# Extensions to regular expressions for Chinese, Japanese, Korean, and Vietnamese scripts

Andrew J. Young

April 9, 2019

**Abstract**

Regular expressions (REs) are principally designed to work with the basic latin alphabet, and with unicode. Their basic syntax and expression makes them incredibly powerful, and this includes syntax to match with classes of character, such as any whitespace character, or any number.

This makes RE principally well suited to alphabetical scripts, particularly ones with small alphabets. But, REs are considerably less useful for alphabets which are large. The limitations of RE make it largely unable to deal with the challenges of parsing Chinese, Japanese, Korean, and old Vietanmese text, commonly known an CJKV text. Because these languages by nature all have large alphabets, RE are effectively unable to be efficiently used to parse text in these languages.

This paper details a proposal for an extension to RE syntax which allows for the parsing of CJKV characters. Extended syntax for RE is not uncommon, and is used in programs such as Libre Office to reduce the number of keystrokes used in searches, and to simplify patterns to make them easier to memorize. Through this extended syntax, we give the ability to phonetically parse hangeul and kana, the phonetic scripts of Korean and Japanese respectively, and also the ability to parse CJKV characters (sinographs) by their radical.

## 1 Background

Regular expressions (REs) are principally designed to work with the basic latin alphabet, and with unicode. Their basic syntax and expression makes them incredibly powerful, and this includes syntax to match with classes of character, such as any whitespace character, or any number.

This makes RE principally well suited to alphabetical scripts, particularly ones with small alphabets. But, REs are considerably less useful for alphabets which are large. The limitations of RE make it largely unable to deal with the challenges of parsing Chinese, Japanese, Korean, and old Vietanmese text, commonly known an CJKV text. Because these languages by nature all have large alphabets, RE are effectively unable to be efficiently used to parse text in these languages.

## 2 Defining the extended syntax

To deal with extended RE, the following syntax is proposed:

`[:\w+]`

This extended syntax is inherited from the extended syntax of RE used in libre office, where `[:\w+:]` indicates an abbreviated or custom RE. There is one modification to the syntax presented in this section, which brings it in line to the syntax already used by RE to parse expressions of the form `[:…:]` and (?…) («`[ ^\^.+]`» and «`(?.+)`»).

It is obviously preferable that syntax not be overlapping or different between different systems. Because of this, it is desirable to place restrictions on what the exact syntax of these boxes can be.

Firstly, it is helpful to denote a script which the extension works with. This should be the 2 letter ISO script or language code, whichever is the most specific, followed by a dash.

After this, the expression should define clearly, ideally in latin script, what characters the expression should match with. It is preferable to use latin script (such as a romanization of the script in question) to name these syntaxes so that they can be easily typed and edited by any

programmer irrespective of the keyboard they use or their locale. This effectively restricts the names of these sequences to ASCII, although the RE themselves will be dealing with unicode.

## 2.1 Syntax for japanese kana

In this section, proposals are presented for extended syntax to match characters from japanese's kana syllabary.

Japanese has 2 syllabaries: katakana and hiragana. These two scripts are almost interchangeable, but not all characters which appear in katakana have a hiragana equivalent.

The following syntax is proposed for matching hiragana:

`[:j-h-(\*(\w)|(\w)\*)]`

where (\w) must match with the legal syllables present in Japanese. For matching katakana, the following syntax is proposed:

`[:j-k-((\w)|(\w))]`

where (\w) and (\W) must match the legal syllables present in Japanese.

As japanese kana are often freely interchangeable (particularly in colloquial japanese), users are likely to want to run "case" insensitive searches which match equivalent hiragana or katakana characters. If a user wants to do a "case" insensitive search, then the syntax will simply be:

`[:j-(\*(\w)|(\w)\*)]`

Japanese romanization is context sensitive, which makes it poorly suited to free-matching of japanese characters. Because of this, we recommend the following:

1. Maintaining minimum compatibility with nihonshiki romanization, which is perhaps the most widely known and understood romanization method within Japan. It is relatively consistent at romanizing.

2. The addition of systematic combining characters, such as *l-* to match with "small" characters such as っ, ぁ, and ぇ, rather than their large forms つ, あ, and え. Hence, `[:j-ltu]` matches with っ.

3. Allow more packages to implement phonetic extensions, but make this non-essential and highly optional.

It should be standard to represent any unicode character with «`U+\d4`», and this comes in useful in the conversion process.

To implement compatibility with this standard is simple. Japanese kana is contained entirely within one unicode block per syllabary: hiragana characters have their own block, and katakana characters have their own block separate to hiragana. The characters within these blocks are in the same relative position from the start of the block, so to convert between them it suffices to numerically add or subtract $96_{10}$ from the unicode code point to get the equivalent character from the other syllabary.

Additionally, characters are arranged within the kana unicode blocks by their position in the Japanese alphabet, which is itself a 2D grid that sorts characters by their vowel and consonant.

Therefore, to search for a character with consonant <C> and the code point U+(3040 + x), where the number of consonants which start with that consonant sound is $n$ :

[:j-h-<C>*] == [U+($3040_{16} + x$)-U+<$3040_{16} + x + n$>]

Vowels need to be handled case by case due to how kana are laid out in unicode. For a vowel <V>, the corresponding syntax is `[:j-*<V>]`

Finally, to match case insensitively:

[:j-<C>*] == ([U+($3040_{16} + x$)-U+<$3040_{16} + x + n$>]|[U+($3040_{16} + x + 96_{10}$)-U+<$3040_{16} + x + n + 96_{10}$>]

In addition to this, we define a sequence for *match any* to either match any hiragana character, any katakana character, or any kana character.

| Type | Syntax | Is equivalent to |
|---|---|---|
| Match any kana | `[:j-(\w)+]` | `[U+3040-U+30FF]` |
| Match any hiragana | `[:j-h-(\w)+]` | `[U+3040-U+309F]` |
| Match any katakana | `[:j-k-(\w)+]` | `[U+309F-U+30FF]` |

### 2.1.1 Examples

1. `[:j-*a]` matches あ, か, さ, た, な, は, ま, や, ら, or わ.
2. `[:j-k*]` matches か, き, く, け, or こ.
3. `[:j-sh*]` matches し.

## 2.2 Syntax for hangeul

Korean text is written in hangeul, a featural alphabet. Although this alphabet is small and phonetic in its components, these individual phonetic components (called *jamo*) combine to form blocks. Each block is one syllable of Korean. The word 🅕🅕 (hangeul) is therefore made up of 6 jamo (🅕, 🅕, 🅕, 🅕, 🅕, and 🅕, spelling out *h-a-n-g-eu-l*).

Like kana, Hangeul is also consistently arranged within unicode. Hangeul only has one alphabet, and so it is simpler to construct the basic syntax of RE compared to doing so for japanese kana. Let the syntax for matching hangeul characters be:

`[:k-(\w)+]`

Where () must match the lowercase revised romanization of the hangeul block being matched. Wildcard jamo are marked with an asterisk.

This makes hangeul matching relatively similar to our existing syntax for regular expressions using latin script text. One key difference is that we need a technical specification for converting korean hangeul matching into syllabic blocks.

Hangeul syllable blocks are placed algorithmically in unicode using the formula:

$[(initial)\times 588 + (medial)\times 28 + (final)] + 44032$

The enumeration of jamo initials, medials, and finals can be found in the appendix.

### 2.2.1 Examples

1. `[:k-h*]` matches
2. `[:k-*eu]` matches
3. `[:k-*eun]` matches

## 2.3 Syntax for CJKV characters

Due to the complexity of CJKV characters both graphically and in terms of pronunciation, the extensions proposed for these character sets are quite simple. It is useful for a programmer to be able to search characters by their radical. Quite often, this is the order by which characters are sorted, and often variant characters (different ways of drawing the ”same” character in terms of meaning) will have the same radical even when the rest of the character looks different. For example, 島 and 🅕 both mean «island», but are made up of the same 2 components: 鳥 and 山.

The standard set of radicals is the Kangxi radicals. These are already widely used in computing, and work with both simplified and traditional character sets, as well as with Japanese *kokuji* and Vietnamese *chu nom*, whose characters may not always be part of other CJKV character sets.

Hence, to search CJKV characters by radical, the following structure should be used:

`[:c-(\w)]`

where (1) matches with one of the Kangxi radicals. This will match with any character from unicode where the radical is (1), regardless of the unicode block in which it is encoded. This does make expanding this expression somewhat more complicated as where to look for these characters is a value which must be regularly updated with new editions of unicode.

### 2.3.1 Examples

- `[:c-水]` matches 清, 水, 泉
- `[:c-火]` matches 燃, 然, 火, 炎
- `[:c-山]` matches 山, 🅕, 島

# 3 Appendix

## 3.1 Enumeration of hangeul jamo

Initial consonants:

0. 🄵
1. 🄵
2. 🄵
3. 🄵
4. 🄵
5. 🄵
6. 🄵
7. 🄵
8. 🄵
9. 🄵
10. 🄵
11. 🄵
12. 🄵
13. 🄵
14. 🄵
15. 🄵
16. 🄵
17. 🄵
18. 🄵

Medial vowels:

0. 🄵
1. 🄵
2. 🄵
3. 🄵
4. 🄵
5. 🄵
6. 🄵
7. 🄵
8. 🄵
9. 🄵
10. 🄵
11. 🄵
12. 🄵

13. 🄵
14. 🄵
15. 🄵
16. 🄵
17. 🄵
18. 🄵
19. 🄵
20. 🄵

Final consonants:

0. none
1. 🄵
2. 🄵
3. 🄵
4. 🄵
5. 🄵
6. 🄵
7. 🄵
8. 🄵
9. 🄵
10. 🄵
11. 🄵
12. 🄵
13. 🄵
14. 🄵
15. 🄵
16. 🄵
17. 🄵
18. 🄵
19. 🄵
20. 🄵
21. 🄵
22. 🄵
23. 🄵
24. 🄵
25. 🄵
26. 🄵
27. 🄵