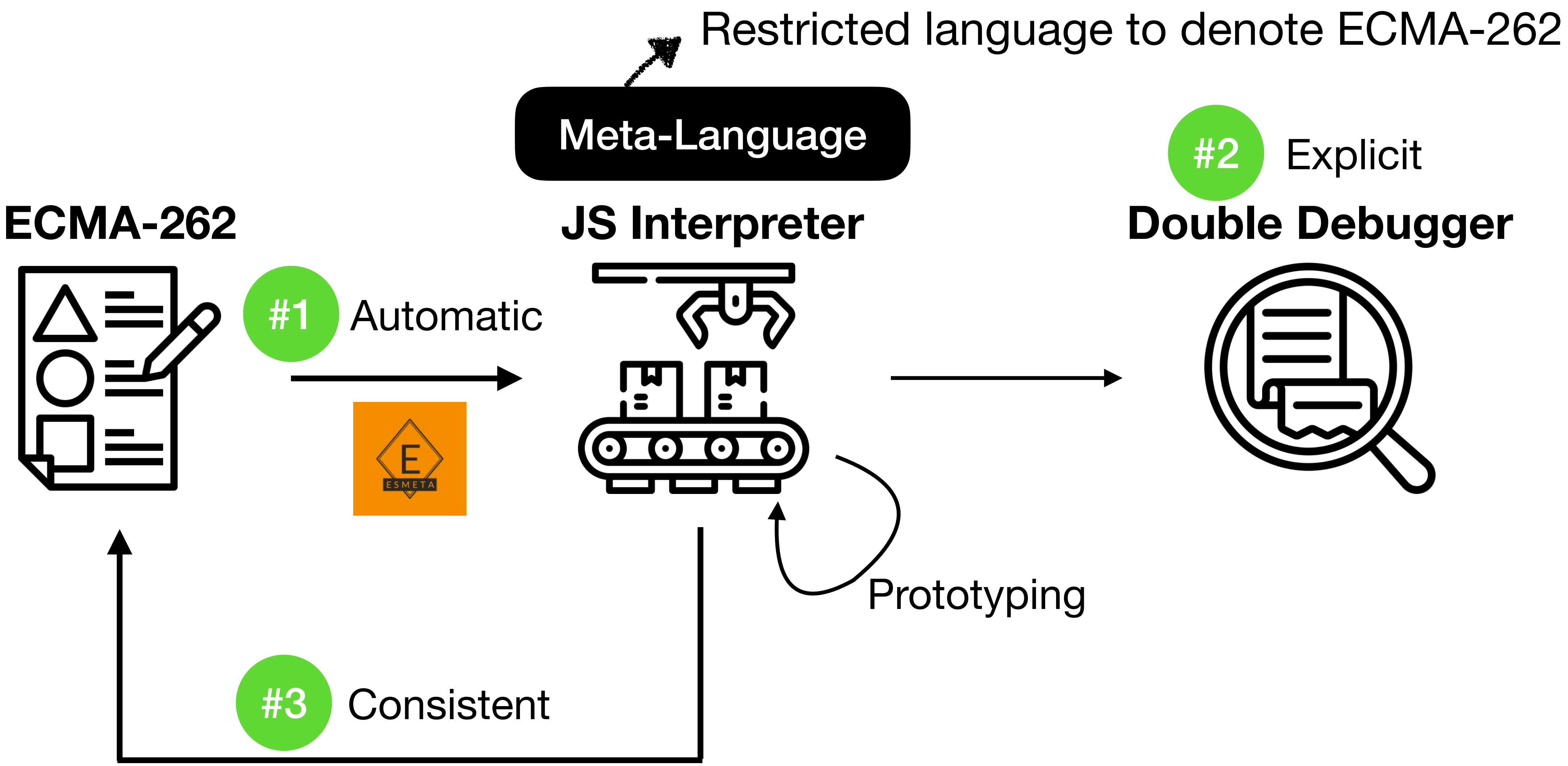
ECMA-262

JS Interpreter

Double Debugger

Prototyping

Our Solution: ESMeta



Restricted language to denote ECMA-262

Meta-Language

ECMA-262

#1 Automatic

JS Interpreter

#2 Explicit

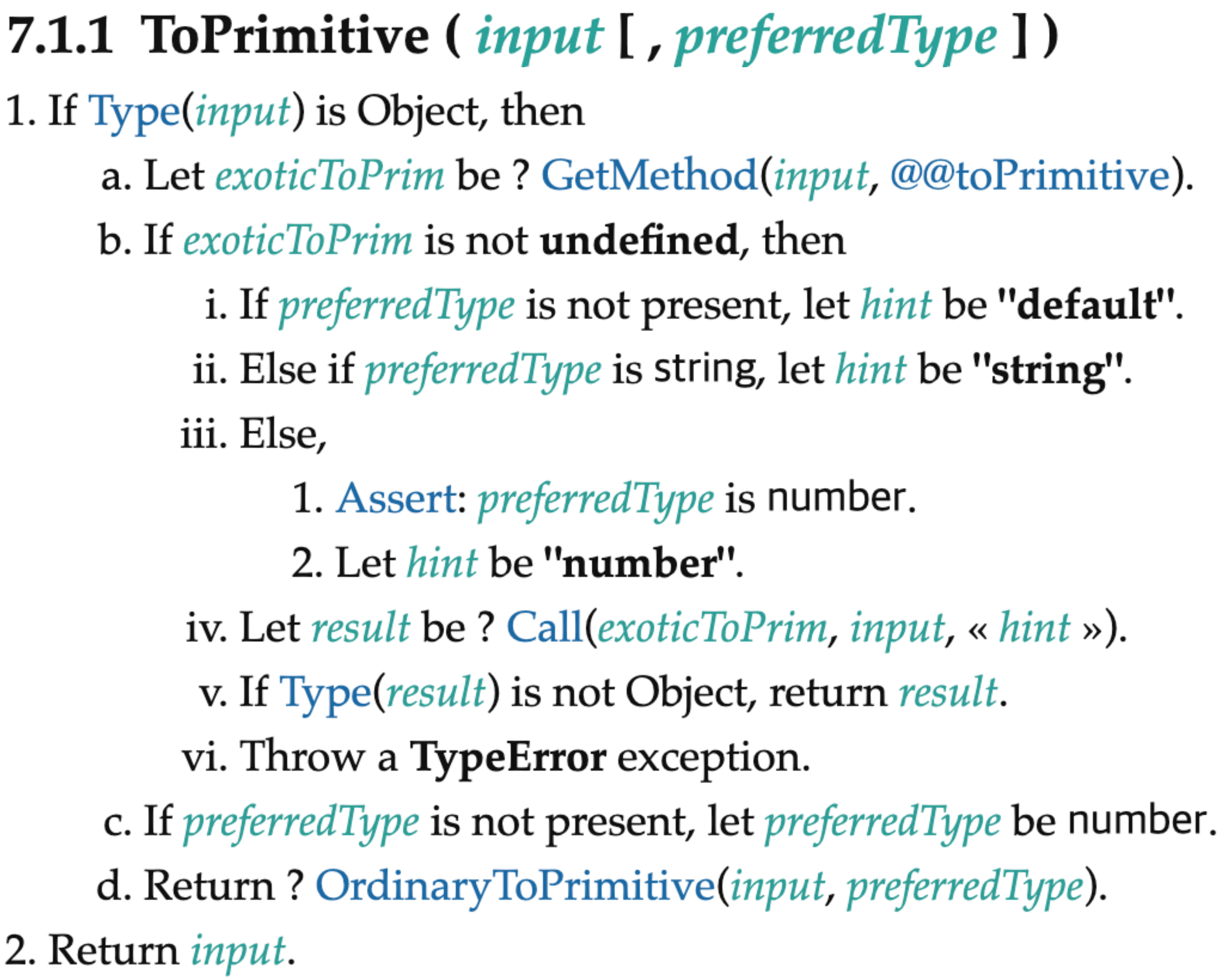
Double Debugger

Prototyping

#3 Consistent

Meta-language

- The bodies of abstract algorithm are written in English prose with patterns.

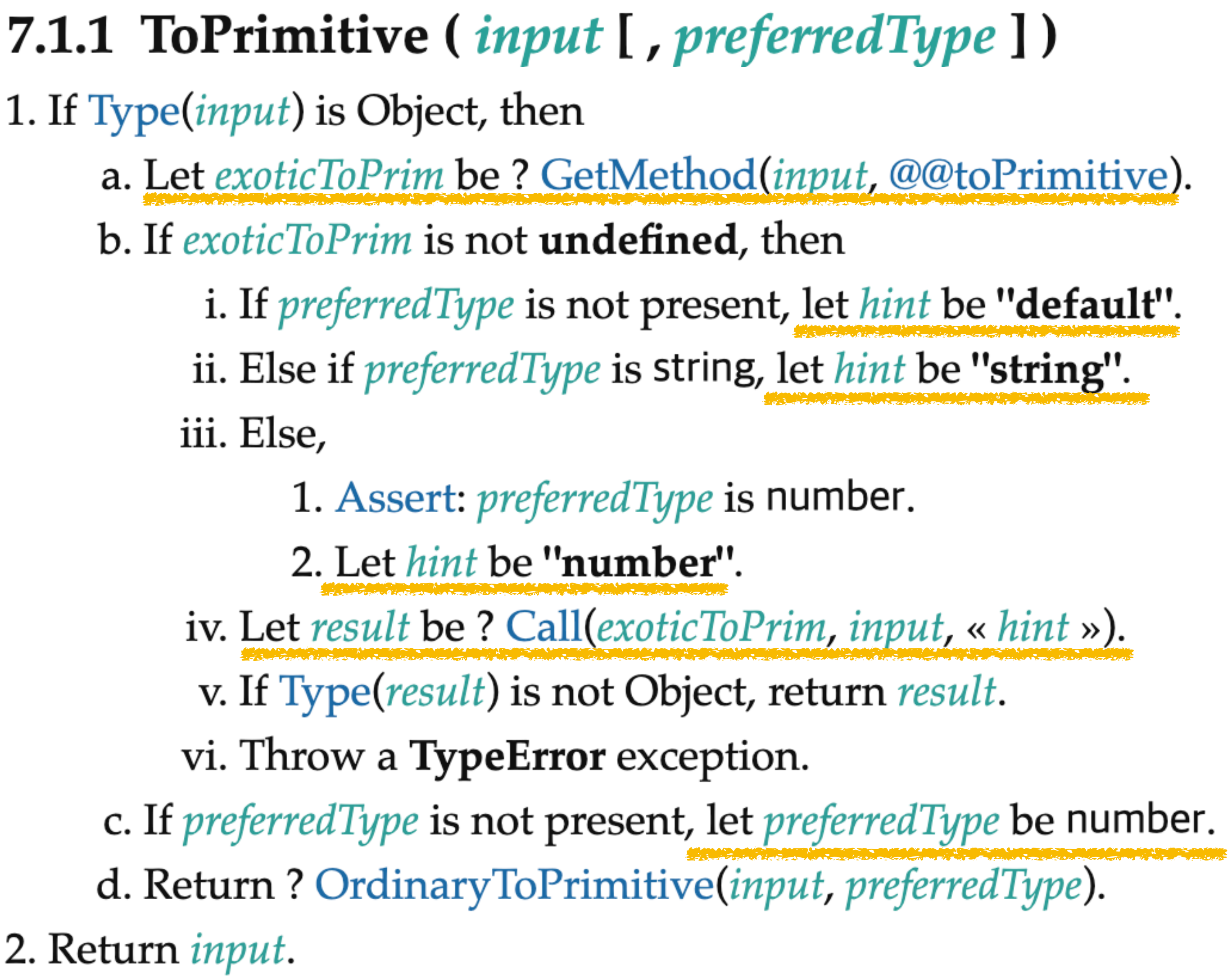


7.1.1 ToPrimitive (*input* [, *preferredType*])

```
1. If Type(input) is Object, then
    a. Let exoticToPrim be ? GetMethod(input, @@toPrimitive).
    b. If exoticToPrim is not undefined, then
        i. If preferredType is not present, let hint be "default".
        ii. Else if preferredType is string, let hint be "string".
        iii. Else,
            1. Assert: preferredType is number.
            2. Let hint be "number".
        iv. Let result be ? Call(exoticToPrim, input, « hint »).
        v. If Type(result) is not Object, return result.
        vi. Throw a TypeError exception.
    c. If preferredType is not present, let preferredType be number.
    d. Return ? OrdinaryToPrimitive(input, preferredType).
2. Return input.
```

Meta-language

- The bodies of abstract algorithm are written in English prose with patterns.



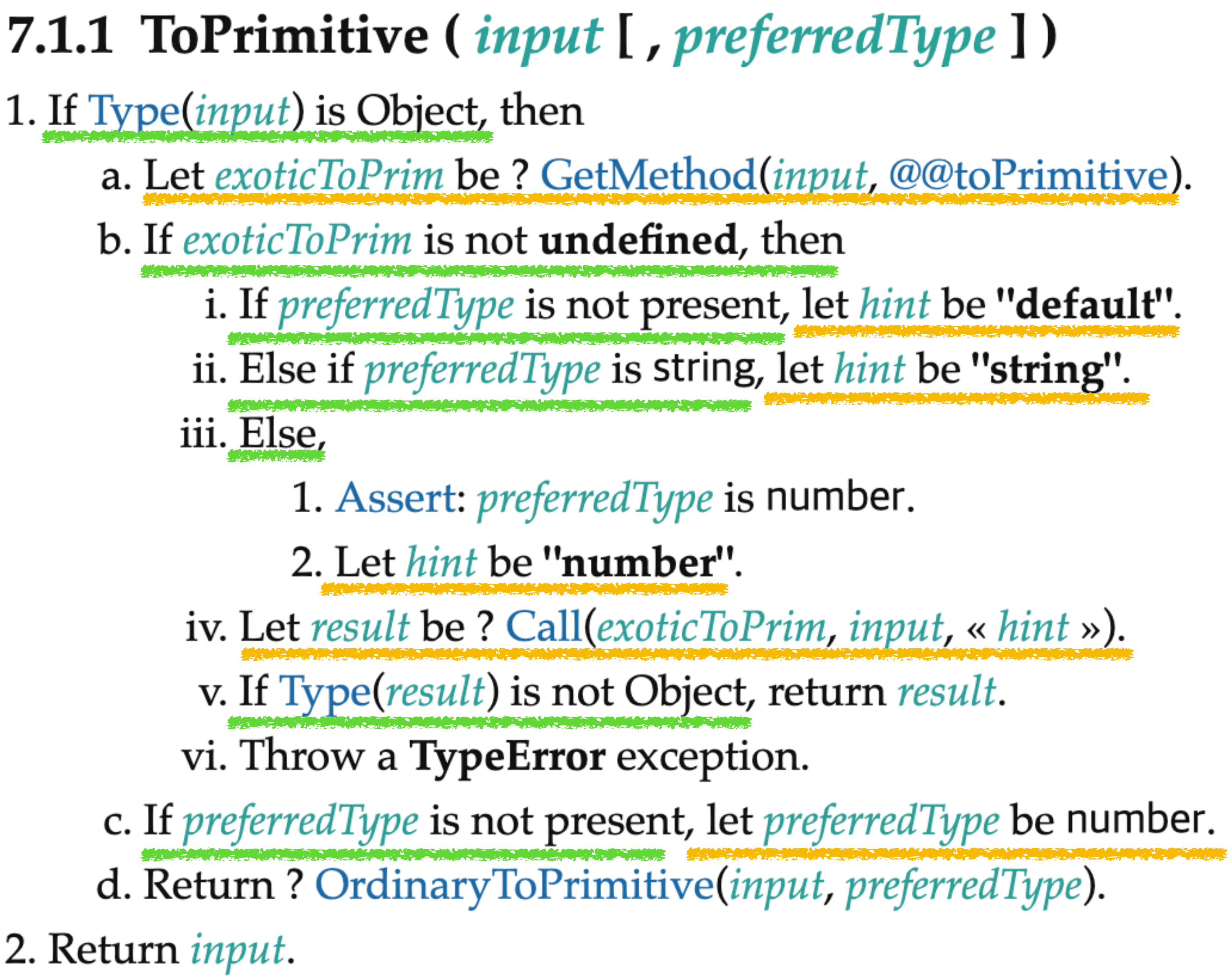
7.1.1 ToPrimitive (*input* [, *preferredType*])

```
1. If Type(input) is Object, then
    a. Let exoticToPrim be ? GetMethod(input, @@toPrimitive).
    b. If exoticToPrim is not undefined, then
        i. If preferredType is not present, let hint be "default".
        ii. Else if preferredType is string, let hint be "string".
        iii. Else,
            1. Assert: preferredType is number.
            2. Let hint be "number".
        iv. Let result be ? Call(exoticToPrim, input, « hint »).
        v. If Type(result) is not Object, return result.
        vi. Throw a TypeError exception.
    c. If preferredType is not present, let preferredType be number.
    d. Return ? OrdinaryToPrimitive(input, preferredType).
2. Return input.
```

Let … be …

Meta-language

- The bodies of abstract algorithm are written in English prose with patterns.



7.1.1 ToPrimitive (*input* [, *preferredType*])

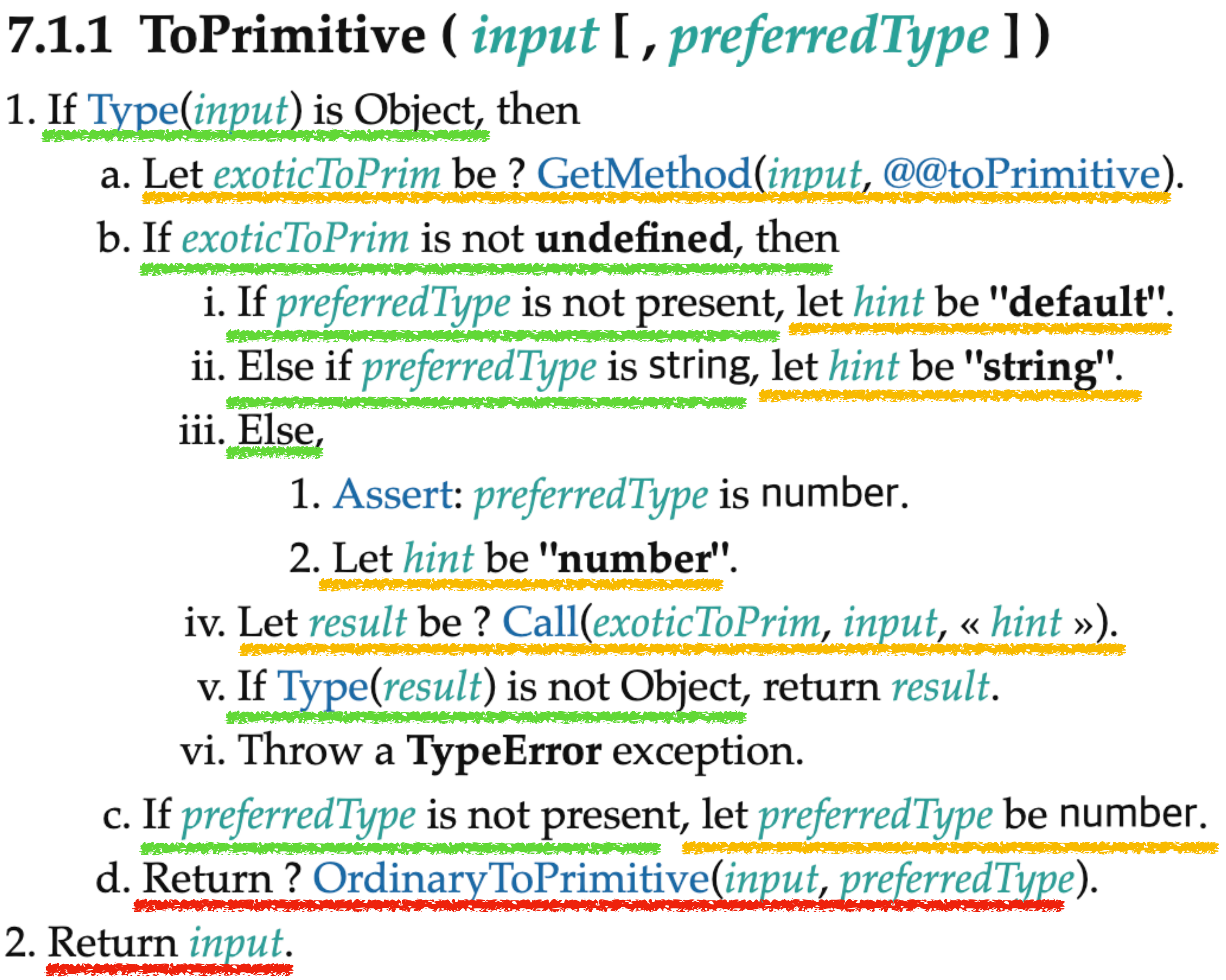
```
1. If Type(input) is Object, then
    a. Let exoticToPrim be ? GetMethod(input, @@toPrimitive).
    b. If exoticToPrim is not undefined, then
        i. If preferredType is not present, let hint be "default".
        ii. Else if preferredType is string, let hint be "string".
        iii. Else,
            1. Assert: preferredType is number.
            2. Let hint be "number".
        iv. Let result be ? Call(exoticToPrim, input, « hint »).
        v. If Type(result) is not Object, return result.
        vi. Throw a TypeError exception.
    c. If preferredType is not present, let preferredType be number.
    d. Return ? OrdinaryToPrimitive(input, preferredType).
2. Return input.
```

Let … be …

If … , then … else, …

Meta-language

- The bodies of abstract algorithm are written in English prose with patterns.



7.1.1 ToPrimitive (*input* [, *preferredType*])

```
1. If Type(input) is Object, then
    a. Let exoticToPrim be ? GetMethod(input, @@toPrimitive).
    b. If exoticToPrim is not undefined, then
        i. If preferredType is not present, let hint be "default".
        ii. Else if preferredType is string, let hint be "string".
        iii. Else,
            1. Assert: preferredType is number.
            2. Let hint be "number".
        iv. Let result be ? Call(exoticToPrim, input, « hint »).
        v. If Type(result) is not Object, return result.
        vi. Throw a TypeError exception.
    c. If preferredType is not present, let preferredType be number.
    d. Return ? OrdinaryToPrimitive(input, preferredType).
2. Return input.
```

Let … be …

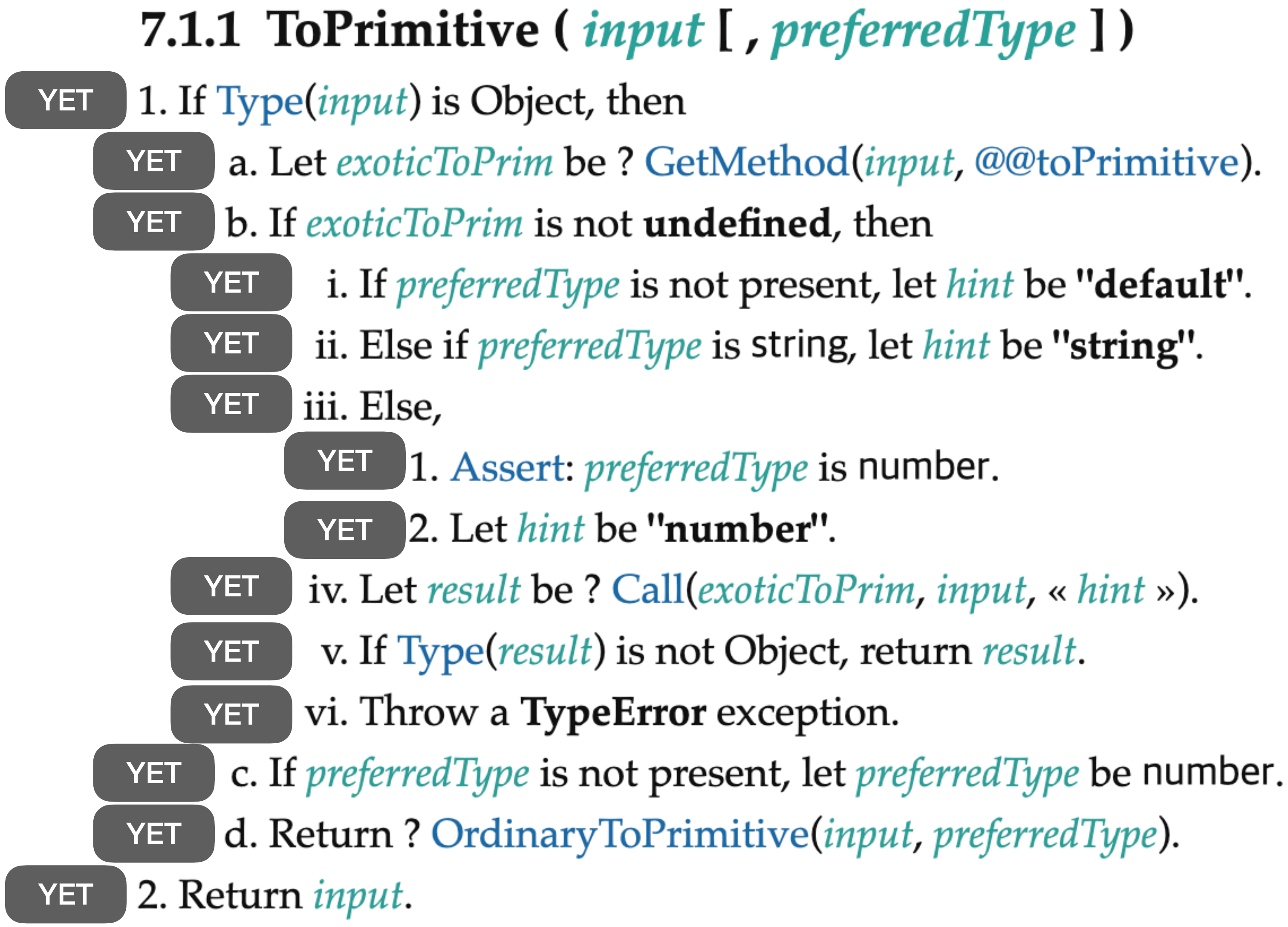
If … , then … else, …

Return …

…

Meta-language

- From writing patterns, we build a parser and incrementally construct a meta-language.



7.1.1 ToPrimitive (*input* [, *preferredType*])

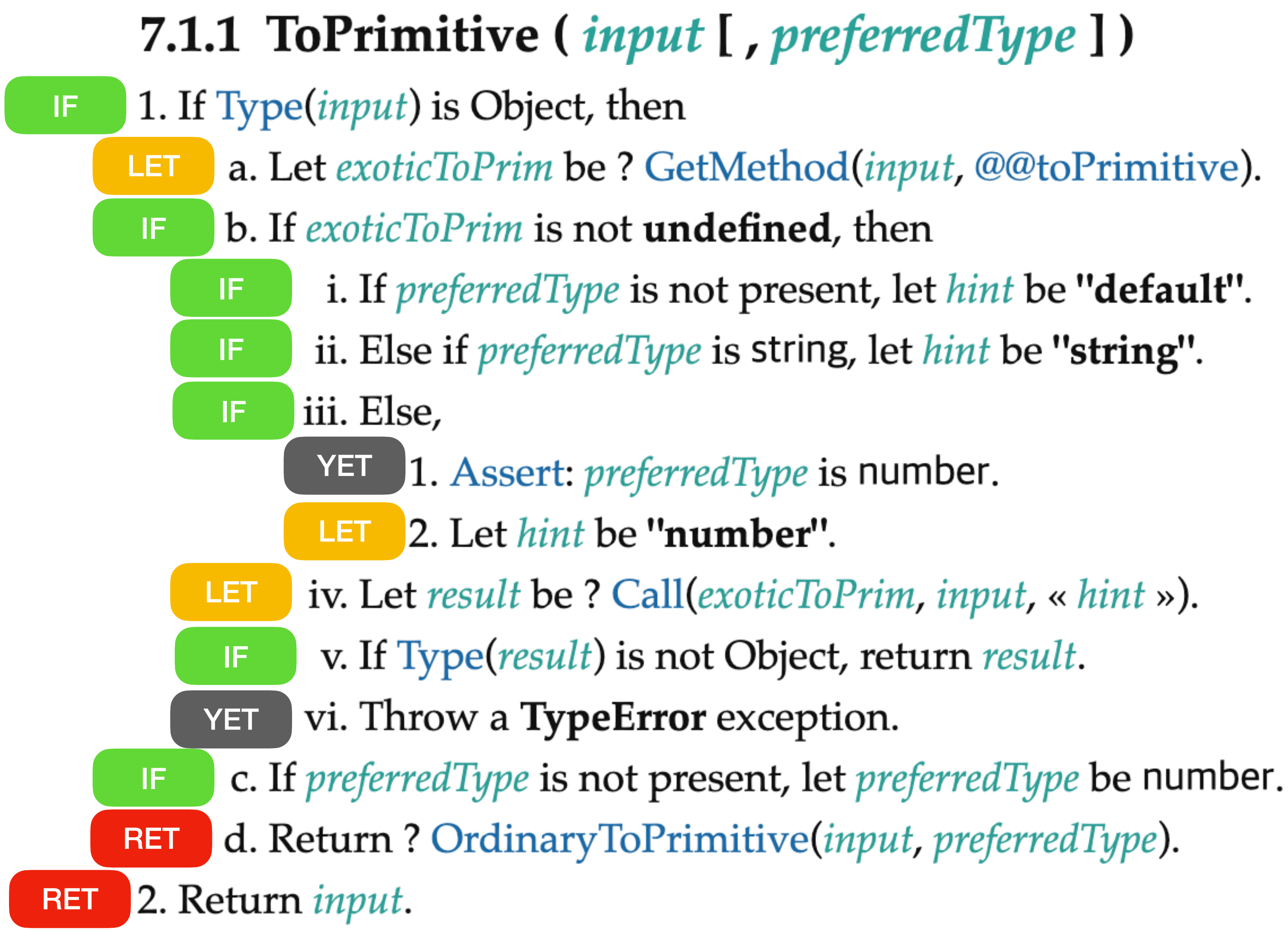
```
YET 1. If Type(input) is Object, then
    YET a. Let exoticToPrim be ? GetMethod(input, @@toPrimitive).
    YET b. If exoticToPrim is not undefined, then
        YET i. If preferredType is not present, let hint be "default".
        YET ii. Else if preferredType is string, let hint be "string".
        YET iii. Else,
            YET 1. Assert: preferredType is number.
            YET 2. Let hint be "number".
        YET iv. Let result be ? Call(exoticToPrim, input, « hint »).
        YET v. If Type(result) is not Object, return result.
        YET vi. Throw a TypeError exception.
    YET c. If preferredType is not present, let preferredType be number.
    YET d. Return ? OrdinaryToPrimitive(input, preferredType).
YET 2. Return input.
```

<parsing rule>

<meta-language>

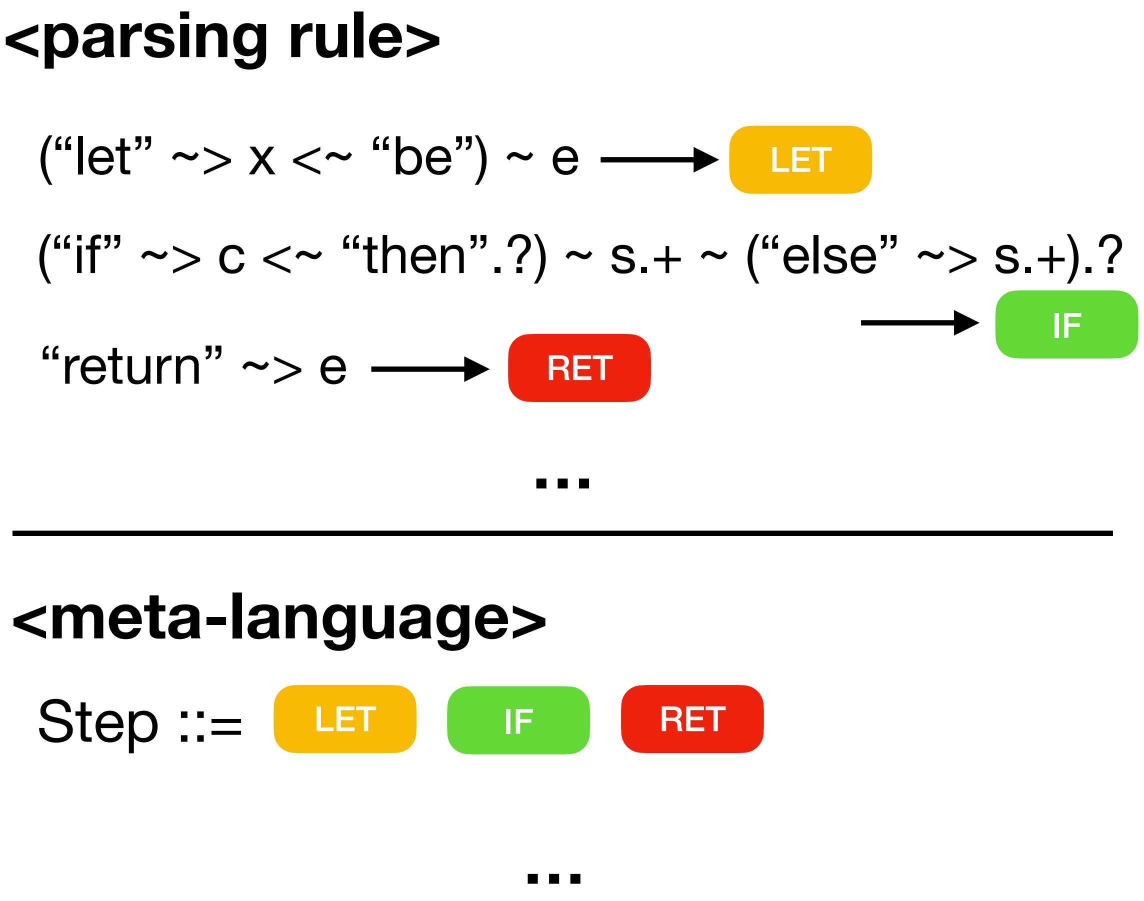
Meta-language

- From writing patterns, we build a parser and incrementally construct a meta-language.



7.1.1 ToPrimitive (*input* [, *preferredType*])

```
IF   1. If Type(input) is Object, then
  LET  a. Let exoticToPrim be ? GetMethod(input, @@toPrimitive).
  IF   b. If exoticToPrim is not undefined, then
    IF   i. If preferredType is not present, let hint be "default".
    IF   ii. Else if preferredType is string, let hint be "string".
    IF   iii. Else,
      YET  1. Assert: preferredType is number.
      LET  2. Let hint be "number".
    LET  iv. Let result be ? Call(exoticToPrim, input, « hint »).
    IF   v. If Type(result) is not Object, return result.
    YET  vi. Throw a TypeError exception.
  IF   c. If preferredType is not present, let preferredType be number.
  RET  d. Return ? OrdinaryToPrimitive(input, preferredType).
RET  2. Return input.
```



<parsing rule>

```
("let" ~> x <~ "be") ~ e → LET
("if" ~> c <~ "then".?) ~ s.+ ~ ("else" ~> s.+).? → IF
"return" ~> e → RET
...
```

<meta-language>

Step ::= LET IF RET

...

PLRG Programming Language Research Group

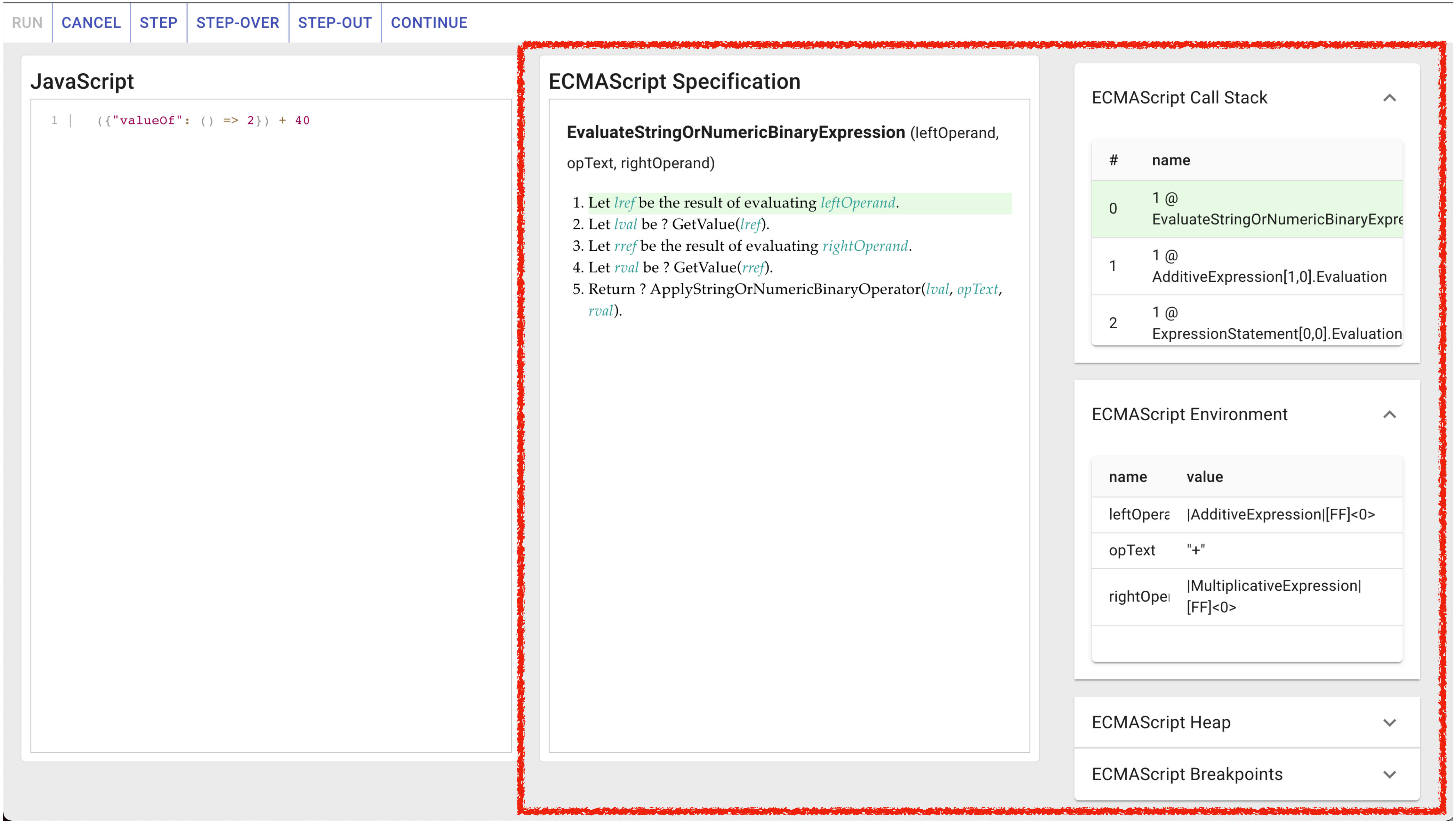
#1: Automatic

Kind	Step	Expression	Condition	Reference	Literal
#	20	26	8	11	29

ECMA-262 Version: cf7145ea3f14943b5aea7d5e05c771f31f989606

- Meta-language is expressive.
  - Steps: 17,763/ 18,789 (94.64%)
  - Algorithms: 2,158/ 2,612 (82.62%)
- Meta-language will not be changed with a high probability.

#2: Explicit



JavaScript

```
({"valueOf": () => 2}) + 40
```

ECMAScript Specification

EvaluateStringOrNumericBinaryExpression (leftOperand, opText, rightOperand)

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

ECMAScript Call Stack

#	name
0	1 @ EvaluateStringOrNumericBinaryExpre
1	1 @ AdditiveExpression[1,0].Evaluation
2	1 @ ExpressionStatement[0,0].Evaluation

ECMAScript Environment

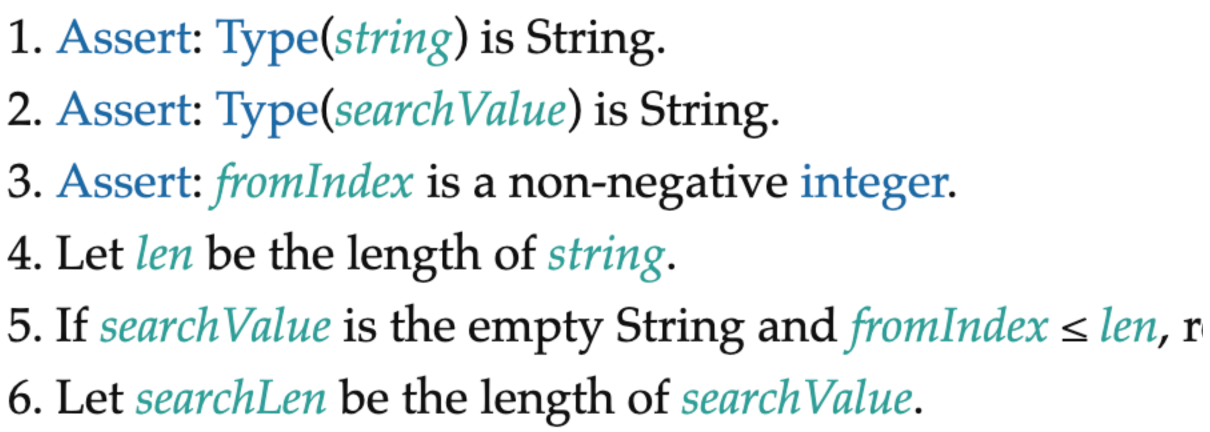
name	value
leftOpera	|AdditiveExpression|[FF]<0>
opText	"+"
rightOpe	|MultiplicativeExpression|[FF]<0>

ECMAScript Heap

ECMAScript Breakpoints

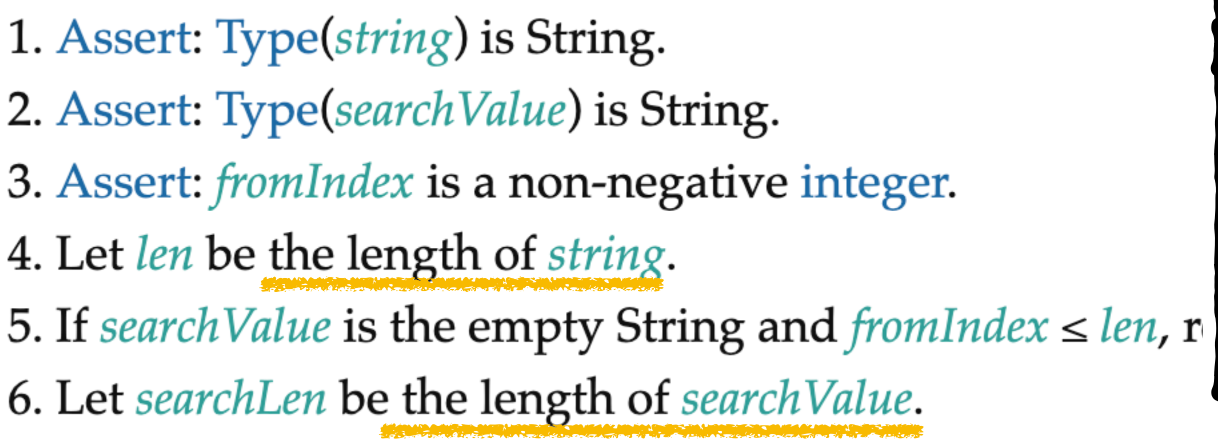
#3: Consistent

#3: Consistent

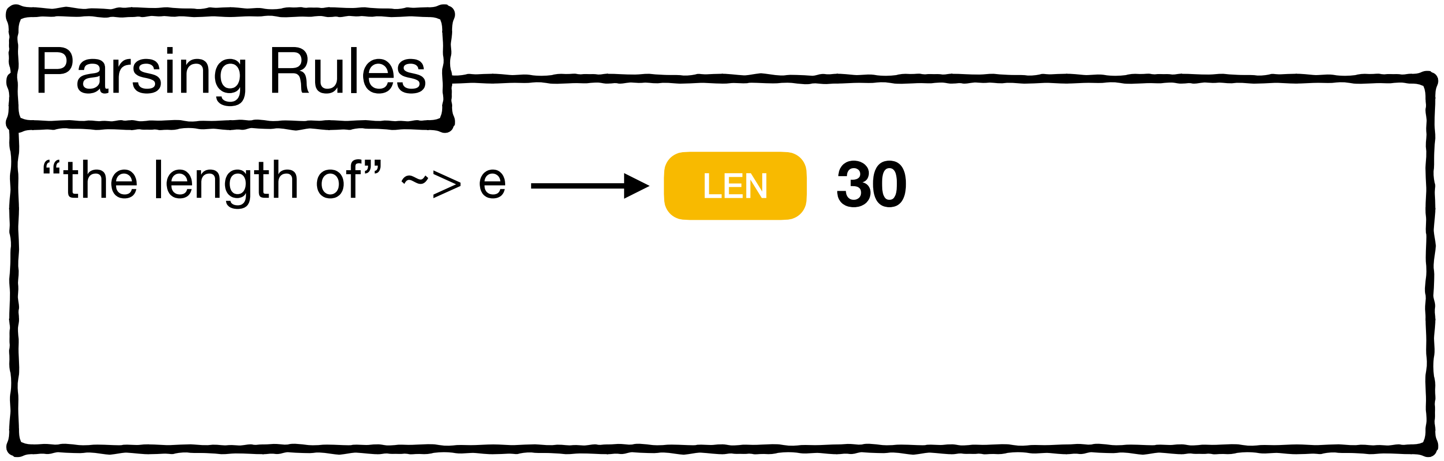


1. Assert: Type(*string*) is String.
2. Assert: Type(*searchValue*) is String.
3. Assert: *fromIndex* is a non-negative integer.
4. Let *len* be the length of *string*.
5. If *searchValue* is the empty String and *fromIndex* ≤ *len*, r
6. Let *searchLen* be the length of *searchValue*.

#3: Consistent



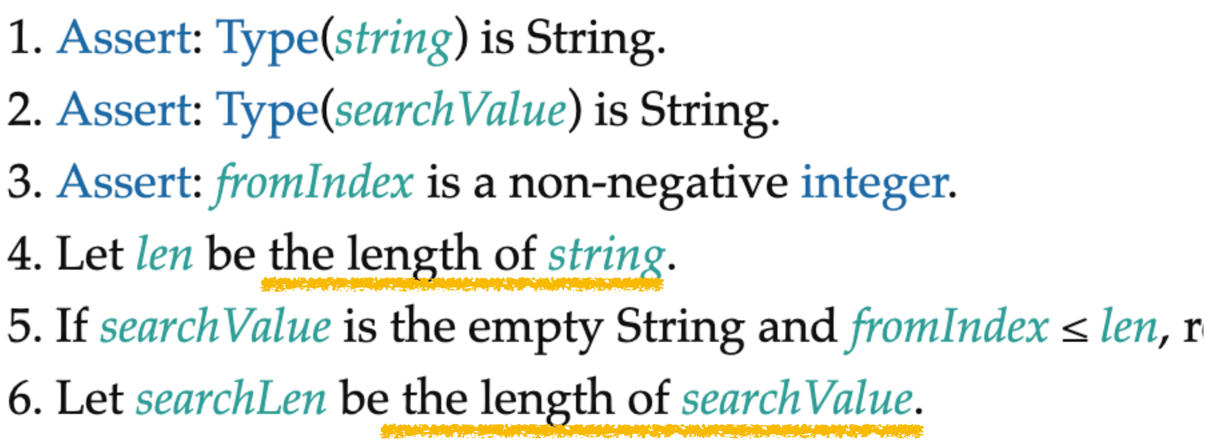
1. Assert: Type(*string*) is String.
2. Assert: Type(*searchValue*) is String.
3. Assert: *fromIndex* is a non-negative integer.
4. Let *len* be the length of *string*.
5. If *searchValue* is the empty String and *fromIndex* ≤ *len*, r
6. Let *searchLen* be the length of *searchValue*.



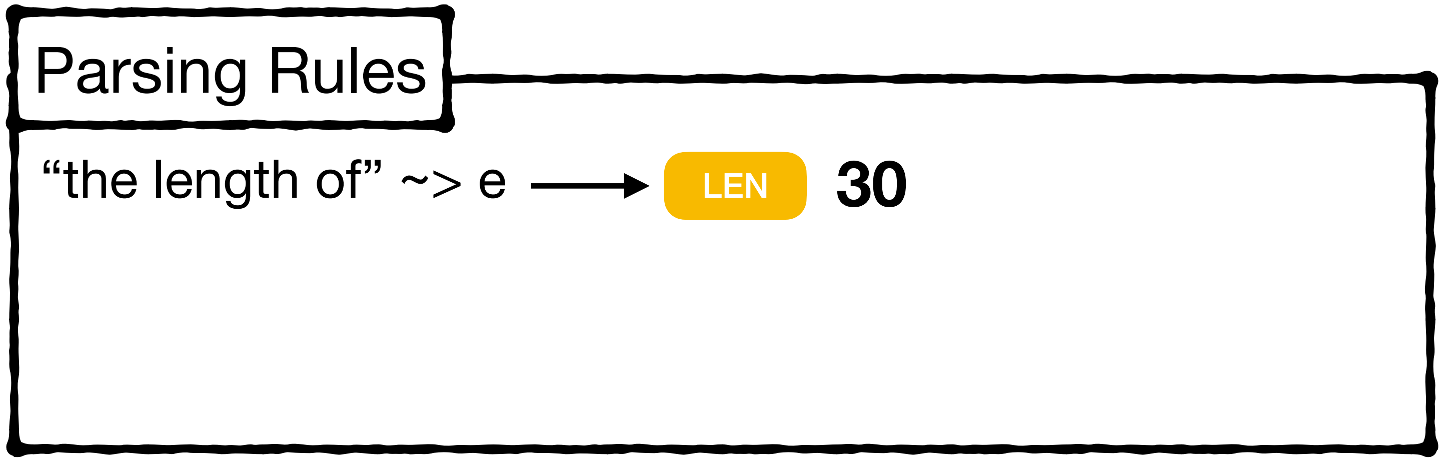
Parsing Rules

"the length of" ~> e → LEN **30**

#3: Consistent

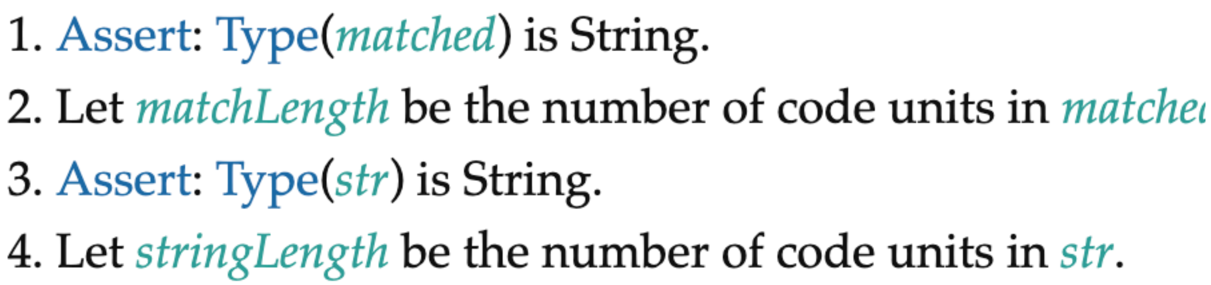


1. Assert: Type(*string*) is String.
2. Assert: Type(*searchValue*) is String.
3. Assert: *fromIndex* is a non-negative integer.
4. Let *len* be the length of *string*.
5. If *searchValue* is the empty String and *fromIndex* ≤ *len*, r
6. Let *searchLen* be the length of *searchValue*.



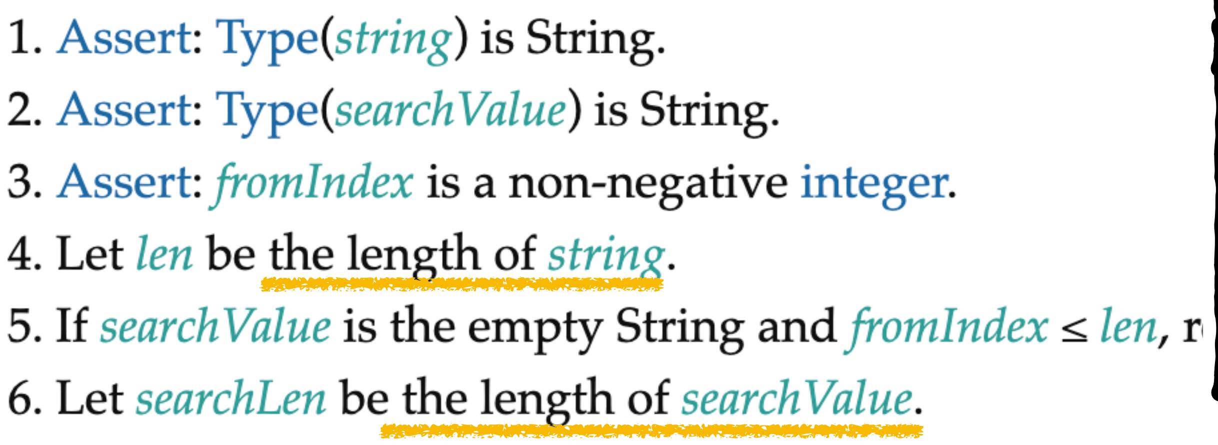
Parsing Rules

“the length of” ~> e → LEN 30

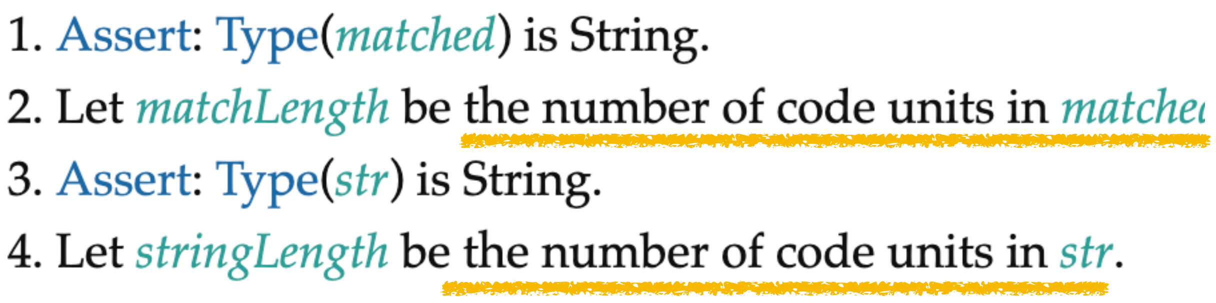


1. Assert: Type(*matched*) is String.
2. Let *matchLength* be the number of code units in *matche*
3. Assert: Type(*str*) is String.
4. Let *stringLength* be the number of code units in *str*.

#3: Consistent



1. Assert: Type(*string*) is String.
2. Assert: Type(*searchValue*) is String.
3. Assert: *fromIndex* is a non-negative integer.
4. Let *len* be the length of *string*.
5. If *searchValue* is the empty String and *fromIndex* ≤ *len*, r
6. Let *searchLen* be the length of *searchValue*.



1. Assert: Type(*matched*) is String.
2. Let *matchLength* be the number of code units in *matche*
3. Assert: Type(*str*) is String.
4. Let *stringLength* be the number of code units in *str*.

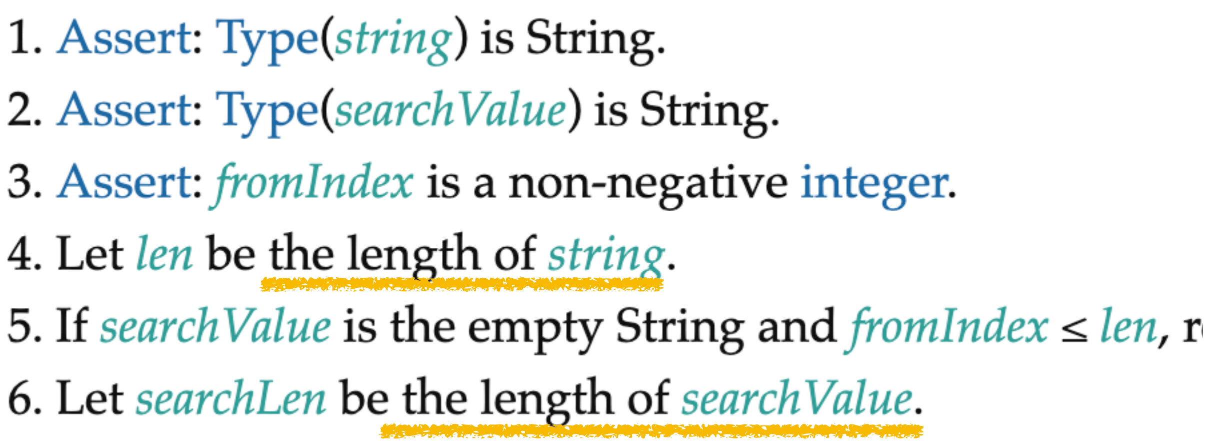
Parsing Rules

"the length of" ~> e → LEN **30**

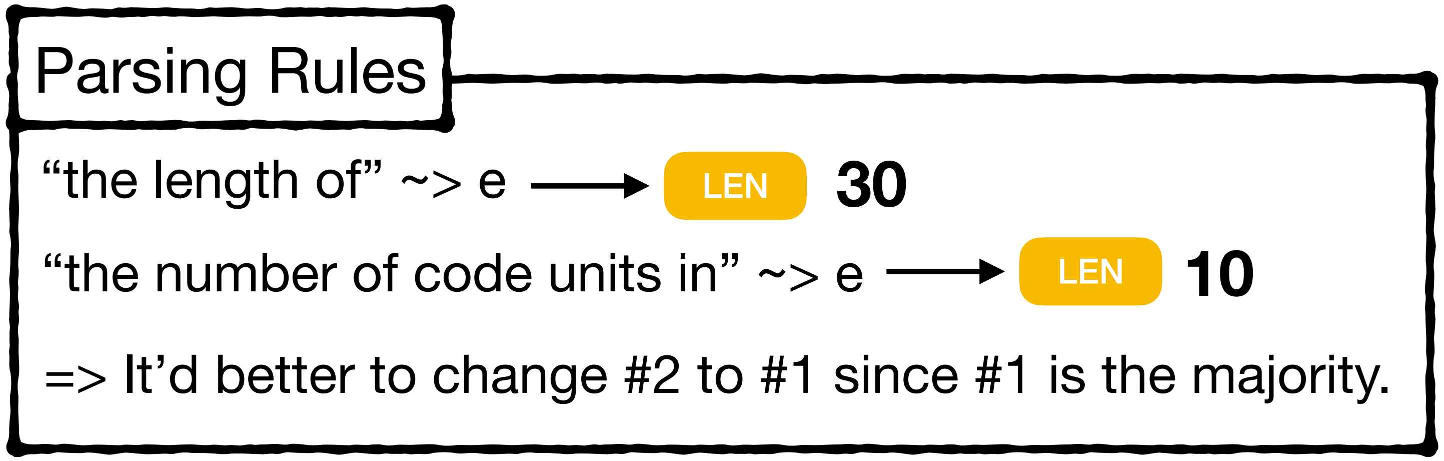
"the number of code units in" ~> e → LEN **10**

=> It'd better to change #2 to #1 since #1 is the majority.

#3: Consistent



1. Assert: Type(*string*) is String.
2. Assert: Type(*searchValue*) is String.
3. Assert: *fromIndex* is a non-negative integer.
4. Let *len* be the length of *string*.
5. If *searchValue* is the empty String and *fromIndex* ≤ *len*, r
6. Let *searchLen* be the length of *searchValue*.

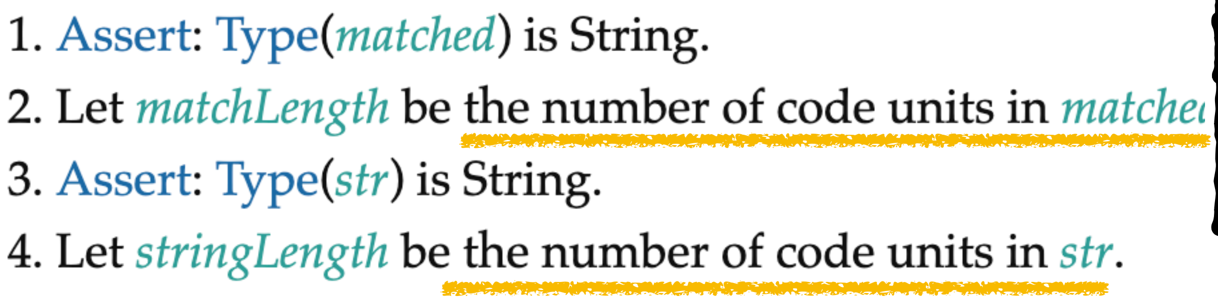


Parsing Rules

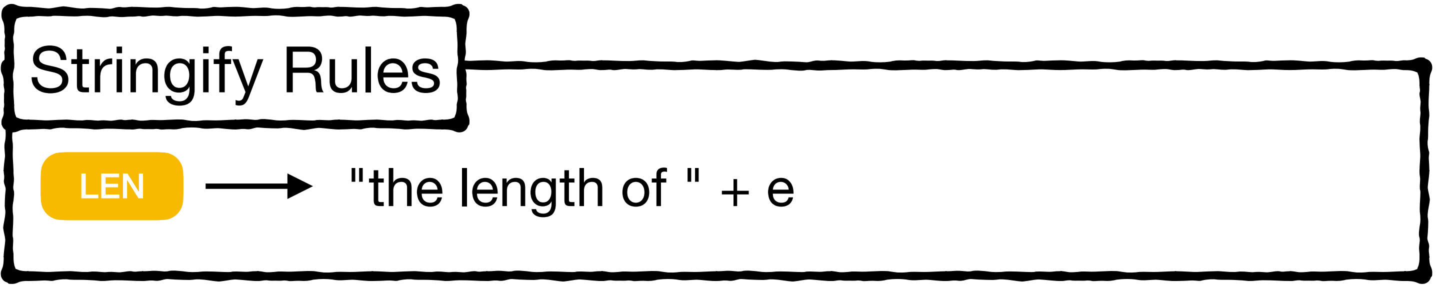
“the length of” ~> e → LEN **30**

“the number of code units in” ~> e → LEN **10**

=> It’d better to change #2 to #1 since #1 is the majority.



1. Assert: Type(*matched*) is String.
2. Let *matchLength* be the number of code units in *matche*
3. Assert: Type(*str*) is String.
4. Let *stringLength* be the number of code units in *str*.



Stringify Rules

LEN → "the length of " + e

#3: Consistent

URL: https://github.com/tc39/ecma262

#	Phrases	Status	PR#
1	the length of string	Already Fixed, Reported	#2746, #2788
2	SDO invocation	Already Fixed	#2626
3	perform/ call	Already Fixed	#2547
4	component property	Reported	#2789
5	the sole element	Reported	#2790
6	empty condition	Reported	#2790
7	the active function object	Reported	#2790
8	the running execution context	Reported	#2790
9	append/ add	Reported	#2790

Conventions of ECMA-262

Background: Syntax

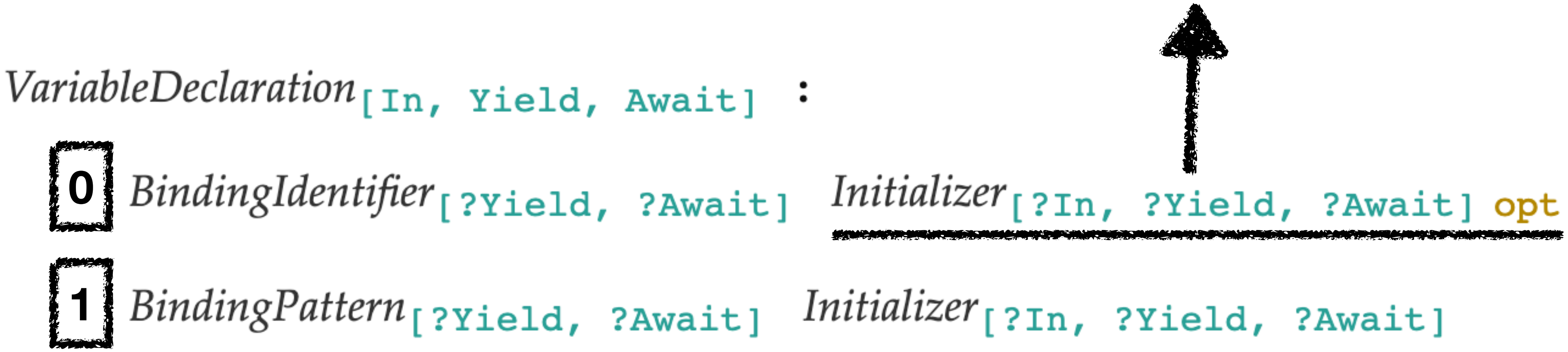
VariableDeclaration`[In, Yield, Await]` :

$\boxed{0}$ *BindingIdentifier*`[?Yield, ?Await]` *Initializer*`[?In, ?Yield, ?Await]` `opt`

$\boxed{1}$ *BindingPattern*`[?Yield, ?Await]` *Initializer*`[?In, ?Yield, ?Await]`

Background: Syntax

two cases for the first alternative



VariableDeclaration$_{\texttt{[In, Yield, Await]}}$:

0 *BindingIdentifier*$_{\texttt{[?Yield, ?Await]}}$ *Initializer*$_{\texttt{[?In, ?Yield, ?Await]}}$ opt

1 *BindingPattern*$_{\texttt{[?Yield, ?Await]}}$ *Initializer*$_{\texttt{[?In, ?Yield, ?Await]}}$

Background: Syntax

two cases for the first alternative

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

$\boxed{0}$ *BindingIdentifier*`[?Yield, ?Await]` *Initializer*`[?In, ?Yield, ?Await]` `opt`

$\boxed{1}$ *BindingPattern*`[?Yield, ?Await]` *Initializer*`[?In, ?Yield, ?Await]`

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

$\boxed{0, 0}$ *BindingIdentifier*`[?Yield, ?Await]`

$\boxed{0, 1}$ *BindingIdentifier*`[?Yield, ?Await]` *Initializer*`[?In, ?Yield, ?Await]`

$\boxed{1, 0}$ *BindingPattern*`[?Yield, ?Await]` *Initializer*`[?In, ?Yield, ?Await]`

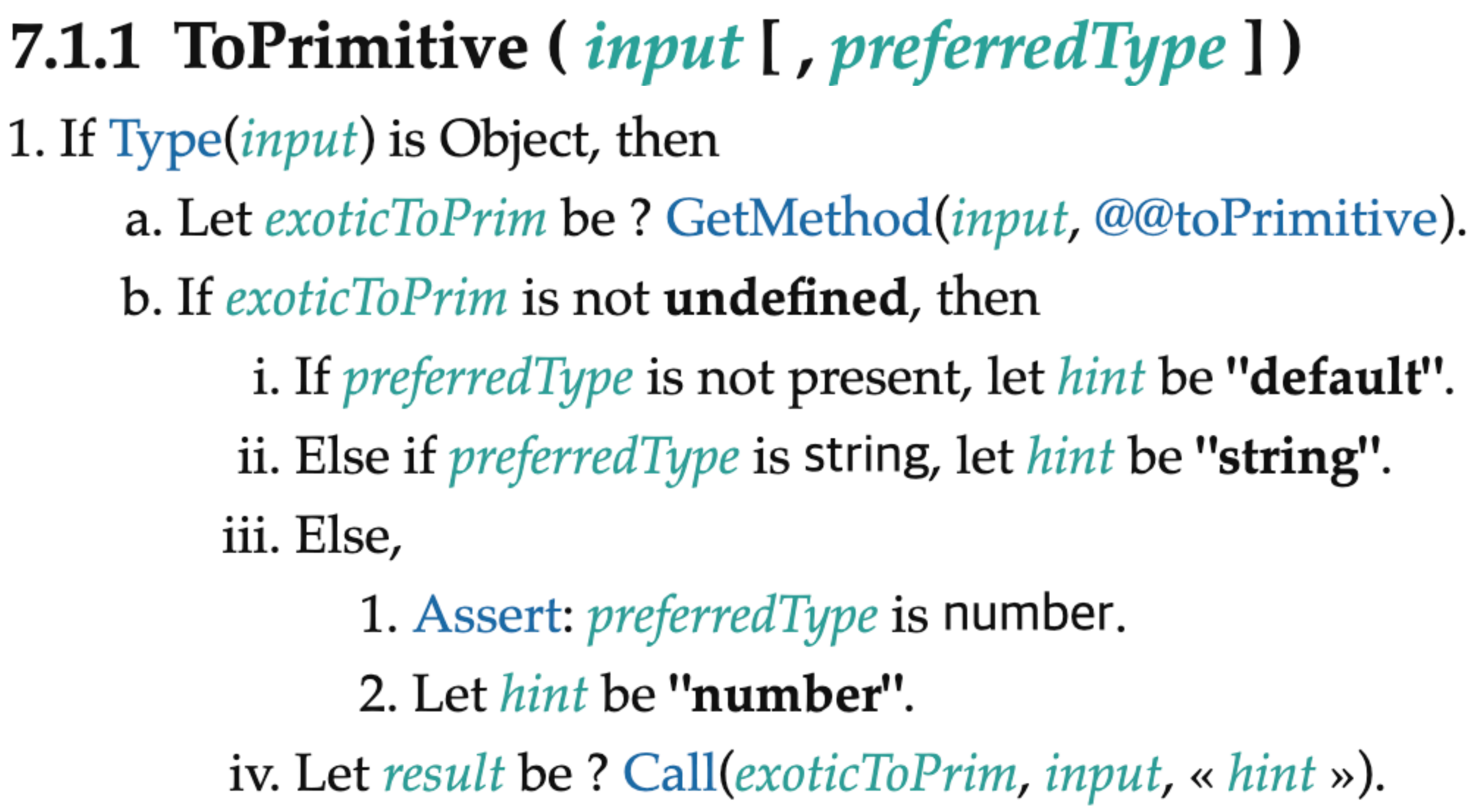
Background: Algorithms

Abstract Operation

Name

Parameters (may be optional)

Header



```
7.1.1 ToPrimitive ( input [ , preferredType ] )
1. If Type(input) is Object, then
    a. Let exoticToPrim be ? GetMethod(input, @@toPrimitive).
    b. If exoticToPrim is not undefined, then
        i. If preferredType is not present, let hint be "default".
        ii. Else if preferredType is string, let hint be "string".
        iii. Else,
            1. Assert: preferredType is number.
            2. Let hint be "number".
        iv. Let result be ? Call(exoticToPrim, input, « hint »).
```

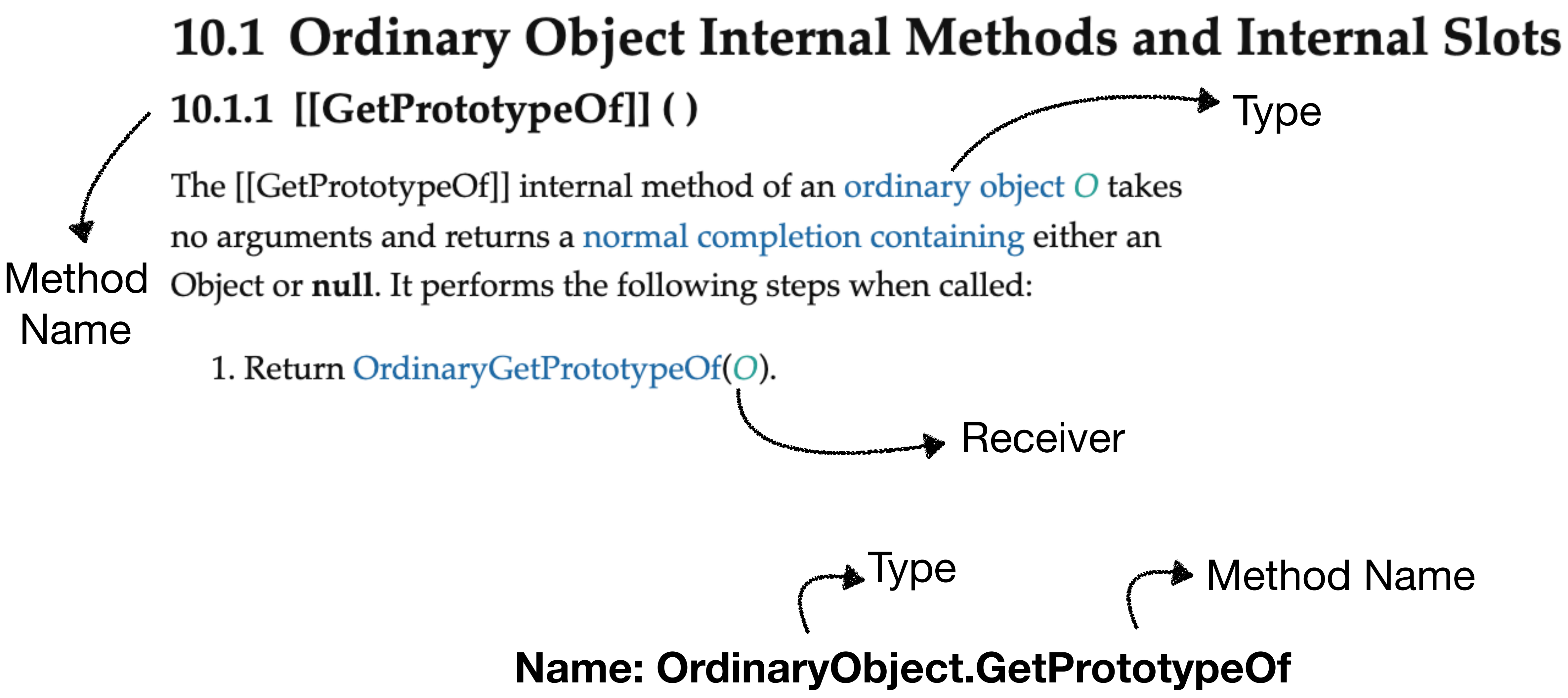
Ordered Steps

Name

Name: ToPrimitive

Background: Algorithms

Method-like Abstract Operation



10.1 Ordinary Object Internal Methods and Internal Slots

10.1.1 [[GetPrototypeOf]] ( )

The [[GetPrototypeOf]] internal method of an ordinary object *O* takes no arguments and returns a normal completion containing either an Object or **null**. It performs the following steps when called:

1. Return OrdinaryGetPrototypeOf(*O*).

Method Name

Type

Receiver

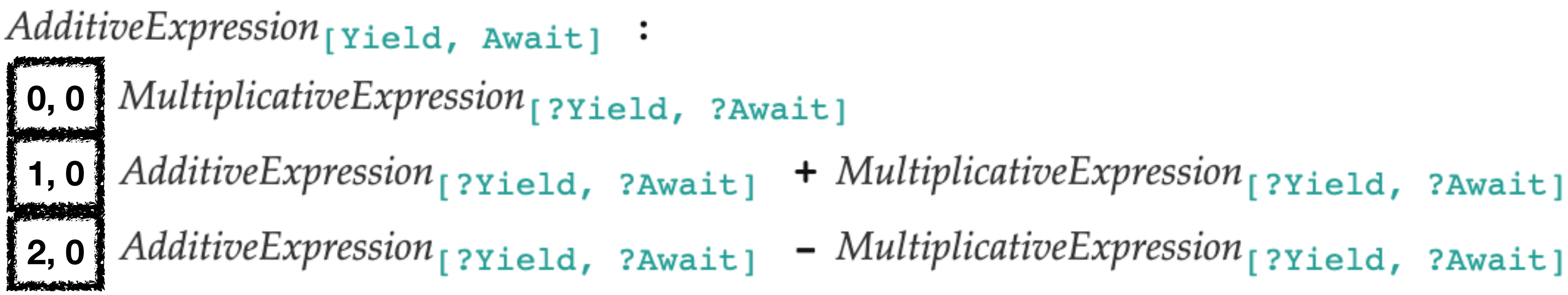
Type

Method Name

Name: OrdinaryObject.GetPrototypeOf

Background: Algorithms

Syntax-Directed Operation



AdditiveExpression[Yield, Await] :

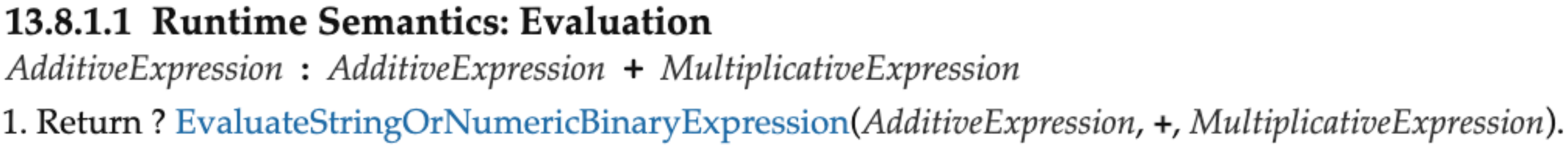
0, 0 *MultiplicativeExpression*[?Yield, ?Await]

1, 0 *AdditiveExpression*[?Yield, ?Await] + *MultiplicativeExpression*[?Yield, ?Await]

2, 0 *AdditiveExpression*[?Yield, ?Await] - *MultiplicativeExpression*[?Yield, ?Await]

Alternatives

Method Name



13.8.1.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

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

Alternative

Method Name

Name: AdditiveExpression[1,0].Evaluation

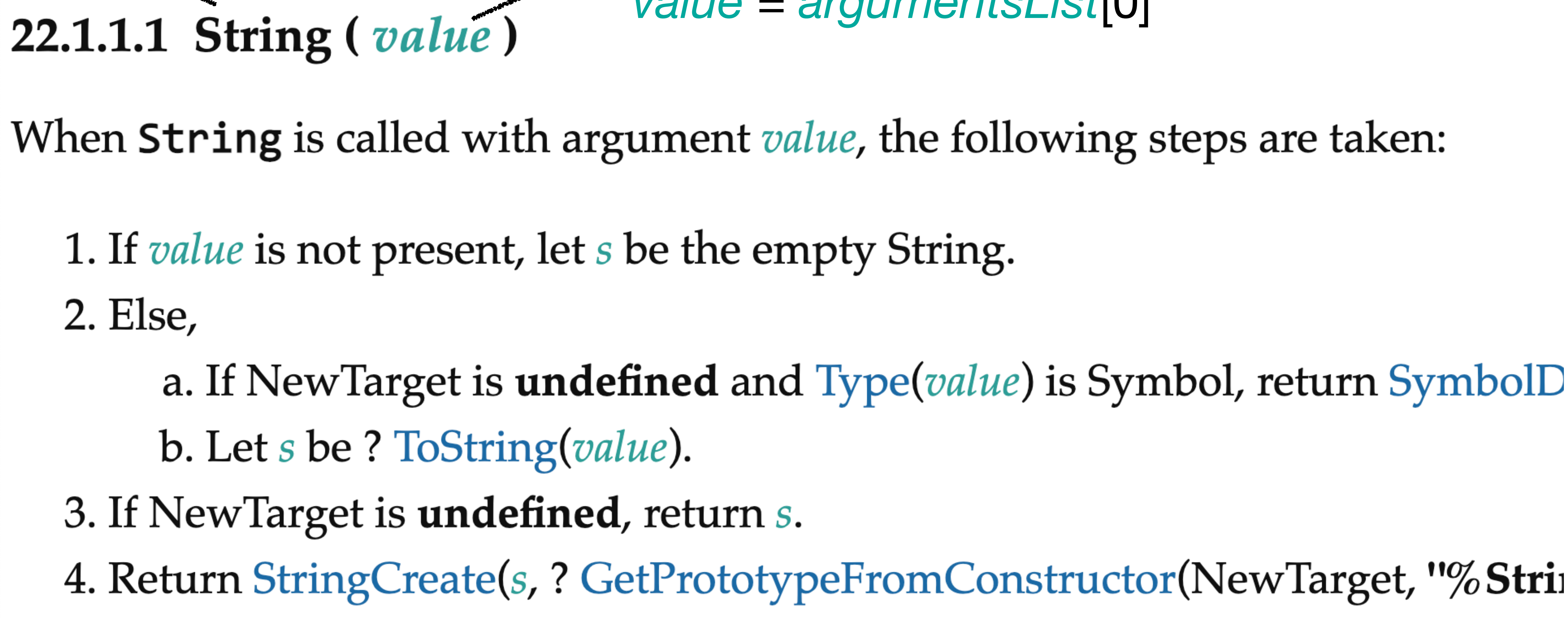
Background: Algorithms

Built-in Operation

Global Name

Parameters are fixed to *this*, *argumentsList*, *NewTarget*

value = *argumentsList*[0]



22.1.1.1 String (*value*)

When `String` is called with argument *value*, the following steps are taken:

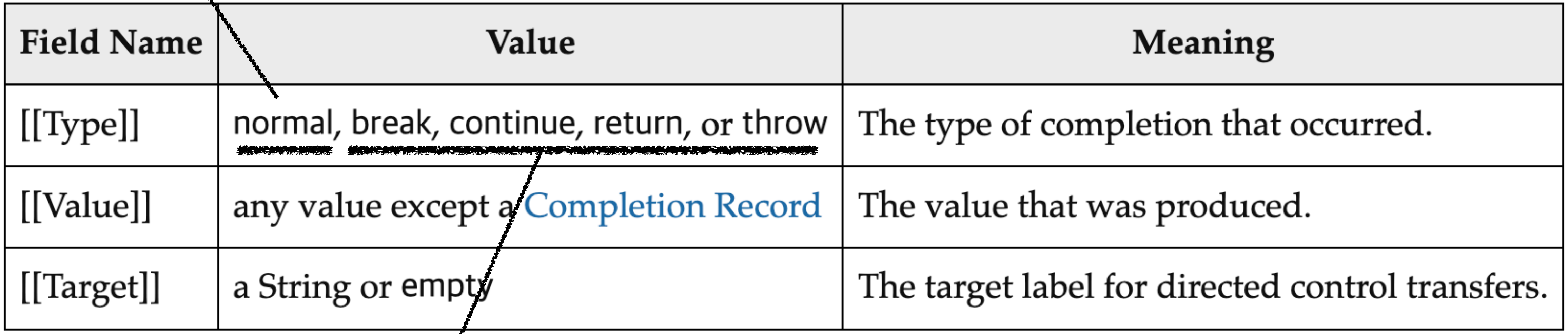
1. If *value* is not present, let *s* be the empty String.
2. Else,
   a. If NewTarget is **undefined** and Type(*value*) is Symbol, return SymbolD
   b. Let *s* be ? ToString(*value*).
3. If NewTarget is **undefined**, return *s*.
4. Return StringCreate(*s*, ? GetPrototypeFromConstructor(NewTarget, "%**Stri**

Global Name

Name: INTRINSICS.String

Background: Completion Record

Normal Completion **N(Value)**



Field Name	Value	Meaning
[[Type]]	normal, break, continue, return, or throw	The type of completion that occurred.
[[Value]]	any value except a Completion Record	The value that was produced.
[[Target]]	a String or empty	The target label for directed control transfers.

From: https://tc39.es/ecma262/#sec-completion-record-specification-type

Abrupt Completion **comp[Type/Target](Value)**

Live Demo

Manual

- Server: `run web` command in sbt.
- Client: `npm start` command in console.
- Default ports of server and client are 8080 and 3000, respectively.

PLRG
Programming Language
Research Group

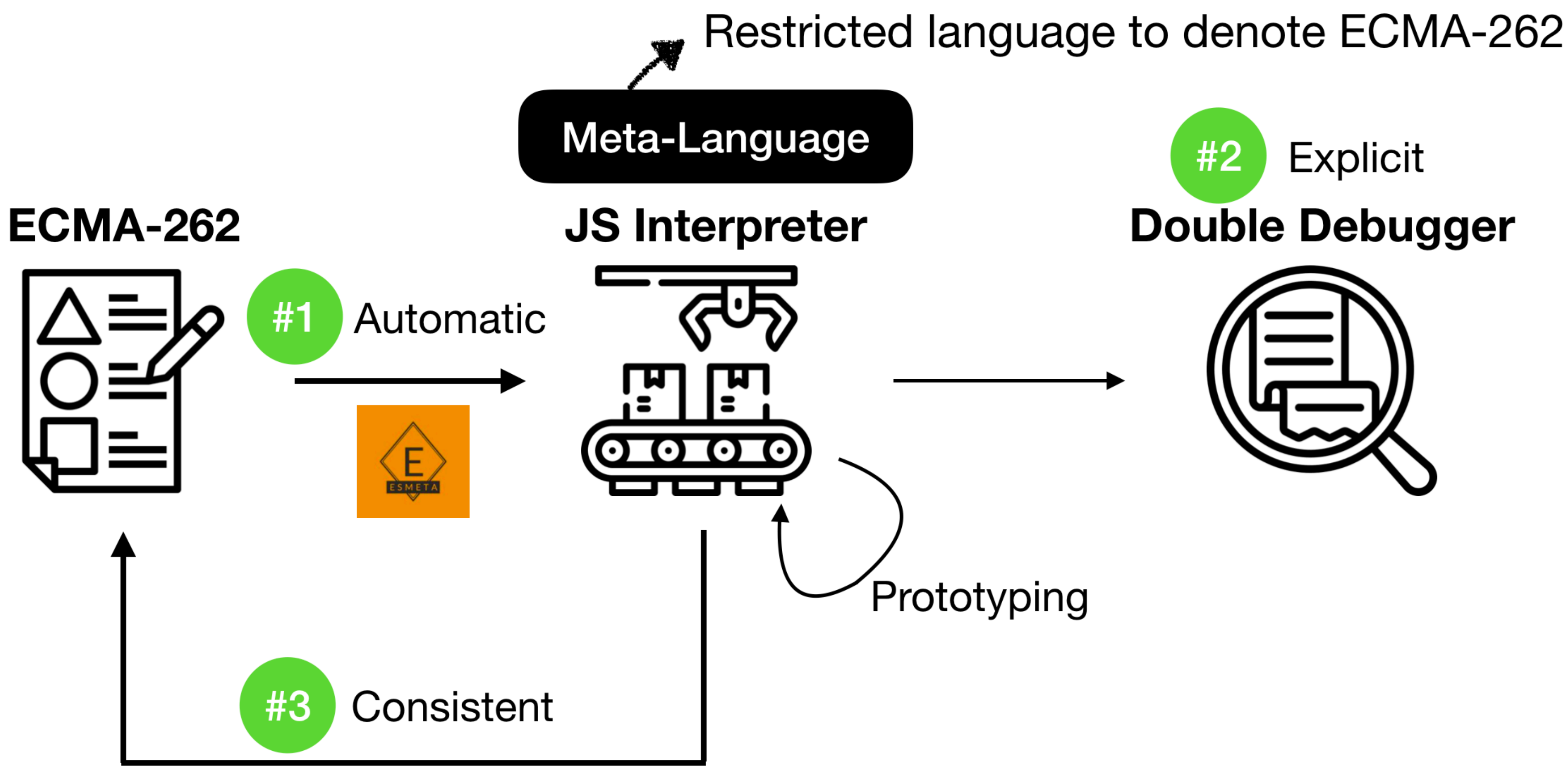
Goal: Understand the Addition

- 1 + 1
- '1' + 1
- 1n + 1
- ({"valueOf": () => 1}) + 1

JavaScript

```
var x = ""; var y = ({valueOf: () => { return x = 3 }) + x;
```

Q. What are the values of x and y?



Restricted language to denote ECMA-262

Meta-Langauge

ECMA-262

JS Interpreter

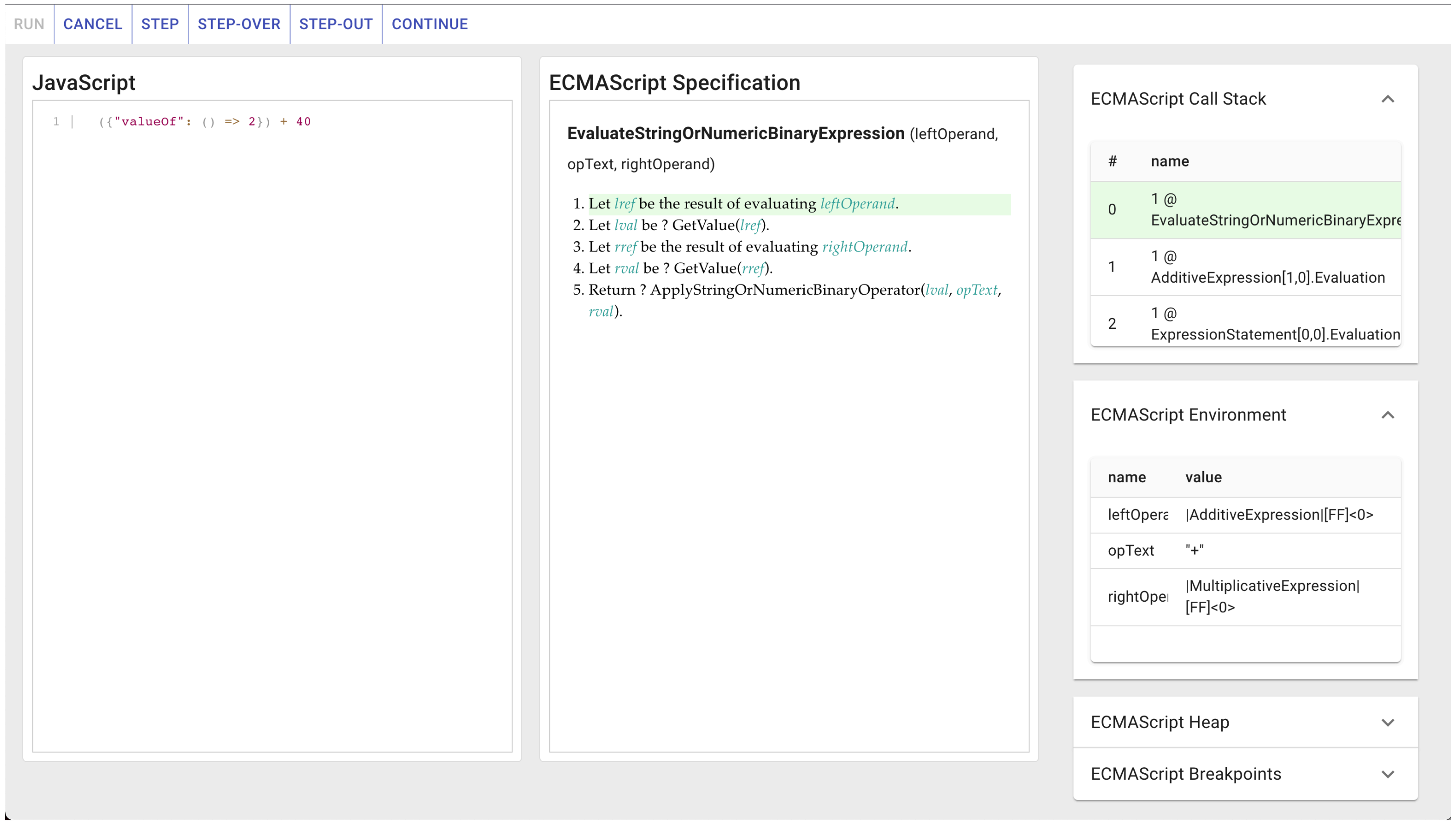
#2 Explicit

Double Debugger

#1 Automatic

Prototyping

#3 Consistent



JavaScript

```
({"valueOf": () => 2}) + 40
```

ECMAScript Specification

EvaluateStringOrNumericBinaryExpression (leftOperand, opText, rightOperand)

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

ECMAScript Call Stack

#	name
0	1 @ EvaluateStringOrNumericBinaryExpre
1	1 @ AdditiveExpression[1,0].Evaluation
2	1 @ ExpressionStatement[0,0].Evaluation

ECMAScript Environment

name	value
leftOpera	|AdditiveExpression|[FF]<0>
opText	"+"
rightOpe	|MultiplicativeExpression|[FF]<0>

ECMAScript Heap

ECMAScript Breakpoints