Recursion Problems in Concatenation: A Case of Korean Morphology

Kiyong Lee
Department of Linguistics
Korea University
Seoul 136-701, Korea

November 10, 1995

Abstract

Without properly constraining recursions in generation, no language system can effectively operate. This is especially so with morphological generation, for each well-formed word must be finite in length and is accepted as such only when it actually occurs in text or ordinary usage. This difficulty is compounded when a language system tries to maintain a single rule of concatenation and apply it repeatedly in order to combine a nominal or verbal stem with a sequence of suffixes in a time-linear manner. In an agglutinative language like Korean, however, it can easily be demonstrated how a language system like Malaga, based on time-linear grammars like Hausser's [1] Left-Associative Grammar, can properly be implemented to constrain undesirable recursive loops in generation. Both nominal and verbal concatenations in Korean are treated in this paper to show how infinite loops can be blocked by imposing appropriate matching conditions on two adjacent input strings to concatenation.

1 Introduction

In Generative Grammar, the infinite magnitude of natural language is accounted for in terms of the notion of recursion. Without it, no syntactic embedding as in relative clause formation or *that*-complementation in English or even repeated modification as in an adjectival phrase "very, very, very long" can be explained with mathematical elegance.

In an ideal agglutinating language there is no problem of morphological recursion because the number of suffix classes is finite and their order of occurrence is fixed:

(1) `Stem--SX1--Sx2--....-SXn`

Furthermore, each position may be filled only once.

However, Korean deviates from the ideal type of an agglutinating language, because certain suffixes may occur in a reversed order:

(2) Stem--...SX1--sx...SXn
(3) Stem--...sx--SX1...SXn

A particular suffix **sx** generally appears after **SX1**, but it may also appear sometimes *before* **SX1**. The recursion problem resides in the possibility of re-adding **sx** after the string **...sx–SX1** because **sx** normally occurs after **SX1**.

Using an implementation of Korean morphology in the Malaga system, this paper will present a straightforward solution to the indicated problem of recursion in Korean morphology. With one single rule for adding the particles or endings, recursively applied, both nominal and verbal forms are successfully generated, each consisting of a stem and a sequence of particles or endings.

2 Morphological Generation using Malaga

Malaga is an acronym for "Malaga Accepts Left-Associative Grammar with Attributes". As a language system[1], it provides a general tool for both analyzing and generating word forms as well as sentences or text from left to right in a time-linear manner. In order to run the tool for a particular language like Korean, several specific files must be written. Among them are two list files, say **korean.seg** and **korean.lex**, and two rule files, **korean.all** and **korean.mor**.[2]

Consider the following nominal and verbal forms in the basic lexical entries of Korean:

```
(4) korean.lex
    a. nominal forms
       [Phon: "pwumo",    Class: N, Sem: "parent"];
       [Phon: "kyoswu",   Class: N, Sem: "professor"];
       [Phon: "sensaeng", Class: N, Sem: "teacher"];
    b. verbal forms
       [Phon: "ka", Class: V, Sem: "go"];
       [Phon: "cap", Class: V, Sem: "catch"];
       [Phon: "ket", Class: V, Default_feat: ir, Sem:"walk"];
```

Each item here has the attribute **Phon** and a string, enclosed in double quotes, for its value.[3] Each form is also marked with an appropriate **Class** and **Sem**antic features.[4] Among the three verbal stems, the first two are regular and the third one is irregular and is marked with **Default_feat: ir**.

The basic lexical entries listed in the file **korean.lex** are transformed into their respective allomorphs by the rule file **korean.all**. This rule file has two functions. One is to generate allomorphic forms based on the basic lexical file **korean.lex** and the other to assign combination features to each generated item. The generated file **korean.lex.cat** shows the result of applying the allormophic rules to the basic lexicon. Consider, first, the following nominal forms generated from the basic noun entries in (4a):

```
(5) Nominal Forms
    a.  surf: "pwumo", base: "pwumo",
        cat: [Class: N,                Sem: "parent",
              Form: <terminal, root>, Syllab: <open>]
    b.  surf: "kyoswu", base: "kyoswu",
        cat: [Class: N,                Sem: "professor",
              Form: <terminal, root>, Syllab: <open>]
    c.  surf: "sensaeng", base: "sensaeng",
        cat: [Class: N,                Sem: "teacher",
              Form: <terminal, root>, Syllab: <closed>]
```

Unlike the basic nominal forms listed in (4a), the newly generated forms all contain the same combination feature **Form : <terminal, root>**. As a **terminal**, each nominal form as given here is a word in itself. As a **root**, however, these nominal forms can undergo concatenation. The nominal stem "pwumo"(parent), for instance, can combine with the Subject particle "ka" because the stem contains the feature **Syllab: <open>** which is required by the particle "ka". On the other hand, the stem "sensaeng"(teacher) cannot take the particle "ka" because it does not have the required feature. Instead, it can combine with the Subject particle ".i" which has the feature **Syllab_req: <closed>**. The allomorphy rule **korean.all** thus automatically assigns necessary combination features to each nominal form.

Verbal forms are also automatically assigned appropriate combination features, as shown below:

```
(6) Verbal Forms
    a. surf: "ka", base: "kata",
       cat: [Class: V,       Sem: "go",      Form: <bse>,
             Stem: <root>, Bridge: <nul>,    Syllab: <open>]
    b. surf: "cap", base: "capta",
       cat: [Class: V,       Sem: "catch",   Form: <bse>,
             Stem: <root>, Bridge: <_u, _a>, Syllab: <closed>]
    c. surf: "ket", base: "ketta",
       cat: [Class: V,       Sem: "walk",    Form: <bse>,
             Stem: <root>,                   Syllab: <closed>]
    d. surf: "kel", base: "ketta",
       cat: [Class: V,       Sem: "walk",    Form: <non_bse>,
             Stem: <root>, Bridge: <_u, _e>, Syllab: <d_liquid>]
```

Among the four blocks listed above, the first two are for the regular verbs "kata"(go) and "capta"(catch) and the other two are for the two allomorphic forms "ket" and "kel" of the irregular verb "ketta"(walk).[5] Each block contains both old and new information: the original values for the attributes **Class** and **Sem** are retained, while the **surface** and **base** forms and combination features are newly introduced. As will be shown presently, a difference in combination features causes different paths of concatenation. Hence, two different allomorphic forms with different combination features undergo different paths of con-

catenation. For example, the path "kel.e.ss.ta" (walk-e_Bridge-Past-Declarative) is acceptable, while the other allomorph "ket" (walk) cannot undergo the same path of concatenation.

Finally, the rule file **korean.mor** consists of three parts: the initial, combinatorial, and terminating. The initial part introduces nominal or verbal stems from the generated allomorph list and then specifies a set of possible rules that may apply to them. These stems may go through combination processes or directly go to the end rule, which recognizes the input as well-formed and then declares it as successful output.

In our Korean_Malaga, the combinatorial part contains only one rule. This single rule concatenates a stem with a particle or ending. It may, however, apply again and again recursively to allow a stem to take more than one suffix.[6] The word form "pwumo.nim.kkeyse.nun" (parent-honorific_suffix-honorific_SBJ-Topic), for instance, consists of a nominal stem "pwumo" (parent) and three particles "nim" (honorific suffix), "kkeyse" (honorific Subject marker), and "nun" (Topic marker). It is thus generated by undergoing the combination rule three times with the stem taking a particle one by one from left to right.

3 Blocking Recursions

Unless properly constrained, Korean_Malaga may generate infinite strings like (7), as has been attested in the process of constructing the system.

```
(7) infinite recursions
    a. pwumo.nim.nim.nim...
       parent-honorific-honorific-honorific...
    b. cap.u.si.ess.u.si.ess.u.si.ess...
       catch-u_Bridge-honcrific_end-Past-u_Bridge...
```

Despite the problem of infinite recursion, our proposed system allows recursive rule application. Since the Korean_Malaga system contains only one combination rule, named **combine**, which may apply recursively, there are only two possible ways of ordering rule appliation:

```
(8) Rule Paths
    a. stem  - success
    b. stem  - combine . . . success
```

In analyzing a word form, Malaga traces the derivational path (or paths), represented in a tree form, which shows what rules have been applied in what order.

The rule **combine** concatenates two input strings in a straightforward manner from left to right without any deletion or alteration of any parts of them. This is based on the fact that all allomorphic forms are pre-generated and ready for morphological combination, as required by the surface compositional approach of Left-Associative Grammar.

Here is an example for combining the allomorphic form "kel" (walk) with the conditional verbal ending "myen" through the bridging vowel ".u".

```
(10)  kel.u.myen
      walk-u_Bridge-Conditional
```

The form "kel" is pre-generated from the basic form "ket" by the allomorphy rule **korean.all** and contains appropriate combination features. Because of these specific features, particularly the feature **Bridge: <_u,_e>**, the form "kel" can take the bridging vowel ".u" and then the ending "myen".[7] The bridging vowel here turns the feature **Syllab: <d_liquid>** of "kel" into **Syllab: <open>** so that the combined string "kel.u" may now combine with the terminal ending "myen" which requires the value of the **Syllable** to be **open**.

It should now become clear that each application of the rule **combine** is constrained by relevant combination features of the two input strings. The rule is designed to apply if a stem or the left element contains the features which are required by a suffix or the right element; otherwise, it fails to apply.

The general mechanism of constraining concatenation processes is simple and straightforward. It is, however, a completely different matter to implement a particular system like Korean_Malaga, especially dealing with recursion. Here are three types of recursion that may be of interest to the implementation of any morphological generator: [i] simple repetition, [ii] reverse order, and [iii] non-adjacent recurrence.

3.1 Simple Repetition

The most frequent problem of recursion occurs from simple repetition as is illustrated by:

```
(11=7a) pwumo.nim.nim.nim...
        parent-honorific-honorific-honorific...
```

Suppose we introduce the particle "nim" into the lexicon, as tentatively specified below:

```
(12)  [Phon: "nim",     Class_req: N,  Form_req: <root>,
       Result_Syllab: <closed>,        Hon: <referent>]
```

Since the particle "nim" has the two features **Class_req: N** and **Form_req: <root>** which require its preceding string to have the values, namely **N** and **root**, for the attributes **Class** and **Form**, it can be attached to any Nominal **root** form, say "pwumo" (parent).

```
(13)  [Phon: "pwumo",     Class: N,     Sem: "parent",
       Form: <terminal, root>,         Syllab: <open>]
```

By combining these two, we obtain:

```
(14) [Phon: "pwumo.nim",          Class: N,
      Sem: "parent",              Form: <terminal, root>,
      Syllab: <closed>,           Hon: <referent>];
```

The newly combined form here has inherited the features **Class: N**, **Sem: "parent"**, and **Form: <terminal, root>** from the stem and the feature **Hon: <referent>** from the particle. On the other hand, the original feature **Syllab: <open>** of the stem has taken a new value **<closed>** because the particle contains the feature **Result_Syllab: <closed>**. The features **Class_req: N** and **Form_req: <root>** of the particle are no longer necessary and thus deleted.

The resultant form (14) with its features, however, allows another concatenation with the particle "nim", for it is again an **N root**. This process may then be repeated infinitely. In order to block such an undesirable repetition, a new feature **Result_Form: <terminal, honorific>** must be added to the original record (12) of the particle "nim", as in (15):

```
(15)  [Phon: "nim",   Class_req: N,  Form_req: <root>,
  ***  Result_Form: <terminal,honored>,
       Result_Syllab: <closed>,
       Hon: <referent>];
```

The added feature, here marked with ***, then successfully eliminates the possibility of repeatedly concatenating the particle "nim" to a nominal root, since the resulting word form "pwumo.nim" is no longer a **root**, but an **honorific** form.

3.2 Reversed Order

The second type of problems in recursion involves the discourse function (**DF** henceforth) particle "man"(only). As shown below, it can occur both before and after a grammatical function (**GF** henceforth) particle : it occurs before the **GF** particle ".i"(Subject) or ".ul"(Object), but after the **GF** particle "kkeyse"(Subject) or ".eykey"(Dative).

```
(16)      N_root      DF      GF
     a. pwumo.nim     man     .i
        parents       only    Subject
     b. pwumo.nim     man     .ul
                               Object

(17)      N_root      GF      DF
     a. pwumo.nim    kkeyse  man
        parents      Subject  only
     b. pwumo.nim    .eykey  man
                     Dative
```

Here the generation process may create a possibly infinite loop from **GF** to **DF**("man") to **GF** and then to **DF**("man"), generating an ill-formed string like "pwu.mo.nim-kkeyse-man-kkeyese-man-kkeyse...man-.ul".

The proposed time-linear approach augmented with feature structures resolves such a recursive loop by setting up two subtypes of **GF**-assigning particles with different combination features. Consider the following two particles:

```
(18) GF-assigning Particles
     a.  [Phon: ".i",   Class_req: N, Syllab_req: <closed>,
          Form_req: <root,honored,plural,DF_man>,
          Result_Form: <terminal>,    GF: <SBJ>];
     b.  [Phon: "kkeyse", Class_req: N,
          Form_req: <root,honored,plural>,
          Result_Form: <terminal,AF>, Result_Syllab: <open>,
          GF: <SBJ>,                  Hon: <SBJ>];
```

Both of the particles carry the information **GF: <SBJ>**, thus each assigning the grammatical function **SBJ** to the nouns to which they are attached. They, however, differ from each other in other respects. Specifically, the genuine GF particle ".i" can be suffixed to a nominal stem that ends in the particle "man" with the feature **Form: <DF_man>**, as specified in (19), but the honorific Subject particle "kkeyse" cannot.

```
(19) [Phon: "man",                          Class_req: N,
      Form_req: <root,honored,plural,AF>,
      Result_Form: <terminal,DF_man>,
      Result_Syllab: <closed>,            DF: <exclusive>];
```

Hence, the modified system can successfully analyze a well-formed string like "pwumo.man.i"(parent-DF(only)-SBJ), while blocking an ill-formed string "pwumo.man.kkeyse"(parent-DF(only)-SBJ).

Here are the actual results of analyzing both of the strings on Korean_Malaga:[8]

```
(20) Successful Analysis
     malaga> roman
     malaga> ma {.pwu.mo.man.i}
     complete analyses for "{.pwu.mo.man.i}":
     1: "{.pwu.mo}/{.man}/{.i}", [Phon: "{.pwu.mo.man.i}",
        Class: N, Sem: "{.pwu.mo}_parent", GF: <SBJ>,
        DF: <exclusive>]
```

The morphological analysis **ma** of the romanized string {.pwu.mo.man.i} here is recognized as successful. On the other hand, the complete analysis of the string {.pwu.mo.man.kkey.se} is shown to fail. The analysis goes through only up to the string {.pwu.mo.man}.

```
(21) Complete Analysis Failed
     malaga> ma {.pwu.mo.man.kkey.se}
     complete analyses for "{.pwu.mo.man.kkey.se}":
     none
     malaga> longest
     longest analyses for "{.pwu.mo.man.kkey.se}":
     1: "{.pwu.mo}/{.man}", [Phon: "{.pwu.mo.man}", Class: N,
        Sem: "{.pwu.mo}_parent", Form: <terminal, DF_man>,
        Stem: nil, Bridge: nil, Syllab: <closed>, DF: <exclusive>]
```

Now to allow the DF "man" to occur after the honorific **SBJ** marker "kkeyse", we can introduce two types of "man", one occurring before a genuine GF marker like ".i" another occurring after the honorific SBJ marker "kkeyse". But this approach does not seem to be too elegant. To avoid introducing two different types of the particle "man," a feature inheritance mechanism is built into the rule of concatenation. In this way, two different occurrences of a single type are treated as two different tokens. Specifically, the occurrence of "man" immediately after the string **N_root** inherits the attribute **root** from the nominal root and thus may concatenate with a GF particle like ".i" or ".ul". On the other hand, the occurrence of "man" after the string **N_root**-"kkeyse"(**GF_AF**)[9] inherits the adverbial function (hence **AF**) attribute instead of the feature **root**. Because of this inherited **AF** attribute, the string **N_root**-"kkeyse"(**GF_AF**)-**DF** may take another DF particle like ".un"(Topic), thus generating a well-formed string like:

```
(22)      N_root     GF_AF      DF     DF
     a. pwumo.nim   kkey.se    man    .un
     b. pwumo.nim   .ey.key    man    .un
```

In this example, the string **N_root**-(**GF_AF**)-**DF**_man-**DF** is complete, because the second DF is a terminal particle. The repetition of "man" occurring in the second DF position is also blocked by a general mechanism, again an inhertance mechanism, that suppresses repetition of tokens of the same type.

3.3 Non-adjacent Recurrence

Recursion phenomena may also occur in verbal concatenations. Consider:

```
(23)   V_root    Bridge   Honorific   Ending
     a.  cap       .u                 myen
         catch                      Conditional
     b.  cap       .u        si         myen
```

Here the ending "myen" requires a bridging vowel ".u" to be concatenated to a verbal root like "cap"(catch) ending in a consonant. The honorific ending "si" has the same requirement for concatenation.

Although the bare root like "ka"(go) may directly concatenate with the ending "myen", the past tense marked stem "cap.a-ss" like the bare root "cap" requires the bridge ".u" because they each end in a consonant.

```
(24)    V_root   Bridge  Past  Bridge  Ending
         cap      .a      ss    .u      myen
```

These two paths in (23b) and (24) may combine into a longer path, as in:

```
(25)    V_root Bridge Honorific Bridge Past Bridge Ending
         cap     .u      si       .e     ss   .u     myen
```

Unless properly constrained, this path may, however, run into an infinite loop, generating strings like:

```
(26) * cap-.u-si-.ess-.u-si-.ess-.u-si-myen
```

This loop can again be broken by differentiating two different tokens of the bridging vowel ".u": the one with the attribute **root** inherited from the adjacent verbal root and the other with the attribute **finite** from the adjacent past particle ".ess". Since the honorific particle "si" only attaches to a root or a string with the attribute **root**, it can attach to a string like "ka" or "cap-.u", but not to a string like "ka-si-.ess-.u" or "cap-.u-si-.ess-.u".

4 Conclusion

In Korean_Malaga, a single concatenation rule applies recursively to combine a stem with a sequence of suffixes. This makes the system simple and elegant. However, unless it is properly constrained, some combination processes never stop, creating the phenomena of infinite loops in recursion. In order to prevent such undesirable recursive loops, a feature inheritance mechanism differentiating tokens from types has been proposed here.

Because of this mechanism, different matching or congruency conditions are presented for concatenating two input strings at each step. Consequently two different occurrences of an identical suffix are treated as different tokens, but of the same type. Hence, such a suffix is listed only once in the lexical entries.

The Korean_Malaga system using Hangul Charaters successfully runs on mule, a multi-lingually enhanced emacs editor, in a shell-mode without undesirable recursive loops. Generation processes can easily be tested by executing the command **paradigm** with a sample list of stems, particles, and endings.[10]

References

[1] R Hausser. Computation of Language. Berlin: Springer-Verlag, 1992.
[2] B Beutel. Malaga: An Implementation Langauge for Left-Associative Grammar. Unpublsihed manuscript. Computerlinguistk, Universität Erlangen-Nürnberg, 1995.

[3] Kiyong Lee. Hangul, the Korean Writing System, and Its Computational Treatment. LDV-FORUM: Forum der Gesellschaft für Linguistische Datenverarbeitung, 11.2, 26-43, 1994.

Acknowledgments: This paper is to be presented at the 10th Pacific-Asian Conference on Language, Information, and Computation hosted by City University of Hong Kong from 27-28 December, 1995. I am grateful to Chin-W. Kim of Illinois and Korea Universities, Jae-Woong Choe, Myung-Yoon Kang, Taegoo Chung of Korea University, Roland Hausser, Björn Beutel, Marcus Schulze, Gerald Schüller, and Oliver Lorenz of Erlangen University, Germany. Without their indispensible and often demanding comments and corrections, this paper would not have survived to see its completion.

Notes

[1]Implemented in ANSI-C by Björn Beutel and his colleagues under the supervision of Roland Hausser, Erlangen, Germany. See Beutel [2] for details.

[2]All these files must have the same name but different extensions. With these files incorporated into Malaga, the compiled system Korean_Malaga runs on mule, an acronym for mult-lingual Enhancement to GNU emacs, in a shell mode.

The file korean.seg lists all the segments like word class names, attributes or value names that are to be used in other files. Examples are Phone, Class, V, Sem and root.

The file korean.lex consists of morphemes or basic lexical items. Each item is represented as a record consisting of attribute-value pairs. Below is listed part of the file for the implemented system Korean_Malaga.

[3]Although the program Korean_Malaga actually uses Hangul characters encoded in KSC5601 Coding System or the so-called precomposed character set, all Hangul strings are romanized by the modified Yale system in this paper for the sake of the foreign audience. See Lee [3] for details.

[4]For the present audience again, the value for Sem is provided with an English equivalent.

[5]The feature Default_feat: ir is deleted here because it is no longer necessary.

[6]The term suffix here is used to cover both nominal particles, verb endings or anything else that may be attached to the end of a stem.

[7]Because the other allomorphic form "ket" does not have this feature, it cannot participate in this combination.

[8]The Korean_Malaga can accept input strings in both Hangul and romanized characters.

[9]Here the particle "kkeyse" is treated as a GF_AF marker, namely a GF-assigning AF marker, although no such segment is actually used for specifying the record of the particle "kkeyse".

[10]A Unix or Linux version of the modified system Korean_Malaga will soon be availabe to the public from the www-site at http://www.linguistik.uni-erlangen.de/Malaga.en.html.