낱말 ▪ 패러다임 형태이론에 입각한 영어동사 굴절 해석 프로그램의 구현

노 용 균
충남대학교

No, Yongkyoon. 1998. **Implementing an Inflection Analyzer Program for English Verbs in a Word-and-Paradigm Morphology**. *Language and Information* 2.2, 121-154. The morphological analyzer is expected to tell attested word forms from imaginable yet unattested ones. An account of the inflectional morphology of English verbs is given in the framework of Word-and-Paradigm morphology, developed mainly by Matthews (1972, 1974, 1991) and further by Aronoff (1994) and Zwicky (1985, 1988), which is free of overrecognition. Thirteen inflectional classes are identified according to the patterns each of them exhibits in filling the slots in the paradigm. Peculiarity in orthography is also considered in assigning each verb lexeme to a class. Modules of a C program which gives associated morphosyntactic properties to all and only attested verb forms are written so that details of this framework can be evaluated explicitly. This program is shown to be superior to existing programs in economy and in the generality it achieves. (**Chungnam National University**)

1. 서언

영어의 낱말 부류들 중에서 굴곡을 하는 것들은 동사, 명사, 형용사, 부사다. 명사는 두 가지 수를 나타내는 데에, 그리고 형용사와 부사는 세 가지 급을 나타내는 데에 굴곡의 통사적 의의가 있다.

동사는 모두 열 다섯 개의 굴절꼴을 가지므로 영어의 다른 낱말 부류들보다 굴절의 복잡도가 더 높다.[1]

비록 낱말 한 개당 수천 내지 수만 개의 굴절꼴을 인정해야 하는 언어들을 고려하면 영어가 확연히 고립어적(isolating) 성격을 갖는다는 것이 명백하나 영어의 동사들의 꼴을 전적으로 자동적으로 해석하는 심리적 기제의 복잡성은 굴곡의 도가 높은(highly inflecting) 언어들의 낱말꼴들을 해석하는 것보다 현저히 낮지는 않다.[2] 접사 첨가의 누적이 없다는 점이 영어의 굴곡이 교착어나 굴절어의 굴곡과 구별되는 특징이지만 (a)동일한 낱말이 둘 이상의 어간으로 다양한 꼴들을 만든다는 점과 (b)동일한 형태통사적 속성들(morphosyntactic properties)이 낱말에 따라 상이한 방식으로 실현된다는 점이 모든 언어에서처럼 영어에서도 관찰되기 때문이다.

이 글에서는 Matthews(1972, 1991), Zwicky(1985, 1988), Aronoff(1994) 등의 낱말·패러다임 형태 이론(Word and Paradigm morphology)의 입장을 따라 영어 동사의 굴절을 기술하고자 한다. 또 하나의 목표는 이 기술의 상세한 부분들을 용이하게 검증할 수 있게 하기 위하여, 널리 쓰이고 있는 컴퓨터 프로그램 언어인 C로 동사의 굴절꼴로부터 그와 연계된 형태통사적 속성들의 집합을 찾아내는 구체적인 절차들을 구현하는 것이다.

부록에 그 원천 꼬치(source code)의 핵심 부분이 제시되어 있는 이 프로그램의 전형적인 실행내역은 아래 (1)과 같다.

```
(1) Enter your word-form: reply
        BASE of {reply}
```

---

[1]영어 동사의 굴절꼴은 정동사(finite verb) 꼴 열 두 개와 분사 두 개, 그리고 원형이며, 정동사의 꼴이 열 두 개인 것은 두 가지 시제에서 각각 여섯 개의 꼴이 있기 때문이다. 이 여섯 개는 세 가지 인칭과 두 가지 수의 조합에 각각 하나씩이다. 실제로 이 열 다섯 가지 꼴이 음운적으로 모두 구별되는 동사는 하나도 없으며 최소 네 개에서 최다 여덟 개의 상이한 꼴들이 동사 어휘소 하나에 연결된다.

[2]어떤 전통에서는 동사의 활용을 굴절(conjugation), 명사의 활용을 곡용(declension)이라고 하기도 한다. 굴절과 곡용을 통칭하기 위해 "굴곡"이라는 용어를 쓰겠는데 이는 대체로 더 넓은 의미로 쓰이는 "굴절"과 동의어다.

```
        1sg PRES of {reply}
        2sg PRES of {reply}
        1pl PRES of {reply}
        2pl PRES of {reply}
        3pl PRES of {reply}
    Enter your word-form: replies
        3sg PRES of {reply}
    Enter your word-form: repli
    Enter your word-form: replie
    Enter your word-form: replying
        PRES PARTICIPLE of {reply}
```

영어의 굴절꼴 해석 기제가 고려해야 할 점은 두 가지이다. 하나는 소위 불규칙 굴절(irregular inflection)이라고 불리는 현상이고 또 하나는 정서법상의 특이성이다. 이 두 가지를 고려하지 않는 파서(parser)는 과수용(over recognition)의 오류를 범할 수밖에 없어서 실재하지 않는 가상적인 낱말꼴들에 대해서 (2)에서와 같이 멀쩡한 해석들을 내놓을 것이다.

(2)
```
Enter your word-form: wents
    3sg PRES of {went}
Enter your word-form: wenting
    PRES PARTICIPLE of {went}
Enter your word-form: cuting
    PRES PARTICIPLE of {cut}
Enter your word-form: writees
    3sg PRES of {write}
```

그러나 이런 해석들을 내어 놓는 파서는 인간인 영어 사용자들의 수행과 큰 차이가 있을 뿐 아니라 그 효용성이 지극히 낮으므로 구

현할 가치가 없는 파서다. 그런 엉성한 파서는 구현하기가 더 쉽기도 하다. 조금이라도 더 도전적인 문제는 실재하는 모든 낱말꼴을 옳게 해석해 내고 실재하지 않는 모든 가상적인 낱말꼴들을 배척하는 파서를 구현하는 것이다. (1)의 실행예에서 "repli"와 "replie"에 대해서 아무런 해석을 내어놓지 않듯이 (2)의 가상적인 낱말꼴들에 대해서도 전혀 해석을 내어놓지 않는 파서가 우리의 관심사다.

2. 낱말·패러다임 형태론

언어 형식과 의미 사이의 연결(pairing) 방식 중에서 가장 단순한 것은 물론 일대 일의 결합이다. 동일한 형식이 한 가지 의미하고만 연결되고 동일한 의미는 한 가지 형식하고만 연결된다면 언어의 탐구는 지극히 용이할 것이다. 비록 형식과 의미가 일대일로 결합하는 예가 그렇지 않은 예보다 더 빈번하기는 하지만 타당성 있는 언어이론이라면 마땅히 덜 빈번하더라도 실재하는 모든 결합방식을 기술하는 데에 적절한 것이라야 한다.

1970년대 이래의 유일 기저 표시체 가설(Unique Underlying Representation Hypothesis)과 "통사부문으로부터 구별되지 않는 형태부문"(Morphology as syntax)가설을 채택하는 연구노선의 문제점은 언어 형식과 의미 사이의 비전형적인 연결 방식들을 기술하는 데에 취약하다는 것이다. Kenstowicz and Kisserberth(1979), Selkirk(1982), Lieber(1992) 등과 같이 생성 음운론의 주류라 할 수 있는 이 노선을 추구하는 학자들과는 딴판으로 Matthews(1972, 1974, 1992)는 일관되게 유일 기저 표시체 가설을 배척하고 동일한 어휘소가 둘 이상의 어간을 가질 수 있다는 가설을 바탕으로 하여 형식과 의미 사이의 비전형적 연결방식들을 기술하는 데에 초점을 맞춘 낱말·패러다임 형태이론을 정교화했다. 이 노력은 미국의 생성음운론 연구그룹의 초기 구성원들 중에서 다수의 지지를 확보하게 되었다. Zwicky(1985, 1988), Anderson(1992), Aronoff(1994), Beard(1987) 등.

이 이론에 따르면 굴곡하는 낱말들은 굴곡부류(inflectional classes)를 형성하는데, 낱말 각각이 어느 굴곡부류에 속하는지는 전적으로 임의적인 것이므로 사전에 제공되어야 하는 순수한 형태론적 성질이다. 동일한 굴곡부류에 속하는 낱말들은 동일한 방식으로 굴곡하고 상이한 부류에 속하는 낱말들은 굴곡 패러다임의 어떤 항을 여타 부류에 속하는 낱말들과는 다른 방식으로 채워 넣는다. 형태 통사적 속성들과 언어 형식들 사이의 연결을 기술하는 규칙들은 실현규칙(rules of realization)이라고 불리는데 이 규칙들은 물론 사전에서 가져 온 굴곡부류에 언급할 수 있다. 굴곡부류 뿐 아니라 형태통사적 속성, 어간, 음운적 조작을 규정하는 부분이다. 아래의 R12와 R23은 영어 동사의 실현규칙의 예다.

R12. 굴절 부류 X에 속하는 낱말에 [PAST, +; PERSON, 3; NUMBER, PLURAL]이 연결된 결합체는 이 낱말의 제1어간에 접미사 /d/를 붙임으로써 실현된다.

R23. 굴절 부류 C에 속하는 낱말에 [PARTICIPLE, PAST]가 연결된 결합체는 이 낱말의 제 2 어간으로 실현된다.

R12는 소위 규칙 동사의 3인칭 복수 과거형을 실현하는 규칙이다. 규칙 동사 뿐 아니라 BE를 제외한 모든 동사가 수와 인칭에 따라 구별되지 않는 단일한 과거형을 갖는 것이 사실이지만 이렇게 수와 인칭을 명시적으로 규정하는 것은 체계의 정합성 때문이다. 소위 규칙 동사의 과거형 접미사가 보이는 삼분적 변동(/d/~/ɪd/~/t/)은 음운적 환경에 의해 전적으로 자동적으로 결정되는 만큼, 별도의 실현규칙이 맡을 현상은 아니다. R23은 불규칙 동사들 중에서 과거 분사형에 하동의 접사가 안들어 있고 동사 굴절 패러다임의 모든 항을 고려할 때 어간이 두 개 이상 발견되는 동사들 중의 일부를 부류 C에 속한다고 전제한다. 어간의 수가 둘 이상이고 과거 분사형에 접사가 안들어 있는 동사들은 COME(/kʌm/과 /kem/) 부류와 THINK 부류, SING 부류가 있는데, 이 것들이 갖는 두 개

이상의 어간들 중에서 어느 어간을 제 2 어간이라고 부르느냐에 따라 C는 이들 중 어느 하나를 가리킬 수 있다.

위와 같이 실현규칙들이 낱말의 굴곡 부류와 특정한 어간에 언급하면 언어 형식과 의미(또는 통사적 기능) 사이의 다양한 관련양상들을 기술할 수 있음에 틀림 없다. 그러나 규칙이 언급하는 요소들이 많으므로 규칙의 수 또한 많다. 따라서 이 이론에 따르는 개별 언어 형태 부문의 기술에서는 경제성의 달성이 우선 고려되어야 하겠다.

정확도에서 만족할 만한 이 모델은 그렇지 못한 모델들에 비해서 물론 경제성이 떨어진다. 그러나 경제성에 있어서의 결손은 두 가지 이유로 그렇게 크지 않다. 첫째, 낱말의 어간 수는 매우 적다. 모든 규칙 동사는 어간을 하나밖에 안갖는다. 불규칙 동사들도 대개 어간 두 개만 갖고 어간의 수가 세 개 이상인 동사의 수는 극히 적다. 영어의 경우에 어간의 수가 네개 이상인 동사는 둘밖에 없다. LIE와 BE가 그들이다.[3]

낱말·패러다임 형태론이 경제성 측면에서 크게 불리하지 않은 두번째 이유는 굴곡 부류의 수 또한 매우 적다는 점이다. 이것은 결국 어떤 형태통사적 속성들의 집합이 실현되는 방식의 수가 둘 이상이라 하더라도 그 수가 논리적으로 가능한 범위보다 훨씬 더 좁은 범위로 국한된다는 것을 의미한다. 예를 들어서, 동사에 [PARTICIPLE, PRES]라는 속성을 실현하는 방식으로 제 2어간에 접사 /ing/를 붙이는 것을 얼마든지 상상할 수 있으나, 그런 동사는 없다. (제2 어간은 [VFORM, BASE]를 실현하는 데에 쓰이는 어간과 다른 어간이라고 정의하자.) 그리고 동사에 [PERSON, 3; NUMBER, SINGULAR; PRES]라는 속성들의 집합을 실현하는 데에 제 2어간과 /z/첨가를 쓰는 동사는 있지만 (does와 do의 모음 차이 유념), 제 1어간에 대한 무조작(identity mapping)이나 제2 어

[3] /lɑɪ/는 동음이의어 둘의 음운표시체다. 이 둘 중에서 하나인 LIE2는 글로 쓰는 영어에서 어간이 네 개인데, 말로 하는 영어에서는 어간이 둘에 지나지 않는다. lie, ly, lay, lai와 /lɑɪ/, /leɪ/를 비교해 보라. 이 논문의 이전 판 원고에서 LIE2를 고려하지 않은 오류를 지적해 준 익명의 심사위원께 감사 드린다.

간에 대한 무조작을 쓰는 동사는 하나도 없다.

요컨대 낱말·패러다임 형태 이론은 정확도가 탁월하며, 결손이 우려되는 경제성 측면 또한 언어의 굴곡 부문이 스스로 갖는 경제성 때문에 걱정할 것이 없는, 굴곡 부문 기술의 훌륭한 모델이다.

3. 철자상의 일탈

글로 쓴 언어는 귀로 듣는 언어에 종속되어 있고 언어의 중요한 성질들은 귀로 듣는 언어에 다 들어 있다. 이것을 인정한다고 하더라도 시각이 청각과 함께 언어 생활에 중요한 절반을 담당하는 만큼 글로 쓴 언어의 독해 과정도 학문적 탐구의 대상이 되어야 한다. 소위 언어 능력에 치중하는 연구에서와는 달리 언어 수행 측면이 주관심사인 심리언어학이나 전산언어학에서는 독해 문제가 유달리 큰 관심을 끌어 왔다. 일례로 Feldman(1995)에 실린 논문 열 일곱 편 중에서 순전히 귀로 듣는 언어만 다루는 논문은 두 편밖에 없다. 이런 맥락에서, Fowler and Liberman(1995: 165)은 "형태음운적 표시체들(morphophonemic representations)은 글을 배운 후에야 비로소 더 충실하게 명세된다고 보아야 할 가능성"이 있다고 결론 짓는다.

영어 동사의 굴곡과 관련해서 철자상의 엉뚱함은 전적으로 규칙적인 것들과 낱말이 무엇이냐에 따라 다르기 때문에 예측 불가능한 것들로 나뉜다. 이 둘 중에서 예측 불가능한 것들은 별도의 굴곡 부류들에 문제의 낱말들을 할당함으로써 해결할 수밖에 없다. 예를 들어서 DENY의 3인칭 단수 현재형은 *denys가 아닌 denies이고 PRAY의 같은 꼴은 prays이지 *praies가 아니다. 어간 끝의 "y"와 "ie" 사이의 변동에 아무런 규칙성이 발견되지 않는다면 DENY류의 동사들은 PRAY류의 동사들과 다른 굴곡부류에 소속시켜야 한다. 그러나 이 변동에는 어간 끝에서 두 번째 글자가 모음자냐 아니냐 하는 조건이 있으므로 이 변동은 자동적인 것이다. (employs, delays, *emploies, *delaies, *occupys, occuppies, *defys, defies.)

귀로 돋는 언어에서의 자동적 음운과정("autphonological processes" à la Zwicky(1988))에 비유 될 수 있다.

어간 끝의 "y"와 "ie" 사이의 변동이 순전히 음운적 환경(여기에서는 문자 환경)에만 민감하므로 "deny"를 제 1어간, "denie"를 제 2 어간으로 갖고 3인칭 단수 현재형과 모든 과거형, 그리고 과거분사형에 제 2어간을 쓰는 굴곡 부류에 이 어휘소 DENY를 소속시킬 필요가 없다. PRAY와 동일한 부류에 넣으면 된다.

그러나 어간의 마지막 글자가 "y"인 동사들 중에는 위의 DENY와 PRAY처럼 "ie"와 "y" 사이의 변동을 보이지 않는 것들이 있다. SAY는 3인칭 단수 현재형을 구성하는 방식이 PRAY와 동일하나 과거형들과 과거 분사형을 구성하는 방식이 전혀 다르다. *sayed, said, prayed, *praid. PRAY가 속하는 부류에 SAY를 소속시키면 이 차이를 포착하기 위해 매우 상세한 문자 연쇄체 정보에 언급해야 하는데 이 것은 어휘소 그 자체에 언급하는 것과 바히 다르지 않다. delayed, *delaid, *layed, laid. 따라서 SAY, PAY, LAY는 단독으로 규칙 동사 부류와 다른 굴곡 부류를 이룬다고 보아야 한다.

어간 마지막의 "y"와 "ie" 사이의 변동이 갖는 복잡성은 위의 두 부류에 속하지 않으면서도 이 변동을 보이는 낱말들의 무리가 있어서 가중된다. PRAY와 SAY는 모두 원형과 현재형들에서 마지막 글자가 "y"인 어간을 쓰나 (그리고 DENY는 3인칭 단수 현재형에서는 자동변화를 입으나), TIE와 LIE는 이 꼴들에서 마지막 글자 연쇄체가 "ie"인 어간을 쓴다. 이들은 현재분사형에서만 마지막 글자가 "y"인 어간을 쓴다.

철자상의 일탈에는 이 밖에도 어간 마지막 자음자의 반복, 어간 끝의 "c"와 "ck" 사이의 변동, 어간 마지막 "e"자의 삭제가 있다. 이것들 중에서 "c"와 "ck" 사이의 변동은 생산성이 낮으므로 별도의 굴절 부류 설정으로 처리하는 것이 자동음운과정처럼 처리하는 것보다 더 자연스럽다. 나머지 변동들은 생산성이 높고 문자 환경에만 민감하다.

4. 사전의 구성

이 프로그램을 위한 사전의 구성은 경제성을 최대한 존중하는 데에 촛점이 맞춰졌다. 지면의 제약 때문에 모든 동사를 제시할 수 없으므로 모든 굴곡 부류를 망라하는 것을 중시한다.[4] 그리고 규칙 동사는 어간만 제시함으로써 사전의 확장을 최대한 용이하게 하도록 배려했다.

각각의 동재항은 "어간 (굴절 부류(어간 번호 어휘소))"의 조직을 갖는다. 어간과 굴절 부류만 제시된 경우에는 어휘소의 이름이 주어진 어간으로부터 그냥 도출되고 어간 번호는 1번인 경우다. 동재항들은 알파베트 순으로 정돈되었다. 이 사전은 "verb.dic"이라는 이름으로 별도의 파일로 존재하며 부록에 제시된 프로그램의 종주함수에서 불려들여 이용된다.

```
am L2 {be}          are L3 {be}          be L
break D             broke D2 {break}     came H2 {come}
come H              cut F                delete
deny                drive E              drove E2 {drive}
```

[4]영어 동사의 굴곡부류는 모두 열 세 개다.

부류	어휘소	제1 어간	제2 어간	여타 어간
A	{mimic}	mimic	mimick	없음
B	{tie}	tie	ty	없음
C	{find}	find	found	없음
D	{get}	get	got	없음
E	{know}	know	knew	없음
F	{put}	put	없음	없음
G	{sing}	sing	sang	sung
H	{come}	come	came	없음
I	{have}	have	has	had
J	{go}	go	went	gone
K	{lie2}	lie	ly	lay, lai
L	{be}	be	am	are, is, was, were
X	{delete}	delete	없음	없음

find C
fled C2 {flee}
flee C
found
found C2 {find}
gave E2 {give}
get D
give E
go J
gone J3 {go}
got D2 {get}
had I3 {have}
has I2 {have}
have I
is L4 {be}
key
kiss
knew E2 {know}
know E
laid C2 {lay}
lain K4 {lie2}
lay C
lay K3 {lie2}
lead C
leave C
led C2 {lead}
left C2 {leave}
lie B1 {lie1}
lie K1 {lie2}
ly B2 {lie1}
ly K2 {lie2}
mimic A
mimick A2 {mimic}
occur
offer
paid C2 {pay}
pat
pay C
pick
pray
program
push
put F
putt
reach
read C
read C2 {read}
reply
said C2 {say}
sang G2 {sing}
saw E2 {see}
say C
see E
set F
sing G
span
spot
sung G3 {sing}
take E
think C
thought C2 {think}
tie B
took E2{take}
tread D
trod D2 {tread}
ty B2 {tie}
was L5 {be}
went J2 {go}
were L6 {be}
work
write E
wrote E2 {write}

5. 프로그램의 구현

이 프로그램은 종주 함수 (main function)를 포함해서 모두 열 한 개의 사용자 정의 함수(user defined functions)로 이루어진다. 이

함수들 각각의 기능을 보이면 (3)과 같다.

(3) 함수 각각의 기능

main 함수 readlines의 도움으로 사전을 읽어들인다. 글쇠판으로부터 해석 대상 낱말꼴을 받아들인다. 이 낱말꼴을 함수 analyze에 넘긴다. 함수 analyze가 내어놓는 해석을 화면에 제시한다. "nomore"라는 연쇄체가 들어올 때까지 이 일을 반복한다.

readlines 함수 getline을 반복적으로 불러 쓴다. 문자 연쇄체의 배열대인 전역변인(global variable) dictionary의 각 셀을 함수 getline이 읽은 행으로 채운다.

getline 전역변인인 파일 지시체 dict로부터 한 행을 읽어들여서 문자의 배열대에 넣는다. 그 행에 들어있는 글자 수를 보고한다.

ends_in_cvo 문자연쇄체를 받아서 그 연쇄체의 끝 석 자가 차례로 자음-모음-순정자음을 나타내는지 검토하고 그러면 1을 안그러면 0을 반환한다.

unduplicate 문자연쇄체를 받아서 그 연쇄체의 끝 두 자가 서로 같고 끝에서 네번째 글자부터 석 자가 자음-모음-순정자음을 나타내는지 검토하고 그러면 본래의 연쇄체에서 끝 글자를 지운 결과 얻는 연쇄체를 반환하며 안 그러면 영연쇄체(null string)을 반환한다.

tt2te 문자연쇄체를 받아서 그 연쇄체의 끝 두 자가 "tt" 또는 "dd" 면 각각 "te" 및 "de"로 바꿔 넣은 연쇄체를 반환한다.

ie2y 문자연쇄체를 받아서 그 연쇄체의 끝 두 자가 "ie"면 "y"로 바꿔 넣은 연쇄체를 반환한다.

lexemerep 문자연쇄체를 받아서 그 연쇄체의 중괄호쌍 안에 있는 문자들을 반환한다. 중괄호쌍이 없으면 그 문자연쇄체의 첫 공백문자 앞의 연쇄체에 중괄호를 입혀 반환한다.

stemsearch 문자연쇄체의 배열대와 문자연쇄체를 받아서 후

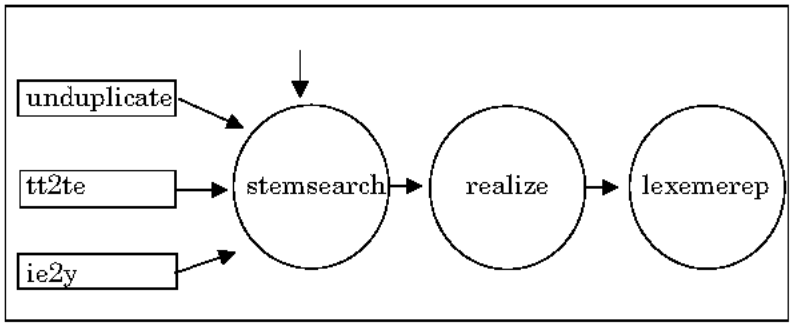
자가 그 배열대에서 발견되면 그 발견되는 셀들의 번호들을 반환한다.

realize 문자 연쇄체 세 개를 받아서 이들이 일련의 조건을 충족하면 1에서 15 사이의 하나 이상의 정수를 돌려 주고 조건을 하나라도 어기면 0을 돌려 준다.

analyze 문자연쇄체를 받아서 그것을 양분하여 앞 부분을 stemsearch에 넘겨 준다. 뒷쪽 절반이 특정한 연쇄체일 때에는 앞쪽 절반에 변화를 가하여 다시 stemsearch에 넘겨준다. stemsearch가 배열대의 셀번호들을 돌려 주면 차례로 각 셀에 들어 있는 문자연쇄체를 위의 앞쪽 절반과 함께 realize에 넘겨 준다. realize가 정수의 연쇄체를 돌려 주면 각 정수를 "1sg PRES of" 따위로 대체하여 화면에 내 놓는다. lexemerep가 불려 쓰인다.

이 함수들 중에서 readlines와 getline은 Kernighan and Ritchie(1988)에서 그대로 가져 온 것이며 나머지 함수들은 필자가 작성했다. 종주함수가 불러서 쓰는 사용자 정의 함수 둘 중에서 analyze가 형태 해석 과제의 대부분을 맡는데, 이 함수의 내부 구성은 (4)와 같다. 종주 함수, analyze, realize의 원천 꼬치(source code)를 부록에 제시한다.

(4) 함수 analyze의 구성



▭ 는 어간에 변형을 가하는 과정들, ⟶ 는 정보 전달의 방향임

프로그램의 구조를 이해하기 위해 "putting"과 "occured"의 해석 과정을 단계별로 추적해 보자. 종주함수는 "putting"을 함수 analyze에 첫번째 논항으로 넘겨 준다. 함수 analyze는 이 연쇄체를 처음에는 "pu"와 "tting"로 나누고 뒤의 연쇄체의 길이가 5이므로 (즉 4 이상이므로) 가능한 굴절꼴일 수 없다고 간주하고 다시 "put"와 "ting"로 분석대상 연쇄체를 가른다. 마찬가지로 이 것 또한 접미사의 길이가 너무 길기 때문에 일찌감치 포기된다. 다시 분석대상을 "putt"와 "ing"로 가르고, 이번에는 "putt"를 함수 stemsearch에 넘긴다. (이 때 사전의 주소, 사전에 포함된 동재항의 수, 그리고 답을 돌려 줄 곳이 "putt"와 함께 넘겨진다.) 함수 stemsearch는 사전에서 "putt"를 어간으로 갖는 동재항을 다 찾아서 그 주소(제4절의 사전에 따르면 56)를 돌려 준다. 경우에 따라서는 이 동재항이 둘 이상일 수도 있기 때문에 정수 연쇄체에 대한 포인터가 쓰인다. 그 다음에는 뒷 연쇄체가 "ing"인지를 검사하고 "ing"면 앞 연쇄체를 unduplicate에 넘긴다. 함수 unduplicate는 자신의 논항인 연쇄체의 마지막 네 글자가 자음자 모음자 순정자음자 순정자음자의 배열인지, 그리고 마지막 두 순정자음자가 동일한지를 검사한다. "putt"는 이 조건을 충족하므로 그 끝 자를 떼어 낸 "put"가 반환된

다. stemsearch에 이 연쇄체를 또 넘겨 준다. 그러면 이 함수는 사전에서 "put"를 어간으로 갖는 동재항의 주소, 즉 55를 돌려 준다.

함수 analyze는 56과 55의 첫머리를 가리키는 포인터를 돌려 받아서 각 동재항의 기재 내용을 함수 realize에 넘긴다. 차례로 "putt"와 "put F"가 넘겨진다. 각각의 기재 내용과 함께 넘겨지는 것은 분석대상 연쇄체의 앞부분(두 경우 다 "putt")과 뒷부분("ing")이다. 이 세 가지 연쇄체를 받은 realize는 사전 동재 내용 "putt"가 하등의 굴곡부류 정보를 안갖고 있으므로 부류 X에 속하는, 번호가 1인 어간으로 간주한다. 접사가 "ing"이므로 먼저 어간의 마지막 글자가 "e"이면서 그 앞자가 "e"자가 아니며 어간의 길이가 2보다 더 길지 않은지 검사한다. "deleteing" 따위는 배제하고 "being"는 수용하기 위해서다. 이 검사 다음에는 접사와 어간 번호, 그리고 굴곡부류 사이의 충돌이 없는지에 대한 검사가 온다. 접사가 "ing"고 어간 번호가 1이니까 (a1)사전 동재항의 어간, 즉 "putt"가 ends_in_cvc를 만족하고 (a2) 넘겨 받은 어간 "putt"의 길이가 사전 동재항의 어간과 동일하거나 (b) 위 (a1)이 거짓이고 넘겨 받은 어간이 사전 동재항의 어간보다 더 긴지를 검사한다. 이 검사에 걸리면 가능한 굴곡형이 아니라고 판정된다. "offerring"은 여기에서 실패하지만 "putting"이나 "deleting", "occurring" 등은 (a1 & a2)에도, (b)에도 해당하지 않으므로 성공한다. 이 근처에서 굴곡부류와 어간 번호가 접사에 비추어 올바른지도 검사된다. 예를 들어 "wenting"의 경우라면, "went"가 부류 J에 속하는 낱말의 번호 2인 어간이므로 불가능한 결합체로 판명된다.

"putt", (그리고 사전 동재내용으로서의) "putt", 그리고 "ing"가 함수 realize에서 앞의 검사 둘을 통과하면 이들이 어떤 형태통사적 속성들을 실현하는지에 따라 일련의 정수가 축적되어 반환된다. 접사가 "ing"라서 현재진행형을 나타내는 정수 14만 반환된다.

함수 realize로부터 14를 넘겨 받은 analyze는 lexemerep에 (사전으로부터 얻은) 해당 어휘소의 정보를 넘겨서 "{putt}"를 반환 받고 이것과 14를 평이한 영어식 해석 "PRES PARTICIPLE of {putt}"

로 바꾸어서 자신의 두번째 논항으로 축적한다. 이로써 아까 stemsearch가 두 차례에 걸쳐서 돌려 준 정수 둘, 즉 56과 55 중에서 첫번째 경우에 대한 작업이 끝났다. 다시 두번 째 동재항인 "put F"를 함수 realize에 넘기는 것이다.

함수 realize는 "put F", "putt", "ing"를 논항으로 받아서, "put F"로부터 "put"가 굴곡부류 F의 번호 1인 어간임을 확정한다. 이들은 앞의 경우와 마찬가지로 검사 두 개를 거치게 되는데 규칙동사 "putt"의 경우와 같이 성공적인 굴곡형이라는 판정을 받는다. 함수 realize는 이번에도 14를 반환한다. 이 것과 "put F"로부터 함수 lexemerep가 얻어낸 "{put}"를 쉬운 영어 표현으로 바꾸어서, "PRES PARTICIPLE of {put}"를 아까 얻은 답, "PRES PARTICIPLE of {putt}" 다음에 놓는다. 더 이상 처리해야 할 사전 동재항의 주소가 없으므로 함수 analyze는 자신을 부른 종주함수한테 이 답을 보고한다. 종주 함수는 이 답을 모니터에 내어 놓고 프로그램 사용자한테 또 다시 "Enter your word form: "이라고 재촉한다.

엄격히 구별되는 두 꼴로 해석되는 "putting"과는 판이하게, "occured"은 아무런 해석도 부여 받지 못한다. 즉, 이것은 프로그램에 의해 배척되는 가상적인 꼴이다. 이 연쇄체가 배척되는 과정을 살펴보자.

함수 analyze는 "oc" "cured", "occ" "ured", "occu" "red"등을 잠시씩 고려하나 모두 접사의 길이가 영어의 동사 굴곡에 쓰이는 접사들보다 길므로 일찌기 배제된다. 이 함수가 "occur" "ed"의 분할로 stemsearch에 전자를 넘기고, 사전에 "occur"가 등재되어 있으므로 그 주소인 46을 넘겨 받는다. 함수 realize에 "occur", "occur", "ed"를 넘겨 준다. 여기에서도 첫 두 논항이 동일하지만, 그것은 어휘소 {occur}가 규칙동사이고 분석대상 문자연쇄체가 앞에서 본 unduplicate등의 과정을 겪지 않기 때문이다. 앞에서 "putt", "putt", "ing"의 경우나 "put F", "putt", "ing"의 경우처럼 적형성 검사를 거치게 되는데, 두 번째 검사에서 실패하고 만다. 그 이유는 (a1)사

전 동재항의 어간, 즉 "occur"가 ends_in_cvo를 만족하지 않는데 (a2) 넘겨 받은 어간 "occur"의 길이가 사전 동재항의 어간의 길이와 동일하기 때문이다. 따라서 함수 realize는 함수 analyze에 특별한 숫자 0을 보고하게 되고, 함수 analyze는 자신에게 임무를 준 종주함수한테 텅 빈 연쇄체(null string)를 반환하고 종주 함수는 모니터에 아무 것도 내놓지 않고 또 "Enter your word form: "으로 사용자를 재촉한다.

6. 평가 및 맺는 말

영어 동사의 굴절꼴을 해석하는 능력을 갖는 프로그램은 이미 많이 개발되어 있다. 여기에서는 이것들 중에서 쉽게 획득할 수 있는 프로그램 세 개를 선택해서 필자의 프로그램과 비교 평가하기로 한다.

비교 대상이 되는 프로그램은 (1) 운영체제인 UNIX의 연장들 중의 하나인 SPELL, (2) TOSCA/LOB Tagger, 그리고 (3) Rule Based Tagger이다. 이 중에서 첫 번째 것은 형태 분석 프로그램이 아니다. 굴절꼴에 관해서 아무런 정보를 제공하지 않고, 단지 입력된 문자연쇄체가 실재하는 영어 낱말인지 아닌지를 판별하는 능력만 갖는다. 나머지 두 개의 프로그램은 모두 낱말들의 품사정보와 굴절꼴 정보를 일정한 범위 안에서 제공해 주는 것을 주목적으로 한다. 이들의 자세한 내용에 관해서는 van Halteren and Oostdijk(1993)와 Brill(1994)를 각각 참고하라.

실재하는 영어 동사의 꼴 50개와 실재하지 않는 가상적인 문자연쇄체 50개에 이 프로그램들을 각각 실행하였다. 이상적인 프로그램이라면 실재하는 꼴들 모두에 대해서 각각의 굴절 특성들을 옳게 제공하고 실재하지 않는 연쇄체들에 대해서는 아무런 굴절특성을 제공하지 않아야 한다. 비교 대상이 되는 이 프로그램들은 다음과 같은 능력을 갖는 것으로 드러났다.[5] (여기에서 정방향 정확도는 실

[5]Rule Based Tagger (Version 1.14)가 TOSCA/LOB Tagger보다 월등히 나은 것

재하는 낱말꼴을 옳게 식별해 내는 비율을, 역방향 정확도는 실재하지 않는 낱말꼴을 옳게 배척해 내는 비율을 뜻한다.)

(5) 프로그램 명칭	정확도 (정방향)	정확도 (역방향)	사전 크기 (바이트)	프로그램 크기(바이트)
UNIX SPELL	100%	98%	906,347	17,024
Rule Based Tagger	82%	72%	1,364,889	59,880
TOSCA/LOB Tagger	100%	98%	2,464,642	>900,000

이 글에서 필자가 소개하는 프로그램 verbinfl은 정방향 정확도와 역방향 정확도가 모두 100%이며 사전의 크기는 170,200바이트이다. 위 프로그램과 verbinfl과의 단순비교는 독자로 하여금 잘못된 결론에 이르게 하기 쉽다. 우선 verbinfl은 동사 이외의 범주에 관해 아무런 처리능력을 갖고 있지 않으므로 영어의 모든 낱말들을 처리대상으로 삼는 위의 세 프로그램에 비해서는 그 규모가 작은 것이 당연하다. 사전의 크기가 verbinfl의 경우에 다른 것들에 비해서 더 작은 것도 당연하다.

verbinfl의 사전과 위 프로그램들의 사전을 크기에 있어서 직접 비교하는 것은 부당하므로 이들 사전들의 동사 정보의 양을 verbinfl의 사전과 비교해야 정당하다고 할 것이다. TOSCA/LOB Tagger의 경우에는 이 비교를 할 수 없다. 사전이 사용자들에게 투명하게 노출되어 있지 않기 때문이다. 그러나 나머지 두 개의 프로그램은 사전을 평범한 텍스트 파일로 갖고 있기 때문에 비교가 용이하다. 이 비교에 의하면 verbinfl의 상대적 우수성이 현저히 드러난다.

UNIX SPELL에 결부된 사전은 모든 낱말꼴들의 나열에 지나지 않는다. 규칙적인 굴절을 하는 동사들조차도 각각의 굴절꼴을 모두

으로 드러나는 이유는 전자가 낱말 하나당 단 한 개의 품사 정보만을 내어 놓는 데에 반하여 후자는 낱말 하나당 한 개 이상의 품사 정보를 내어 놓기 때문이다. 이 두 프로그램은 모두 텍스트 안의 낱말들에, 특정 문맥 속에서의 품사정보를 주게끔 의도된 것들이다.

등재하고 있다. Brill의 Tagger에 결부된 사전은 품사정보가 부착된 코퍼스로부터 단순한 과정을 거쳐서 구축되는 것으로서 분석된 코퍼스 안에 있는 모든 낱말꼴을 그대로 포함한다. 입력된 문자연쇄체가 이 사전에 등재되어 있지 않은 경우에는 추측절차(heuristics)들을 써서 품사 정보를 내어 놓는다.[6]

결국 이들 프로그램들과 verbinfl과의 가장 중요한 차이는 경제성에 있다. 모든 낱말꼴들을 포함하는 방대한 사전을 구성해서 쓰면 이 논문에서처럼 굴절부류들을 설정할 필요가 없어 보인다. 그러나 다른 모든 점에서 동일하다면 더 경제적인 해법이 우월하다. 특히 동일한 해법이 다른 언어의 기술에도 그대로 적용될 수 있을 때에는 더욱 그렇다. 동사 어휘소 하나가 수천 개 또는 수만 개의 굴절꼴들을 갖는 언어의 처리에 위의 세 프로그램의 해법들이 어떻게 적용될 수 있을지 상상하기 어렵다.[7]

낱말·패러다임 형태 이론에 입각한 영어 동사 굴절 해석 프로그램을 구현함으로써 통사 및 의미 해석의 기초를 제공함과 동시에 이 이론의 명시적인 검증을 가능하게 하였다. 모든 불규칙 현상과 함께 철자법 고유의 일탈조차 감안하면서도 일체 과수용(overrecognition)이 없는 체계를 완벽히 구축한 것으로 보인다. 사전 구성 또한 지극히 경제적인 것이며, 프로그램의 실행 속도도 인간인 영어 사용자의 형태 해석 속도보다 빠르므로, 이 프로그램에서 채택한 각종 해법은 그 기초 이론인 낱말·패러다임 이론과 함께 형태 부문 처리에 관한 심리적 모델로서의 타당성도 갖는다고 결론 지을 수 있다.

---

[6]연쇄체가 "ing"나 "ed"나 "en"으로 끝나면 동사로 간주하고 그렇지 않으면, 고유명사나 보통명사로 간주한다. 연쇄체의 첫 글자가 대문자이면 고유명사로, 그렇지 않으면 끝 글자가 "s"냐에 따라서 복수형 보통명사(NNS) 아니면 단수형 보통명사(NN)로 간주하는 것이다.

[7]Hankamer(1992: 403)에 의하면 터키어 동사 어근 하나당 1,830,248개의 굴절꼴이 존재한다. 보수적인 견해에 따르면 2,000개의 굴절꼴들이 터키어 동사 어휘소 하나에 연계되어 있다고 한다.

참고 문헌

Anderson, Stephen R. 1992. *A morphous morphology.* Cambridge: Cambridge University Press.

Aronoff, Mark. 1994. *Morphology by itself.* Cambridge, Massachusetts: MIT Press.

Beard, R. 1987. "Morpheme order in a lexeme/morpheme based morphology," Lingua 72. 73 116.

Brill, E. 1994. "Some advances in rule based part of speech tagging," in *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI 94)*, Seattle, Washington, 1994.

Feldman, Laurie Beth. 1995. *Morphological aspects of language processing.* Hillsdale, New Jersey: Lawrence Erlbaum Associates, Publishers.

Fowler, Anne E. and Isabelle Y. Liberman. 1995. "The role of phonology and orthography in morphological awareness," in Feldman(1995).

van Halteren, H. and N. Oostdijk. 1993. "Towards a syntactic database: The TOSCA analysis system," in Aarts, J., P. de Haan and N. Oostdijk (eds.) *English Language Corpora: Design, Analysis, and Exploitation.* Amsterdam: Rodopi. 145 161.

Hankamer, Jorge. 1992. "Morphological parsing and the lexicon," in Marslen Wilson (ed.) *Lexical Representation and Process.* Cambridge, Massachusetts: MIT Press.

Kenstowicz, Michael and Charles Kisseberth. 1979. *Generative phonology: Description and theory.* New York: Academic Press.

Kernighan, Brian W. and Dennis M. Ritchie, 1988, *The C*

programming language, second edition, Englewood Cliffs, New Jersey: Prentice Hall PTR.

Lieber, Rochelle. 1992. *Deconstructing morphology.* Chicago: University of Chicago Press.

Matthews, P. H. 1972. *Inflectional morphology: A theoretical study based on Latin verb conjugation.* Cambridge: Cambridge University Press.

Matthews, P. H. 1974. *Morphology: An introduction to the theory of word structure.* Cambridge: Cambridge University Press.

Matthews, P. H. 1991. *Morphology: An introduction to the theory of word structure. 2nd edition.* Cambridge: Cambridge University Press.

Selkirk, Elizabeth. 1982. *The syntax of words.* Cambridge, Massachusetts: MIT Press.

Zwicky, Arnold M. 1985. "How to describe inflection," Berkeley Linguistic Society.

Zwicky, Arnold M. 1988. "Morphological rules, operations, and operation types," *ESCOL 87: Proceedings of the Fourth Eastern States Conference on Lingusitics.* 318 34.

부 록

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#define MAXLINES 1000
#define MAXLEN 1000
#define CHARALLOC(n) (char *) malloc (n * sizeof(char))
#define ERRORMSG { printf("malloc failed\n"); exit(1); }
#define is_vowel(c) (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
#define is_cons(c) (c >= 'a' && c <= 'z' && !is_vowel(c))
#define is_obstru(c) (is_cons(c) && c != 'y' && c != 'w')

char line[MAXLEN];
char *dictionary[MAXLINES];
int nlines;
FILE *dict;
void analyze(char *t, char *result);

int main()
{
  char *candidate, *analyses;

  candidate = CHARALLOC(30);
  if (candidate == (char *) NULL)
    ERRORMSG
  dict = fopen("verb.dic", "r");
  if ((nlines = readlines(dictionary, MAXLINES)) >= 0)   ;
/*
```

```
    qsort(dictionary, 0, nlines-1);
*/
  else {
    printf("error: input too big to sort\n");
    return 1;
  }
  analyses = CHARALLOC(500);
  if (analyses == (char *) NULL)
    ERRORMSG
  do {
    printf("Enter your word-form: ");
    scanf("%s", candidate);
    analyze(candidate, analyses);
    if (*analyses)
      printf("%s", analyses);
  } while (strncmp(candidate, "nomore", 6));
  free(analyses);
  free(candidate);
  fclose(dict);
  return 0;
}

char *realize(char *l, char *s, char *affix)
{
 #define IDEN 0
 #define S 1
 #define ES 2
 #define D 3
 #define ED 4
 #define N 5
```

```
#define EN 6
#define ING 7
#define UNLIKELY 8
#define SUFFIX(s) !strcmp(affix, s)
#define CLS(x) iclass == x

char *p, i=0, stemnum, aftype;
char iclass, *stemfinal, *stem_in_dict, *t;

p = CHARALLOC(16);
if (p == (char *) NULL)
{
 printf("error: no space left\n");
 return p;
}
*p = -1;
while (*(l + i) != ' ' && *(l + i))
 i++;
if (*(l + i))
 iclass = *(l + i + 1);
else
 iclass = 'X';
if (!(CLS('X')) && *(l + i + 2))
 stemnum = *(l + i + 2) - '0';
else
 stemnum = 1;
if (SUFFIX("\0"))
 aftype = IDEN;
else if (SUFFIX("s"))
 aftype = S;
```

```
 else if (SUFFIX("es"))
  aftype = ES;
 else if (SUFFIX("d"))
  aftype = D;
 else if (SUFFIX("ed"))
  aftype = ED;
 else if (SUFFIX("ing"))
  aftype = ING;
 else if (SUFFIX("n"))
  aftype = N;
 else if (SUFFIX("en"))
  aftype = EN;
 else
  aftype = UNLIKELY;
 stemfinal =s;
 while (*stemfinal)
  stemfinal++;
 stemfinal--;
 stem_in_dict = CHARALLOC(15);
 strncpy(stem_in_dict, l, i);
 *(stem_in_dict + i) = '\0';
 switch (aftype)
 {
  #define HSXZO (strchr("hsxzo", *stemfinal) != (char *) NULL)
  #define ENDS_IN(c) (*stemfinal == c)

  case ED: if (ENDS_IN('e') || (ENDS_IN('y') && !is_vowel(*(stemfinal - 1)))) *p =
0;
    if (*(stem_in_dict + i - 1) == 'e') *p = 0; /* *deletted */
    break; /* *deleteed *denyed prayed */
```

```
  case D: if (!ENDS_IN('e')) *p = 0; break; /* deleted *kissd */
  case S: if (HSXZO || (ENDS_IN('y') && !is_vowel(*(stemfinal - 1)))) *p = 0;
   break;  /* *kisss *denys prays deletes */
  case ES: if (!HSXZO) *p = 0; break;
  case N: if (!ENDS_IN('e') && !ENDS_IN('w')) *p = 0; break;
  case EN: if ((ENDS_IN('e') && !(CLS('L'))) || ENDS_IN('w'))    *p = 0; break;
  case ING: if (ENDS_IN('e') && *(stemfinal - 1) != 'e' && strlen(s) > 2)    *p =
0; break;
  default: break;
 }
 if (*p == 0)
  return p;
 switch (aftype)
 {
  case IDEN:
   if (stemnum == 2 && (CLS('A') || CLS('B') || CLS('K')))
    *p = 0;
   break;
  case S:  if  (is_vowel(*(stem_in_dict  +  i  -  2))  &&  strlen(s)  !=
strlen(stem_in_dict))
    *p = 0; /* *praie-s pray-s denie-s */
  case ES:
   if (CLS('I') || CLS('L'))
    *p = 0;
   switch (stemnum) {
    case 1: if (CLS('A'))
      *p = 0;
     break;
    case 2:
     if (CLS('B') || CLS('C') || CLS('D') || CLS('E') || CLS('H') || CLS('G') ||
```

```
CLS('H') || CLS('J') || CLS('K'))
      *p = 0;
     break;
    default: *p = 0; break;
   }
   break;
  case D: if (strlen(s) != strlen(stem_in_dict) && is_vowel(*(stem_in_dict + i -
2))) *p = 0;
/* *praie-d denie-d */
          if (CLS('K')) *p = 0;
    break;
  case ED: switch (iclass) {
   case 'X':   if (ends_in_cvo(stem_in_dict) && strlen(s) == strlen(stem_in_dict))
   *p = 0;
    /* *occured *spaned offered */
    if (!ends_in_cvo(stem_in_dict) && strlen(s) != strlen(stem_in_dict)) *p = 0;
/* *offerred */
    break;
   case 'A': if (stemnum == 1)
     *p = 0;
    break;
   case 'B': if (stemnum == 2)
     *p = 0;
    break;
   default: *p = 0;
   }
   break;
  case N: if (CLS('E') && !strcmp("ite", stemfinal - 2)) *p = 0;
  case EN: if (!((CLS('E') && stemnum == 1) || (stemnum == 2 && (CLS('D')))
 || (CLS('H') && stemnum == 3) || (CLS('L') && stemnum == 1)))
```

```
    *p = 0;
   if (CLS('D') && stemnum == 2 && ends_in_cvo(s))
    *p = 0;
   break;
  case ING: switch (stemnum) {
   case 1: switch (iclass) {
    case 'X': if ((ends_in_cvo(stem_in_dict) && strlen(s) == strlen(stem_in_dict))
||
    (!ends_in_cvo(stem_in_dict) && strlen(s) > strlen(stem_in_dict))) /* offerr vs.
delet */
      *p = 0;
     break;
    case 'A': case 'B': *p = 0; break;
    case 'F':
     if (strlen(s) == strlen(stem_in_dict))
      *p = 0;
     break;
    case 'K': *p = 0; break;
    default: break;
    }; break;
   case 2: switch (iclass) {
    case 'C': case 'D': case 'E': case 'G': case 'H': case 'I': case 'J': case 'L':
*p = 0;
             break;
    default: break;
    }; break;
   default: *p = 0; break;
  }; break;
  default: break;
 }
```

```
 if (*p == 0)
  return p;
 i = 0;
 switch (aftype)
 {
  case IDEN: switch (stemnum) {
   case 1: *(p + i++) = 1;
    if (!(CLS('L'))) {
     *(p + i++) = 2; *(p + i++) = 3; *(p + i++) = 5; *(p + i++) = 6; *(p + i++) =
7;
    }
    if (CLS('F')) {
     *(p + i++) = 8; *(p + i++) = 9; *(p + i++) = 10;
     *(p + i++) = 11; *(p + i++) = 12; *(p + i++) = 13; *(p + i++) = 15;
    }
    if (CLS('H')) { *(p + i++) = 15; }
    break;
   case 2:
    if (CLS('I'))
     *(p + i++) = 4;
    else if (CLS('L'))
     *(p + i++) = 2;
    else
    {
     *(p + i++) = 8; *(p + i++) = 9; *(p + i++) = 10; *(p + i++) = 11; *(p + i++) =
12;
     *(p + i++) = 13;
    }
    if (CLS('C'))
     *(p + i++) = 15;
```

```
    break;
   case 3: if (CLS('G') || CLS('J'))
     *(p + i++) = 15;
    else if (CLS('K'))
    {
     *(p + i++) = 8; *(p + i++) = 9; *(p + i++) = 10; *(p + i++) = 11; *(p + i++) =
12;
    }
    else if (CLS('L'))
    {
     *(p + i++) = 3; *(p + i++) = 5; *(p + i++) = 6; *(p + i++) = 7;
    }
    else
    {
     *(p + i++) = 8; *(p + i++) = 9; *(p + i++) = 10; *(p + i++) = 11; *(p + i++) =
12;
     *(p + i++) = 13; *(p + i++) = 15;
    }
    break;
   case 4: if (CLS('K')) *(p + i++) = 15;
    else *(p + i++) = 4; break;
   case 5: *(p + i++) = 8; *(p + i++) = 10; break;
   case 6: *(p + i++) = 9; *(p + i++) = 11; *(p + i++) = 12; *(p + i++) = 13;
break;
   default: break;
  }
  break;
  case S:
  case ES: *(p + i++) = 4; break;
  case D:
```

```
  case ED: *(p + i++) = 8; *(p + i++) = 9; *(p + i++) = 10; *(p + i++) = 11; *(p +
i++) = 12;
           *(p + i++) = 13; *(p + i++) = 15;
    break;
  case N: case EN: *(p + i++) = 15; break;
  case ING: *(p + i++) = 14; break;
  default: break;
 }
 *(p + i) = 0;
 return p;
}

void analyze(char *cand, char *ana)
{
  char *result, *stem, *suffix, *lexeme, *temp, *property;
  int boundary, len, i, *stemloc, *locorigin;

  result = ana;
  *result = '\0';
  len = strlen(cand);
  stem = CHARALLOC(30);
  if (stem == (char *) NULL)
    ERRORMSG
  locorigin = (int *) malloc (300 * sizeof(int));
  if (locorigin == (int *) NULL)
    ERRORMSG
  stemloc = locorigin; *stemloc = -1;
  suffix = CHARALLOC(30);
  if (suffix == (char *) NULL)
    ERRORMSG
```

```
lexeme = CHARALLOC(30);
if (lexeme == (char *) NULL)
 ERRORMSG
for (boundary=2;boundary<=len;boundary++)
{
 strncpy(stem, cand, boundary);
 *(stem + boundary) = '\0';
 strcpy(suffix, cand+boundary);
 if (len - boundary < 4)
 {
  stemsearch(dictionary, nlines, stem, stemloc);
  if (!strcmp(suffix, "ing")) {
   temp = unduplicate(stem);
   if (*temp)
    stemsearch(dictionary, nlines, temp, stemloc);
   else
    free(temp);
  }
  if (*stemloc == -1) {
   if (!strcmp(suffix, "ed") || !strcmp(suffix, "en")) {
    temp = unduplicate(stem);
    if (*temp)
     stemsearch(dictionary, nlines, temp, stemloc);
    else
     free(temp);
    temp = tt2te(stem);
    if (*temp)
     stemsearch(dictionary, nlines, temp, stemloc);
    else
     free(temp);
```

```
  }
  else if (!strcmp(suffix, "ing")) {
    temp = CHARALLOC(20);
    strcpy(temp, stem);
    strcat(temp, "e");
    stemsearch(dictionary, nlines, temp, stemloc);
    free(temp);
  }
  else if (!strcmp(suffix, "s") || !strcmp(suffix, "d")) {
    temp = ie2y(stem);
    if (*temp)
      stemsearch(dictionary, nlines, temp, stemloc);
    else
      free(temp);
  }
}
if (*stemloc == -1)
  ;
else
  while (*stemloc != -1)
  {
    char *tmp;

    strcpy(lexeme, dictionary[*stemloc]);
    tmp = CHARALLOC(30);
    if (tmp == (char *) NULL)
      ERRORMSG
    property = realize(lexeme, stem, suffix);
    i = 0;
    while (*(property + i))
```

```
{
 #define CAT(s) strcat(result, s);
 lexemerep(lexeme, tmp);
 switch (*(property + i)) {
  case 1: CAT("\tBASE of ");
    break;
  case 2: CAT("\t1sg PRES of ");
    break;
  case 3: CAT("\t2sg PRES of ");
    break;
  case 4: CAT("\t3sg PRES of ");
    break;
  case 5: CAT("\t1pl PRES of ");
    break;
  case 6: CAT("\t2pl PRES of ");
    break;
  case 7: CAT("\t3pl PRES of ");
    break;
  case 8: CAT("\t1sg PAST of ");
    break;
  case 9: CAT("\t2sg PAST of ");
    break;
  case 10: CAT("\t3sg PAST of ");
    break;
  case 11: CAT("\t1pl PAST of ");
    break;
  case 12: CAT("\t2pl PAST of ");
    break;
  case 13: CAT("\t3pl PAST of ");
    break;
```

```
            case 14: CAT("\tPRES PARTICIPLE of ");
              break;
            case 15: CAT("\tPAST PARTICIPLE of ");
              break;
            default: break;
          }
          CAT(tmp);
          i++;
        }
        free(property);
        stemloc++;
        free(tmp);
      }
    }
  }
  free(stem);
  free(locorigin);
  free(suffix);
  free(lexeme);
}
```

대전광역시 유성구 궁동 220
충남대학교 문과대학 언어학과
305-764
E-mail: ynoling@hanbat.chungnam.ac.kr
FAX: +82-42-823-3667

접수일자: 1998. 5. 10
게재결정: 1998. 9. 30