Pipe operator

Update and bikeshedding

J. S. Choi
Indiana University
2022-03
GitHub repository

The long-awaited pipe operator is nearly ready for Stage 3. We have one more big hurdle:

We are stuck on bikeshedding its spelling for a crucial piece of its syntax: its topic reference's token.

Why a pipe operator

Developers often transform raw data with a sequence of steps in "dataflows".

Dataflows with prototype method chains benefit from a linear, left-to-right word order:

```
    0           1          2     3
kitchen.getFridge().find(pred)
  .count().toString()
      4          5
```

In contrast, dataflows that use other expressions (especially function calls) result in deeply nested expressions, which have nonlinear zigzagging word orders:

```
  4      2       0           1
count(find(kitchen.getFridge(),
  pred)).toString()
    3          5
```

Developers should be able to express non-`this`-based dataflows as fluent interfaces – linear left-to-right chains – just as they can with `this`-based property/method chains. A pipe operator `|>` would make this possible.

```
    0           1          2     3
kitchen.getFridge() |> find(@, pred)
  |> count(@).toString()
       4          5
```

`|>` creates a lexical context around its RHS, within which it binds `@` (the topic reference) to the result of its LHS.

Developers have been using progressively more APIs with many separately importable functions, acting on objects with smaller prototypes and fewer methods – rather than objects with larger prototypes and many methods. See Firebase JS SDK v9's changes for an example of such an API.

Long road to get here

The pipe operator in JavaScript has a long and twisty history since its first proposal in 2015. There is a history document with more details.

The proposal champions previously were split between two possibilities ("F# style" with tacit unary function calls – and "Hack style" with lexical topic references). But, since the summer of 2021, we have had consensus for Hack style as the way forward due to concerns from the Committee about F# style.

The developer community remains split, but there appears to be large support for pipeline syntax in whatever form we decide on.

Since the winter of 2021, we have also been discussing the pipe operator in the greater context of proposals for dataflow, which also include the bind-this operator, the Extensions syntaxes, partial-function-application (PFA) syntax, and Function.pipe.

Although we will continue this holistic dataflow discussion later in this meeting, a general consensus has formed that the pipe operator with topic reference is worth adding to the language (although it must be added with a call-this operator).

The other last major hurdle is bikeshedding the spelling of the topic reference, which involves several cross-cutting concerns.

Criteria for choosing the topic reference's token

Parsing simplicity
Does the topic make parsing more complex or contextual for computer or human readers?

Visual distinguishability
Is the topic easy to identify when humans quickly scan code? How often will other symbols resemble the topic reference? Does visual noise frequently blur into ASCII soup?

Textual brevity
Does it make code excessively more verbose?

Typing easiness
Is typing it difficult in many keyboard formats?

When judging how each candidate topic token fulfills these criteria, we must weigh its benefits and costs by how often we expect them to occur in code.

Expected Benefit
= Expected Number of Occurrences
× Expected Benefit per Occurrence

Expected Cost
= Expected Number of Occurrences
× Expected Cost per Occurrence

For example, the candidate topic ## and tuple literals #[ ] resemble each other, which has a cost to visual distinguishability. If tuple literals (and therefore that cost) are expected to occur frequently, then the cost must be multiplied by that large number of occurrences.

Candidate topic tokens (1/4)

```
x |> f(@, 0)
x |> f(^^, 0)
x |> f(%%, 0)
x |> f(@@, 0)
x |> f(#_, 0)
x |> f(##, 0)
```

Wiki page (with table)

Issue #91 (very long thread)

^ and % were previously candidates but have been excluded, due to concerns by an implementer about lexing complexity.

also was a candidate but has been excluded. `#[ ]` syntax for tuple literals would unacceptably require `x |> #[0]` to be parenthesized as `x |> (#)[0]`.

Additionally, identifiers like `it`, `$`, `_`, and `$_` were also deemed too hazardous to refactoring.

Candidate topic tokens (2/4)

`x |> f(@, 0)`

@ is the **only** viable single-character token.

To prevent an ASI hazard with `@(expr)` decorator syntax:

```
x |> @(@) // There is no semicolon.
class C {}
```

…we could make the previous lines an **early error**. To compile, the developer must explicitly separate the pipe body from the class:

```
x |> @(@); // Here, both @s are topic references.
class C {}
```

…or explicitly parenthesize the decorated class:

```
x |> (
  @(@) // Here, the first @ is a decorator indicator, not a topic reference.
  class C {}
)
```

Also, developers should be discouraged from putting complex expressions like classes and functions in pipe bodies anyway.

`x |> f(^^, 0)`

^^ would require separation from the bitwise-xor operator ^:

```
x |> ^^ ^ 2
```

…although bitwise xor is quite rare in most JavaScript code.

^^ can be typed even in keyboard layouts with circumflex-accent dead keys, although some platforms require 3–4 keystrokes.

Candidate topic tokens (3/4)

```
x |> f(%%, 0)
```

%% would require separation from the remainder operator %:

```
x |> %% % 2
```

…although the remainder operator is uncommon in most JavaScript code (albeit more common than bitwise xor).

```
x |> f(@@, 0)
```

Not to be confused with the single-character @ token.

Candidate topic tokens (4/4)

`x |> f(#_, 0)`

`#_` would not be ambiguous with private fields (which must be qualified with `this.`) or record/tuple literals:

```
x |> f(this.#y, #_)
```

```
x |> f(#[#_])
```

…except in one special case:

```
x |> #_ in this
```

`#_` would also preclude future bare private fields.

`x |> f(##, 0)`

`##` is also not ambiguous with record/tuple literals or with private fields (although it arguably is difficult to read when mixed with any of them):

```
x |> f(this.#y, ##)
```

```
x |> f(#[##])
```