Response to L2/19-056 on broadening the scope of “properties”

by Mathias Bynens (mths@google.com)
Shared publicly

Previously:

- TC39 proposal to support `\p{Sequence}` in JS RegExp
- L2/18-337 Broaden the scope of what Unicode calls “properties”
- L2/19-056 Comments on L2/18-337

This document is a response to claims made in L2/19-056.

Claim w.r.t. boolean mappings

> **Michael:**
> In most current cases, sequences are boolean, either a string matches or it doesn’t. Properties on the other hand can have many possible values. Even the canonical equivalent sequences are still boolean mappings.

Mathias:
The statement “either a string matches or it doesn’t” applies to both properties of characters and properties of strings, and either “can have many possible values”. For example, `Script=Greek` matches both "π" and "ↄ ", and `Emoji_Keycap_Sequence` matches both "4⃣" and "2⃣". `Basic_Emoji` includes both single code points such as “🧐” and multiple-code point strings such as “☹”. (All properties of code points are logically also properties of strings, just those of single code points.)

> **Michael:**
> Certainly there are boolean properties, but there are also other types like String and Enumeration. For example the full case folding property maps single code points to one or more code points. Most of the line, sentence, word and other breaks are enumerations and not boolean properties. For sequences, matching string is sufficient. For non-boolean properties, determining a codepoint’s value involve more than just matching.

Mathias:
We need to be clear about nomenclature. Properties in UTS #18 are mappings from a domain to a range. When UTS #18 talks about an enumerated property (implicitly “of code points”), that means a mapping from code points to an enumerated type; when it talks about string properties that means a mapping from code point to string. A code point property of strings would be a mapping from string to code point, on the other hand.

So in Unicode terms, a “non-boolean property” (implicitly “of code points”) is one that maps code points to a non-boolean value. An example is `Word_Break`, where the expression `\p{Word_Break=ALetter}` means the set of all elements of the range whose `Word_Break` value is `ALetter`. The meaning of “determining a code point’s value involves more than just matching” is unclear in that context. Note that UTC is currently weighing whether it would be better to have a unified `\p{...}` syntax or to have a separate syntactic structure for properties of strings.

Markus: To clarify, while it is true that the properties-of-strings proposed in L2/18-337 for use in regex are “boolean properties” (although mostly called “binary properties”), it is possible that future properties-of-strings may map those strings to enumerated, numeric, or other types of values. This could be handled using the same `prop=value` syntax as for properties-of-code-points. (ECMAScript does not allow the `prop=value` syntax for boolean/binary properties, but UTS #18 does, and it should do so for properties-of-strings as well.)

Claim w.r.t. named sequences

Michael: *“In general, sequences are used to describe strings that have meanings distinct from their individual code points. L2/18-337 is focused primarily on five Emoji sequences. These sequences are lists of strings (or list of sequences) that define a boolean group membership. While L2/18-337 does provide for future lists of sequences, it doesn’t address the myriad of current Unicode sequences, e.g. LATIN CAPITAL LETTER A WITH OGONEK AND ACUTE.”*

Markus: Syntax for “match any Emoji keycap sequence” is very different from syntax for “match the sequence of characters with this name”. Both make sense, just for different use cases. The former is an extension of what `\p{prop=value}` does, the latter is an extension of what `\N{name}` does. Note that UTS #18 `\q{string}` takes a literal string, not a name or alias.

Claim w.r.t. incompatibility with existing code

Michael:
Finally, by reusing syntax, we will likely introduce incompatibilities in deployed code. Consider existing code that accepts regular expression components as strings, including properties, where a property name is inserted in a \p{…} or \P{…} construct. Those computed property escapes could run into the syntax issues described above without the associated error handling. Such error handling would likely not be anticipated since it wasn’t needed for properties. The

only error checking required today is validating if existence of a property, and not its acceptable usage within a regular expression.

Mathias:
The claim that “such error handling would likely not be anticipated since it wasn’t needed for properties” is incorrect. Any code that inserts a user-provided property into `\p{…}` or `\P{…}` today already must have error handling, in case the user passes an invalid or unsupported property.

Furthermore, properties of strings/sequences aren’t currently supported anywhere. Therefore, such code would currently throw an exception when the user passes in a string property. Even if the code has the error checking you describe, i.e. *“validating [the] existence of a property”*, properties of strings wouldn’t be in the safelist as they’re currently not supported anywhere. If they are in the safelist, then the code is already broken today. If they aren’t, then the code won’t suddenly break when sequences gain support. As such, there are no “incompatibilities in deployed code” at all. No existing code would break.

Michael:
The point I’m making is that a regular expression constructed via code like:

```
function doStuff(myProperty) {
  try {
    new RegExp('\p{' + myProperty + '}');  // Is `myProperty` valid?
  } catch (err) {
    return;
  }
  let reString = '[\p{' + myProperty + '}abc]';
  let re = new RegExp(reString);
  let match = re.exec(someString);
  return match;
}
```

…will fail in an unexpected way if `myProperty` could be a sequence. This doesn’t seem to be a far fetched way to implement a safelist.

Mathias:
It does seem like a far-fetched use case (but it’s hard to qualify that). But even given the use case, the particular implementation you describe *is* far-fetched. The code constructs two

different regular expressions, where it seems simpler and more straightforward to just create a single one and then use it if that doesn't fail:

```
function doStuff(property) {
  try {
    // Is `property` valid?
    const pattern = '[\\p{' + property + '}abc]';
    const re = new RegExp(pattern, 'u');
    const match = re.exec(someString);
    return match;
  } catch {
    return;
  }
}
```

This (IMHO more likely) implementation doesn't break when sequence properties become supported.

However, this code is still inherently insecure, just like the original snippet. The user can inject arbitrary regular expression patterns by passing values like:

```
'Unassigned}|arbitrary_pattern|pattern_that_is_not_in_string['
```

Note that there is no way to escape any `}` within property escapes, so to be robust against such an attack, the code would have to check if `myProperty` includes `}` separately. All in all, this pattern seems increasingly unlikely to be used in a real-world code. **Regardless of our decision w.r.t. adding sequence property support, the particular code example you describe is already broken.**