ECMAScript仕様の読み方ガイド ～比較演算子編～

syumai

Meguro.es # 26 (2024/1/25)

自己紹介

syumai

- ECMAScript 仕様輪読会 主催
- 株式会社ベースマキナで管理画面のSaaSを開発中
  - GoでGraphQLサーバー (gqlgen) や TypeScriptでフロントエンドを書いています
- Software Design 12月号からCloudflare Workersの連載をしてます

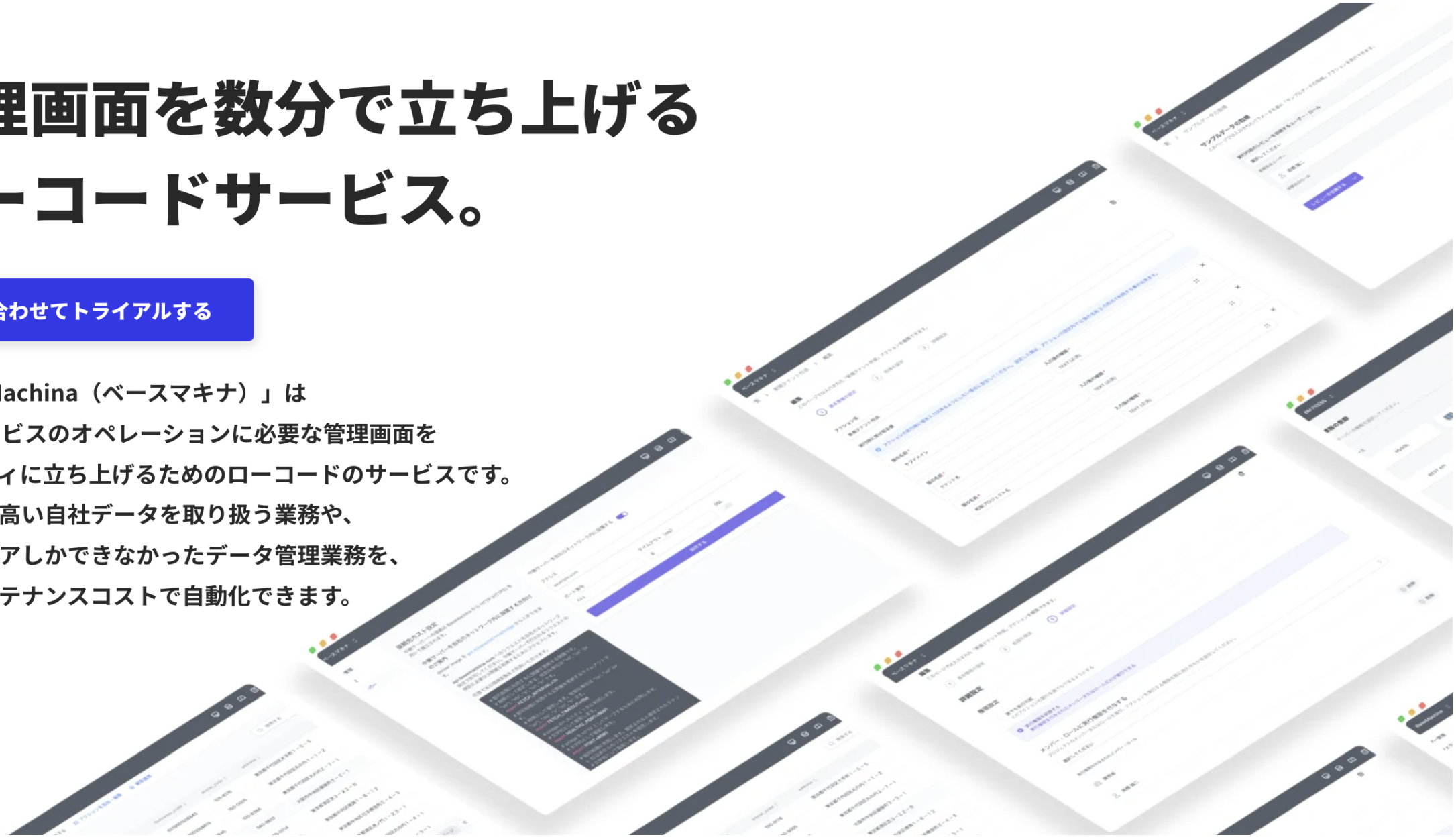
Twitter: @__syumai

Website: https://syum.ai

管理画面を数分で立ち上げる
ローコードサービス。

問い合わせてトライアルする

「BaseMachina（ベースマキナ）」は
Webサービスのオペレーションに必要な管理画面を
スピーディに立ち上げるためのローコードのサービスです。
機密性の高い自社データを取り扱う業務や、
エンジニアしかできなかったデータ管理業務を、
低いメンテナンスコストで自動化できます。



ベースマキナとは？

- DBやAPIの接続設定 & 呼び出し設定をするだけで、簡単にUI生成が行える管理画面SaaS
- API呼び出しへの権限設定や、レビュー依頼 / 承認機能も簡単に使えます
- https://about.basemachina.com

本日話すこと

- ECMAScriptの仕様の読み方について
  - 比較演算子の仕様を読みながら、登場順に紹介

ECMAScript仕様書

ECMAScript仕様書

TC39のサイトから閲覧可能

- https://tc39.es/
- TC39 -> 標準化団体Ecma InternationalのTechnical Committee 39
  - Ecma Internationalには他にも沢山のCommitteeがある
    - https://ecma-international.org/technical-committees/
    - サイトを見る限り54まであるらしい

TC39

JavaScriptを定義する。

TC39

Ecma InternationalのTC39は、JavaScript開発者、実装者、専門家などのグループで、
JavaScriptの仕様をメンテナンスし発展させるためにコミュニティと協力しています。

我々は
ecma INTERNATIONAL
所属の委員会です。

ECMAScriptはECMA-262の仕様

TC39

ECMAScript

General | Activities | Task Groups | Published Standards

Number	Description	Last Change
ECMA-262	ECMAScript® 2023 language specification	June 2023
ECMA-402	ECMAScript® 2023 internationalization API specification	June 2023
ECMA-404	The JSON data interchange syntax	December 2017
ECMA-414	ECMAScript® specification suite	December 2017

https://ecma-international.org/technical-committees/tc39/?tab=published-standards

ECMAScript仕様書

- ECMA-262は毎年版が更新される
  - 現在の最新はECMAScript 2023 (第14版)
- Ecma Internationalのサイトから、PDFまたはHTMLで閲覧可能
  - https://ecma-international.org/publications-and-standards/standards/ecma-262/
    - 今回はES2023のHTML版を見ます
      - https://262.ecma-international.org/14.0/
  - 古い仕様のアーカイブもここで読める
- 版が切られる前の最新のドラフトも閲覧可能
  - https://tc39.es/ecma262/

比較演算子の仕様を読む

ゴール

- 以下のコードの振る舞いを説明できる
  - `0 < 1` -> true
  - `1 < 0` -> false
  - `"a" < "b"` -> true
  - `"b" < "a"` -> false
  - `undefined < 1` -> false
  - `undefined < -1` -> false
  - `null < 1` -> true
  - `null < -1` -> false
  - `1 < ({})` -> false
  - `({}) < 1` -> false

文法

13.10 Relational Operators

> NOTE 1 The result of evaluating a relational operator is always of type Boolean, reflecting whether the relationship named by the operator holds between its two operands.

Syntax

RelationalExpression[In, Yield, Await] :

ShiftExpression[?Yield, ?Await]

RelationalExpression[?In, ?Yield, ?Await] < *ShiftExpression*[?Yield, ?Await]

RelationalExpression[?In, ?Yield, ?Await] > *ShiftExpression*[?Yield, ?Await]

RelationalExpression[?In, ?Yield, ?Await] <= *ShiftExpression*[?Yield, ?Await]

RelationalExpression[?In, ?Yield, ?Await] >= *ShiftExpression*[?Yield, ?Await]

RelationalExpression[?In, ?Yield, ?Await] instanceof *ShiftExpression*[?Yield, ?Await]

[+In] *RelationalExpression*[+In, ?Yield, ?Await] in *ShiftExpression*[?Yield, ?Await]

[+In] *PrivateIdentifier* in *ShiftExpression*[?Yield, ?Await]

13: ECMAScript Language: Expressions 内の Relational Operators に記載

文法

- 文脈自由文法で表現される
- `Syntax` に記載される *斜体文字* は非終端記号
  - ここでは *RelationalExpression* など
- `:` は非終端記号の定義 (文法生成規則とも言う)
  - 左辺の非終端記号を構成するトークン列が `:` の右辺に示される
    - 右辺のトークン列には、非終端記号および終端記号が並ぶ
  - 左辺の非終端記号が同一となる生成規則が複数並ぶ場合、縦に並べて記載される
    - 繰り返し同じ記載をするのを避けるための省略記法
      - 共通した左辺を持つ複数の生成規則を指して代替 (alternatives) とも言う

Syntax

RelationalExpression[In, Yield, Await] :

ShiftExpression[?Yield, ?Await]

RelationalExpression[?In, ?Yield, ?Await] < *ShiftExpression*[?Yield, ?Await]

文法

- 等幅フォントの文字は終端記号
  - ここでは `<`
- 終端記号は、非終端記号と異なり、記載された通りにソースコード中に現れる
  - 非終端記号は最終的にこれらの記号に辿り着くため、終端記号と呼ばれる

Syntax

RelationalExpression[In, Yield, Await] :

ShiftExpression[?Yield, ?Await]

RelationalExpression[?In, ?Yield, ?Await] < *ShiftExpression*[?Yield, ?Await]

セマンティクス

Semantics (セマンティクス)

プログラミングでは、**セマンティクス**とは、コードの断片の意味を指します。たとえば、「JavaScript でその行を実行すると、どのような効果があるのか？」、「その HTML 要素には、どのような目的や役割があるのか？」 (「どのように見えるのか？」ではなく)。

MDN Web Docs 用語集: ウェブ関連用語の定義 > Semantics (セマンティクス)

https://developer.mozilla.org/ja/docs/Glossary/Semantics

13.10.1 Runtime Semantics: Evaluation

RelationalExpression : *RelationalExpression* < *ShiftExpression*

1. Let *lref* be ? Evaluation of *RelationalExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be ? Evaluation of *ShiftExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *r* be ? IsLessThan(*lval*, *rval*, **true**).
6. If *r* is **undefined**, return **false**. Otherwise, return *r*.

13.10.1 Runtime Semantics: Evaluation に記載

Syntax

RelationalExpression[In, Yield, Await] :

Syntax-Directed Operations (4)

RelationalExpression[?In, ?Yield, ?Await] < *ShiftExpression*[?Yield, ?Await]

SYNTAX-DIRECTED OPERATIONS FOR

RelationalExpression : RelationalExpression < ShiftExpression

8.4.2 IsFunctionDefinition

8.6.4 AssignmentTargetType

13.10.1 Evaluation

15.10.2 HasCallInTailPosition

セマンティクスが気になる文法生成規則にホバーすると出てくるリンクから飛べます

比較演算子 `<` のセマンティクス

- 文法生成規則に対応するRuntime Semanticsが定義されている
  - 実行時 (runtime) のセマンティクスを示すためのアルゴリズム列
- セマンティクスはStatic Semanticsによって定義されるケースもある
  - 文脈自由文法だけでは入力列が有効か判断できないようなケースで使う
    - Runtime Semanticsの前段のバリデーション的な用法が多い (Early Errors等)

13.10.1 Runtime Semantics: Evaluation

RelationalExpression : *RelationalExpression* < *ShiftExpression*

1. Let *lref* be ? Evaluation of *RelationalExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be ? Evaluation of *ShiftExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *r* be ? IsLessThan(*lval*, *rval*, **true**).
6. If *r* is **undefined**, return **false**. Otherwise, return *r*.

アルゴリズムステップを読む (`<` のRuntime Semantics)

1. Let lref be ? Evaluation of *RelationalExpression*.

- Let lref be
  - → エイリアス宣言。アルゴリズムステップ中でのみ有効。
    - ここでは、`? ...` 以下の結果に `lref` と言うエイリアスを付けている

アルゴリズムステップを読む (`<` のRuntime Semantics)

1. Let lref be ? Evaluation of *RelationalExpression*.

- Evaluation of *RelationalExpression*.
  - → `Evaluation` と言う **Syntax-Directed Operation** の呼び出し
    - Syntax-Directed Operationは、 `Operation` of `Syntax` の形式で呼ばれる
    - 実は、今読んでいる箇所が *RelationalExpression* に対する `Evaluation` 呼び出しそのもののアルゴリズムです
      - ※ *RelationalExpression* には複数の表現があり、それぞれのセマンティクスが定義されている点に注意

アルゴリズムステップを読む (`<` のRuntime Semantics)

1. Let lref be ? Evaluation of *RelationalExpression*.

- ?
  - → ReturnIfAbruptの省略記法
    - これが書かれている箇所の右側の処理が失敗した場合、ステップをここで中断し、失敗した結果を返す

アルゴリズムステップを読む (`<` のRuntime Semantics)

1. Let lref be ? Evaluation of *RelationalExpression*.

まとめると

→ *RelationalExpression* に対して `Evaluation` を評価し、その結果に `lref` と言うエイリアスを割り当てる。 `Evaluation` の評価に失敗したら、失敗した結果を返して中断する。

アルゴリズムステップを読む (`<` のRuntime Semantics)

2 ~ 4 はスキップします

13.10.1 Runtime Semantics: Evaluation

RelationalExpression : *RelationalExpression* < *ShiftExpression*

1. Let *lref* be ? Evaluation of *RelationalExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be ? Evaluation of *ShiftExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *r* be ? IsLessThan(*lval*, *rval*, **true**).
6. If *r* is **undefined**, return **false**. Otherwise, return *r*.

アルゴリズムステップを読む (`<` のRuntime Semantics)

5. Let r be ? IsLessThan(lval, rval, **true**).

- IsLessThan(lval, rval, **true**).
  - → `IsLessThan` と言う **Abstract Operation** の呼び出し
    - Abstract Operationは、`()` に引数を伴う、関数に似た形式で呼び出される

6. If r is **undefined**, return **false**. Otherwise, return r.

- → `r` が `undefined` なら `false` を、それ以外なら `r` の値を返す

Abstract OperationとSyntax-Directed Operationのおさらい

- ECMAScriptの仕様内で記述されるアルゴリズムは、Abstract OperationかSyntax-Directed Operationを通じて呼び出される。
- **Abstract Operation**は `Operation(arg1, arg2)` といった関数的なスタイルで、**値**を受け取ってアルゴリズムを処理する。
- **Syntax-Directed Operation**は `Operation` of `Syntax` のスタイルで、**文法生成規則**を受け取ってアルゴリズムを処理する。
  - 文法生成規則に複数の代替 (alternatives) が存在する場合、それぞれに対するアルゴリズムが定義される

IsLessThan の仕様を読む

7.2.13 IsLessThan (*x*, *y*, *LeftFirst*)

The abstract operation IsLessThan takes arguments *x* (an ECMAScript language value), *y* (an ECMAScript language value), and *LeftFirst* (a Boolean) and returns either a normal completion containing either a Boolean or **undefined**, or a throw completion. It provides the semantics for the comparison $x < y$, returning **true**, **false**, or **undefined** (which indicates that at least one operand is **NaN**). The *LeftFirst* flag is used to control the order in which operations with potentially visible side-effects are performed upon *x* and *y*. It is necessary because ECMAScript specifies left to right evaluation of expressions. If *LeftFirst* is **true**, the *x* parameter corresponds to an expression that occurs to the left of the *y* parameter's corresponding expression. If *LeftFirst* is **false**, the reverse is the case and operations must be performed upon *y* before *x*. It performs the following steps when called:

1. If *LeftFirst* is **true**, then
    a. Let *px* be ? ToPrimitive(*x*, number).
    b. Let *py* be ? ToPrimitive(*y*, number).
2. Else,
    a. NOTE: The order of evaluation needs to be reversed to preserve left to right evaluation.
    b. Let *py* be ? ToPrimitive(*y*, number).
    c. Let *px* be ? ToPrimitive(*x*, number).
3. If *px* is a String and *py* is a String, then
    a. Let *lx* be the length of *px*.
    b. Let *ly* be the length of *py*.
    c. For each integer *i* such that $0 \le i < \min(lx, ly)$, in ascending order, do
        i. Let *cx* be the numeric value of the code unit at index *i* within *px*.
        ii. Let *cy* be the numeric value of the code unit at index *i* within *py*.
        iii. If $cx < cy$, return **true**.
        iv. If $cx > cy$, return **false**.
    d. If $lx < ly$, return **true**. Otherwise, return **false**.
4. Else,
    a. If *px* is a BigInt and *py* is a String, then
        i. Let *ny* be StringToBigInt(*py*).
        ii. If *ny* is **undefined**, return **undefined**.
        iii. Return BigInt::lessThan(*px*, *ny*).
    b. If *px* is a String and *py* is a BigInt, then
        i. Let *nx* be StringToBigInt(*px*).
        ii. If *nx* is **undefined**, return **undefined**.
        iii. Return BigInt::lessThan(*nx*, *py*).
    c. NOTE: Because *px* and *py* are primitive values, evaluation order is not important.
    d. Let *nx* be ? ToNumeric(*px*).
    e. Let *ny* be ? ToNumeric(*py*).
    f. If Type(*nx*) is Type(*ny*), then
        i. If *nx* is a Number, then
            1. Return Number::lessThan(*nx*, *ny*).
        ii. Else,
            1. Assert: *nx* is a BigInt.
            2. Return BigInt::lessThan(*nx*, *ny*).
    g. Assert: *nx* is a BigInt and *ny* is a Number, or *nx* is a Number and *ny* is a BigInt.
    h. If *nx* or *ny* is **NaN**, return **undefined**.
    i. If *nx* is $-\infty_{\mathbb{F}}$ or *ny* is $+\infty_{\mathbb{F}}$, return **true**.
    j. If *nx* is $+\infty_{\mathbb{F}}$ or *ny* is $-\infty_{\mathbb{F}}$, return **false**.
    k. If $\mathbb{R}(nx) < \mathbb{R}(ny)$, return **true**; otherwise return **false**.

https://262.ecma-international.org/14.0/#sec-islessthan

死ぬほど長い

時間がないのでざっくり説明

`IsLessThan(x, y, LeftFirst)` に対して、

1. x, y に `ToPrimitive` を適用して、値をプリミティブ型にする。
2. String同士の組なら、同じindex同士のcode unitの大きさで比較して終了。
3. BigIntとStringの組なら、BigInt側に揃えて比較して終了。
    - StringからBigIntに変換を試みる。
4. `ToPrimitive` を適用した x, y に、更に `ToNumeric` を適用する。
5. Number同士またはBigInt同士の組となった場合、それらを比較して終了。
6. x, y のいずれかが `NaN` であれば、 `undefined` を返して終了。
7. `−∞ < +∞` なら `true` 、 `+∞ < −∞` なら `false` を返して終了。
8. x, y を実数に変換したものを比較した結果を返して終了。

冒頭の例を振り返る

- `0 < 1` -> true
- `1 < 0` -> false
- `"a" < "b"` -> true
- `"b" < "a"` -> false

これらは、Number同士の組、およびString同士の組なので、特に懸念なし

冒頭の例を振り返る

- `undefined < 1` -> false
- `undefined < -1` -> false

これは、

- `ToPrimitive(undefined)` が `undefined`
- `ToNumeric(undefined)` が `NaN`

となるため、

- `NaN < 1`
- `NaN < -1`

となり、

- IsLessThanが `undefined` を返すため、 `Evaluation` で `false` を返す

13.10.1 Runtime Semantics: Evaluation

RelationalExpression : *RelationalExpression* < *ShiftExpression*

1. Let *lref* be ? Evaluation of *RelationalExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be ? Evaluation of *ShiftExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *r* be ? IsLessThan(*lval*, *rval*, **true**).
6. If *r* is **undefined**, return **false**. Otherwise, return *r*.

13.10.1 Runtime Semantics: Evaluation の再掲

冒頭の例を振り返る

- `null < 1` -> true
- `null < -1` -> false

これは、

- `ToPrimitive(null)` が `null`
- `ToNumeric(null)` が `0`

となるため、

- `0 < 1`
- `0 < -1`

となり、

- IsLessThanが `undefined` ではない値を返すため、 `Evaluation` で `true` および `false` を返す

冒頭の例を振り返る

- `1 < ({})` -> false
- `({}) < 1` -> false

これは、

- `ToPrimitive({})` が `"[object Object]"`
- `ToNumeric("[object Object]")` が `NaN`

となるため、

- `NaN < 1`
- `NaN < -1`

となり、

- IsLessThanが `undefined` を返すため、 `Evaluation` で `false` を返す

まとめ

- ECMAScriptの仕様を読むには、知っておくべき概念がいくつかある
  - セマンティクスは、アルゴリズムステップで示される
  - アルゴリズムステップは、Syntax-Directed Operation、Abstract Operationを通じて呼び出される
  - その他色々ありますが、今回は紹介し切れていません
- ECMAScript仕様書には、パーマリンクが大量に貼ってあって便利
- 一見複雑そうに見えるが、ルールがわかってきたら意外と読める

宣伝

- ECMAScriptの仕様を読むのに興味が出た方は、ぜひECMAScript仕様輪読会にお越しください！
  - 2週に1回、火曜に開催しています
  - https://esspec.connpass.com/

ご清聴ありがとうございました！

おまけ

ECMAScriptの仕様を読むにあたって便利なリンク集

- esmeta
  - https://github.com/es-meta/esmeta
  - ECMAScript仕様書をステップ実行できるツール
- How to Read the ECMAScript Specification
  - https://timothygu.me/es-howto/
  - 読み方の基本的なルールについてよくまとまっています
- Understanding the ECMAScript spec
  - https://v8.dev/blog/tags/understanding-ecmascript
  - V8公式ブログの記事。全4記事に跨って説明しています