Mixed-script detection in identifier chunks

To: UTC
From: Robin Leroy, Source code ad hoc working group
Date: 2022-10-20

This document defines a mechanism for detecting confusing mixtures of scripts that could result in usability or spoofing issues, while accepting those that would arise from legitimate usage for linguistic reasons (and somewhat improving the situation when scripts are mixed for technical applications). Note that spoofing is more comprehensively handled by confusable detection on the set of identifiers in use; but depending on the specifics of its implementation, it may leave the door open to deeply confusing compilation errors.

> **Note:** Document L2/22-229 proposes incorporating the definitions from this document into a new Unicode Technical Standard. The purpose of this document is to serve as a more detailed rationale for its technicalities.

An implementation which diagnoses confusable identifiers at the lexical level (on the set of identifier tokens that may be in scope) prior to successful compilation has no need for this mechanism, as it then remedies the usability issues as well as the spoofing issue. This is, for instance, the case of the Rust compiler.

The mechanism described here could, for instance, be applied by an editor which is only capable of performing lexical analysis on the file currently being edited, and thus cannot obtain the set of visible names defined by other files.

This file uses the regular expression syntax defined in UTS #18 *Unicode Regular Expressions*, version 23.

Definition

An *identifier word boundary* is defined as any of the following:

- 🐫 a *CamelBoundary*, defined as the position after the group in a sequence matching the following regular expression:
  `( [ \p{Ll} [\p{Lt}-\p{Grek}] ] [\p{Mn}\p{Me}]* ) [\p{Lu}\p{Lt}]`,
- 🎩 a *HATBoundary*, defined as as the position before a sequence matching the following regular expression:
  `[\p{Lu}\p{Lt}] [\p{Mn}\p{Me}]* \p{Ll}  |  [\p{Lt}-\p{Grek}]`.
- 🐍 a *snake_boundary*, defined as the positions either side of a Punctuation character which is not an Other Punctuation character, *i.e.*, either side of a sequence matching `[\p{P}-\p{Po}]`.

An identifier splits into *identifier chunks* delimited at identifier word boundaries. Note that multiple kinds of boundaries can coincide.

Examples of the separation into identifier chunks are given in the table below; emoji mark the various boundaries.

Identifier	Identifier chunks	Notes
`dromedaryCamel`	`dromedary`🐫🎩`Camel`	🐫
`snakeELEPHANTSnake`	`snake`🐫`ELEPHANT`🎩`Snake`	🌹📦
`TypeII`	🎩`Type`🐫`II`	
`OCaml`	`O`🎩`Caml`	The HATBoundary is designed to accommodate the common practice of keeping acronyms in upper case in a CamelCase identifier.
`HTTPзапрос`	`HTTP`🎩`Запрос`	
`UAX9ClauseHL4`	`UAX9`🎩`Clause`🐫`HL4`	
`LOUD_SNAKE`	`LOUD`🐍`_`🐍`SNAKE`	
`Fancy_Snake`	🎩`Fancy`🐍`_`🐍🎩`Snake`	
`snake-kebab`	`snake`🐍`-`🐍`kebab`	Assuming a profile allowing hyphen-minus in identifiers.
`Paral·lel`	🎩`Paral·lel`	Other Punctuation does not separate words; indeed it is used within words in Catalan.
`microB`	`micro`🐫`B`	
`microb`	`microb`	The sequence `\p{Ll}\p{Lo}` is not a CamelBoundary, and should not be one: this Other Letter is confusable with a Lowercase Letter.
ᾼΔᾙ	ᾼΔᾙ	No boundaries despite the Titlecase Letters: depending on font, they might not look like they are titlecase.
ᾼιΔᾙι	🎩ᾼι🐫Δ🎩ᾙι	May render identically to the one above depending on the font, but this one will never look like uppercase with diacritics.
HTTPῼδή	HTTP🎩ῼδή	Here there are other Lowercase Letters after the Greek Titlecase letter, so we have a boundary.
HTTPसर्वर	HTTPसर्वर	Here a visible word boundary is not detected, but the resulting multi-word chunk is visibly mixed-script.

An identifier chunk X is *confusing* if both of the following are true:

1. X has a restriction level greater than Highly Restrictive, as defined in UTS #39, section 5.2;
2. There exists a string Y such that all of the following are true:

a. Y is confusable with X;
b. The resolved script set of Y is neither ∅ nor ALL;
c. The resolved script set of Y is a subset of the union of the Script_Extensions of the characters of X.
d. Y is in the General Security Profile for Identifiers.

Note: Criteria a through c of condition 2 are similar to "X has a whole-script confusable in the union of its Script_Extensions", but do not require X to be single-script.

An identifier chunk for which condition 1 holds but condition 2 does not hold is called *visibly mixed-script*.

Note: Visibly mixed-script identifier chunks are not confusing.

An implementation implementing mixed-script detection in identifier chunks shall diagnose confusing identifier chunks in identifier tokens.

Examples of confusing and non-confusing mixed-script identifier chunks are given in the following table; all have a restriction level greater than Highly Restrictive.

`Строка`	Confusing, confusable with all-Cyrillic `Строка` and all-Latin `Строка`.
`Δt`	Visibly mixed-script, `t` is not confusable with a Greek letter, nor is `Δ` confusable with a Latin letter.
`μэow`	Visibly mixed-script, `μ` is not confusable with a Cyrillic letter nor with a Latin letter.
`MIKPA`	Confusing, confusable with all-Greek `MIKPA` and all-Latin `MIKPA`.
`HTTPसर्वर`	Visibly mixed-script, `H` is not confusable with a Devanagari letter, nor is `स` confusable with a Latin letter.
`microb`	Confusing, confusable with all-Latin `microb`.

Rationale

Why mixed-script detection to start with?

Assuming good confusable data on the characters used, spoofing issues arising from confusable identifiers can be adequately mitigated by solely detecting the coëxistence of confusable identifiers without any mixed-script detection, *e.g.*, by warning about

```
std::string строка;
std::string строка;  // Latin a,
```

but not about

```
std::string строка;  // Latin a.
```

However, when working with multiple scripts, there is a common usability issue whereby one accidentally changes a letter while using the wrong keyboard layout, *e.g.*, editing the following line:

```
std::vector<std::string> строки;  // "strings",
```

removing the following struck-out letters and typing (using a Latin keyboard layout) the underlined letters:

```
~~std::vector<std::string>~ const~ строки~  // "strings",
```

to produce the following line:

```
std::string const строка;  // "string".
```

Trying to refer to the resulting identifier `строка` will lead to a compilation error (because it is actually `строка`, with a Latin a).

If confusable detection operates on the set of declarations, it will fail to detect this situation. Similarly, if confusable detection is performed by a linter operating globally on a code base after it has compiled (recall that confusable detection is a global operation, since it requires collecting the set of all identifiers), it will not get to run. The user will then be faced with an inscrutable compilation error.

Note that this means that mixed-script detection acts primarily to remedy a usability concern; in adversarial scenarii, the detection of confusable identifiers is a more effective remedy.

Why not whole-identifier mixed-script detection?

Industry standard terms are often in another script: consider the real-life identifier `HTTPЗапрос` (*HTTP Request*). Note that if that identifier consisted of one identifier chunk, it would be confusing, because the Cyrillic-only identifier `HTTPЗапрос` (*NTTR Request*) would be confusable with it.

Why not simply flagging mixed-script identifier chunks?

This is the approach taken by ocaml-m17n. We believe that the refinement used here is useful.

An identifier like `μэow`, which consists of a visibly mixed-script identifier chunk, is neither problematic from a usability nor a security standpoint; the reader knows that scripts are being mixed, and cannot realistically have expectations on the nature of the "`o`". Some single-chunk visibly mixed-script identifiers such as `Δt` are common. Identifier chunk detection fails when unicameral scripts are involved, so that a warning would be issued about a legitimate and harmless identifier such as `HTTPसर्वर`.

What's the deal with the `\p{Grek}` stuff?

The Greek Titlecase Letters may not look titlecase at all depending on the font, and may instead look like a capital letter with a diacritic (indeed they are canonically equivalent to a sequence `\p{Lu}\p{Mn}`), so they do not reliably signal a visible word boundary. We posit that detecting a HATBoundary if they are followed by a Lowercase Letter is more than enough to properly handle Ancient Greek identifiers.

Why `[\p{P}-\p{Po}]` rather than `\p{Pc}` or `\p{P}` in snake_boundary?

The characters in `[[:Po:] & [:XID_Continue:]]` are in `[:XID_Continue:]` not as word separators, but because they are needed as part of words in Catalan; see UAX #31 *Unicode Identifier and*

Pattern Syntax, Section 2.4 “Specific Character Adjustments”; they are not expected to separate words, let alone scripts.

The sets `[[:P:] & [:XID_Continue:] - [:Po:]]` and `[:Pc:]` are equal. However, if an implementation uses a profile for UAX31-R1, it may allow punctuation characters from other general categories as word separators, *e.g.*, the Dash Punctuation hyphen-minus.

What about unicameral scripts?

While the case-based identifier chunk detection fails for those, note that while `HTTPसर्वर` consists of a single identifier chunk, it is neither confusable with a Devanagari string nor with a Latin string, and is therefore not confusing. We posit that in the cases where characters from a unicameral script are often confusable with characters from another script also in use, a Connector Punctuation character would help with legibility, *e.g.*, that it would be more readable to write `HTML_dY:` than `HTMLdY:` (which forms a confusing identifier chunk because of its confusability with the Lisu-only `HTMLdY:`), or `micro_b` than `microb` (confusing because confusable with the Latin-only `microb`).

Note that mechanisms enforcing CamelCase identifier styles should be generalized to allow a low line adjacent to a unicameral script. See L2/22-232.

If however the requirement to add a Connector Punctuation character proves too onerous for some frequently-confusable pair of scripts where one is unicameral, the diagnostic described in this document could be suppressed.

Does this solve the issues of mixed-script mathematical notation?

To a limited extent. As mentioned above, `Δt` is non-confusing. The case of `δt` is more interesting: were it not for criterion 2.d in the definition of “confusing”, that chunk would be confusing, because `δ` is confusable with the Latin letter `δ` (and this is a case of perfect confusability assuming good font support, contrary to, *e.g.*, `p` and `ρ`; it cannot be solved with stricter confusable data). However, Latin letter delta is Restricted with Identifier_Type=Technical, so it doesn’t count.

However, the limitations of confusability itself can lead to issues: `dρ` is confusing because it is confusable with the all-Latin `dp`.

The diagnostic may need to be suppressed for those applications, much like diagnostics implementing confusable detection should be suppressed in applications that make use of restricted scripts (for which confusable data is not available).

Acknowledgements

The usability issue dealt with here was brought to the author’s attention by Catherine “whitequark”. The mitigation is heavily influenced by the one implemented by her ocaml-m17n package (from which we take the term “identifier chunk”), with adjustments for case-based identifier word boundaries and accepting visibly mixed-script identifier chunks. We thank her for feedback on an early version of this document.