Ecma/TC39/2010/016

1. Hygienic Macros For EcmaScript

2. Why Macros?

EcmaScript is primarily used to **generate** and **manipulate** content in **other languages** : HTML, CSS, JSON, XML.

Domain Specific Languages (DSLs) integrate languages into ES.

3. **Why Macros?**

DSLs are used for query languages, and to generate content.

Already widely used to query DOMs: XPATH and CSS selectors.

jQuery: `$("#foo li.bar")` → all items with `class="bar"` in el with `id=foo`

Interpreter overhead.

Content generation is ad-hoc and often inefficient and/or insecure.

4. **Why Macros?**

Standardizing DSLs : burden on browser implementor → low rev rate.

E4X took a long time, and is still BEA/Mozilla only.

DSLs via Macros : burden on library developers

→ high rev rate.

5. Why Macros?

DSLs specified as macros can expand to easily **inlinable code** — no per-iteration function call or interpreter overhead.

An unsound content-generation **idiom** (e.g. +=) must be fixed **everywhere** it is used, but if a content-generation **DSL** has problems, checks and fixes can often be applied **once** per library.

Burden for security shifted to library from app developers. Surprisingly this leads to lower effective rev rate.

6. Why Macros?

Dynamic Regexs. A multipart mime boundary may can contain '+'.

```
re`^--${boundary}$`
```

→ `new RegExp('^--' + escapeRegexpSpecials(boundary) + '$')`

Date formats with qualifiers

```
date`${day}d/${month}m/${year}Y`
```

Control Structures

```
using` $={k} in ${file} do ${foo(k)} ` →
let channel = open(file);
try { while (!channel.empty()) k
= channel.next(), foo(k);
} finally { channel.close(); }
```

String Interpolation

```
s`<b>$foo</b>`
```

7. **Why Hygiene?**

- Correctness — no namespace collisions or masking.
- Encapsulation — only substitutions and expander seen by expansion.
- Strictness — don't compromise analyzability of ES5 strict mode

8. **Anatomy of a Macro**

```
name`literal0${subst0}literal1`
```

Macro expander specified by a function name

Literal portions : data

Substitutions : code

Data/code split → non-intrusive injection-resistant content generation

9. **Desugaring**

```
name`literal0${subst0}literal1`
```

→

```
//
resolved in macro scope
name(
    // literals convey no
authority
    "literal0", "literal1", …)(
    // substitutions convey
authority
    { get: function () { return (subst0)
 } }, …)
```

Substitutions passed as property descriptors.

If λ available, then substitutions can be *Program* s.

Otherwise, ban free `arguments` and use *GroupingExpression* bodies.

10. Allowing Mutation

```
matcher`<div>$=foo</div>
 as $=bar`
```

= prefix adds a setter to the property descriptor.

Query languages can contain their output location.

11. Performance

All inputs statically known.

All inputs share same environment.

If macro expander is `const`, and does not compare substitutions by identity, highly inlineable after first call.

12. Syntax Open Questions

Goal: a syntax that is familiar to most JavaScript developers. Many know string interpolation from Python/Perl/Ruby/PHP.

¿Boundaries between literal portions and substitutions independent of vendor extensions and future language changes?

¿Literals raw escapes?

¿New Quotes `foo`...` vs brackets `foo {{...}}` vs old quotes `foo"..."` vs smileys `foo(:...:)`?

¿Substitution Marker $ vs # vs %?