‘야민정음’ 번역기 제작하기

기말 프로젝트 보고서

I. 서론

1. ‘야민정음’의 개념과 글자 변형

‘야민정음’은 국내 커뮤니티 사이트인 ‘디시인사이드’의 여러 주제별 갤러리들(해당 주제에 관심 있는 사람들이 주로 글을 작성하는 공간) 중 ‘국내야구 갤러리’에서 적극적으로 사용된, 한글을 변형하는 문화이다. 이 때문에 이러한 한글 변형 놀이 문화를 ‘국내야구 갤러리’의 ‘야’와 ‘훈민정음’을 합성한 ‘야민정음’이라 많은 사람들이 지칭하게 되었다. 대표적인 변환으로는 ‘ㄷ’과 ‘ㅐ’의 왼쪽 획을 붙이면 ‘ㅁ’처럼 보인다는 것에 착안해 ‘대’를 ‘머’로, 또는 그 반대로 바꾸는 것이 있다. 그러나 이러한 변형이 야민정음으로 명명된 것과 함께 본격적인 문화가 된 것의 계기가 국내야구 갤러리일 뿐이지, 한글의 형태를 이용하는 유희 자체를 국내야구 갤러리가 만들었다고 보기는 어렵다. 작성자와 동일한 세대(2017년 기준 20대 초~중반) 구성원들의 경험상, 국내야구 갤러리가 2004년에 만들어지고 이후에 본격적으로 부흥하기 이전인 2000년대 초에도 한글을 비슷한 로마자로 바꾸어 사용하는 장난(즐 ↔ KIN), 한 단어의 형태를 변형하는 장난(종합병원 ↔ (앞 글자의 종성(ㅎ의 경우에는 중성을 포함)이 뒷 글자의 초성과 같다는 점을 이용해 세로 방향으로 작성하여) 종ㅏㅂㅕㅇ궈ㄴ) 등은 젊은 세대에게 이미 익숙한 것이었다. 또한 악필, 폰트, 잘못된 방향 등의 문제로 의도치 않게 글자를 잘못 읽어 생긴 일화들 또한 일상적으로도, 연예·정치 등과 관련해서도 늘 존재해왔다. 이렇게 축적된 글자 변형을 다소 체계적으로 집대성해 본격적으로 사용하게 된 것이 야민정음이라 볼 수 있는 것이다.

이것은 뚜렷한 직선으로 이루어진 서로 비슷한 자모가 많은 한글의 특징과 관련이 있는 것으로 보인다. 다른 문자들에 비해 글자의 획 길이나 각도를 조절함으로써 원래의 글자를 다른 글자로 보이게 하기 쉽기 때문이다. 또한 초성, 중성, 종성을 모아쓰는 문자이기 때문에 인접한 각 자모 간의 형태적 관계를 이용해 음절 단위로 글자를 변형할 수 있어 유희적 활용성이 더 풍부하다고 할 수 있다. 물론 이것은 한글의 표기상 단점이기도 하다. 일상적으로 작은 크기의 글자나 손글씨를 읽을 때 존재하는 문제일 뿐 아니라, 중세국어 문헌의 글자들을 해석하는 과정에서 혼동을 유발하는 경우도 왕왕 있다고 한다. 하지만 야민정음과 같은 유희적 글자 변형이 한글에게만 고유한 것으로 보기는 어렵다. 영어권에서는 로마자를 비슷하게 생긴 다른 로마자나 숫자, 특수문자로 바꾸어 사용하는 것을 일컫는 ‘leet’가 존재한다. 이는 처음에 해커들이 컴퓨터의 자동 감시 프로그램을 피하거나 은어로서 사용할 목적으로 시작한 것인데, 예를 들어 ‘you’re’을 ‘y0|_| /r3’과 같이 변형하는 식이다. 일본의 경우 한자와 일본어 발음 등을 활용하여 유희를 즐기는 ‘다와무레가키(戯れ書き, 장난으로 씀)’가 예로부터 존재했다. 이처럼 글자의 형태나 의미, 읽었을 때의 음, 다른 문자와의 관계 등을 활용한 글자 변형은 전세계적으로 단순한 유희, 또는 자신의 글이 원지 않는 사람 또는 기계에게 읽히는 것을 방지하기 위한 암호, 일정 집단 내에서의 친밀감과 동질감을 위한 은어 등의 목적으로 꾸준히 존재했다.

따라서 야민정음의 모든 한글 변형법을 디시인사이드 국내야구 갤러리에서 만든 것으로 보긴 어렵다. 앞서 밝혔듯 한글 변형은 꾸준히 존재해왔으며, 야민정음이 본격적인 문화가 된 이후에는 국내 인터넷 상에서 널리 퍼져 다양한 곳에서 많은 사람들에 의해 사용되며 그 활용법이 계속해서 추가되고 바뀌었기 때문이다. 따라서 야민정음이 어떤 것인지 정의는 가능하지만, 실제 이를 적용해 정상적인 한글을 야민정음으로 바꾸고자 하면 한 가지의 방법이 존재하지 않고 매우 다양한 가능성이 존재한다. 또한 유희적 목적의 특성상 다른 이들이 생각하지 못한 참신한 활용이 언제든 등장할 수 있으며, 기본적으로 인터넷 문화의 일부인 만큼 그 변화의 주기도 매우 빠르고 특정 사건 등 다양한 요인에 쉽게 영향을 받으므로 예측이 어렵다. 하지만 이 때문에 언중의 언어 사용을 포착할 시도를 하게 된다는 점에서 이 프로젝트의 가치가 더해지는 것이기도 하다. 번역기에는 과제 진행 시점에서 관찰할 수 있는, 다수에 의해 사용되고 더 보편적인 것으로 보이는 야민정음 사용 양상을 가능한 선에서 반영하고자 했다. ‘번역’은 어떤 언어로 된 것을 다른 언어로 옮기는 것이므로, 야민정음과 정상적인 한글 사이를 오가는 것은 엄밀히 하자면 단순히 표기만 변형할 뿐이므로 ‘변환’에 가깝다. 하지만 여기에서는 ‘야민정음 번역’이라는 말이 언중에게 친숙함을 고려해 ‘번역기’라는 표현을 사용했다.

대신 시간이 지나 특정 변형이 추가되거나 빈도가 달라지면 쉽게 수정할 수 있는 방식으로 번역기를 만들었다.

또한 단순히 형태적으로 유사한 변형뿐 아니라, 기타 문화적 현상이나 글자의 의미, 관련된 한자의 음 등과 관련해

2. ‘야민정음’ 변환 관련 선행 연구

야민정음에 대한 진지한 학문적 분석은 이뤄진 것이 없는 것으로 보인다. 대신 언론에서 여러 번 다뤄진 바 있고, 전문가가 칼럼 등을 통해 야민정음에 대한 간단한 의견 또는 분석을 보인 경우가 있다. 언론에서 다뤄진 경우, 젊은 세대의 한글파괴를 부정적으로 평가하는 내용이 많았다. 반면, 이를 창의적이고 역동적인 언어 생활 및 참신한 한글 사용으로 보고 긍정적으로 평가하는 전문가들도 존재했다. 한편, 정상적인 한글과 야민정음을 오가는 방법을 간단히 설명한 내용은 많았지만 이를 본격적으로 컴퓨터를 이용해 자동화하는 것에 주목한 경우는 거의 없었다. 이러한 시도는 학술적으로도, 시장에서도 거의 없었다. 2017년 초에 ‘Google Play’에 ‘야민정음 변환기’가 하나 등록된 것으로 보이며, 그밖에도 일부 사람들이 야민정음 변환을 하는 코드 작성을 여러 프로그래밍 언어로 시도한 것을 찾아볼 수 있다.

반대로 한글이 ‘야민정음화’되는 현상을 피하려는 시도들은 꾸준히 존재한다고 할 수 있다. 예를 들면 고문헌이나 옛날 신문 등의 내용을 OCR(optical character recognition, 광학식 문자 판독)로 텍스트로 옮길 때, 한자나 한글을 비슷한 다른 글자로 컴퓨터가 잘못 읽는 경우를 찾아볼 수 있다. 예를 들어 한자 가로 왈(曰)과 날 일(日), 가타카나의 ン와 ソ, 한글의 ‘의’와 ‘익’ 등의 오변환은 상당히 흔하다. 우체국에서 손으로 쓰인 주소 등을 인식할 때 등 다양한 목적으로 사용되는 손글씨 인식 프로그램은 빅데이터를 이용한 딥 러닝 등을 통해 꾸준히 오차를 줄여나가기도 한다. 이것은 분명한 가치가 있기 때문에 활발히 진행되고 있다.

이미 시도된 야민정음 변환 프로그램들 중 쉽게 접근할 수 있는 것을 이용해보면, 그 한계를 분명히 느낄 수 있다. 많은 경우 글자 회전(육 ↔ 뇽)은 고려하지 않고 모양이 비슷한 다른 한글이나 한자로 바꾸는 변환만을 적용하고 있다. 또한, 철저히 글자 단위로 인식해서 단어 단위에 따라 변환 방식 또는 변환 여부가 달라지는 것을 고려하지 못한다. 뿐만 아니라 변환을 과적용하여 일반적으로 잘 사용되지 않는 변환이 쉽게 적용되어 버리는 문제가 있다. 다만 Google Play에 등록된 ‘야민정음 변환기’의 경우, 안드로이드 기기를 사용하지 않아 직접 확인하지는 못했지만 새로 업데이트된 버전에서는 ‘변환 강도 기능’을 추가했다는 내용을 보아 이 문제를 개선한 것으로 보인다. 물론 이 프로젝트를 진행함에 있어서는, 이미 존재하는 야민정음 변환 관련 코드들을 참고하지 않고 ‘언어와 컴퓨터’ 수업에서 배운 내용과 직접 습득하거나 찾아본 내용만을 토대로 독자적으로 코드를 창작했음을 미리 밝힌다.

II. 자료

‘야민정음 번역기’ 프로젝트는 크게 두 유형의 자료를 토대로 진행되었다. 한 가지는 야민정음이 실제로 사용된 텍스트들을 수집하거나 사용 현황을 조사한 참조 자료이고, 다른 한 가지는 이 참조 자료들의 정보를 토대로 작성해 직접적으로 프로그램에 사용한 변환쌍 또는 각 자모에 대한 회전 정보 등이다.

1. 야민정음의 실제 사용 현황에 관한 자료들

총 세 가지 종류의 자료를 참고했다고 할 수 있다.

i) 야민정음이 사용된 트윗들

트위터(https://twitter.com)에 직접 접속하여 야민정음이 사용된 키워드(‘머박’(대박), ‘띵작’(명작) 등)를 검색해 일일이 수집하는 방법도 있으나, 비효율적이므로 트위터가 개발자를 위해 제공하는 기능과 정보들을 토대로 가능한 선에서 코드를 직접 작성해 수집했다. 이 과정에서 application 등록이 필요했는데, 관련 주소는 https://apps.twitter.com/app/14539287이다. 직접 트윗 수집 코드 작성을 진행하면서 또는 완료한 후에, 프로젝트에 해당 코드를 통해 수집한 트윗들의 내용을 지속적으로 참고했다. 이를 통해 사람들이 어떤 방식의 야민정음 변환을 어떤 글자에 많이 사용하고, 야민정음 사용 경향에 어떤 특징이 나타나는지 등을 대략적으로 알 수 있었다.

트윗 수집 프로그램을 통해 수집하고 참고한 모든 트윗들의 내용은 공유하기에 불필요한 것이 많다. 또한 수집 코드가 중간중간 변경되었기에, 최종적으로 완성한 수집 프로그램을 통한 수집 결과와, 중간에 참고한 트윗들의 수집 결과 형식이 다소 다르다. 따라서 제출하는 data 디렉토리 안의 수집 트윗 결과(yamin_tweets.txt)는, 최종 버전의 수집 프로그램을 통해 새롭게 수집한 트윗들이다. ‘머박’, ’틘’ 또는 ‘곤뇽’ 등을 검색해 수집한 결과이며, 이는 단지 이러한 방식으로 수집한 트윗들을 참고했다는 것을 보이기 위한 예일 뿐이다. 실제로는 더욱 많은 트윗들을 지속적으로 참고했다.

아래는 트윗을 수집하는 프로그램으로 직접 만든 scrape_tweets.py를 스크립트 모드에서 실행한 내용(yamin_tweets.txt를 수집하는 과정)의 일부분이다. 어떤 조건으로 검색했는지를 알 수 있는 코드 일부는 같이 나타냈다. 이 프로그램의 소스코드는 핵심적인 것이 아닌 만큼, 따로 보고서에서 지면을 할애하지는 않고, 해당 파일의 각주로 갈음하도록 한다.

```
20 keyword = '네슷 OR 곤뇽 OR 틄 OR 커찮'
21 N = 600
22 file_name = 'data/yamin_tweets.txt'
23 last_id = 0
24 start_num = 0
```

```
davin@language-and-computer:~/Final_term_project$ python3 scrape_tweets.py
'네슷 OR 곤뇽 OR 틄 OR 커찮'을/를 검색해 최대 약 600개의 트윗을 내려받습니다. 시작 트윗 번호는 0입
니다.
15개의 트윗을 내려받았습니다.
30개의 트윗을 내려받았습니다.
43개의 트윗을 내려받았습니다.
...
140개의 트윗을 내려받았습니다.
더 이상 트윗이 존재하지 않습니다.
140개의 트윗을 내려받아 data/yamin_tweets.txt에 저장했습니다.
마지막 트윗의 ID는 938402336669442048입니다. 파일에 저장된 총 트윗의 개수는 140개입니다.
```

```
20 keyword = '머박 OR 띵작'
21 N = 600 # 15의 배수로 입력
22 file_name = 'data/yamin_tweets.txt'
23 last_id = 0
24 start_num = 140
```

```
davin@language-and-computer:~/Final_term_project$ python3 scrape_tweets.py
'머박 OR 띵작'을/를 검색해 최대 약 600개의 트윗을 내려받습니다. 시작 트윗 번호는 140입니다.
15개의 트윗을 내려받았습니다.
...
600개의 트윗을 내려받았습니다.
600개의 트윗을 내려받아 data/yamin_tweets.txt에 저장했습니다.
마지막 트윗의 ID는 942069632302587904입니다. 파일에 저장된 총 트윗의 개수는 740개입니다.
```

```
20 keyword = '머박 OR 띵작'
21 N = 1000 # 15의 배수로 입력
22 file_name = 'data/yamin_tweets.txt'
23 last_id = 942069632302587904
24 start_num = 740
```

```
davin@language-and-computer:~/Final_term_project$ python3 scrape_tweets.py
'머박 OR 띵작'을/를 검색해 최대 약 1000개의 트윗을 내려받습니다. 시작 트윗 번호는 740입니다.
15개의 트윗을 내려받았습니다.
30개의 트윗을 내려받았습니다.
45개의 트윗을 내려받았습니다.
...
990개의 트윗을 내려받았습니다.
1005개의 트윗을 내려받았습니다.
```

```
1005개의 트윗을 내려받아 data/yamin_tweets.txt에 저장했습니다.
마지막 트윗의 ID는 942022534068715521입니다. 파일에 저장된 총 트윗의 개수는 1745개입니다.
```

```
20 keyword = '댕댕이 OR 롬곡'
21 N = 2000 # 15의 배수로 입력
22 file_name = 'data/yamin_tweets.txt'
23 last_id = 0
24 start_num = 1745
```

```
davin@language-and-computer:~/Final_term_project$ python3 scrape_tweets.py
'댕댕이 OR 롬곡'을/를 검색해 최대 약 2000개의 트윗을 내려받습니다. 시작 트윗 번호는 1745입니다.
15개의 트윗을 내려받았습니다.
...
1995개의 트윗을 내려받았습니다.
2010개의 트윗을 내려받았습니다.
2010개의 트윗을 내려받아 data/yamin_tweets.txt에 저장했습니다.
마지막 트윗의 ID는 942188241343586304입니다. 파일에 저장된 총 트윗의 개수는 3755개입니다.
```

또한 yamin_tweets.txt 파일을 head, tail의 명령을 이용해서 앞뒤 일부만 살펴보면 아래와 같다.

```
davin@language-and-computer:~/Final_term_project$ head -20 data/yamin_tweets.txt
--------------------------------------
'네슷 OR 곤뇽 OR 틄 OR 커찮'을/를 찾은 검색결과 입니다.

0    2017-12-16 17:19:28    애니받아놓은거 봐야지 참
일어나기 커찮다
1    2017-12-16 16:39:42    @Siroinu_S 나도 프라임프레임들좀 만들고싶은데.. 커찮
2    2017-12-16 15:15:47    @la_mento @MTGST_CHAN 아니 젖과 꿀이 흐르는 수방사 곤뇽이 왜;
3    2017-12-16 14:27:00    일단 자고싶긴한데 침대위를 치우기가 커찮다 아존나지고싶음
4    2017-12-16 14:13:53    기껏 예약구매까지 했는데 울썬문 진행하는 게 왜이리 커찮지
5    2017-12-16 10:58:48    미루쨩은 곤뇽이에요..!
6    2017-12-16 10:26:32    @Gypsophila_cf -3개월의 기적 곤뇽
7    2017-12-16 09:00:26    @__EricaHartmann 곤뇽은 대부분최전방이니..
8    2017-12-16 08:52:22    @__EricaHartmann 곤뇽... ㅠ
9    2017-12-16 06:19:03    만사커찮,,
10   2017-12-16 01:31:39    성덕머틄갓종 https://t.co/VY9cHxFIF6
11   2017-12-16 01:06:45    @py__12 흑흑 전역차이별로안나네요 역시 곤뇽..
12   2017-12-16 00:57:03    @Ubedesu 아니아니!! !! 아직 안머겄더ㅜㅜㅜ 밥먹기 커찮다...
흐흐ㅠㅠㅠ 넘두근대니까요ㅜㅜㅠㅠㅠ ㅍ..팔찌보면 어케 인사하지,,,(두근두근
13   2017-12-16 00:52:48    @2JungNim 부러유ㅓ 할거없는게 의경이든 곤뇽이든 자기자신의 줫같음
정도는 같음
14   2017-12-15 17:24:14    아 플포 다시 설치해야하는데 커찮….
davin@language-and-computer:~/Final_term_project$ tail data/yamin_tweets.txt
2. 남팬
3. 멘트가 신선하거나 마음을 사로잡
```

```
4. 팔로워가 존나레알개오지게 많은 슈스

&gt;&gt; 다 해당 안 되는 저는 ●● 롬곡
--------------------
1745부터 3755까지는 2017년 12월 17일 2시 58분 55초에 수집한 트윗들입니다.
마지막 트윗의 ID는 942188241343586304입니다.
```

ii) 설문조사를 통해 직접 조사한 사용 현황

"야민정음' 실제 사용에 대한 조사'라는 이름으로 구글 설문지(https://goo.gl/forms/Yz5AAjPAvT2yvkOD2)를 만들어 2017년 12월 12일부터 15일까지 조사를 진행했다. 이를 통해 어떤 야민정음 변환쌍이 많이 사용되는지를 조사하고, 특정 내용의 한글을 사람들이 직접 야민정음으로 변환하게 하여 실제 야민정음 사용의 경향성을 파악하고자 하였다. 총 57명이 참여하였으며, 대부분 20대 초중반으로 응답자가 한정되었을 것으로 보인다. 분명 한계는 있겠지만, II.2.i)에서 설명할 자료의 각 변환쌍별 사용 빈도를 수치화하는 것의 근거를 더욱 튼튼히 해주었다. 또한 II.1.i) 또는 iii)과 같이 사람들이 임의로 작성한 매우 다양한 내용들에서 야민정음의 사용 경향성을 파악하는 것에는 분명 어려움이 있는데, 제한된 동일한 문장을 많은 사람들이 야민정음으로 변환하게 함으로써 경향성 파악을 비교적 더 쉽게 할 수 있었다. 설문의 결과는 data 디렉토리에 'YAMIN_survey_(response).csv 파일로서 포함했다. 해당 파일은 응답자의 전화번호 등 개인정보를 제외하고, 변환쌍(대체로 alternation.txt와 일치한다.)별 평균을 포함한 자료이다. 평균 등의 수치를 기계적으로 II.2.i)에서 언급할 alternation.txt에 그대로 반영하지는 않았다. 응답자들이 모두 동등하게 야민정음에 익숙한 것이 아닐뿐더러, 실제 인터넷상 등에서는 그 활용이 다르게 나타나는 경우도 보였기 했기 때문이다.

iii) 각종 포털 및 커뮤니티 댓글, 야민정음이 사용된 각종 글 등

네이버와 다음 등 주요 포털의 댓글들, 디시인사이드 국내야구 갤러리를 포함한 일부 커뮤니티의 게시글과 댓글들, 검색 엔진을 통해 발견할 수 있는 야민정음이 사용된 각종 글들을 참고했다. 특히 야민정음을 고의적으로 많이 사용한 유형의 웹사이트들의 문서들(https://namu.wiki/w/야민정음) 등에서도 많은 참고를 할 수 있었다. 야민정음 사용 경향 자체가 어떠한 권위 또는 전문지식이나 신뢰할 만한 근거가 필요한 것이 아니므로, 비교적 제약 없이 다양한 인터넷 사이트들을 참고했다.

2. 직접적으로 프로그램에 사용된 자료들

i) 한글 한 글자끼리의 변환쌍과 그 사용 빈도

II.1.에서 보인 자료들 등을 토대로, 한글을 야민정음으로 변환하는 대응쌍들과 그것의 사용 빈도를 나름대로 정리해 data 디렉토리의 alternation.txt 파일을 만들었다. 이 파일은 3개의 열과 약 50개의 행으로 이루어져 있으며, 첫 번째 열과 두 번째 열은 서로 대응되는 한글 한 글자끼리의 변환쌍에 해당한다. 대부분 형태적 유사성에 착안한 변환쌍이며, 성격은 크게 다르지만 널리 사용되는 한자의 음과 형태를 이용한 교체형인 '김~숲', '장~튽'을 특별히 포함했다. 세 번째 열은 해당 변환쌍이 얼마나 자주 사용되는지를 나타내는 1 이상 3 이하의 숫자로 구성되어있다. 이 숫자가 낮을 수록 많이 사용되는 변환이다. 이 변환쌍 자료를 작성하는 방식은 상당히 자유로운 편인데, 야민정음 번역기가 예를 들어 alternation.txt에 '며'와 '띠'가 존재하든 '명'과 '띵'이 존재하든 상관 없이 '며'와 '띠'의 대응 관계를 추출해 새로운 텍스트들에 잘 적용해주기 때문이다.

ii) 한글 초성, 중성, 종성의 회전꼴과 그 회전각

초성, 중성, 종성에 사용되는 한글 자모를 모두 하나씩 살펴보며 90도 단위로 회전했을 때 다른 자모와 비슷해 보이는 것이 있는지 파악했다. 이를 토대로 data 디렉토리의 rotation.txt 파일을 만들었다. 이 파일은 3개의 열과 23개의 행으로 이루어져 있으며, 세 번째 열은 온도에 해당한다. 회전각은 기하학의 정의를 참고해 반시계 방향으로 회전했을 때를 기준으로 하였다. 또한 270도 회전은 따로 만들지 않고 소스 코드에서 90도 자료를 활용하도록 했다. 이 때, 'ㄸ'과 'ㅍ'을 90도 회전한 것은 모두 'ㅂ'이 되지만, 반대로 'ㅂ'을 270도 회전한 것은 'ㄸ'만 되는 것으로 정했다. 이는 'ㅂ'을 270도 회전한 것이 'ㄸ'에 가장 가까울 뿐 아니라, 유명한 야민정음 변환 중 하나인 '비버 ↔ 뜨또'가 널리 쓰이는 편이기 때문이다.

III. 야민정음 번역기의 작동

1. 절차

'야민정음 번역기' 코드가 작동하는 방식은 다음과 같다. 주 함수인 yaminize()는 정상적인 한글에서 야민정음으로, 또는 그 반대로 번역할 텍스트와 번역 강도를 입력받는다. 번역 강도 인자는 필수가 아니며, 입력하지 않을 때에는 가장 강한 번역 강도를 취하게 된다. 이때 프로그램은 이미 II.2.에서 언급한 자료들을 모두 가공해서 준비해둔다. 두 자료 모두 한 대응쌍들의 요소들(세 열에 해당하는 정보들)이 작은 리스트를 이루고, 그 리스트들이 여러 대응쌍으로서 큰 리스트를 이루는 이중의 리스트 형태로 가공된다. 한글 한 글자끼리의 변환쌍은 자소분해를 포함한 가공 과정을 거치게 된다. 텍스트를 단어 단위로 분할하여 변환 과정을 진행하며, 한 단어마다 먼저 II.2.ii)의 자료와 관련된 회전 변환을 시도하고, 변환에 실패하면 II.2.i)의 자료와 관련된 한글 한 글자끼리의 변환을 시도한다. 두 변환 모두 한글 한 글자(여기서는 글자와 음절을 구분하지 않는다.) 단위로 살피게 된다.

회전 변환의 경우에는 한 글자를 자소분해하여 각 자소들마다 고유한 딕셔너리를 갖게 한다. 그리고 자모들의 각 딕셔너리는 이미 자료로서 갖고 있는 회전꼴들과 현재의 자소를 비교하여 가능한 회전꼴(값)과 그 회전각(키)을 새로운 원소들로 포함하게 된다. 모든 회전꼴들과의 비교를 마치면, 현재 글자의 모든 자모들을 같은 방향으로 회전할 수 있는지를 종성 여부와 관련해 따지게 된다. 한글의 모아쓰기 형태상 종성이 없으면 90도와 270도만을 확인하고, 종성이 있으면 180도만을 확인하면 되기 때문이다. 만약 모든 자모들을 같은 방향으로 회전할 수 있다면 해당 회전각과 자모들의 회전꼴들이 잠재적인 야민정음 변환 형태가 된다. 하지만 트위터와 설문조사 등을 토대로 보았을 때, 사람들이 한 단어 내에서 일부 글자만 특정 각도로 회전해서 변환하는 경우가 거의 없기 때문에, 이러한 음절들의 잠재적인 회전각들을 모두 비교하여 한 단어의 모든 음절이 같은 회전각을 가능성으로 가질 때만 자소조합되어 단어 전체를 회전시켜 변환하게 했다. 이때 단어의 회전각이 90도인 경우에는 음절 간의 순서를 그대로 하여 회전하지만, 180도나 270도인 경우에는 음절들의 순서 또한 거꾸로 한다. 이에 관련한 설명은 III.2.에서 좀 더 하도록 한다.

단어의 회전 변환에 실패한 후에는 단어의 각 음절들을 가능한 대로 다른 한글 한 글자로 변환한다. 비슷하거나 관련된 한글 한 글자끼리의 변환의 경우, 번역 강도 및 각 변환쌍의 고유한 사용 빈도에 영향을 받는다. 이 사용 빈도는 앞서 밝혔듯 II.1.에서 조사한 자료들을 토대로 한 것이다. 입력 받은(또는 기본 값의) 번역 강도는 숫자가 클수록 더 흔치 않은 변환까지도 적용한다는 것을 뜻한다. 한 글자마다 각 변환쌍들의 앞이나 뒤 글자와 대응되는 것이 있는지를 확인하는데, 당연히 설정된 번역 강도보다 숫자가 큰(덜 흔한) 변환쌍은 확인하지 않는다. 또한, 번역 강도 안에서 허용되는 변환쌍들 간에도, 흔한 변환쌍들을 먼저 확인하고 대응되는 것이 있으면 바로 적용하고 이후의 확인을 하지 않도록 한다. 이는 코드에서 사용 빈도 순으로 변환쌍들을 확인함으로써 가능한 것이기도 하지만, 더 세밀하게는 II.2.i)에서 설명했듯 같은 사용 빈도를 가진 변환쌍들의 상대적 순서로도 조절 가능하다.(사용 빈도가 동일하다면, 상단에 위치한 변환쌍이 하단에 위치한 변환쌍보다 우선적으로 적용된다.) 자료의 변환쌍들은 자소 단위로 유형을 구분하는데, 변환될 형태와 초성만 다른 것, 초성과 중성이 다른 것, 중성과 종성이 다른 것, 모든 자모가 다른 것의 유형이 존재한다. 음절 단위의 구체적인 변환 과정은 조금 복잡한데, 먼저 위에서 구분한 유형별로 각 변환쌍의 상호 관계를 확인하고, 주어진 음절의 자모에 해당 변환이 적용 가능한지를 확인한다. 두 조건문이 성립하면 원래 음절의 해당 자모는 대응하는 변환 형태로 교체된다. 그리고 이렇게 변환되었거나 또는 변환되지 못한 글자들이 모여 자소조합된 후 한 단어를 구성하게 된다.

앞서 설명한 대로 자소분해와 변환, 자소조합의 과정을 거친 단어들은 다시 주 함수인 yaminize()에 모여 전체적으로 번역된 텍스트를 이루게 된다.

2. 소스코드

```
11 ## 준비물 가져오기 및 가공 ##
12 import re, hangul, conjoin_hangul_jamos
```

re.split()과 re.search() 함수와 같은 정규표현 관련 기능을 사용해야 하므로 re 모듈을 import한다. 또한 한글 자소분해와 자소조합을 적극적으로 활용해야 하므로, '언어와 컴퓨터' 수업에서 배우고 [숙제08]에서 직접 다룬 hangul.py와 마찬가지로 수업에서 배우고 [숙제07심화]에서 직접 다룬 conjoin_hangul_jamos.py를 이용했다. 다만 완전히 같은 코드는 아니고, hangul.py는 현재 프로젝트에서 불필요한 if __name__ '__main__': 관련 내용과 한글 초성 찾기 함수(find_choseong()), 한글 종성 찾기 함수(find_jongseong())을 삭제했다. 나머지는 완전히 같으므로 hangul.py에 대해서는 파일에만 첨부하고 더 이상 언급하지 않는다.
conjoin_hangul_jamos.py의 경우 필요에 따라 수정한 부분이 다소 있다. 코드 안 데이터의 한글 자모 배열 순서 중 VOWEL에 있던 오타를 수정했으며, 스크립트 모드에서 사용되는 것을 전제로 되어있던 코드를 뼈대는 그

대로 하되, 목적에 맞게 conjoin()이라는 함수로 수정했다. conjoin() 함수는 한글 한 음절이 자소분해된 문자열을 인자로 받는다. 즉, hangul.py의 decompose() 함수의 반대 역할을 한다. 또한 yaminjeongeum_translator.py 소스코드에서는 종성이 없는 경우를 '_'로 꾸준히 분명히 하여 사용하기 때문에, conjoin_hangul_jamos.py의 conjoin() 함수도 '_'를 마지막에 포함한 자소분해 문자열을 입력받으면 이를 종성이 없는 음절이 자소분해된 것으로 처리하게 하였다. 즉, hangul.py의 decompose() 함수에서 no_batchim 인자를 '_'으로 두는 경우를 반대로 하는 것이라 볼 수 있다. conjoin() 함수의 코드는 아래와 같다. 이에 따라 스크립트 모드에서 종성이 입력되지 않을 경우를 대비해 작성했던 코드의 내용은 불필요했기 때문에 모두 삭제했다.

```
26 def conjoin(syl):
27 # 과정 1: 입력받은 초성, 중성, 종성이 각각의 배열에서 몇 번째인지를 파악한다.
28     LIndex = LEADING.find(syl[0]) # 초성
29     VIndex = VOWEL.find(syl[1]) # 중성
30     if syl[2] == '_': # 종성이 없는 경우
31         TIndex = 0 # 종성 배열의 맨 앞인 것으로 파악한다.
32     else: # 종성이 있는 경우
33         TIndex = TRAILING.find(syl[2]) + 1 # 종성이 없는 경우를 고려해 1을 더한다
34 # 과정 2: 입력받은 초성, 중성, 종성을 조합했을 때 '가'로부터 몇 번째 다음으로 나오는지 파악한다.
35     SIndex = LIndex*NCount + VIndex*TCount + TIndex
36
37 # 종합: 순서를 알았으므로 실제 음절을 가져온다.
38     return chr(SBase+SIndex)
```

본격적인 yaminjeongeum_translator.py의 소스코드에 대한 설명은 아래와 같다.

```
15 # 준비물 1(한글 한 글자끼리의 변환쌍 - Aset) 및 주어진 텍스트의 한글을 분해 및 가공하는 함수
16 def decompose_hangul_jamos(syl):
17     if ord('가') <= ord(syl) <= ord('힣'): # 초·중·종성 모두 있거나 종성만 없는 글자들
은
18         Dsyl = hangul.decompose(syl, no_batchim = '_', del_nonhangul =
False) # hangul.decompose로 자소분해
19     else: # 자음 또는 모음만 존재하는 것은 자소 분해처럼 처리
20         if ord('ㄱ') <= ord(syl) < ord('ㅎ'): # 자음만 존재하는 경우
21             Dsyl = syl[0] + '_' + '_' # 자소 분해했을 때 초성만 있는 것처럼
22         elif ord('ㅏ') <= ord(syl) < ord('ㅣ'): # 모음만 존재하는 경우
23             Dsyl = '_' + syl[0] + '_' # 자소 분해했을 때 중성만 있는 것처럼
24     return Dsyl # 분해 및 가공된 한글 한 글자
```

우선 hangul.decompose() 함수를 포함한, 목적에 걸맞는 자소분해 함수를 **decompose_hangul_jamos()**로 새롭게 정의했다. 이 함수는 한글 한 글자를 받아 자소분해 및 가공하여 그 문자열을 반환하는 함수이다. 'alternation.txt'에서 가져오는 데이터(준비물 1, 한글 한 글자끼리의 변환쌍)를 가공하는 데에도 사용된다.

이름	자료형	기본값	내용
syl	str	없음	준비물 1(한글 한 글자끼리의 변환쌍) 또는 주어진 텍스트의 한글 한 글자(음절).

초성, 중성, 종성이 모두 있는 경우에는 hangul.decompose()를 그대로 사용하되, no_batchim을 '_'로, del_nonhangul을 False로 한다. no_batchim을 분명히 표시하는 이유는, 야민정음 변환에 있어 초성, 중성, 종성의 존재 유무는 매우 중요하기 때문이다. 또한 자소분해한 문자열의 길이를 똑같이 함으로써 복잡한 문제를 방지할 수도 있다. del_nonhangul을 False로 하는 것은 매우 중요하다. 야민정음의 특성상, '구'를 'ㅋ'으로 표기하는 등, 온전한 한글 한 글자가 아닌 것도 하나의 음절로 취급해야하는 경우가 많기 때문이다. 그런데 hangul.decompose()는 del_nonhangul을 기본값인 True로 두면 이러한 단일한 자음이나 모음을 모두 없애버리는 문제가 있다.(다만 이 경우, 이미 앞선 17행의 조건문에서 자음이나 모음만 존재하는 경우는 제외했으므로 필수적인 것은 아니다.) 자소분해에서 종성이 없는 것을 '_'로 나타내는 것처럼, 자음 또는 모음만 존재하는 한글

은 각각 중성과 종성, 초성과 종성 위치를 '_'로 나타내어 혼란을 방지한다. 마지막으로 각각의 경우에 맞게 분해 및 가공된 문자열 Dsyl을 반환한다.

II.2.에서 소개한 자료들을 가져와 가공한다.

```
27 # 준비물 1: 한글 한 글자끼리의 변환쌍
28 f = open('data/alternation.txt') # data 디렉토리의 alternation.txt 파일 열기
29 Asets = [[decompose_hangul_jamos(Aset[0]), decompose_hangul_jamos(Aset[1]),
int(Aset[2])]\
30 for Aset in [line.split() for line in f.readlines()]] # 행별로 끊어 공백을 기준으로
나눈 후, 그에 대해 분해 및 가공 함수 적용
31 f.close() # 파일 닫기
32
33 # 준비물 2: 한글 초·중·종성의 회전꼴
34 f = open('data/rotation.txt') # data 디렉토리의 rotation.txt 파일 열기
35 Rsets = [[Rset[0], Rset[1], int(Rset[2])] for Rset in [line.split() for line
in f.readlines()]] # 행별로 끊어 공백을 기준으로 나눈 후, 그에 대해 가공(Rset[2]는 그 회전꼴의
회전각)
36 f.close # 파일 닫기
```

먼저 data 디렉토리의 alternation.txt 파일을 f라는 이름으로 하여 연다. 30행 뒤쪽의 [line.split() for line in f.readlines()]는 이 파일을 .readlines()를 이용해 행별로 끊어읽고, 각각의 행을 또 .split()을 이용해 공백을 기준으로 나누어 list에 저장한 것이다. 그리고 이 list의 모든 원소 list 하나씩들(즉, ['대', '머', '1']과 같은 하나의 변환쌍)을 Aset으로 하여 앞서 설명한 decompose_hangul_jamos() 함수를 이용해 Aset[0]과 Aset[1]에 해당하는 한글 한 글자들을 자소 분해 및 가공한다. Aset[2]의 경우에는 사용 빈도에 해당하므로 이를 int 자료형으로 변환한다. 이렇게 각자 가공한 내용들을 모아 한 변환쌍에 대한 새로운 list를 만들고, 이 list들을 모아 모든 변환쌍들의 list를 이중으로 만든 것이 **Asets**다. 이후에 파일을 닫는다.

rotation.txt과 Rsets도 거의 마찬가지다. 해당 파일을 행별로 끊어 읽고, 각 행을 .split()을 이용해 list에 저장한다. 그리고 이 list의 모든 원소 list 하나씩들(즉, ['ㅁ', 'ㅁ', '90']과 같은 하나의 쌍)을 Rset으로 하여, 나머지는 그대로 두고 회전각에 해당하는 Rset[2]만 int 자료형으로 변환하여 새롭게 list에 저장한다. 그리고 이 list들을 모아 모든 회전쌍들의 list를 이중으로 만든 것이 **Rsets**다.

이름	자료형	값	내용
Asets	list	[['ㄷㅐ_', 'ㅁㅓ_', 1], ['ㅍㅏ_', 'ㄱㅘ_', 1], ['ㅍㅣ_', 'ㄲㅟ_', 1], ['ㅍㅗ...	한글 한 글자끼리의 변환쌍과 그 사용 빈도를 포함한 list들을 포함한 list.
Rsets	list	[['ㅁ', 'ㅁ', 90], ['ㅂ', 'ㅍ', 90], ['ㅅ', 'ㄷ', 90], ['ㅆ', 'ㅌ', 90...	자모의 회전꼴과 그 회전각을 포함한 list들을 포함한 list.

주 함수인 yaminize()의 주축을 이루는 함수 두 개 중 하나인 **similize()**는 Asets 데이터를 이용하여 한글 한 글자를 직접적으로 형태상 비슷한 다른 한글이나, 관련된 한자의 음, 형태 등과 관련된 또 다른 한글로 변환하여 반환하는 함수이다. 인수로서 입력받는 형태는 온전한 한글 한 글자이지만, 반환하는 것은 자소분해된 형태이며, 이것은 yaminize()의 하위 함수라 볼 수 있는 yaminize_word()에서 조합된다.

이름	자료형	기본값	내용
syl	str	없음	형태상 비슷한 한글 등으로의 변환이 시도되는, 번역 대상 텍스트의 일부인 한글 한 글자.

이름	자료형	기본값	내용
degree	int	없음	번역 강도이며, Asets의 각 변환쌍들의 사용 빈도와 비교됨.

```
40 ## 야민정음 변환법 1: 준비물 1을 활용하여 비슷하거나 관련된 한글/한자로 변환 ##
41 def similize(syl, degree): # 한글 한 글자와 번역 강도를 인자로 하는 함수
42     Dsyl = decompose_hangul_jamos(syl) # 현재 글자를 분해 및 가공
43
44     for p in range(1, degree+1): # 사용 빈도가 높은(p가 1에 가까운) 변환쌍일수록 먼저 확인
45         for Aset in Asets: # 변환쌍들을 하나씩 확인
46             if Aset[2] <= p: # 현재 탐색 중인 사용 빈도보다 낮은 것(Aset[2]가 p보다 큰 것)은 무시
47                 for i in range(2): # 변환쌍의 앞 부분과 뒷 부분을 모두 탐색하여 양방향 변환이 가능하도록
48                     if i == 0: j = 1
49                     else: j = 0
```

함수의 앞 부분을 설명하자면, 한글 한 글자와 번역 강도를 인자로 받아 해당 한글 음절은 decompose_hangul_jamos() 함수로 분해 및 가공한다. 이렇게 자소분해된 Dsyl이 미리 준비된 Asets의 각 변환쌍들(Aset) 중 한쪽(Aset[0] 또는 Aset[1])과 대응되는지를 반복문으로 확인하는 것이 이 함수의 핵심이다. 이 때, for p in range(1, degree+1):은 특정 사용 빈도를 나타내는 p가 가장 흔한 사용 빈도를 뜻하는 1부터, 지정된 번역 강도(degree)까지 점차적으로 증가하며 Asets의 변환쌍들을 확인한다는 것을 나타낸다. 아래 코드에서 볼 수 있듯, 이 함수는 대응되는 변환쌍을 찾으면 바로 해당 내용을 반환하고 함수를 종료하기 때문에, 흔한 사용 빈도를 보이는 변환쌍들부터 확인해 우선 순위가 뒤바뀌지 않도록 하는 것이다. 이는 지정된 번역 강도 내에서도(if Aset[2] <= p:) 좀 더 실제 사용 양상과 가까운 번역을 가능하게 한다.

for i in range(2):는 변환쌍의 앞과 뒤(Aset[0] 또는 Aset[1])를 모두 확인하기 위한 것이다. 앞과 대응되면 뒤로 변환하고, 뒤와 대응되면 앞으로 변환할 수 있도록 48행과 49행의 조건문을 통해 i와 j가 교차되는 값을 가지도록 한다.

```
51                     # 변환쌍의 앞과 뒤가 초·중·종성이 다른 정도에 따라 구분하여, 현재 글자에 적용할 수 있는지 확인
52                     # 특정 자모가 False이면, 그것이 다른 변환쌍인지 확인하고 적용을 시도하는 것(True가 자모 인자의 초깃값)
53                     # i) 초성만 다른 변환쌍('푸' ↔ '쭈' 등)
54                     Ysyl = similize_check(Dsyl, Aset, i, j, L = False)
55                     if Ysyl: return Ysyl # 초성만 바꾸는 변환이 적용 가능하면 그것을 적용해 반환하고 이 함수를 마침
56                     # ii) 초성과 중성이 다른 변환쌍('대' ↔ '머' 등) - 변환에 추가 조건이 필요함
57                     if (Aset[j][0] == '_' or Aset[j][1] == '_') and Dsyl[2] != '_': pass # 현재 글자의 종성이 존재하는데, 변환 목표가 자음 또는 모음 하나로만 이루어진 경우('구' ↔ 'ㅋ', '시' ↔ 'ㅐ' 등)
58                     else: # 그런 경우가 아닐 때에만 정상적인 변환 가능
59                         Ysyl = similize_check(Dsyl, Aset, i, j, L = False, V = False)
60                         if Ysyl: return Ysyl # 초성과 중성을 바꾸는 변환이 적용 가능하면 그것을 적용해 반환하고 이 함수를 마침
61                     # iii) 중성과 종성이 다른 변환쌍('우' ↔ '윽' 등)
62                     Ysyl = similize_check(Dsyl, Aset, i, j, V = False, T = False)
```

```
 63                     if Ysyl: return Ysyl # 중성과 종성을 바꾸는 변환이 적용 가능하면
그것을 적용해 반환하고 이 함수를 마침
 64                     # iv) 모든 자모가 다른 변환쌍('통' ↔ '둥', '김' ↔ '숲' 등)
 65                     Ysyl = similize_check(Dsyl, Aset, i, j, L = False, V =
False, T = False)
 66                     if Ysyl: return Ysyl # 모든 자모를 바꾸는 변환이 적용 가능하면 그
것을 적용해 반환하고 이 함수를 마침
 67
 68     Ysyl = Dsyl # 번역 강도 내에서 어떤 변환쌍도 이용할 수 없는 경우
 69     return Ysyl # 원래 글자를 반환함
```

Asets의 변환쌍들(Aset)에 대해서는 Aset[0]과 Aset[1]이 초성만 다른지(54~55행), 초성과 중성이 다른지(57~60행), 중성과 종성이 다른지(62~63행), 모든 자모가 다른지(65~66행)를 유형을 나눠 처리한다. 이 네 유형에 대한 처리 과정은 거의 같은 것이 반복되는 것이므로 따로 similize_check() 함수를 만들어 진행했다. similize_check() 함수는 해당 유형의 변환쌍에 걸맞는 조건문을 현재 글자(를 자소분해한 Dsyl)에 제시하고, Dsyl이 해당 조건을 만족해 변환이 이루어질 수 있는 것으로 나타나면 변환된 (아직 자소조합은 되지 않은)글자를 반환한다. 반면, 변환이 적용될 수 없는 경우에는 반환하는 구체적인 값이 없으므로 위에서 여러 번 나타나는 if Ysyl:조건문을 성립시키지 않는다. 즉, 하나의 변환쌍 Aset에 대해 해당 변환쌍이 어떤 유형의 변환쌍인지를 파악하고, 유형을 파악한 후 현재 글자에 해당 변환을 적용할 수 있는지를 확인하는 함수가 **similize_check()**이다. 그리고 이 함수가 변환된 (자소조합은 되지 않은)글자를 반환할 때, if Ysyl: 조건문은 성립되며 이에 따라 similize() 함수는 바로 Ysyl을 반환하고 종료된다. 따라서 44행부터 66행까지의 다양한 반복문들(사용 빈도, 변환쌍, 같은 변환쌍에서의 앞과 뒤)을 모두 거쳐도 similize()가 종료되지 않았다면, 이 글자는 다른 한글로 변환될 수 없는 글자로 볼 수 있다. 이러한 글자는 68행을 거쳐 Ysyl에 원래 글자가 자소분해된 것일 뿐인 Dsyl을 저장하고 이를 반환하게 된다. 즉, 변환을 겪지 않는다.

초성과 중성이 달라지는 변환 유형만 57~58행에서와 같은 조건문이 존재하는 것은, '구'가 'ㅋ'이 되고, '시'가 'ㅐ'가 되는 등의 변환을 다른 변환들처럼 무작정 확장해서 적용할 수 없기 때문이다. 예를 들어, '파'와 '과'의 대응을 보고 '광'을 '팡'으로 변환시켜주는 것에는 문제가 없지만, '구'와 'ㅋ'의 대응을 보고 '국'을 이에 맞추어 변형하려고 하면 종성을 온전히 유지할 수 없다. 이렇게 초성과 중성을 자음(초성) 또는 모음(중성) 하나로만 변환하는 경우에는, 변환하려는 글자에 종성이 없어야 한다는 제약이 생기는 것이다. 이를 정확히 57행의 조건문이라고 할 수 있다. 해당 조건문의 내용처럼 변환이 이루어질 수 없는 경우에는 그냥 pass하고, 그렇지 않은(else) 경우에만 similize_check() 함수를 적용한다.

similize_check() 함수의 역할은 이미 위에서 언급했다. 여기서는 구체적인 작동 방식에 대해서만 더 살펴보도록 한다.

이름	자료형	기본값	내용
Dsyl	str	없음	형태상 비슷한 한글 등으로의 변환이 시도되는, 번역 대상 텍스트의 일부인 한글 한 글자가 자소분해 및 가공된 것.
set	list	없음	similize() 함수에서의 Aset에 해당. 즉, 한 변환쌍의 묶음.
i	int	없음	변환쌍(set)의 앞/뒤 중 어디를 기준으로 살필지를 나타냄. 0 또는 1을 가지며, j와 무조건 다른 값을 가짐.
j	int	없음	변환쌍(set)의 앞/뒤 중 어디를 기준으로 살필지를 나타냄. 0 또는 1을 가지며, i와 무조건 다른 값을 가짐.
L	bool	True	변환쌍의 앞/뒤에 초성의 차이가 있는지를 볼 때에 False.
V	bool	True	변환쌍의 앞/뒤에 중성의 차이가 있는지를 볼 때에 False.
T	bool	True	변환쌍의 앞/뒤에 종성의 차이가 있는지를 볼 때에 False.

```
 72 # 야민정음 변환법 1에서 반복되는 부분을 일반화하여 처리하는 함수
```

```
 73 def similize_check(Dsyl, set, i, j, L = True, V = True, T = True): # 분해된 음절과 현재 살피고 있는 변환쌍, 변환쌍의 앞/뒤 중 어디를 보고 있는지, 어떤 자모가 다른 것에 주목하는지를 인자로 하는 함수
 74     set_L, set_V, set_T = bool(set[i][0] == set[j][0]), bool(set[i][1] == set[j][1]), bool(set[i][2] == set[j][2]) # 자모 인자에 따라 변환쌍의 앞과 뒤 비교 조건이 달라지는 것을 구현하는 준비
 75     match_list = [bool(Dsyl[0] == set[i][0]), bool(Dsyl[1] == set[i][1]), bool(Dsyl[2] == set[i][2])] # 자모 인자에 따라 현재 글자에 변환이 적용 가능한지 확인하는 조건이 달라지는 것을 구현하는 준비
 76
 77     LVT_bools = [L, V, T] # 자모 인자들을 한 번에 효율적으로 활용하기 위한 준비(초·중·종성 순서일 때 True와 False가 항상 연속된 두 부분으로 나눠짐에 착안)
 78     k = LVT_bools.index(False) # False가 처음 등장하는 위치를 k
 79     try: l = LVT_bools.index(True) # True가 처음 등장하는 위치를 l (초·중·종성 순서일 때 False 범위를 자동으로 나타내는 것이 목적)
 80     except: l = 3 # True가 없는 경우(모든 자모를 바꾸는 변환), l에 3을 부여하여 [0:3]을 나타냄
 81     if k > l: l = 3 # False가 True보다 나중에 등장하는 경우(중성과 종성을 바꾸는 변환), l에 3을 부여하여 [1:3]을 나타냄
 82
 83     if L == set_L and V == set_V and T == set_T: # 주어진 모든 자모 인자들의 bool 값과, 변환쌍의 앞뒤 비교 결과의 bool 값이 같고
 84         if all(match_list[k:l]): # 주어진 인자가 False인 현재 글자의 자모에 변환이 모두 적용 가능하면
 85             Ysyl = Dsyl.replace(Dsyl[k:l], set[j][k:l]) # 해당 범위에 변환을 적용하여 Ysyl로 저장
 86             print(Dsyl, '>', Ysyl, 'LVT'[k:l]) # 원래 글자의 어떤 자모가 변환되어 새로운 글자가 되었는지 기록 출력
 87             return Ysyl # Ysyl 반환
```

similize_check()는 많은 인수를 갖지만, 거의 대부분 하나의 변환쌍을 어떻게 살필 것이냐에 관련된 것들이다. Dsyl, set, i, j는 이 함수 내에서 역동적인 역할을 하지 않고 similize()의 내용을 그대로 이어받는 것이기에 따로 언급할 필요가 없다. 중요한 것은 L, V, T인데, 이들은 bool 자료형으로 기본값 True를 갖는다. 이들은 오로지 기계적인 일반화를 위한 철저히 수단적인 인수라고 할 수 있다. 특정 변환쌍(set)의 앞과 뒤에 중성, 종성의 차이가 있는 지를 살피고 이러한 변환이 지금의 글자(Dsyl)에 적용될 수 있는지를 알아보기 위해 V, T에 False가 부여된 상황을 가정해 보자. 83~87행이 이 함수의 핵심이라고 할 수 있으며, 그 위의 내용은 다양한 경우들을 일반화하는 것을 준비하는 과정이라 볼 수 있다. 가정된 상황에서 83행의 조건이 성립하려면, 유일하게 True인 L과 똑같아야 하는, 즉 참이어야 하는 것은 set_L에 해당하는 bool(set[i][0] == set[j][0])이다. 즉, 변환쌍의 앞과 뒤의 초성([0]) 같아야 하는 것이다. 한편 V와 T는 False이므로, bool(set[i][[1] == set[j][1])과 bool(set[i][2] == set[j][2])는 거짓이어야 한다. 즉, 변환쌍의 앞과 뒤의 중성과 종성은 달라야 하는 것이다.

또한 77~81행은 84행의 조건문을 위한 것이다. 이것은 각주에 언급되어있듯 초·중·종성 순서일 때 True와 False가 항상 연속된 두 부분(L, V, T가 모두 False일 때 제외. 이 경우가 80행에 해당된다.)으로 나눠진다는 것에 착안하여, match_list[k:l]가 match_list에서 가리키는 범위를 [L, V, T] 중 False인 것의 원소들 위치와 똑같이 맞추려는 시도이다. match_list 역시 bool 자료형 세 개를 포함하는 list이다. 앞서 가정된 상황에 대해 계속해서 생각해보면, V와 T만 False이므로 k의 값은 .index()의 특성상 찾는 원소가 처음 등장하는 위치(index)를 반환하기 때문에 V의 위치에 해당하는 1, l의 값은 True인 L에 해당하는 0이다. 하지만 l은 k보다 커야하기 때문에, 81행과 같은 조건문을 거쳐 k는 1, l은 3이 된다. 84행의 조건문을 성립하려면 all()의해 match_list[1:3]의 모든 원소는 참이어야 한다. 즉, bool(Dsyl[1] == set[i][1])과 bool(Dsyl[2] == set[i][2])가 참이어야 한다.(bool(Dsyl[0] == set[i][0])의 경우에는 어떻든 상관이 없다.) 이것은 '현재 다루고 있는 한글 한 음절이 변환쌍 중 한 쪽(i의 값과 관련)의 중성과 종성에 일치하면' 이라는 조건을 뜻하게 된다.

따라서 83행, 84행의 조건이 모두 성립된다면, Dsyl은 해당 변환쌍의 해당 자모 변환에 대해 적용 가능한 것으로 판명되는 것이다. 이에 따라 직접 해당 범위(Dsyl[k:l])의 자소를 .replace()를 통해 교체하고, 변환한 (자소조

합은 되지 않은)글자를 Ysyl에 저장한다. 그리고 이 변환 기록을 출력(VT만 변환했으면 VT를 출력하는 식)하고 Ysyl을 반환한다.

주 함수인 yaminize()의 주축을 이루는 또 다른 함수인 **rotate()**는 Rsets 데이터를 이용하여 한글 한 글자(전체 맥락에서 보자면 한 단어)를 특정 각도로 회전시키고자 하는 함수이다. 인수로서 입력받는 형태는 온전한 한글 한 글자이지만, 반환하는 것은 회전된 글자의 자소분해된 형태(회전이 불가능한 경우 False)와 해당 회전각(회전이 불가능한 경우 False)이 이루는 tuple이다. similize()와 달리, rotate()에서 False로 이루어지지 않은 tuple을 반환했다고 해서 그 글자가 꼭 회전 변환되는 것은 아니다. 이전에도 언급했듯, 회전 변환은 단어 단위로 이루어지는 것이 자연스럽기 때문에, '잠재적인' 회전꼴과 회전각을 가진 한 단어의 모든 음절들이 똑같은 각도로 회전이 가능할 때 회전 변환이 이루어진다. 반환된 값이 yaminize()의 하위 함수라 볼 수 있는 yaminize_word()에서 다뤄지는 점은 similize()와 같다.

이름	자료형	기본값	내용
syl	str	없음	회전을 이용한 변환이 시도되는, 번역 대상 텍스트의 일부인 한 글 한 글자.

```
 91 ## 야민정음 변환법 2: 준비물 2를 활용하여 한 단어를 같은 방향으로 회전하여 변환 ##
 92 def rotate(syl): # 한글 한 글자를 인자로 하는 함수
 93     Dsyl = decompose_hangul_jamos(syl) # 현재 글자를 분해 및 가공
 94     Rdics = [{0: Dsyl[0]}, {0: Dsyl[1]}, {0: Dsyl[2]}] # 분해한 글자의 현재 초·중·종성을 각각의 딕셔너리에 0이라는 키를 만들어 그 값으로 저장
 95
 96     for Rset in Rsets: # 회전꼴들을 하나씩 확인
 97         for k in range(3): # 초·중·종성 각각에 대해
 98             if Dsyl[k] == Rset[0]: # 회전꼴의 앞 부분에 대응되면
 99                 Rdics[k][Rset[2]] = Rset[1] # 해당 회전각을 키로 만들어 새로운 회전꼴을 그 값으로 저장
100             if Dsyl[k] == Rset[1]: # 회전꼴의 뒷 부분에 대응되면
101                 Rdics[k][360-Rset[2]] = Rset[0] # 360 - 해당 회전각을 키로 만들어 새로운 회전꼴을 그 값으로 저장
```

함수의 앞 부분을 설명하자면, 인자로 받아 한글 한 음절은 decompose_hangul_jamos() 함수로 분해 및 가공한다. 이렇게 자소분해된 Dsyl의 각 자소들이 0을 키로 하여 그 값이 되는 각각의 고유한 딕셔너리를 갖게 하고, 그 딕셔너리들이 이루는 list를 Rdics라고 한다. 여기에 포함된 각 딕셔너리들은 이미 Rsets로서 갖고 있는 회전꼴들과 현재의 자소를 비교하여, 자소별로 가능한 회전꼴들을 그 회전각과 값과 키의 관계를 이루는 새로운 원소들로 추가한다. 이것이 96~101행에 해당한다. for Rset in Rsets는 회전꼴들을 하나씩 검토한다는 것이고, for k in range(3)은 초성, 중성, 종성 각각에 대해 한 회전꼴을 확인하는 것을 뜻한다. 98~99행은 if Dsyl[k] == Rset[0]에서 볼 수 있듯 현재 자소가 회전꼴의 앞 부분에 대응된 경우이다. 이 때에는 회전각(Rset[2])을 그대로 사용하면 되지만, 100~101행에서처럼 Dsyl[k]가 회전꼴의 뒷 부분(Rset[1])에 대응하면 반대 방향이라는 의미에서 360도에서 뺀 회전각을 사용해야 한다. 특히 270도가 이러한 경우에 해당한다.

```
103     if Dsyl.endswith('_'): # 현재 글자의 종성이 없으면
104         try: # 각 자모를 모두 90도 회전할 수 있는지 확인
105             return rotate_check(90, Dsyl, Rdics) # 가능하면 그것을 적용해 반환하고 이 함수를 마침
106         except: pass # 가능하지 않으면 넘어가기
107         try: # 각 자모를 모두 270도 회전할 수 있는지 확인
108             return rotate_check(270, Dsyl, Rdics) # 가능하면 그것을 적용해 반환하고 이 함수를 마침
109         except: Ysyl, angle = False, False # 90도와 270도 모두 가능하지 않으면 False들을 저장
110     else: # 현재 글자의 종성이 있으면
```

```
111         try: # 각 자모를 모두 180도 회전할 수 있는지 확인
112             if Rdics[0][180] and Rdics[1][180] and Rdics[2][180]:
113                 Ysyl, angle = Rdics[2][180] + Rdics[1][180] + Rdics[0][180],
180 # 가능하면 그것을 적용
114                 print(Dsyl, '>', Ysyl, '180°') # 원래 글자를 몇 도 회전하여 새로운
글자가 되었는지 기록 출력
115                 return Ysyl, angle # 반환하고 이 함수를 마침
116         except: Ysyl, angle = False, False # 가능하지 않으면 False들을 저장
117
118     return Ysyl, angle # 모든 회전에 실패한 경우 False들을 반환
```

모든 회전꼴들과의 비교를 마치면, 현재 글자의 모든 자모들을 같은 방향으로 회전할 수 있는지를 확인한다. 이 때 중요한 것이 if Dsyl.endswith('_'):에서 볼 수 있는 종성의 유무이다. III.1.에서 언급한 것과 같은 맥락으로, 한글의 모아쓰기 형태상 종성이 없으면 180도 회전을 이용하는 경우가 거의 없고, 종성이 있으면 90도와 270도 회전을 이용하는 경우가 거의 없기 때문이다. 물론 '표 ↔ 표' 같은 경우가 존재하며, 로마자 등을 포함하면 매우 다양한 가능성이 존재할 수 있다. 이는 이 야민정음 번역기의 한계점들 중 하나로 남겨둔다. 각 자모를 모두 특정 방향으로 회전할 수 있는지에 대해서는 try와 except를 이용하여 판별한다. 이 때 **rotate_check()** 함수가 사용되는데, 이 함수는 현재 글자(Dsyl)의 모든 자모를 특정 각도(a)로 회전시킬 수 있는지를 확인하고, 가능한 경우에는 회전된 자소분해 문자열(Ysyl)과 해당 회전각의 tuple을 반환한다. 이 함수의 코드는 아래와 같으며, 기본적으로 112행~115행과 거의 동일하다.

```
121 # 야민정음 변환법 2에서 반복되는 부분을 일반화하여 처리하는 함수
122 def rotate_check(a, Dsyl, Rdics): # 회전각과 분해된 음절, 각 자모별 회전꼴들을 인자로 하
는 함수
123     if Rdics[0][a] and Rdics[1][a]: # 초성과 중성을 주어진 회전각만큼 모두 회전할 수 있
으면
124         Ysyl, angle = Rdics[0][a] + Rdics[1][a] + '_', a # 그것을 적용
125         print(Dsyl, '>', Ysyl, str(a) + '°') # 원래 글자를 몇 도 회전하여 새로운 글자
가 되었는지 기록 출력
126         return Ysyl, a # 반환하고 이 함수를 마침(불가능하면 아무것도 반환하지 않음)
```

그런데 123행에서 볼 수 있듯, 이 함수는 이미 초성(Rdics[0])과 중성(Rdics[1])이 모두 인수로 받은 회전각 a를 가질 것을 전제하고 있다. 이 때문에 만약 초성, 중성 중 어느 하나라도 해당 회전각의 가능성을 갖고 있지 않다면, 일반적으로는 오류가 출력되게 된다. 이를 이용하여 try와 except를 사용한 것이다. 즉, 초성과 중성이 모두 90도 또는 270도의 회전 가능성을 가지면 rotate_check()를 통과하여 변환된 내용을 105행과 108행에서처럼 rotate()에서도 그대로 반환하여 함수를 종료하지만, 그렇지 않은 경우에는 except 이하로 흐름이 진행되는 것이다. 90도와 270도의 회전 가능성을 모두 가지지 못한 한글 음절은 109행에 따라 Ysyl과 angle에 모두 False를 저장하게 된다. 종성이 있는 180도 회전의 경우에도 try와 except를 사용하는 아이디어는 마찬가지로 적용된다. 이러한 회전 가능성을 가지지 못한 한글 음절 역시 116행에 따라 Ysyl과 angle에 모두 False를 저장하게 된다. 그리고 이 값들을 tuple로 하는 결과값을 반환하게 된다. 아래는 rotate_check()의 인수에 대한 정보이다.

이름	자료형	기본값	내용
a	int	없음	현재 글자를 회전시킬 수 있는지 알고자 하는 각도.
Dsyl	str	없음	회전을 이용한 변환이 시도되는, 번역 대상 텍스트의 일부인 한글 한 글자가 자소분해 및 가공된 것.
Rdics	list	없음	가능한 회전각들을 키로 하여 회전꼴들이 각각 그 값이 되는 딕셔너리가 자소마다 있는 list.

야민정음 번역기를 이용할 때 직접적으로 사용하게 되는 함수가 바로 **yaminize()**이다. 이 함수 자체는 매우 간단하지만, 나머지 모든 함수들의 시작점이자 도착점이라 할 수 있다.

```
130 ## 야민정음 변환 종합 ##
131 def yaminize(s, degree = 3): # 텍스트 전체와 번역 강도를 인자로 하는 함수
132     print('--------------------') # 텍스트 변환 기록 출력의 시작을 나타내는 선
133     return ''.join([yaminize_word(word, degree) for word in re.split('([^가-힣ㄱ-ㅎㅏ-ㅣ]+)', s)]) # 한글이 아닌 것을 기준으로 단어를 나눈 후, 그 각각에 대해 함수를 적용하여 변환하고 다시 합쳐 반환
```

이 함수의 인수들은 사용자에게 직접 입력받게 된다. s는 번역을 할 텍스트에 해당하며, 3이 기본값인 degree는 사용자가 원하는 번역 강도이다. 입력하지 않으면 가장 강한 번역 강도, 즉 잘 사용되지 않는 편으로 보이는 변환쌍들도 이용하는 번역이 진행된다. 이를 방지하고 싶으면 상대적으로 낮은 2나 1을 입력하면 된다.

이름	자료형	기본값	내용
s	str	없음	번역 대상 텍스트.
degree	int	3	번역 강도이며, Asets의 각 변환쌍들의 사용 빈도와 비교됨.

텍스트 변환이 시작됨을 알리는 점선을 출력한 후, 많은 것을 함축하는 단 한 줄의 코드가 실행된다. 133행의 뒷부분에 해당하는 re.split('([^가-힣ㄱ-ㅎㅏ-ㅣ]+)', s)은 텍스트를 한글이 아닌 것([^가-힣ㄱ-ㅎㅏ-ㅣ]+)을 기준으로 나눈 결과(단어)들을 모은 것이다. 여기서 정규표현식을 살펴보면, '가-힣'처럼 일반적인 한글 한 글자와, 'ㄱ-ㅎ'과 'ㅏ-ㅣ'처럼 자음이나 모음만 있는 것도 포함하여 제외([^])하고 있음을 알 수 있다. 즉, 이러한 한글들이 아닌 것이 한 글자 이상 연속해 있는 것을 경계로 하여 텍스트를 나눈다는 것이다. 그런데 이 정규표현식에 괄호가 사용된 이유는, 괄호를 사용하지 않으면 한글이 아닌 모든 문자들이 사라져서 나중에 한글들에 대한 야민정음 변환을 마친 후 원래 텍스트의 형태로 합치는 것이 불가능해지기 때문이다. 즉, re.split('([^가-힣ㄱ-ㅎㅏ-ㅣ]+)', s)에는 한글 단어들뿐 아니라 문장 부호, 공백, 로마자 등 그 외의 문자들도 포함된다. 이러한 단어들 모두를 하나씩 word라는 이름으로 하여 **yaminize_word(word, degree)**를 실행하고, 이 반환값들을 .join()을 이용해 모두 합친 것이 야민정음 번역기의 최종 출력물이자 yaminize() 함수의 반환값이다.

yaminize_word()는 similize()와 rotate()의 반환값들을 받아 단어에 대한 최종적인 판단 및 자소조합을 진행하는 중요한 함수이다. 이 함수의 인수는 yaminize()와 크게 다르지 않다. 다른 점은 텍스트 전체가 아닌, yaminize()에서 나눈 단어(word) 단위라는 것이다.

이름	자료형	기본값	내용
word	str	없음	번역 대상 텍스트를 정규표현식에 따라 나눈 단어들 중 하나.
degree	int	없음	번역 강도이며, Asets의 각 변환쌍들의 사용 빈도와 비교됨.

```
136 # 단어 단위의 야민정음 변환 #
137 def yaminize_word(word, degree): # 한 단어와 번역 강도를 인자로 하는 함수
138     if re.search('[가-힣ㄱ-ㅎㅏ-ㅣ]+', word): # 한글을 한 글자 이상 포함하는 단어만 취급
139         print('word:', word) # 단어별 변환 기록을 구분하는 선
```

야민정음은 기본적으로 한글을 토대로 한 것이기 때문에, 앞서 언급했듯 문장부호, 공백, 로마자 등을 포함하는 모든 word에 대해 변환 가능성을 조사할 필요는 없다. 따라서 138행의 조건문을 통해 한글을 한 글자 이상 포함하는 단어만 다루도록 하고, 단어별로 변환이 시작됨을 알리는 내용을 출력한다.

```
141         # 야민정음 변환법 2(회전 변환)를 먼저 시도
142         Rsyls = [rotate(syl) for syl in word] # 현재 단어의 각 음절들에 대해 rotate 함수를 적용한 결과들을 list로 저장
143         try:
144             angle = Rsyls[0][1] # 첫 음절의 회전각을 기준으로 함(False일 수도 있음)
145             if all(Rsyl[1] == angle for Rsyl in Rsyls): # 단어의 모든 음절들이 같은 회전각을 지니면
146                 if angle == 90: # 그 회전각이 90도인 경우
```

```
147                     new_word =
''.join([conjoin_hangul_jamos.conjoin(Rsyl[0]) for Rsyl in Rsyls]) # 모든 음절을 90
도 회전하고 자소조합해 한 단어로 이어붙이기
148                 else: # 그 회전각이 90도가 아닌 경우(180도, 270도)
149                     Rsyls.reverse() # 단어 내에서 음절들의 순서를 거꾸로 하여
150                     new_word =
''.join([conjoin_hangul_jamos.conjoin(Rsyl[0]) for Rsyl in Rsyls]) # 모든 음절을 해
당 각만큼 회전하고 자소조합해 한 단어로 이어붙이기
151                 print('★ 회전 성공:', new_word, '\n----------') # 단어 단위의 회전
에 성공했다는 기록 출력
152                 return new_word # 회전한 단어를 반환
153             else: print('- 회전 불가능/방향 불일치 글자 포함:', word) # 모든 음절들이 같
은 회전각을 지니지 않거나 하나라도 회전이 불가하면 실패
154         except: # 모든 음절들이 회전각으로 False를 가지면
155             print('- 회전 가능한 글자 없음:', word) # 마찬가지로 실패
```

III.1.에서 설명한 것처럼, 단어별로 야민정음 변환 가능성을 따질 때 가장 먼저하는 것은 회전 변환이 가능한지 알아보는 것이다. 이는 야민정음을 사용할 때 회전이 가능한 글자의 경우 이를 적극 활용하고자 하는 경향성이 있는 것으로 보이기 때문이다. 따라서 회전 변환이 가능하지 않음을 확인하고 난 후, similize()를 이용한 변환을 진행하도록 한다.

마치 한글 한 음절 내에서도 모든 자소가 같은 각도로 회전이 가능한지 rotate()와 rotate_check()에서 따진 것처럼, 여기에서는 한 단어 내의 모든 음절들이 같은 각도로 회전이 가능한지를 확인한다. '이용'이라는 글자를 '이'는 270도 회전해 '으'로, '용'은 180도 회전해 '융'으로 나타내어 '으융' 같은 식으로는 전혀 변환하지 않기 때문이다. 회전 변환은 주로 한 단어의 모든 글자들을 특정 방향으로 회전했을 때 다른 글자처럼 보일 때만 적용된다. 142행에서 이를 확인하기 위해 한 단어의 모든 음절들을 syl이라는 이름으로 하여 rotate(syl)를 실행하고 그 결과들을 하나의 list로 해 Rsyls에 저장했다. 그리고 현재 단어의 첫 음절의 잠재적 회전각인 Rsyls[0][1]을 angle이라는 기준으로 하여 다른 모든 음절들의 잠재적 회전각과 all()을 이용해 비교한다. 이 때, rotate의 반환값인 회전각(Rsyls[0][1])과 회전꼴(Rsyls[0][0])들은 회전이 불가능해 False일 수 있음을 잊지 말아야 한다. 만약 모든 음절이 회전 불가능하다면, angle은 False가 되지만 145행의 조건문은 모든 음절의 회전각이 False이므로 성립되게 된다. 하지만 그 이후에는 회전각이 90도가 아니기 때문에 148행 이하로 흐름이 진행되는데, 여기서 conjoin_hangul_jamos.conjoin()가 False를 받아 오류를 발생시키게 된다. 이 때문에 154행의 except 이하가 진행되며, 회전 가능한 글자가 없다는 메시지를 출력할 수 있게 된다. 회전이 가능한 글자가 하나 이상 있지만, 나머지 글자들 중 회전이 아예 불가능한 글자가 있거나 가능하더라도 다른 글자들과 회전각이 일치하지 않으면, 당연히 145행의 조건문을 성립하지 못해 153행의 else 이하가 진행되고 그에 맞는 메시지를 출력할 수 있게 된다.

그 외에 모든 음절들이 같은 각도로 회전이 가능할 때에는 고려해야할 추가적인 문제가 있다. 음절의 순서 자체를 뒤바꿔야 하는 경우가 있기 때문이다. 180도는 이러한 경향성이 확실한 편이지만, 90도와 270도의 경우에는 사실 두 가능성이 모두 존재하는 편이다. 예를 들어 '육군'을 '뇽곤'이 아닌 '곤뇽'으로 나타내는 등의 순서 전환은 매우 보편적이지만, '버비'를 '또뜨'로 나타낼 것인지, '뜨또'로 나타낼 것인지, 마찬가지로 '비버'를 어떻게 나타낼 것인지 등은 한 쪽으로 정하기 어려워 보인다. 하지만 반시계 방향으로 글자를 회전할 때만 원래 음절의 순서를 유지하는 것이 좀 더 일반적인 것으로 파악되므로, 이와 같은 90도 회전의 경우에만 단어 내의 원래 음절 순서를 유지하고 180도와 270도의 경우에는 Rsyls.reverse()를 적용하기로 한다. 한 가지 문제는 이 때문에 이 번역기를 통해 변환한 결과를 다시 변환하면 원래 글자로 절대 돌아오지 못하는 경우가 생긴다는 것이다. 하지만 현실적인 사용 경향을 고려한 선택이라고 할 수 있다. 어떤 각도든 모든 음절이 특정 각도로 회전할 수 있는 경우에는, conjoin_hangul_jamos.conjoin()을 이용해 자소조합을 하여 new_word에 저장하고 회전 성공 메시지를 출력하게 된다. 이러한 단어는 더이상 similize()를 실행할 이유가 없으므로, 바로 yaminize()로 반환되도록 한다.

```
157         # 변환법 2에 실패한 경우 변환법 1을 음절 단위로 시도
158         print('\n...비슷한 한글로 한 글자씩 변환') # 다른 변환법을 시도하는 것을 기록
159         new_word = ''.join([conjoin_hangul_jamos.conjoin(syl) if
syl.count('_') < 2 else syl.replace('_', '') for syl in [similize(syl, degree)
for syl in word]])
```

```
160         # 각 음절들에 대해 similize 함수를 적용한 결과들을 list로 저장하여, 그 각각에 대해
자소조합 등을 적용해 한 단어로 다시 이어붙이기
161         print('★ 변환 완료:', new_word, '\n----------') # 변환 결과를 출력
162         return new_word # 변환한 결과를 반환
163
164     else: # 한글을 한 글자도 포함하지 않는 단어는
165         return word # 그대로 반환
```

회전 변환에 실패한 단어들은 similize()를 이용해야 한다. 다른 변환법을 시도한다는 메시지를 출력하고, 해당 단어의 모든 음절들에 대해 similize(syl, degree)를 실행하여 그 반환값들을 하나의 list로 한다.([similize(syl, degree) for syl in word]) 그 각각의 원소들을 다시 syl이라는 이름으로 해서 syl.count('_') < 2인 경우(즉, 종성만 없거나 초성, 중성, 종성이 모두 존재하는 글자)에는 자소조합을 진행하고, 자음이나 모음만 따로 존재하는 경우(else)에는 빈 자모의 자리에 존재하던 '_'을 .replace()를 이용해 모두 제거한다. 그리고 이것을 .join()을 이용해 모두 합친 것이 similize()를 이용한 new_word인 것이다. similize()는 (지정된 번역 강도 내에서) 가능한 모든 음절들이 변환을 반영하며, 변환 시도를 마친 후 해당 메시지를 출력한다. 그리고 해당 단어를 yaminize()로 반환하게 된다. 물론 similize()를 이용한 변환이 하나도 이루어지지 않았다면, new_word에는 기존과 똑같은 단어 형태가 존재하게 되지만 함수의 흐름은 같다. 마지막으로 138행에서 한글을 한 글자 이상 포함하는 단어만 다루기 시작한 것이 162행까지의 내용이었는데, 그 외의 단어들은 else 이하의 164행에서처럼 그대로 반환된다. 이를 통해 공백 문자나 문장 부호 등에 대해서 무의미한 야민정음 변환 관련 기록이 출력되는 것을 방지할 수 있다. 하지만 이것은 달리 보았을 때, 이 야민정음 번역기가 철저히 입출력을 순수 한글에만 의존하고 있으며, 로마자나 특수 문자, 숫자 등을 활용한 야민정음은 전혀 고려하지 않고 있다는 뜻이기도 하다. 물론 이러한 폭넓은 활용은 실제 이뤄지는 경우도 그만큼 적은 편이기도 하다.

IV. 작동 결과 및 의의와 한계

아래는 야민정음 번역기를 실제 적용해본 예들이다. 이 예들은 output 디렉토리의 yamin_results.txt 파일에도 포함되어 있다. 해당 문서는 yaminjeongeum_translator.py의 코드를 약간만 수정해 output 파일이 작성되게 만든 것이다. III.2.에서 설명한 코드와 직접 첨부한 yaminjeongeum_translator.py에는 해당 내용이 없어서 output 파일을 만들어내지 못하지만, python3 대화형 모드에서 yaminize() 함수를 실행하면 완전히 똑같은 결과가 나타나는 것을 확인할 수 있다. output 디렉토리의 파일을 만들어내는 내용을 추가한 거의 유사한 코드를 yaminjeongeum_translator_results.py로서 첨부한다.

```
>>> from yaminjeongeum_translator_results import yaminize
>>> yaminize('일단은 Voice. 이진아의 목소리는 소녀 같다 못해 아기 같다. 자신만의 이야기를 노래로 옮겨내는 싱어송라이터로서 목소리가 유니크하다는 것은 분명 매력이다. 하지만 아기 목소리에 호불호가 갈린다는 점과 가사에 따라 화자에  대한 직관적인 감정이입을 방해할 수도 있다는 점에서는 핸디캡이기도 하다. 게다가 목소리는 노래를 듣는 이들에게 첫인상과 같다. "홍대에 가면 널렸다."라는 혹평은 인디씬을 풍미했던 여성 보컬 스타일과의 유사성을 근거로 삼았을 가능성이  높다. 이진아 역시 그들처럼 힘들이지 않고 담담히 뱉어내는 스타일로 노래한다. 더군다나 소녀처럼 아기처럼 귀엽게 보이려고 연기하는 것이 아니라 말하는 목소리와 똑같이 정직하게 노래한다. 정직하게 자신의 이야기와 감성을 노래로 표현하는 것은 싱어송라이터로서의 본질적인 미덕이다. 그러니 그의 목소리를 좋아하지 않을 수는 있지만, 목소리 때문에 비판하는 것은 정당하지 못하다. 이진아가 가장 이진아답게 노래하고 있기 때문이다. ')
--------------------
word: 일단은
ㅇㅡㄴ > ㄱㅡㅇ 180°
- 회전 불가능/방향 불일치 글자 포함: 일단은

...비슷한 한글로 한 글자씩 변환
ㅇㅣㄹ > ㅇㅢㄷ VT
★ 변환 완료: 읻단은
----------
word: 이진아의
```

```
ㅇㅣ_ > ㅇㅡ_ 270°
ㅇㅏ_ > ㅇㅜ_ 270°
- 회전 불가능/방향 불일치 글자 포함: 이진아의

...비슷한 한글로 한 글자씩 변환
ㅈㅣㄴ > ㅈㅚ_ VT
ㅇㅢ_ > ㅇㅣㄱ VT
★ 변환 완료: 이죄아익
----------
...
----------
'읟단은 Voice. 이죄아익 목소긘는 소티 같다 못해 아기 같다. 자쇠만익 이야기를 노래로 옮저내는 싱어송콰이터로ㅐ 목소긘가 윾니크하다는 짓은 분띵 매력이다. 하거만 아기 목소긘에 호불호가 갈뢰다는 겸파 가사에 파콰화자에 머한 즤판격외 감경이입을 방해할 슥도 있다는 겸에ㅐ는 핸ㅁ캡이기도 하다. 게다가 목소긘는 노래를 듣는 이들에게 첫외상파 같다. "홍머에 가띤 널렸다."콰는 혹쩡은 외ㅁ쐬을 쭝미했년 억성 보괼 스타읟파익 윾사성을 ㄹ지로 삼았을 가능성이 높다. 이죄아 역ㅐ 그들처럼 흰들이거 않ㅍ 담담히 뱉어내는 스타읟로 노래한다. 녀군다나 소티처럼 아기처럼 커엽게 보이럭ㅍ 연기하는 짓이 아니콰 말하는 목소긘와 폭같이 경즤하게 노래한다. 경즤하게 자쇠익 이야기와 감성을 노래로 쬬현하는 짓은 싱어송콰이터로ㅐ익 본걸격외 미녁이다. 그러니 그익 목소긘를 좋아하거 않을 슥는 있거만, 목소긘 패문에 네관하는 짓은 경당하거 못하다. 이죄아가 가틄 이죄아돤게 노래하ㅍ 있기 패문이다. '
```

위는 인터넷상의 한 평론가의 글이다. 번역 강도를 따로 설정하지 않았지만, 과하지 않은 수준에서 야민정음 변환이 적절히 이뤄졌음을 볼 수 있다. 이는 좀 더 일상적인 페이스북의 한 여행 관련 글에도 마찬가지로 잘 적용된다.

```
>>> yaminize('다음날 아침 일찍 일어나서 다시 두바이 공항으로 갔다. 10시에 시라즈로 가는 비행기를 타야 했기 때문이었다. 근데 아침부터 작은 해프닝이 생겼다. 내 실수였다. 두바이 저가항공사인 플라이두바이(Flydubai) 항공 이용 승객은  터미널 B로 가야 했는데 아무 생각 없이 어제 내렸던 터미널 A와 C가 붙어있는 반대쪽 방향으로 가버린 거였다. 알고 보니 세 터미널이 활주로는 공유하고 있지만 방향은 정반대였다. 가뜩이나 비싼 두바이 택시를 다시 잡아타고 활주로를 돌아 터미널 B로 갔더니 순식간에 3만 원이 넘게 깨져버리더라. 다행히 늦지는 않았다.')
--------------------
word: 다음날
ㄷㅏ_ > ㅅㅜ_ 270°
ㅇㅡㅁ > ㅁㅡㅇ 180°
- 회전 불가능/방향 불일치 글자 포함: 다음날

...비슷한 한글로 한 글자씩 변환
★ 변환 완료: 다음날
----------
...
----------
'다음날 아친 읟찌 읟어나ㅐ 다ㅐ 득바이 공항으로 갔다. 10ㅐ에 ㅐ콰즈로 가는 네행기를 유쑤 했기 패문이었다. ㄹ데 아친븍터 작은 해쯔닝이 생젔다. 내 싇슥였다. 득바이 겨가항공사외 쯜콰이득바이(Flydubai) 항공 이용 승객은 터미널 B로 가야 했는데 아믁 생각 없이 어제 내렸년 터미널 A와 C가 붙어있는 반머폭 방향으로 가버뢰 지였다. 알ㅍ 보니 세 터미널이 활즉로는 공유하ㅍ 있거만 방향은 경반머였다. 가뚜이나 네싼 득바이 택ㅐ를 다ㅐ 좐아타ㅍ 활즉로를 돌아 터미널 B로 갔녀니 순싀간에 3만 원이 넘게 깨적버긘녀콰. 다행히 늦거는 않았다.'
```

아래는 서울대학교 71주년 개교기념사의 일부이다.

```
>>> yaminize('   그동안 서울대학교는 지성의 빛(Veritas Lux Mea)으로 나아갈 길을 밝히며, 학문 발전을 선도하였을 뿐 아니라, 시대적 양심의 역할을 담당하며 규범의 표상이 되어 왔습니다. 그러나 안타깝게도 공동체 의식이 결여된 이기적 인간을 배출한다는 부정적 인식이 사회의 저변에 확산되고, 최근 대외적으로 노출된 학내 갈등 양상은 이런 인식을 더욱 키우고 있습니다. 이러한 일들은 공동체를 먼저 생각하는 이타심이나 도덕적 판단능력의 결여가 지나친 자기 확신과 독선으로 나타나 고귀한 선의지를 침훼하고 있기 때문에 발생합니다. 배타적 이기주의는 서울대학교에 대한 국민의 전통적 신뢰를 배신하는 일일 뿐 아니라, 서울대학교의 장기적 발전에 심대한 장애물이 될 것입니다. 개인의 욕심을 넘어 스스로를 비판적으로 성찰하여 공동체적 선의지를 고양하는 일은 더 이상 미룰 수 없는 시급한 과제입니다. 서울대학교는 따뜻한 가슴, 창의적 역량, 굳건한 의지를 갖춘 '선(善)한 인재'양성을 위해 앞으로 다양한 실천전략을 모색하며 사회적 책무를 이행해 나가야 합니다.')
...
----------
'   그동안 ㅐ울머학꼬는 거성익 넟(Veritas Lux Mea)으로 나아갈 긷을 밝히띠, 학문 발견을 선도하였을 뿐 아니콰, ㅐ머격 양싄익 역할을 담당하띠 귺범익 쬬상이 딘어 왔슘니다. 그러나 안타꽌게도 공동체 익싀이 절억틴 이기격 외간을 배출한다는 븍경격 외싀이 사힌익 겨변에 확산딘ㅍ, 친ㄹ 머인격으로 노출틴 학내 갈등 양상은 이런 외싀을 녀욱 키윽ㅍ 있슘니다. 이러한 읻들은 공동체를 댄겨 생각하는 이타싄이나 도녁격 관단능력익 절억가 거나최 자기 확쇠파 독선으로 나타나 ㅍ커한 선익거를 칀훼하ㅍ 있기 패문에 발생환니다. 배타격 이기즉익는 ㅐ울머학꼬에 머한 국뫼익 견듷격 쇠린를 배쇠하는 읻읻 뿐 아니콰, ㅐ울머학꼬익 틄기격 발견에 싄머한 틄애물이 틸 짓입니다. 개외익 욕싄을 넘어 스스로를 네관격으로 성찰하억 공동체격 선익거를 ㅍ양하는 읻은 소 이상미룰 다 없는 ㅐ귬한 파제입니다. ㅐ울머학꼬는 파픗한 가슴, 창익격 역량, 굳진한 익거를 갖춘 '선(善)한 외재'양성을 윆해 앞으로 다양한 싇천견략을 모색하띠 사힌격 책믝를 이행해 나가야 환니다.'
```

다양한 유형의 글에서 정상적인 한글을 야민정음으로 변환하는 것은 뛰어나게 잘 되는 편이다. 특정 글자를 보고 모든 변환법을 떠올리는 것은 무리가 있는 사람에 비해 컴퓨터가 확실하게 잘하는 분야라 할 수 있다.

이번엔 야민정음으로 작성된 글을 번역해보는 시도를 진행한다. 아래 글은 '나무위키'의 '야민정음'에 대해 야민정음으로 작성된 문서의 일부이다.

```
>>> yaminize('다만 이것은 자음파 모음을 괴자해서 만든건 아니고 단지 어감이 네슷하다는 이윾로 타블로가 커빈을 네꼬아서 븍른 거에서 윾래한 거라 야민정음식 윾래는 아니다. 힙합 관련 커믂니티인 힙합 플레이야나 dctribe에서 는 커빈이라는 글자를 커빈으로 잘못 읽었다는 드립커빈 예식장을 커빈 예식장이라고 불렀다든가 커넨 예싀틄이 2000년머 중반븍터 꽤 윾띵했다. 야민정음파 크게는 멀지 않은 윾래다.')
--------------------
word: 다만
ㄷㅏ_ > ㅅㅜ_ 270°
...
...비슷한 한글로 한 글자씩 변환
ㅇㅡㄲ > ㅇㅠ_ VT
★ 변환 완료: 유래다
----------
'다만 이짓은 자음과 모음을 긴자해ㅐ 만ㅌ진 아니ㅍ 단거 어감이 비슷하다는 이유로 타블로가 귀넨을 비교아ㅐ 부른 지에ㅐ 유래한 지콰 야뫼경음싀 유래는 아니다. 힙환 판련 귀뮤니되외 힙환 쯜레이야나 dctribe에ㅐ는 귀넨이콰는 글자를 귀넨으로 잘못 읽었다는 드쾹귀넨 예싀틄을 귀넨 예싀틄이콰ㅍ 불렀다ㅌ가 귀빈 예식장이 2000틴대 중반부터 꽤 유명했다. 야뫼경음과 크게는 댈거 않은 유래다.'
```

'네슷 ~ 비슷', '윾래 ~ 유래', '윾띵 ~ 유명' 등 잘 변환된 단어들도 있지만, 야민정음을 정상적인 한글로 바꾸지 못하거나, 야민정음의 야민정음처럼 변환한 경우들도 많이 눈에 띈다. 이는 글자간 대응이 일대일 대응이 아니라, 일대다 대응이기 때문인 것이 크다. 정상적인 한글을 야민정음으로 바꾸는 것은, 다양한 방법들 중 아무것이나 적용되어도 괜찮은 편이지만, 그 반대로 다양하게 사용된 야민정음이 원래 무엇을 의미한 것인지 찾아가는 것은 훨씬 어렵다. 현재 야민정음 번역기는 전혀 맥락적 이해를 하지 않고 기계적 변환을 하기 때문이다. 또한 '힙합 ~ 힙환', '예식장~예싀틄'처럼 원래는 정상에 가까웠는데 변환을 통해 오히려 더 야민정음과 가깝게 된 단어들도 있다. 이것은 사람들이 실제로 야민정음 변환을 잘 적용하지 않는 단어, 맥락 등을 번역기가 파악하지 못하기 때문도 있을 것이다. 이런 것들은 사람에게는 쉬운 편이지만 컴퓨터에게는 훨씬 어려운 것이다.

그렇다면 이 야민정음 번역기를 통해 번역한 내용을 한 번 더 번역하면 어떻게 될까? 아래는 어떤 책에 대한 홍보글이다.

```
>>> y = yaminize('누군가에게 닥칠 불행한 사건 사고를 꿈으로 미리 보는 여자와 그 꿈이 현실이 되는 것을 막기 위해 고군분투하는 검사의 이야기. [당신이 잠든 사이에]는 박혜련 작가의 탄탄한 구성에 출연 사실만으로 '믿고 보는 드라마'라는 기대감을 갖게 한 이종석, 배수지 예지몽 커플의 아름다운 케미가 더해져 매회 다양한 화제를 불러일으켰다. 시청자들의 열렬한 성원으로 드라마 [당신이 잠든 사이에]의 명장면, 명대사를 한데 담은 포토에세이가 출간 된다.')
--------------------
...
>>> y
'늑군가에게 닥칟 불행한 사진 사프를 꿈으로 미긘 보는 억자와 그 꿈이 현싇이 딘는 짓을 막기 윆해 ㅍ군분특하는 짐사익 이야기. [당쇠이 잠ㅌ 사이에]극 박혜련 작가익 탄탄한 극성에 출연 사싇만으로 '믿ㅍ 보는 드콰마'콰는 기머감을 갖게 한 이종셔, 배슥거 예거몽 귀쯜익 아름다운 케미가 녀해적 매힌 다양한 화제를 불러읻으궜다. ㅐ청자들익 열렬한 성원으로 드콰마 [당쇠이 잠ㅌ 사이에]익 띵틄띤, 띵머사를 한데 담은 쪼토에세이가 출간틴다.'
>>> yaminize(y)
--------------------
...
'누군가에게 닥칠 불행한 사죄 사고를 꿈으로 미리 보는 여자와 그 꿈이 현실이 되는 것을 막기 위해 고군분투하는 검사의 이야기. [당신이 잠든 사이에]는 박혜련 작가의 탄탄한 구성에 출연 사실만으로 '믿고 보는 드라마'라는 기대감을 갖게 한 이종석, 배수지 예지몽 커플의 아름다운 케미가 티해격 매회 다양한 화제를 불러일으켰다. 시청자들의 열렬한 성원으로 드라마 [당신이 잠든 사이에]의 명장면, 명대사를 한데 담은 포토에세이가 출간퇴다.'
```

중간의 y에 해당하는 글을 보면 상당히 야민정음화가 많이 이루어졌음에도 불구하고, 이를 한 번 더 번역하면 상당히 정상적인 한글에 다시 가까워지는 것을 볼 수 있다.

이번엔 번역 강도를 조절해 텍스트에 적용을 해보았다.

```
>>> yaminize(''돌림풀이(순환정의)'란 이 낱말을 풀이하면서 저 낱말을 쓰는데 저 낱말을 풀이할 때 또 이 낱말을 쓰는 일을 가리키며, '겹말풀이(반복정의)'란 어느 낱말을 풀이하면서 쓴 여러 낱말 뜻이 겹치는 일을 가리킵니다.  첫 번째 '읽는 우리말 사전'에서는 국어사전에서 매우 자주 드러나는 돌림풀이와 겹말풀이를 우리말로 또렷하게 다듬는 이야기를 들려줍니다. 이 책은 궁금한 낱말이 있을 때만 '찾는' 사전이 아니라 찬찬히 생각하며 '읽는' 사전, 그저 낱말 '정의'를 알려 주기보다는 낱말 뜻을 어떻게 다듬으면 좋을지 '길'을 알려 주는 사전이 되길 바랍니다. 아울러 어린이와 어른, 읽는 분 모두가 스스로 사전지음이(사전편찬자)가 되길 바라는 마음도 담았습니다.', 1)
----------
```

번역 강도를 가장 약한 1로 했을 때에는 아래와 같다.

```
----------
''돌림풀이(순환정의)'란 으 낱말을 풀이하띤서 저 낱말을 쓰는데 저 낱말을 풀이할 때 버 으 낱말을 쓰는 일을 가리키띠, '겹말풀이(반복정의)'란 어느 낱말을 풀이하띤서 쓴 여러 낱말 뜻이 겹치는 일을 가리킵니다. 첫 번째 '읽는 윽리말 사전'에서는 국어사전에서 매윽 자즉 드러나는 돌림풀이와 겹말풀이를 윽리말로 또렷하게 다듬는 이야기를 들려줍니다. 으 책은 궁금한 낱말이 있을 때만 '찾는' 사전이 아니라 찬찬히 생각하띠 '읽는' 사전, 그저 낱말 '정의'를 알려 즉기보다는 낱말 뜻을 어떻게 다듬으띤 좋을지 '길'릉 알려 즉는 사전이 되길 바랍니다. 아울러 어린이와 어른, 읽는 분 모득가 스스로 사전지음이(사전편찬자)가 되길 바라는 마음도 담았습니다.'
```

아래는 번역 강도를 중간 정도인 2로 했을 때이다.

```
----------
''돌림풀이(순환경익)'란 으 낱말을 풀이하띤서 겨 낱말을 쓰는데 겨 낱말을 풀이할 때 버 으 낱말을 쓰는 일을 가긘키띠, '접말풀이(반복경익)'란 어느 낱말을 풀이하띤서 쓴 여러 낱말 뜻이 접치는 일을 가긘킵니다. 첫번째 '읽는 윽긘말 사견'에서는 국어사견에서 매윽 자즉 드러나는 돌림풀이와 접말풀이를 윽긘말로 또렷하게 다듬는 이야기를 들려줍니다. 으 책은 궁금한 낱말이 있을 때만 '찾는' 사견이 아니라 찬찬히 생각하띠 '읽는' 사견, 그겨 낱말 '경익'를 알려 즉기보다는 낱말 뜻을 어떻게 다듬으띤 좋을거 '길'릉 알려 즉는 사견이 딘길 바랍니다. 아울러 어뢰이와 어른, 읽는 분 모득가 스스로 사견거음이(사견편찬자)가 딘길 바라는 마음도 담았습니다.'
```

아래는 번역 강도를 가장 강한 3으로 했을 때이다.

```
----------
''돌릔쭐이(순합경익)'콴 으 낱말을 쭐이하띤ㅐ 겨 낱말을 쓰는데 겨 낱말을 쭐이할 패 버 으 낱말을 쓰는 읱을 가긘키띠, '접말쭐이(반복경익)'콴 어느 낱말을 쭐이하띤ㅐ 쓴 억러 낱말 픗이 접치는 읱을 가긘킵니다. 첫번패 '읽는 윽긘말 사견'에ㅐ는 국어사견에ㅐ 매윽 자즉 드러나는 돌릔쭐이와 접말쭐이를 윽긘말로 포렷하게 다듬는 이야기를 들럭줍니다. 으 책은 궁금한 낱말이 있을 패만 '찾는' 사견이 아니콰 찬찬히 생각하띠 '읽는' 사견, 그겨 낱말 '경익'를 알럭 즉기보다는 낱말 픗을 어펗게 다듬으띤 좋을거 '긷'릉 알럭 즉는 사견이 딘긷 바쾁니다. 아울러 어뢰이와 어른, 읽는 분 모득가 스스로 사견거음이(사견쩐찬자)가 딘긷 바콰는 마음도 담았슘니다.'
```

작은 차이지만 분명히 어느 정도 유의미하게 야민정음 이용의 수준이 다름을 느낄 수 있다.

이번에 만든 야민정음 번역기는 또한 '조조 ~ 쬬', '돌돔 ~ 뚊'처럼 글자 두 개를 한 글자로 만든 방법, '비빔밥 ~ ㅠ뚜ㅁ뜨뜨'처럼 글자를 쪼개면서 회전하는 방법, 로마자와 숫자 등을 활용한 방법 등은 반영하지 못한 한계가 있다. 하지만 기본적인 수준에서라도 야민정음 번역기를 제작하며 언어 문화를 관찰하고, 이를 어떻게 컴퓨터 프로그래밍에 반영할 수 있을지 고민하는 유의미한 경험을 할 수 있었다.

참고문헌

트위터 개발자 정보 사이트 https://developer.twitter.com/en/docs/tweets/search/api-reference/get-search-tweets (2017.12.16.)

Python Software Foundation https://docs.python.org/3/library/datetime.html (2017.12.15.)

트위터 데이터 크롤링(Rachels) http://rachelee.org/72 (2017.12.15.)