Summary: Owing to the high expressiveness of regular expression, it is frequently used in searching and manipulation of text based data. Regular expression is highly applicable in processing Latin alphabet based text, but the same cannot be said for Hangeul*, the writing system for Korean language. Although Hangeul possesses alphabetic features within the script, expressiveness of regular expression pattern using Hangeul is hindered by the absence of syllable decomposition. Without decomposition support in regular expression, searching through Hangeul text is limited to a string literal matching. Literal matching has made enumeration of syllable candidates in regular expression pattern definition indispensable, albeit impractical, especially for a large set of syllable candidates. Although the existing implementation of canonical decomposition in Unicode standard does reduce a pre-composed Hangeul syllable into smaller unit of consonant-vowel or consonant-vowel-consonant letters, it still leaves quite a number of the individual letters in compounded form. We have observed that there is a necessity to further reduce the compounded letters into unit of basic letters to properly represent the Korean script in regular expression. We look at how the new canonical decomposition technique proposed by Kim can help in handling Hangeul in regular expression. In this paper, we examine several of the performance indicators of full decomposition of Hangeul syllable to better understand the overhead that might incur, if a full decomposition were to be implemented in a regular expression engine. For efficiency considerations, we propose a semi decomposition technique alongside with a notation for defining Hangeul syllables. The semi decomposition functions as an enhancement to the existing regular expression syntax by taking in some of the special constructs and features of the Korean language. This proposed technique intends to allow an end user to have a greater freedom to define regular expression syntax for Hangeul.

Key words: regular expression, Hangeul, Unicode, NFD, Korean script

1. Introduction

Regular expression pattern matching has been included in many programming languages for text searching and replacing. It can also be applied to user input validation against a specific format defined using regular expression. XML Schema, for example, has a sub-element pattern that requires an attribute value in regular expression to check for a match between a defined format and the element value in an XML document [3].

The existing implementation of regular expression feature found in numerous programming languages and applications works well if texts are in Latin based alphabet like those found in English language; but when text is in Hangeul script, however, we observed several limitations such as the following example.

Table 1 shows an example of conjugation and inflection of a commonly used Korean verb 하다 hada ‘to do’. Notice that all of the phrases have the prefix ‘하’ appearing in the romanisation column next to each phrase. Searching for text in the romanisation column containing the literal ‘하’ using regular expression might be as straightforward as /하.+./. However, if we were to substitute the search pattern with Hangeul as /하.+./, we would only be able to match 5 out of the total 12 instances using the same literal character-by-character matching technique as used by the romanised method.

One workaround to the above problem is to use alternation by enumerating the different possible forms of the verb, where we can modify the previous regular expression to be in the pattern /하|한|하|할|하다|합|하/.+./. Although this can solve the problem temporarily, it is not a perfect technique; we might stumble upon yet another conjugation or inflection that is unaccounted for in our pattern. The modified pattern for example, did not apply to nominalised verb 한다 hada.

In addition, enumerating all possible permutations is not practical as well. Take for example the case where we were to search for word begins with ‘하’ (h). There are at least 588 possible permutations available [2] that we need to list out in our pattern. Clearly, we need a better solution to deal with this limitation and other associated issues pertaining to Hangeul in regular expression.

Section 2 introduces the Hangeul script and its representation in Unicode while presenting related works on the topic. In Sect. 3, we explain about our proposal on how to properly handle Hangeul script in regular expression. Sec-

Copyright © 2011 The Institute of Electronics, Information and Communication Engineers
tion 4 demonstrates several motivating examples using the proposed improvement to regular expression. We conclude our findings for this paper in Sect. 5.

2. Background & Related Work

2.1 Hangeul

The Korean script, more specifically modern Hangeul contains 24 basic letters (known as 자모 ‘jamo’ in Korean), with 14 consonants and 10 vowels [4]. Using South Korean alphabetical order, the 14 consonants are $g/k$, $n$, $d/t$, $r/l$, $m$, $b/p$, $s/-ng$, $j$, $ch$, $k$, $t$, $p$, and $h$; the 10 vowels include $a$, $ya$, $eo$, $yeo$, $o$, $yo$, $u$, $yu$, $eu$, and $i$. Using the basic letters, complex (or compound) consonants can be formed by combining basic consonant with another basic consonant, while complex vowels can be formed likewise. Complex vowel $wa$ for example is made up of basic vowel $o$ and $a$ [4]. Figure 1 shows the modern Hangeul letters in alphabetical order, with shaded letters signifying complex letters. Notice there are two sets of consonants available, leading consonants and trailing consonants.

Unlike Latin script where we form a syllable or a word by just juxtaposition the letters side by side, forming a Hangeul syllable however requires clustering together consonants and vowels into a single block [5]. Modern Hangeul syllables can be formed by combining with at least a leading consonant (L) and a vowel (V). An optional trailing consonant (T) can also be added to the end of syllable, giving us a modern Hangeul syllable in either LV formation or LVT formation. The word 한글 Hangeul for example, is formed by joining $h$ (L) + $a$ (V) + $n$ (T) to form the LVT syllable han, and $geul$ is formed by $g$ (L) + $eu$ (V) + $l$ (T).

2.2 Hangeul in Unicode

A modern Hangeul syllable in Unicode can either be represented in conjoining jamos (sequence of individual complex and basic letters) or as a single pre-composed syllable (known as 완성 wanseong) [2]. Either representation should render an identical output that is transparent to users, where the former will be treated as a grapheme cluster [6]. Unicode standard has allocated a block of code points for Hangeul jamo (both modern and old Hangeul jamo) ranging from $U+1100$ to $U+11FF$. The subset of the block where modern Hangeul jamos are available can be seen from Fig. 1 alongside with jamo’s corresponding code point.

Pre-composed Hangeul syllables on the other hand are available in a continuous block ranging from $U+AC00$ to $U+D7A3$. The 11,172 pre-composed Hangeul syllables from this block are formed as a result of permuting all basic and complex jamos and sorted alphabetically, yielding 399 syllables in LV form and the remaining 10,733 syllables in LVT form. If we were to arrange the pre-composed syllables by rows of leading consonants and columns of vowels and trailing consonants permutation, we can have a chart similar to Fig. 2.

2.3 Canonical Decomposition

The permutation arrangement in Unicode for pre-composed Hangeul syllables allows us to convert each syllable into conjoining jamo representation and vice versa algorithmically. Representation of pre-composed syllable in conjoining jamo is known as Normalisation form D (NFD) or canonical decomposition, while the reverse is known as Normalisation form C (NFC) or canonical composition. Unicode standard contains algorithm for normalising Hangeul syllable from one form to another [2], [7], as illustrated conceptually in Fig. 3.

The canonical decomposition algorithm works by applying modular arithmetic and offset shifting to the syllable code point value from a set of corresponding jamo cat-
egory’s first code point value. For a syllable in LV form, we
will have 2 conjoining jamos while LVT form will have 3 instead. The resulting canonical decomposition of a
pre-composed Hangeul syllable will have conjoining jamos
found in Fig. 1. Notice that NFD for the Hangeul syllable
in Fig. 3 using method shown in [7] still contains complex
jamo. As explained in Sect. 2.1, complex jamos are made
up of simpler basic jamos. Kim [8] has proposed that in-
stead of decomposing a syllable into L, V, T form in exist-
ing NFD and stop, any complex jamo found within the re-
sult, decomposition should be continued until all jamos are
in basic form. We will explain more detail in Sect. 3, on why
this is essential for regular expression to work in Hangeul.

In Kim’s proposed decomposition process, a Hangeul
syllable will first go through a canonical decomposition just
like how it was done in [7], after which if jamo were still
in complex form, it will be decomposed into basic jamos.
Figure 4 shows new canonical decomposition process ap-
plied to the same Hangeul syllable ꪙ kkwaelm from Fig. 3.
Table 2, 3, and 4 summarise on how complex jamo found
in Fig. 1 can be decomposed using this new method. Basic
jamos will remain the same when applying this new decom-
position technique since there is no further decomposition
possible.

The rationale behind this new canonical decomposi-
tion (and also composition) process is to treat both mod-
ern Hangeul jamos and old Hangeul jamos in a consistent
manner [8]. The author has pointed out that existing nor-
malisation process in Unicode standard did not handle old
Hangeul syllable well. Similar concern about old Hangeul
syllable handling in Unicode has also been highlighted by

Ahn and Park [9], [10].

Ahn and Park have explained how existing normali-
sation of Hangeul syllable was done incorrectly, and pro-
posed on how normalisation should be applied to both old
and modern Hangeul syllable [9]. Citing efficiency reasons,
the authors have also proposed several changes to Hangeul
encoding in Unicode. They have suggested that modern
Hangeul should only be represented in pre-composed form,
while old Hangeul should only be represented in conjoining
jamo. In addition, they have proposed inserting an extra set
of old Hangeul jamo into the existing Unicode standard [10].

Taking into account the limitation on representing and
processing of old Hangeul syllables in current version of
Unicode, plus the majority of commonly found Korean texts
are in modern Hangeul, we will therefore be focusing on the
more stable modern Hangeul syllables in this paper.

2.4 Unicode Regular Expression

Many of the existing regular expression engines can only
handle ASCII encoded characters. Unicode consortium has
released a technical specification for supporting Unicode in
regular expression. It describes three levels of Unicode sup-
port in regular expression engine by taking into considera-
tion the large repertoire of characters and writing systems
supported within Unicode [11]. In order for Unicode to
work for regular expression engine, at least Level 1 support
is required. Level 1 covers the use of Unicode by specifying
code point directly in regular expression syntax using hex-
decimal notation. It also covers the use of Unicode prop-
erties to define character set, alongside with set operations
of union, intersection and relative complement. Basic word
and line boundary should be available in this level, together
with case insensitive matching [12].

Level 2 includes support for extended features on top of
features from Level 1, and contains canonical equivalent
matching. Regular expression syntax in this level should
be more intuitive to user. In addition to using hexadecimal
notation found in Level 1, named character can be used as
well in this level. It is noted in the specification that lan-
guage or locale specific handling is not done in this level
but on Level 3 for tailored support. Regular expression en-

Fig. 4 New canonical composition and decomposition for Hangeul [8].

Table 2 Complex leading consonants decomposition.

<table>
<thead>
<tr>
<th>Leading Consonant</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>₋ U+1101</td>
<td>₋ U+1100 ₋ U+1100</td>
</tr>
<tr>
<td>₋ U+1104</td>
<td>₋ U+1103 ₋ U+1103</td>
</tr>
<tr>
<td>₋ U+1108</td>
<td>₋ U+1107 ₋ U+1107</td>
</tr>
<tr>
<td>₋ U+110A</td>
<td>₋ U+1109 ₋ U+1109</td>
</tr>
<tr>
<td>₋ U+110D</td>
<td>₋ U+110C ₋ U+110C</td>
</tr>
</tbody>
</table>

Table 3 Complex vowels decomposition.

<table>
<thead>
<tr>
<th>Vowel</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>₋ U+1162</td>
<td>₋ U+1161</td>
</tr>
<tr>
<td>₋ U+1164</td>
<td>₋ U+1163</td>
</tr>
<tr>
<td>₋ U+1166</td>
<td>₋ U+1165</td>
</tr>
<tr>
<td>₋ U+1168</td>
<td>₋ U+1167</td>
</tr>
<tr>
<td>₋ U+116A</td>
<td>₋ U+1169</td>
</tr>
<tr>
<td>₋ U+116B</td>
<td>₋ U+1169</td>
</tr>
<tr>
<td>₋ U+116C</td>
<td>₋ U+1169</td>
</tr>
<tr>
<td>₋ U+116F</td>
<td>₋ U+1166</td>
</tr>
<tr>
<td>₋ U+1170</td>
<td>₋ U+116E</td>
</tr>
<tr>
<td>₋ U+1171</td>
<td>₋ U+116E</td>
</tr>
<tr>
<td>₋ U+1174</td>
<td>₋ U+1173</td>
</tr>
</tbody>
</table>

Table 4 Complex trailing consonants decomposition.

<table>
<thead>
<tr>
<th>Trailing Consonant</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>₋ U+11A9</td>
<td>₋ U+11A8 ₋ U+11A8</td>
</tr>
<tr>
<td>₋ U+11AA</td>
<td>₋ U+11A8 ₋ U+11A8</td>
</tr>
<tr>
<td>₋ U+11AC</td>
<td>₋ U+11A8 ₋ U+11A8</td>
</tr>
<tr>
<td>₋ U+11AD</td>
<td>₋ U+11A8 ₋ U+11A8</td>
</tr>
<tr>
<td>₋ U+11B0</td>
<td>₋ U+11A8 ₋ U+11A8</td>
</tr>
<tr>
<td>₋ U+11B1</td>
<td>₋ U+11A8 ₋ U+11B7</td>
</tr>
<tr>
<td>₋ U+11B2</td>
<td>₋ U+11A8 ₋ U+11B8</td>
</tr>
<tr>
<td>₋ U+11B3</td>
<td>₋ U+11A8 ₋ U+11B8</td>
</tr>
<tr>
<td>₋ U+11B4</td>
<td>₋ U+11A8 ₋ U+11B8</td>
</tr>
<tr>
<td>₋ U+11B5</td>
<td>₋ U+11A8 ₋ U+11C0</td>
</tr>
<tr>
<td>₋ U+11B6</td>
<td>₋ U+11A8 ₋ U+11C1</td>
</tr>
<tr>
<td>₋ U+11B7</td>
<td>₋ U+11A8 ₋ U+11C2</td>
</tr>
<tr>
<td>₋ U+11B9</td>
<td>₋ U+11A8 ₋ U+11B8</td>
</tr>
<tr>
<td>₋ U+11BB</td>
<td>₋ U+11B8 ₋ U+11A8</td>
</tr>
</tbody>
</table>
gines that have Unicode support are generally conformed to Level 1 [13]. Java regular expression class has a Level 1 conformance together with Canonical Equivalents (RL2.1) from Level 2 [14]. IBM’s ICU regular expression package is also conformed to Level 1, but instead has support for Default Word Boundaries (RL2.3) and Name Properties (RL2.5) from Level 2 [15].

Heninger [13] has explained on some of the aspects of applying Unicode in regular expression syntaxes. He has looked at how to define character set using character classes and ranges, together with Unicode character properties. If we need characters for a particular language or writing system, character properties such as script and block can be used instead. Also, he has presented some issues about the use of different normalisation form in representing characters and how this could affect regular expression matching.

3. Handling Hangeul in Regular Expression

Hangeul script is known to be a featural syllabic script or alphabetic syllabary, where it is a syllabary but at the same time exhibits alphabetic characteristics [2], [4], [12]. The alphabetic part of Hangeul script is the jamos. Unlike other syllabic script such as Hiragana or Katakana used in Japanese, we can apply regular expression syntax for Hangeul just like how we would do it for Latin alphabets. In order for us to make use of the alphabetic characteristics, pre-composed Hangeul syllables need to be decomposed into a sequence of jamos.

Although modern Hangeul text can be represented in either in NFC or in NFD, we would consider the input text represented in pre-composed Hangeul syllables having the code points ranging from U+AC00 to U+D7A3, which is more efficient to process as suggested by Ahn and Park [10]. Normalisation process can be applied to a given text if it is not in NFC. For texts that require the use of Hangeul jamo, which commonly found in texts dealing with Korean grammar, jamo shown in Fig. 1 can also be used, as well as the two filler characters U+115F (leading consonant filler) and U+1160 (vowel filler).

Regular expression engines that can handle Hangeul syllables can easily match a pattern literally to a Hangeul text in NFC, just like how it would match a pattern to an ASCII text without further processing on the Hangeul text itself. In this case, the regular expression engine treats the Hangeul syllable as if it is a set of distinct characters, albeit greater in number compared to ASCII.

3.1 Decomposition

Decomposition algorithm given in UAX#15 is insufficient to properly handle Hangeul to produced desired result of finding the word 하 to using the same example in Sect. 1 as shown in Table 5. However, using the new decomposition process shown in the third column can achieve the result. Implementation of Hangeul syllable decomposition in various libraries, such as those found in IBM ICU 4.2, Microsoft .NET Framework 3.5 and Java 6 are based on UAX#15. The new decomposition process however, has made its way to a utility function in Perl [17].

The new decomposition process transforms Hangeul from syllabic representation into alphabetic representation, giving us more flexibility to define our regular expression syntax. The decomposition process can be applied to a given input text in two different ways before evaluating on the regular expression syntax, namely Full Decomposition and Semi Decomposition.

3.1.1 Full Decomposition

In Full Decomposition, input text is fully decomposed into a sequence of basic syllables internally before regular expression pattern is applied. This allows us to define a pattern using jamo directly. The matching of a pattern with a text is straightforward, and can be done using the same method as matching English letters. For example, we need to match the pattern /+/ against an input text of 공무원-이요 gonghuhaess-eoyo, the input text will first be decomposed as 꾹또요 + signs, and the matching process will follow subsequently. We can specify the pattern as a pre-composed form /+/ as well and applied the same decomposition to the pattern.

Decomposing syllables across a phrase or a string of words with similar jamo does not create ambiguity to match jamo that can function as both leading and trailing consonant as shown in Fig. 5. 꾹 for example has two different code points in Unicode: U+1100 for leading consonant and U+11A8 for trailing consonant. This separation of code point property for representing leading consonant and trailing consonant in Unicode eliminates the need for any syllabic mark to demarcate the boundary of a syllable. The examples in Fig. 5. 닭과 dalggwa will not be interpreted as 닭과 dalggwa, and 헛살 haess-ssal will not be interpreted as 헛살 haess-ssal and vice versa.

It is interesting to note that although we can process regular expression via romanisation, where the Hangeul text

<table>
<thead>
<tr>
<th>Table 5</th>
<th>Decomposition of first syllable in Table 1 examples.</th>
</tr>
</thead>
<tbody>
<tr>
<td>함 han</td>
<td>함 han</td>
</tr>
<tr>
<td>해 hae</td>
<td>헤 hae</td>
</tr>
<tr>
<td>할 hal</td>
<td>할 hal</td>
</tr>
<tr>
<td>하 해스-살 haess-sal</td>
<td>하 해스-살 haess-sal</td>
</tr>
<tr>
<td>할 할 bal bal</td>
<td>할 할 bal bal</td>
</tr>
</tbody>
</table>

Fig. 5 Full decomposition example.
and pattern are romanised beforehand without any decomposition involved. However, we do not recommend such approach. Not only it is unnatural for end user to represent Korean, it has ignored the structure of the Hangeul script, and thus giving incorrect result along the way. Take for example an input of the word “데포판”, in which we wish to find a match for “ppy”. Although using romanisation we can find a match, but judging from the Hangeul text, we can clearly see that this is a mismatch.

Implementing full decomposition to Hangeul text and pattern can be done by allowing a flag to be set. Without enabling full decomposition, pattern matching will be done on the literal level. Java has a named constant called Pattern.CANON_EQ flag for canonical equivalent matching, but that will only work for an exact match of all decomposed character sequence of pattern, but not just a part of it. Also, it is meant to be set for pattern only and using jamo directly does not result a match. For example, a pattern “/서/” will match the text “서” in Java, but not the reverse.

Deploying full decomposition to regular expression engine to support Hangeul enables Korean text to be searched like alphabetic text. However, it requires the text to be decomposed entirely and this can have some impact on performance, especially for long text or pattern. Using a set of Korean texts that has a balance of both spoken and written Korean, we have analysed the full decomposition overhead incurred to the UTF-8 encoded text files. Each text file has a size of 44,600 ± 5.5% bytes, with an average of 12,153 pre-composed Hangeul syllables in it.

Figure 6 shows the average time for full decomposition on our test data using UAX #15 algorithm, new decomposition and also internal library of programming languages. An additional external library ICU4J is added in Java for comparison. We have included a control for each test to single out the overhead of file I/O. All internal and external libraries returned an identical output to UAX #15 algorithm that decompose Hangeul syllable that retains complex jamo as it is. Both C# and Java gave similar result for the same test but varied on different decomposition approaches within a same programming environment.

The number characters increase substantially after decomposition as shown in Fig. 7. Decomposing Hangeul syllables into jamos has made an approximate 146% increase on average in character count for UAX #15 approach. An additional 8.3% increment of characters can be observed from further decomposing using new decomposition. The file size after decomposition recorded on average 95.7 KB and 103 KB for UAX #15 and new decomposition respectively. The big step increment of characters could have a toll on regular expression engine, making full decomposition unsuitable for long input. Instead of full decomposition, we recommend using semi decomposition.

3.1.2 Semi Decomposition

As opposed to full decomposition, semi decomposition does not require an input text to be decomposed before processing. Semi decomposition will decompose Hangeul syllable on demand depending on how the regular expression syntax is defined. Analogous to turning on and off case insensitive matching mode for portion of a pattern by using a flag (?i), we can use similar construct for enabling and disabling decomposition matching mode for a certain portion of a pattern.

Using the example shown earlier, we can redefine the pattern to have a semi decomposition construct, such as (?d) and (?-d) to become “/서/” to become “/서/”. The flag d defined in the construct will initiate the decomposition of a Hangeul syllable and determine whether the syllable contains a leading consonant h, a vowel a and any other letters (the .* sub-expression) that might follow after the decomposition. Compared to full decomposition, the literal 공부 will first be matched and only upon finding a match, we will proceed with decomposing on what follows after the literal. This can help to speed up the matching, where the engine will first have to fulfill the literal matching using existing matching technique without having to run a decomposing process.

Apart from using a decompose flag, we propose another shorthand [=L:V:T=], inspired by the POSIX character equivalents syntax for defining Hangeul syllable that requires decomposition. This is useful to solve the ambiguous matching that might stem from using character class
with repetition quantifier. In the previous example, we define \( (?d) \& \cdot \star (?d) \) as one way to take in all the remaining jamo that might result from decomposing a Hangeul syllable, if any. This creates an ambiguity on the engine to decide when to stop decomposition. Using the proposed shorthand of \([=\text{L} \cdot \text{V} \cdot \text{T} =\], we can change the decomposition portion to become \([=\& \cdot \cdot \cdot \& =\]). Thus, we have made it clear that the decomposition will stop once we found a match from the original pre-composed syllable itself.

3.2 Any Character

Regular expression has a special metacharacter “." (dot) for matching any character. In Unicode, what constitute a character in a writing system can be different depending on how a character is constructed. In certain implementation of regular expression such as Perl, there is another metacharacter “\( \&X \)” that operates on the base character together with any remaining combining characters [16]. We do not require this additional metacharacter to be supported for our proposed matching technique, since our input text is generally made up of pre-composed Hangeul syllables, using \( \& \) is sufficient for referring to any character.

Combining “.” metacharacter with semi decomposition, the \( \& \) will now carry two meanings. If the \( \& \) is located within a decomposition section, it refers to a jamo. If it is located outside a decomposition section, it refers to any literal syllable or letter as what a dot would normally be referred to. Consider the pattern \( /[^{+\&}]\{=\& : \cdot \cdot \cdot \& =\}[^{+\&}]\) for example, it will match these following input texts: 더 빛여요, deo meog-eyo, 만히 빛여요, manh-i meog-eo-ye, 만히 빛여요, manh-i meog-eoss-eyo.

3.3 Character Range

Representing modern Hangeul character range in regular expression can be done using Unicode properties, \( \text{script} \) and \( \text{block} \)[16]. Unicode has provided a list of properties in which we can use it to refer to a defined range in Unicode standard. These properties can be accessed using the “\( \&\)’ construct. For example, using this regular expression pattern \( \&p\{\text{script=Hangul}\}+, \) we can search in a given text for Hangeul characters. The \( \text{script} \) value \( \text{Hangul} \) consists of more than just the jamos and the pre-composed syllables. It also includes all the half-width Hangeul letters, circled Hangeul letters like \( \&F \) and some other characters that made up of the total 11,737 code points [19].

If we are interested to access a subset of characters from the script defined range, we can apply the set operators available, such as intersection or subtraction [16]. For using a specific range of characters within a script, there is another property called \( \text{block} \). Block value \( \text{Hangul} \) Jamo is a set of characters ranging from U+1100 to U+11FF for jamo characters while the value \( \text{Hangu} \) Syllables is for all the pre-composed Hangeul syllables. Alternately, we can also use this pattern \( /[(\text{\&AC00-\&D7A3})|\text{\&1100-\&11C2})+]/ \) for searching all modern Hangeul syllables and jamos\(^\dagger\).

The use of character class range construct via code point may look cryptic for users who are not familiar with the exact code point value. Instead, we proposed the use of Hangeul character directly just like how we would do it for ASCII to match character ranging from \( d \) to \( m \) /\( [d-m] \). Since Hangeul syllables in Unicode are sorted, using Hangeul directly would only require a direct mapping translation. There are several user aspects to be taken into consideration, however, if we were to use Hangeul to specify range.

For example, we would like to search for the first four ordered list heading, commonly written in this order: \( \text{가 \( g \)} \), \( \text{나 \( n \)} \), \( \text{다 \( d \)} \), \( \text{라 \( l \)} \). Intuitively, we would specify the pattern as /\( +[가나다라]+\)/, where we assume the list heading can be written in a single phrase. If character range implementation is done using direct code point mapping, the pattern would also match \( \text{가 \( g \)} \) or even \( \text{산들 \( s \)} \) (sandalk) which is not what we have expected in the first place. This problem appears due to both of the character \( \text{가 \( g \)} \) U+AC01 and \( \text{산 \( s \)} \) U+B2ED fall within the defined range of U+AC00 to U+B77C.

One approach to deal with this limitation is to use alphabetic range instead of syllabic range. We assume the regular expression engine has the ability to handle our proposed decomposition shorthand, the pattern for the previous example can be defined as /\( +[^\&-\&: \cdot \cdot \cdot \& =\}]\)/. Notice that we are embedding the character range metacharacter directly into the decomposition shorthand. Also, We do not need to specify the exclusion list of \( ^\& \text{\&-\&: \cdot \cdot \cdot \& =\} \), since decomposing those two will result in two basic leading consonants, but in our pattern, we are accepting only one jamo.

3.4 Word Boundary

Word boundary defines the location where a start and an end of a word. In English text, a word in a sentence can be easily spotted by identifying a continuous group of characters between two spaces or punctuation marks. Korean text contains space to separate part of speech and grammatical components [5]. However, each phrase that is separated by spaces does not fit the true meaning of word, since a phrase might contain more than one word, with or without grammatical particles. Segmenting Korean text into words within a single phrase would require some form of dictionary lookup [6] or by analysis through natural language processing. Word boundary is useful for information retrieval, machine translation and other useful linguistic processing of words [20].

Given that segmentation of Hangeul text into words requires additional lexical processing, for general use of regular expression, we will treat a word based on phrases sep-

\(^\dagger\)This range of jamos is stated in a generalised manner for simplicity, where in fact the range from U+1100 to U+11C2 contains jamos for old Hangeul as well. For strictly modern Hangeul jamo only, use \( [\text{\&1100-\&1112}], [\text{\&1161-\&1175}] \) and \( [\text{\&11A8-\&11C2}] \) instead.
ated by space and punctuation. Although this does not fit the actual definition of a word boundary, we assume a user can decide on how to formulate the regular expression syntax to achieve a desirable result using just punctuation or space separated phrases. The following examples explain on how space between phrases can be useful and how a user could decide on the processing of word needed.

Assume we have a regular expression enabled text editor, and we would like to replace a negated transitive verb “N을/를 안 V” into a non-negated form, we can use back references to extract the text with a pattern. If the given input can match this pattern /\([\ldots](\text{음/을})?\)\s\an\s(\ldots)\/, we can extract the transitive verb out by using /\1 \2/. Thus, for a given input 아이를 안 안아요 aileul an anayo, using search and replace, we would get a result of 아이를 안아요 aileul anayo using back references. The use of space helps us to distinguish the negation word 안 an and the verb conjugation 안아요 anayo that share the same first syllable.

4. Applied Examples

4.1 Trailing Consonant Detection

It is common in Korean language to append a suffix to a word to indicate part of speech or as a grammatical element. Choice of suffixes depends on a presence or an absence of trailing consonant (받침 badchim) in a word that is going to be attached. One naive way of achieving this, is to enumerate the words with trailing consonant. As we have explained in Sect. 2.2, it is impractical as there are 10,733 syllables that contain a trailing consonant. One way to detect for words with trailing consonant is to define a pattern similar to /\([\ldots]+\ldots+:.*\)=\2/. The third part of the shorthand indicates that we need a syllable that contains at least one trailing consonant.

4.2 Causative Verb to Basic Verb

Causative verbs are formed by adding a suffix such as 이 i, 히 hi, 리 li, 리 gi, 우 u or 추 chu to the basic verb form [21]. One particular suffix, the 우 u suffix requires a change in basic form with an additional 르 i. Extracting basic verb from 우-suffixed causative like obtaining 타다 tada from 태우다 taeuda and 서다 seoda from 새우다 se-uda for example requires decomposition of complex jamos || ae and || e into basic jamos of ] a i and [ e o i. Using the decomposition shorthand, we can define the search pattern as /\([\ldots](\ldots)\)\{i\} \1/ and the replace pattern as /\1:2::=\2/ to obtain basic verb form.

4.3 Irregular Verb Conjugation

In some operations, such as spell checking, we would like to verify for proper consonant change in a word when a criterion is met. d irregular verb for example, will change its trailing consonant to l whenever a (null) is present in the next word. Irregular verbs like 간다 geoddar or 둘다 deurda for example, can be conjugated as 간아요 geol-eo-yyo and 둘어져 deul-eoseo, where the subsequent verb contain the (null) consonant. Thus, we can check for correct consonant was used by applying this pattern /\([\ldots]:\ldots\)\{i\}=\1\{i\}\{r\}/. If a mistake is found, we can replace it with /\1:2::=\2/.

4.4 Past Tense Verb Formation

Verbs that end with a vowel o with no trailing consonant such as 보다 boda and 오다 oda can change from present tense to past tense by adding 이-ass to the base verb, and at the same time allows a contraction by forming a diphthong na. For searching verbs with o o in a text, we can use the pattern /\([\ldots]:\odot=\1\)/ and the substitution pattern can be defined as /\1:1:1:1:1:1:1:1:1:1:/2/.

4.5 Attaching 아 Particle

The eu particle is usually appended to the end of a word when a trailing consonant is present. Unlike other particles that are appended to the end of a word by judging on a presence of trailing consonant, eu eu has a special exception where words that end with 는 l are not included. Thus, for appending suffix like (으) for example, we can search using the pattern /\([\ldots]:\odot=\1\\{\odot\}\)/ and substitute the result with this pattern \1\odot. Similarly, for words without trailing consonant or they contain trailing consonant 는 l, we can append 변 myeon by using this search pattern /\1\odot=\2/.

4.6 Forming Pronunciation Guide

Pronunciation of words in Korean can differ from actual spelling depending on pronunciation rules [22]. We can generate a simple pronunciation guide by applying a number of regular expressions to map Hangeul syllables to the target pronunciation step by step. The 27 trailing consonants for example can be mapped to the 7 sounds of ㄱ k, ㄴ n, ㄷ t, ㄹ l, ㅁ m, ㅂ p, and ㅇ ng using a regular expression for each sound. Search pattern of /\([\ldots]:\ldots\)\{개\}\{\odot\}\{\?\}=\1/ can be used to map for ㅅ s, and the remaining sounds can be done likewise.

For a more refined pattern, exclusion of ㅎ h from the search pattern may be useful to avoid sound lost for possible consonant assimilation, or by adding an exclusion of (null) to subsequent leading consonant for checking possible linking or liaison. Both of which we can apply a separate regular expression to handle them accordingly.

Pronunciation guide for the other pronunciation rules that involve sound change such as nasalisation, palatalisation, or tensification can be formed easily with the aid of complex consonant decomposition. Tensification of ㄱ g, ㄷ d, ㅅ s, and ㅈ j for example after ㅅ(ㅈ) n(ŋj) and ㅈ(ㅊ) m(lm) can be searched by /\([\ldots]:\ldots\)\{\?\}\{\?\}=?\{\?\}/.
5. Conclusion

In this paper, we have seen how canonical decomposition can help to expand the expressibility of a regular expression pattern for Hangeul. The innate alphabetic nature of Hangeul allows us to break it down into smaller jamo, and processing of regular expression can be carried out just as it is now done for existing Latin based script. We have explained why the existing canonical decomposition algorithm for Hangeul in UAX #15 is insufficient to properly handle Hangeul text, instead, we need to use the proposed enhancement of canonical decomposition by Kim.

The simplicity of fully decomposing Hangeul syllable into individual basic jamo comes with a price. Data length grows double as we decompose the Hangeul syllable. One solution to overcome this issue is by decomposing Hangeul syllable on demand, or we called it semi decomposition. We have also proposed a shorthand notation that can be used in semi decomposition, which extends the existing processing scope of Hangeul text beyond literal matching, and is especially useful on context related to Korean grammar. Our proposed method can be applied in various contexts such as validation of well formed grammar phrases, extraction of base form word from inflection and conjugation, transformation of word using suffixes, generation of pronunciation guide, and many more possible areas of application.

The lack of decomposition process in numerous regular expression engines in text manipulating program not only limits the scope of processing Hangeul text, but also the absence of the ability to analyse word structure and context can affect the accuracy of identifying the correct Hangeul text. We are exploring the possibility of integrating another technique such as morphological analysis to enhance our method to manipulate Hangeul text, especially in handling irregular and exception cases. We are also looking into the composition process for Hangeul syllable to further improve on the expressibility and application of regular expression for Hangeul text.

References


Hee Yuan Tan received his Bachelor IT degree majoring in Software Engineering from the Multimedia University, Malaysia in 2004, and his Master in Ubiquitous IT from Dongseo University, Korea in 2010. He is currently a technical project manager in Mobile FPX, and has worked for more than 3 years as a software developer prior to enrolling in his graduate study. His research interests are currently in the fields of ubiquitous computing, and software development technique.
Hyotaek Lim received his B.S. in Computer Science from Hongik University in 1988, M.S. in Computer Science from POSTECH and the Ph.D. in Computer Science from Yonsei University in 1992 and 1997 respectively. From 1988 to 1994, he had worked for Electronics and Telecommunications Research Institute as a research staff. Since 1994, he has been with Dongseo University, where he is currently a professor in the Div. of Computer & Information Engineering. His research interests include ubiquitous and mobile networking, storage area networks, and network management.