ISO/IEC JTC 1/SC 2 N 3925

DATE: 2007-04-17

L2/07-112

ISO/IEC JTC 1/SC 2
Coded Character Sets
Secretariat: Japan (JISC)

DOC. TYPE		Working group document
TITLE		A proposed method of preprocessing Hangul to be included as an Annex of ISO/IEC 14651
SOURCE		Republic of Korea
PROJECT		JTC 1.02.14651.00.00.02
STATUS		This document will be considered at the OWG-SORT meeting to be held in Frankfurt, Germany, April 2007.
ACTION ID		FYI
DUE DATE		
DISTRIBUTION		P, O and L Members of ISO/IEC JTC 1/SC 2 ; ISO/IEC JTC 1 Secretariat; ISO/IEC ITTF
ACCESS LEVEL		Open
ISSUE NO.		269
FILE	**NAME** **SIZE (KB)** **PAGES**	02n3926.pdf 2687 16

Secretariat ISO/IEC JTC 1/SC 2 - IPSJ/ITSCJ (Information Processing Society of Japan/Information Technology Standards Commission of Japan)* Room 308-3, Kikai-Shinko-Kaikan Bldg., 3-5-8, Shiba-Koen, Minato-ku, Tokyo 105-0011 Japan *Standard Organization Accredited by JISC
Telephone: +81-3-3431-2808; Facsimile: +81-3-3431-6493; E-mail: kimura @ itscj.ipsj.or.jp

ISO/IEC JTC1/SC2/SC2 N 3xxx

(= Korea JTC1/SC2 K1562B)

2007-04-xx

Universal Multiple Octet Coded Character Set
International Organization for Standardization
Organisation internationale de normalisation

Doc Type: Working Group Document

Title: A proposed method of preprocessing Hangul to be included as an annex of ISO/IEC 14651

Source: Republic of KOREA (KIM, Kyongsok)

Status: National Position

Action: For consideration by JTC1/SC2/OWG-SORT

A proposed method of preprocessing Hangul
to be included as an annex of ISO/IEC 14651

Purpose:

Currently ISO/IEC 14651 cannot collate Hangul data in ISO/IEC 10646 properly, especially data in Old Hangul.

Therefore, we propose a method of preprocessing Hangul data in ISO/IEC 10646 so that the output can be used as an input to ISO/IEC 14651 supporting program, which will then collate Hangul properly.

1. Introduction

1.1 BNF

We want to specify the rules of transforming Hangul data in UCS so that Hangul can be easily collated by ISO/IEC 14651-supporting software.

Since we will specify the transforming rules in a widely used notation, called a context-free grammar (or grammars, for short) or BNF (for Backus-Naur Form or Backus-Normal Form), we will briefly introduce BNF.

The following explanations come from [Compilers, Principles, Techniques, and Tools. Aho, Sethi, and Ullman. Addison-Wesley Publishing Company. 1985]. Some parts are slightly edited so that we can better understand in ISO/IEC 14651 context.

For example, an if-else statement in C has the form

if (expression) statement else statement

The if-else statement is the concatenation of the keyword if, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword else, and another statement. The structure can be expressed in BNF as

<stmt> -> if (<expr>) <stmt> else <stmt>

in which the arrow may be read as "can have the form". Such a rule is called a production. The keyword if and the parentheses are called "tokens". <expr> and <stmt> represent sequence of tokens and are called non-terminals.

A context-free grammar has four components:

1) A set of "tokens", known as "terminal symbols".
2) A set of "non-terminals".
3) A set of productions where each production consists of a non-terminal, called the left side of the production, and arrow ("->"), and a sequence of tokens and/or non-terminals, called the right side of the production.
4) A designation of one of the non-terminals as the start symbol.

We specify the transformation rules (or grammars) by listing their productions, with the productions for the start symbol listed first.

In this paper, non-terminals are shown enclosed within a pair of brackets, e.g., <si>, <si1>, <si2>, <si3>. Terminals are shown without brackets, e.g., U1100, U1162, where U1100 is Hangul letter Giyeog and U1162 is Hangul letter Nieun).
Productions with the same non-terminal on the left can have their right sides grouped, with the alternative right sides separated by the vertical bar symbol "¦", which we read as "or".

Example 1.1. Consider expressions consisting of two digits separated by plus or minus signs, e. g., 9 + 2, and 3 - 1. The following grammar describes the syntax of these expressions. The productions are:

<expr> -> <term> + <term> (production 1a)
<expr> -> <term> - <term> (production 1b)
<term> -> 0 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9 (production 1c)

The right sides of the two productions with non-terminal <expr> on the left side can equivalently be grouped:

<expr> -> <term> + <term> ¦ <term> - <term>

<expr> and <term> are non-terminals with <expr> being the starting non-terminal because its productions are given first. +, -, 0, 1, ..., and 9 are terminals (or tokens).

A grammar derives strings by beginning with the start symbol and repeatedly replacing a non-terminal by the right side of a production for that non-terminal. The strings that can be derived from the start symbol form the language defined by the grammar.

Example 1.2. The language defined by the grammar of Example 1.1 consists of two digits separated by a plus or minus sign.
The ten productions for the non-terminal <digit> allow it to stand for any of the 0, 1, ..., 9. From production 1c, a single digit by itself is a term. Productions 1a and 1b express the fact that if we take any digit and

follow it by a plus or minus sign and then another digit we have an expression.

a) 9 is a <term> by production 1c
b) 9 - 5 is an <expr> by production 1b, since 9 is a <term> and 5 is also a <term>

Example 1.3. An English alphabet consists of 26 letters; 5 of them are vowels and the others are consonants. That can be expressed as follows:

```
<eng-alphabet> -> <vowel> | <consonant>
<vowel> -> a | e | i | o | u
<consonant> -> b | c | d | f | g | h | j | k | l | m | n |
               p | q | r | s | t | v | w | x | y | z
```

1.2 Syntax-directed translation

A translation scheme is a context-free grammar in which program fragments called "semantic actions" are embedded within the right sides of productions. The position at which an action is to be executed is shown by enclosing it between braces ("{ }") and writing it within the right side of a production, as in

```
<expr> -> <term> + <term> { print('+') }
<expr> -> <term> - <term> { print('-') }
<term> -> 0 {print ('0')}
<term> -> 1 {print ('1')}
...
<term> -> 9 {print ('9')}
```

A translation scheme generates an output for each sentence x generated by the underlying grammar by executing the actions in the order they appear.

The above translation scheme translates a given expression into postfix form. This scheme accepts expressions having only two numbers and a plus or minus in between. For example, '9 + 5' or '9 - 5' is accepted, but '1 + 2 - 3' or '9 - 8 - 7' is not.

Expressions such as 3 + 5 or 9 - 8 are called infix notation, since a plus or minus sign, which is a binary operator, are written between two numbers. With a postfix notation, the binary operator (a plus or minus sing) is put after two numbers.
For example, the postfix notation for 3 + 5 is 3 5 + (plus sign is put 'after' two numbers, 'not between' two numbers).

Let's see how 9 - 5 is translated into 9 5 -. We start with the production "<expr> -> <term> + <term> {print('+')}". The first part of the right side is <term>. Then the production "<term> -> 9" matches "9" and '9' is printed. Now '+' of the right side does not match with '-'. Therefore we give up "<expr> -> <term> + <term> {print('+')}".

Now we try the next production "<expr> -> <term> - <term> {print('-')}". The production "<term> -> 9" matches with '9' and prints '9'. Then '-' in the production matches with '+'; however nothing is printed at this point. Now the production "<term> -> 5" matches with '5' and prints '5'.

The production "<expr> -> <term> - <term> {print('-')}" matches with the given string '9 - 5'. At this point, '-' is printed. We are done. Therefore, the final output is '9 5 -', which is a postfix notation for the given expression '9 - 5'.

2. A proposed method of preprocessing Hangul

In Section 1, we studied the basic concept of BNF and translation scheme. With this background, let's see examples showing how to transform data in Hangul.

2.1 Example 01A (a simple example using only a few Hangeul characters)

- For simplicity, we included only two syllable-initial characters, two syllable-peak characters, two syllable-final characters, and two fill characters.

- Some exercises are shown below to show how input characters are transformed according the to given rules.

```
------------------------------------------------------------------------
Example 01A (abridged from Example 02A; this Example is for demo purpose)
------------------------------------------------------------------------

/* constants */
SI-FILL == U115F /* syllable-initial FILL character */
SP-FILL == U1160 /* syllable-peak    FILL character */

/* Hangeul syllables */
/* LS: Left side */
/* RS: Right side or pattern */
/* we start from <root> */
/* FAIL cancels temporary OUTPUT */
/* 'finalize OUTPUT' finalizes temporary OUTPUT */
/* action is shown within { }.  */
/* Most actions are to output some characters. */

/* rule R01B accepts four combinations of characters:
   U1100 U1161 ¦ U1100 U1163 ¦  U1102 U1161 ¦ U1102 U1163
   GA            GYA           NA            NYA */

Rule#  LS         RS (or pattern)
-----  -------    ----------------------------------
ROOT <root>   -> <hg-syl> {finalize OUTPUT}
R01B <hg-syl> -> <si> <sp> ¦
R01F            SI-FILL { print('SI-FILL') } <sf>

/* <si> : syllable-initial letters
   <si1>: syllable-initial simple letters */
```

```
R11D <si>  -> <si1>
R12A <si1> -> U1100 { print('U1100') } ¦
R12B          U1102 { print('U1101') }
              /* output without any transform */

/* <sp> : Syllable-peak letters
   <sp1>: syllable-peak simple letters */
R21D <sp>  -> <sp1>
R22A <sp1> -> U1161 { print('U1161') } ¦
R22B          U1163 { print('U1163') }
              /* output without any transform */

/* <sf> : syllable-final letters
   <sf1>: syllable-final simple letters */
R31D <sf>  -> <sf1>
R32A <sf1> -> U11A8 { print('U11A8') } ¦
R32B          U11AB { print('U11AB') }
              /* output without any transform */
```

Exercise 1.1 input string -> U1100 U1161

- An input string represents Hangeul syllable "GA".
- Unless "FAIL" is mentioned, the pattern match succeeds.
- When "finalize OUTPUT" is executed, a temporary output becomes final.

- Using the above rules, we will process the input string.
- We start with rule ROOT <root>. Its right side is <hg-syl>.

- Then we go to rule R01B <hg-syl>. Its right side is <si>.
  Then we go to rule R11D <si>. Its right side is <si1>.
  Then we go to rule R12A <si1>. Its right side "U1100" matches input character U1100. At this point, 'U1100' is printed by the action { print('U1100') } in rule R12A.
  Now we are done with rule R12A <si1>.
  Then we back up to R11D <si1>. We are done with rule R11D <si1>.
  Then we back up to R01B <si>. We are done with rule R11D <si1>.

- Then we back up to rule R01B <hg-syl>. We are done with <sp> of rule R01B and then we try <sp> in rule R01B.
  Then we go to rule R21D <sp>. Its right side is <sp1>.
  Then we go to rule R22A <sp1>. Its right side "U1161" matches input character U1161. At this point, 'U1161' is printed by the action { print('U1161') } in rule R22A.
  Now we are done with rule R22A <sp1>.

Then we back up to R21D <sp1>. We are done with rule R21D <sp1>.
Then we back up to R01B <sp>.
At this point, we are done with rule R01B <hg-syl>.
Then we back up to ROOT <root>. We are done with rule ROOT <root>.
At this point, temporary OUTPUT is finalized.

- In this exercise, we do not change anything. We just try to match the rules against the input string and then print out without any transformation.

pattern	MATCH/FAIL	OUTPUT by actions
ROOT <root>		
R01B <hg-syl>		
R11D <si>		
R12A <si1>	R12A MATCH	U1100
	R11D MATCH	
	R01B MATCH	
R01B <sp>		
R21D <sp1>		
R22	R22A MATCH	U1161
	R21D MATCH	
	R01B MATCH	
	ROOT MATCH	finalize output

* final output -> U1100 U1161

Exercise 1.2 input -> U115F U11A8

- input file represents Hangul syllable-final letter "Giyeog".

pattern	match/FAIL	OUTPUT by actions
ROOT <hg-syl>		
R01B <si>		
R11D <si1>		
R12A	R12A FAIL	
R12B	R12B FAIL	
	R11D FAIL	
	R01B FAIL	

```
R01F            SI-FILL MATCH     U115F
R31D <sf1>
R32A            R32A MATCH        U11A8
                R31D MATCH
                R01F MATCH
                ROOT ROOT         finalize output

* final output = U115F U11A8
```

2.2 Exmaple 02A

- This example transforms one Hangeul syllable into 9 code positions: 3 code positions for each of syllable-initial, syllable-peak, and syllable-final character, respectively.
- Some EMPTY characters are intentionally inserted so that we can collate Old Hangeul correctly.

Example 02A (modified from Example 01A; this Example is for demo purpose)

```
/* constants */
SI-FILL == U115F
SP-FILL == U1160

/* Hangeul syllables */
      LS          RS (or pattern)
-------------     ---------------------------------------------
ROOT <root>    -> <hg-syl> { print('finalize OUTPUT') }
R01B <hg-syl> -> <si> <sp> { print('U0000 U0000 U0000') }
                 /*FAIL cancels relevant temp output*/
R01F             SI-FILL { print('SI-FILL U0000 U0000 U0000 U0000 U0000') }
<sf>

/* syllable-initial letters:  <si1> a syllable-initial simple letter */
R11D <si>  -> <si1> { print('U0000 U0000') }
R12A <si1> -> U1100 { print('U1100') } ¦
R12B          U1102 { print('U1102') }  /* output without any transform */

/* Syllable-peak letters:  <sp1> a syllable-peak simple letter; */
R21D <sp>  -> <sp1> { print('U0000 U0000') }
R22A <sp1> -> U1161 { print('U1161') } ¦
R22B          U1163 { print('U1163') } /* output without any transform */
```

```
/* syllable-final letters:  <sf1> a simple letter; */
R31D <sf>  -> <sf1> { print('U0000 U0000') }
R32A <sf1> -> U11A8 { print('U11A8') } ¦
              U11AB { print('U11AB') } /* output without any transform */


Exercise 2.1 input = U1100 U1161

  - input file represents Hangeul syllable "GA".


----------     ------------      ----------------------------------------
pattern        match/FAIL        OUTPUT by actions
----------     ------------      ----------------------------------------

ROOT <hg-syl>
R01B <si>
R11D <si1>
R12A           R12A MATCH        U1100
               R11D MATCH        U0000 U0000
               R01B <si> MATCH
R01B <sp>
R21D <sp1>
R22A           R22A MATCH        U1161
               R21D MATCH        U0000 U0000
               R01B MATCH        U0000 U0000 U0000
               ROOT MATCH        finalize output

* final output = U1100 U0000 U0000   U1161 U0000 U0000   U0000 U0000 U0000



Exercise 2.2 input = U115F U11A8

  - input file represents Hangul syllable-final letter "Giyeog".

----------     ------------      ----------------------------------------
pattern        match/FAIL        OUTPUT by actions
----------     ------------      ----------------------------------------
ROOT <hg-syl>
R01B <si>
R11D <si1>
R12A           R12A FAIL
               R12B FAIL
```

```
               R11D FAIL
               R01B FAIL
R01F SI-FILL   SI-FILL           U115F U0000 U0000  U0000 U0000 U0000
R01F <sf>
R31D <sf1>
R32A U11A8     R32A U11A8        U11A8
               R31D <sf1>        U0000 U0000
               R01F <sf>
               ROOT <hg-syl>     finalize output = (shown below)

final output = U115F U0000 U0000   U0000 U0000 U0000   U11A8 U0000 U0000
-------------------------------------------------------------------------
```

2.3 Example 11A

- This is more or less a real example.

- This example can preprocess 11,172 Modern Hangeul syllables and other incomplete syllables.

```
-------------------------------------------------------------------------
Example 11A:
-------------------------------------------------------------------------

/* constant */
SI-FILL == U115F
SP-FILL == U1160

/* Hangeul syllable */
ROOT <root>   -> <hg-syl> {finalize OUTPUT}
R01A <hg-syl> -> <si> <sp> <sf> ¦
R01B             <si> <sp> { print('U0000 U0000 U0000') } ¦
R01C             <si> SP-FILL
                   {print('SP-FILL U0000 U0000 U0000 U0000 U0000') } ¦
R01D             SI-FILL { print('SI-FILL U0000 U0000') } <sp> <sf> ¦
R01E             SI-FILL { print('SI-FILL U0000 U0000') } <sp>
                   { print('U0000 U0000 U0000') } ¦
R01F             SI-FILL { print('SI-FILL U0000 U0000 U0000 U0000 U0000') }
                 <sf>


/* syllable-initial letters:
```

```
  <si1> a syllable-initial simple letter
  <si2> a syllable-initial 2-complex letter (composed of 2 simple letters)
  <si3> a syllable-initial 3-complex letter (composed of 2 simple letters)
*/

R11A <si>  -> <si1> <si1> <si1> ¦
R11B          <si1> <si1> { print('U0000') } ¦
R11C          <si1> <si2> ¦
R11D          <si1> { print('U0000 U0000') } ¦
R11E          <si2> <si1> ¦
R11F          <si2> { print('U0000') } ¦
R11G          <si3>

R12A <si1> -> U1100 { print('U1100') } ¦
R12B          U1102 { print('U1102') } ¦
R12C          U1103 { print('U1103') } ¦
R12D          U1105 { print('U1105') } ¦
R12E          U1106 { print('U1106') } ¦
R12F          U1107 { print('U1107') } ¦
R12G          U1109 { print('U1109') } ¦
R12H          U110B { print('U110B') } ¦
R12I          U110C { print('U110C') } ¦
R12J          U110E { print('U110E') } ¦
R12K          U110F { print('U110F') } ¦
R12L          U1110 { print('U1110') } ¦
R12M          U1101 { print('U1101') } ¦
R12N          U1102 { print('U1102') }

/* output without any transform */

R13A <si2> -> U1101 { print('U1100 U1100') } ¦
R13B          U1104 { print('U1103 U1103') } ¦
R13C          U1108 { print('U1107 U1107') } ¦
R13D          U110A { print('U1109 U1109') } ¦
R13E          U110D { print('U110C U110C') }

/* R14 <si3> ->  no si3 for modern Hangeul */

/* Syllable-peak letters:
  <sp1> a syllable-peak simple letter
  <sp2> a syllable-peak 2-complex letter (composed of 2 simple letters)
  <sp3> a syllable-peak 3-complex letter (composed of 3 simple letters) */

R21A <sp>  -> <sp1> <sp1> <sp1> ¦
R21B          <sp1> <sp1> { print('U0000') } ¦
```

```
R21C          <sp1> <sp2> ¦
R21D          <sp1> { print('U0000 U0000') } ¦
R21E          <sp2> <sp1> ¦
R21F          <sp2> { print('U0000') } ¦
R21G          <sp3>

R22A <sp1> -> U1161 { print('U1161') } ¦
R22B          U1163 { print('U1163') } ¦
R22C          U1165 { print('U1165') } ¦
R22D          U1161 { print('U1161') } ¦
R22E          U1169 { print('U1169') } ¦
R22F          U116D { print('U116D') } ¦
R22G          U116E { print('U116E') } ¦
R22H          U1172 { print('U1172') } ¦
R22I          U1173 { print('U1173') } ¦
R22J          U1175 { print('U1175') }

R23A <sp2> -> U1162 { print('U1161 U1175') } ¦
R23B          U1164 { print('U1163 U1175') } ¦
R23C          U1166 { print('U1165 U1175') } ¦
R23D          U1168 { print('U1167 U1175') } ¦
R23E          U116A { print('U1169 U1161') } ¦
R23F          U116C { print('U1169 U1175') } ¦
R23G          U116F { print('U116E U1165') } ¦
R23H          U1171 { print('U116E U1175') } ¦
R23I          U1174 { print('U1173 U1175') }

R24A <sp3> -> U116B { print('U1169 U1161 U1175') }
R24B          U1170 { print('U116E U1165 U1175') }


/* syllable-final letters:
  <sf1> a syllable-final simple letter
  <sf2> a syllable-final 2-complex letter (composed of 2 simple letters)
  <sf3> a syllable-final 3-complex letter (composed of 3 simple letters) */

R31A <sf>  -> <sf1> <sf1> <sf1> ¦
R31B          <sf1> <sf1> { print('U0000') } ¦
R31C          <sf1> <sf2> ¦
R31D          <sf1> { print('U0000 U0000') } ¦
R31E          <sf2> <sf1> ¦
R31F          <sf2> { print('U0000') } ¦
R31G          <sf3>

R32A <sf1> -> U11A8 { print('U11A8') } ¦
```

```
R32B          U11AB { print('U11AB') } |
R32C          U11AE { print('U11AE') } |
R32D          U11AF { print('U11AF') } |
R32E          U11B7 { print('U11B7') } |
R32F          U11B8 { print('U11B8') } |
R32G          U11BA { print('U11BA') } |
R32H          U11BC { print('U11BC') } |
R32I          U11BD { print('U11BD') } |
R32J          U11BE { print('U11BE') } |
R32K          U11BF { print('U11BF') } |
R32L          U11C0 { print('U11C0') } |
R32M          U11C1 { print('U11C1') } |
R32N          U11C2 { print('U11C2') }  /* output without any transform */

R33A <sf2> -> U11A9 { print('U11A8 U11A8') } |
R33B          U11AA { print('U11A8 U11BA') } |
R33C          U11AC { print('U11AB U11BD') } |
R33D          U11AD { print('U11AB U11C2') } |
R33E          U11B0 { print('U11AF U11A8') } |
R33F          U11B1 { print('U11AF U11B7') } |
R33G          U11B2 { print('U11AF U11B8') } |
R33H          U11B3 { print('U11AF U11BA') } |
R33I          U11B4 { print('U11AF U11C0') } |
R33J          U11B5 { print('U11AF U11C1') } |
R33K          U11B6 { print('U11AF U11C2') } |
R33L          U11B9 { print('U11B8 U11BA') } |
R33M          U11BB { print('U11BA U11BA') }

/* R34  <sf3> ->    no sf3 for modern Hangeul */
```

3. Conclusions

Currently ISO/IEC 14651 cannot collate Hangul data in ISO/IEC 10646, especially in Old Hangul, properly.

Therefore, we proposed a method of preprocessing Hangul data in ISO/IEC 10646 so that the output can be used as an input to ISO/IEC 14651 supporting program, which will then collate Hangul properly.

Rules in Section 2.3 transform one modern Hangul syllable (including incomplete syllables) is into 9 code positions. When Hangul data is transformed this way, it can be collated properly by ISO/IEC 14651

supporting program.

Once a collating rule for Old Hangul is established, rules in Section 2.3 can be easily extended so that Old Hangul data can be preprocessed..

* * *