# DDAG: 효율적인 한국어 형태소 해석 방법

김 덕봉, 최 기선

한국과학기술원 전산학과

# DDAG: An Efficient Method for Morphological Analysis of Korean

Deok-Bong Kim and Key-Sun Choi

Dept. of Computer Science, KAIST

**요 약**

기존의 한국어 형태소 해석 시스템들은 철자 변화형 어절에 대한 처리가 매우 효율적이지 못했다. 대개가 문제를 일으키는 형태소들의 변형들을 모두 사전에 등록하여 후처리 형태로 다루려 하거나, 각 형태/음운 규칙을 적용한 다음 거기에 대응하는 후보 단어들을 사전 검색을 통해 확인하는 방법들을 취하고 있다. 그러나 이러한 방법들은 과다한 사전 정보의 중복이나 계산의 중복으로 인하여 비효율적인 면을 많이 내포한다. 또한, 기존의 한국어 형태소 시스템들은 거의 모두가 형태소 해석 엔진과 언어학적인 지식(특히, 철자 규칙과 형태소 배열 규칙)이 제대로 분리되지 않아 시스템 확장이 매우 어려웠다.

이 논문에서는, 철자 변화형 어절을 후처리에 의하지 않고, 사전 검색과 함께 하나의 오토마타에 의해 처리하면서, 형태소 해석시 발생하는 중복 계산을 최대한 배제하고, 또한 형태소 해석 엔진과 언어학적인 지식을 완전히 분리하여 시스템의 확장성을 한층 높인, 효율적인 한국어 형태소 해석 시스템 DDAG를 소개한다. 이 시스템의 주요 알고리즘의 계산적인 복잡도는 n이 입력 어절의 길이이고, m이 입력 어절을 이루고 있는 형태소의 최대 수라고 할 때 다음과 같이 분석된다: (1) 철자 변화의 처리와 사전 검색 부분의 계산적인 복잡도는 $O(n^2)$이고, (2) 형태소 배열 검사와 모든 가능한 결과를 출력해 내는 부분은 $O(2^m)$이다. 여기에서 m의 실질적인 값은 복잡한 한국어 용언의 경우 최대 8이다.

## 1. 서론

다른 시스템들과 마찬가지로 형태소 해석 시스템을 구축하는데 있어서도 시스템의 초기 구축으로부터 그 시스템을 확장하는 과정이 필요하다. 그러나, 이러한 확장에는 일반적으로 많은 비용과 어려움이 따른다. 특히, 사전의 확장과 규칙의 확장은 그것의 대표적인 본보기라 할 수 있다. 따라서, 이러한 확장을 저렴하고 편리하게 하기 위해서는 다음과 같은 것들이 요구된다:

- 첫째, 사전 표제어는 원칙적으로 기본 이형태(basic allomorph)로만 하고, 다른 이형태들은 알고리즘에서 지원해 주어야 한다. 이는 사전 정보의 중복 가능성을 미리 없애 줄 뿐만 아니라 사전 구축과 확장의 비용을 줄여 주는 효과를 준다[10].

- 둘째, 어휘에 종속적이 아닌 사전 정보는 사전에서 최대한 배제시켜야 한다. 예를 들어, 기존의 한국어 형태소 해석 시스템에서의 좌우 접속 정보는 어휘의 품사에 따라 중복되는 요소가 많기 때문에 사전의 각 어휘마다에는 품사 정보만을 두고, 접속 정보에 관한 것은 따로 두는 것이 바람직하다.

- 세째, 알고리즘과 언어학적인 지식(문법, 사전 등)은 완전히 분리되어야 한다.

- 네째, 규칙들간의 상호 작용을 최대한 배제하고, 언어학자도 받아 들일 수 있는 규칙 기술 언어가 제공되어야 한다.

- 다섯째, 개발과 확장을 돕는 다수의 도구들(사전 에디터, 규칙 컴파일러 등)이 마련되어야 한다.

본 논문은 위와 같은 조건을 만족하는 효율적인 한국어 형태소 해석 시스템의 설계 및 구현에 대하여 다룬다. 우리는 이 시스템을 DDAG (Dictionary + Dynamic Programming + Automata + Grammar)라 부른다. DDAG의 효율성은 먼저 기존의 한국어 형태소 해석 시스템들이 갖고 있는 심각한 문제점들이 무엇인지를 알 때 뚜렷하게 드러나리라고 생각된다. 하지만, 지면 관계상 그 모든 것을 살펴 볼 수는 없고, 여기에서는 철자 변화형 어절에 대한 처리에 촛점을 맞춰 기존의 한국어 형태소 시스템들이 안고 있는 문제점들을 점검해 본다.

**예제 1** 한국어 어절 "도와"는 문맥에 따라 *3*가지 해석이 가능하다*:*

*1.* "도와"는 명사 "도"와 조사 "와"로 구성된 것이다.

*2.* "도와"는 동사 "돕"과 어미 "아"로 구성된 철자 변화형 어절이다.

*3.* "도와"는 그 자체가 명사이다.

기본적으로 예제 1의 3 가지 해석을 위해서는 주어진 어절로부터 그 어절을 이루고 있는 모든 형태소들의 순서쌍들을 추출하는 형태소 분절 과정이 필요하다. 이 과정은 대부분의 시스템에서 사전 탐색과 병행하여 수행된다. 그러나, 이 과정은 그다지 쉬운 작업이 아니다. 만약 기본 이형태만이 사전에 들어 있다고 가정할 경우, 1과 3을 위한 형태소 분절은 사전 탐색만으로도 가능하지만, 2는 그렇지 못하다. 따라서, 이 경우에, 2를 위해서는 좀 더 복잡한 뭔가가 요구된다.

기존의 한국어 형태소 시스템들이 철자 변화형 어절을 다루는 방법에는 크게 나누어 두 가지가 있다: 사전에 의존하는 방법[3]과 규칙에 의존하는 방법[2]. 사전에 의존하는 방법은 대개가 예제 1에 있는 2의 해석을 위하여 동사 "돕"의 이형태들인 "돕"과 "도", 어미 "아"의 이

형태들인 "아"와 "와"를 모두 사전에 등록한다. 이러한 방법이 갖는 장점은 간단한 사전 탐색만으로 모든 어절의 형태소 분절이 가능하다는 점이다. 그러나, 이 방법을 위해서는 가능한 모든 이형태들을 고려해서 그것들을 사전에 모두 등록해야 한다는 단점이 있다.

사전에 의존하는 방법이 갖는 더 심각한 문제는 이형태들의 사전 등록으로 인하여 과잉 해석의 오류가 야기될 수 있다는 점이다. 단순히 "아"의 이형태인 "와"가 어미라고만 되어 있다라고 한다면, 예제 1의 "도와"는 명사 "도"와 어미 "와"로 구성된 것으로 하나 더 해석될 수 있다. 이러한 과잉 해석을 막기 위해서는 어미 "와"에 대하여 보다 세밀한 범주의 분류와 접속 정보의 부여가 필요하다. 이러한 정보는 대개 어휘에 종속적인 경향을 띤다. 예를 들면, 어미 "와"는 반드시 'ㅂ'-불규칙 용언의 어간하고만 결합될 수 있다라는 제약들을 갖고 있어야만 한다. 결국, 이 방법은 사전 구축 및 확장시에 모든 어휘의 이형태들에 대하여 어휘에 종속적인 접속 정보 등을 고려해 주어야 하기 때문에 사전 구축이나 확장이 어렵고, 또한 비용도 많이 드는 심각한 면을 지닌다고 할 수 있다.

이와는 반대로, 규칙에 의존하는 방법은 원칙적으로 기본 이형태만을 사전에 두고, 나머지 이형태들은 모두 규칙으로 처리한다. 지금까지 규칙에 의존하는 방법으로 우리에게 가장 많은 관심을 받았던 모델은 두 단계 모델[2][6][9]이다. 두 단계 모델은 개개의 철자 규칙을 독립된 오토마타의 일종인 FST(Finite-State Transducer)로 바꾼 다음, 이들을 병렬적으로 구성하여 철자 변화형 어절을 처리한다. 이 모델이 갖는 최대의 장점은 FST의 성질로 인해 하나의 규칙을 가지고 형태소 해석과 생성을 함께 할 수 있다는 점이다. 그러나, 두 단계 모델은 Barton(1986)[7]이 증명했듯이 계산적인 효율성을 보장하지 못한다. 이는 FST의 비결정성과 규칙의 교차 현상에 의해 과다한 백트랙킹이 야기되기 때문이다.

## 2. DDAG의 구조

DDAG는, 시스템의 효율과 확장성을 높이기 위하여, 그림 1에서와 같이 형태소 해석을 하는데 필요한 3가지의 지식(철자 규칙, 형태소 배열 규칙, 사전)을 형태소 해석 엔진으로부터 분리하여 두 가지 형태로 유지시킨다. 하나는 시스템 개발자 이외의 사람들(예, 언어학자)도 쉽게 다룰 수 있는 것이고, 나머지 하나는 형태소 해석 엔진을 위한 것이다.

또한, DDAG는 언어학자가 쉽게 철자 규칙과 형태소 배열 규칙을 작성하고 확장할 수 있도록 하기 위하여 두 개의 다른 규칙 기술 언어를 제공한다. 철자 규칙의 기술을 위한 음절 기반 유한 언어 SBFL과 형태소 배열 규칙의 기술을 위한 문맥 자유 문법 CFG 형태가 바로 그것이다. 필요에 따라서, 이들 규칙 기술 언어로 작성된 규칙들은 규칙 컴파일러에 의해 오토마타(상태 전이 테이블)로 변환된다. 이 오토마타는 형태소 해석 엔진이 규칙의 적용을 결정적으로 하게 하여 알고리즘의 계산적인 복잡도를 상당히 낮추는 효과를 준다.

사전은 언어학자 등이 다룰 수 있는 개발용 사전과 형태소 해석 시스템에서 사용하는 시스템 사전으로 분류된다. 개발용 사전은 기능별(품사별)로 여러개가 유지된다. 이렇게 하게 된 커다란 이유는 사전이 대개 기능별로 비슷한 정보를 갖고 있어, 기능별로 유지하는 것이 사전 개발이나 관리 측면에서 효과적이기 때문이다. 그러한 사전의 종류로는 체언 사전, 용언 사전,

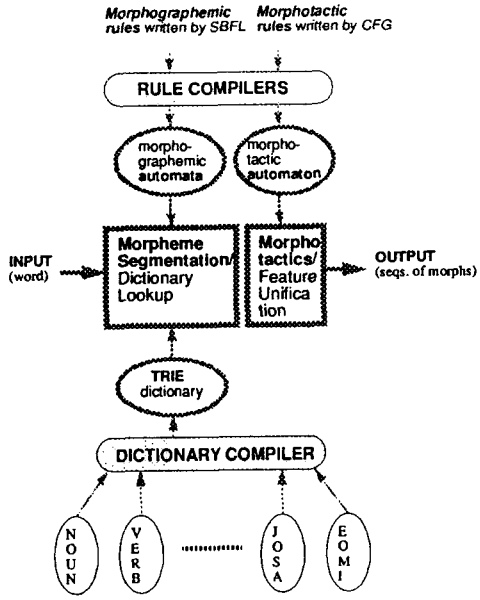



그림 1: DDAG의 구조

독립언 사전, 조사 사전, 어미 사전, 파생접사 사전 등이 있다. 이들의 사전 정보는 표제어를 단위로 하여 보통 자질-값의 쌍들로 나타난다.

그러나, 시스템 사전은 원칙적으로 하나만을 갖는다. 이 사전은 개발용 사전으로부터 사전 컴파일러에 의해 자동으로 얻어진다. 이 사전은 2개의 화일로 유지된다: TRIE 화일과 INFO 화일. TRIE 화일은 음절을 기반으로 하면서 음소를 기본 단위로 하여 표제어를 구성하는 일종의 인덱스 화일이다. 시스템의 효율을 위하여, TRIE 화일은 보통 시스템 실행시에 메모리에 상주한다. 이에 비해, INFO 화일은 각 표제어에 대한 사전 정보를 담고 있는 일종의 데이타 화일이다. 이 화일은 응용 분야에 따라 많은 양의 정보를 갖을 수 있기 때문에 TRIE 화일과 같이 메모리에 두는 것보다 하드 디스크에 두는 것이 바람직하다. 그러나, 시스템의 효율을 위해서는 다음과 같이 운영될 수도 있다: 조사나 어미와 같은 기능어에 대한 정보는 메모리에 두고, 나머지 정보는 하드 디스크에 둔다.

## 3. 철자 규칙과 오토마타

철자 변화형 어절 "도와"를 해석하는데 필요한 규칙을 SBFL 규칙으로 표현해 보자.

**예제 2** "도와"에 대하여 예제 *1*의 *2*와 같은 해석을 위해서는 다음과 같은 한글 맞춤법(제 *18*항 *6*의 일부)[5]이 요구된다: '돕-, 곱-'과 같은 단음절 어간에 어미 '-아'가 결합되어 '와'로 소리나는 것은 '-와'로 적는다.

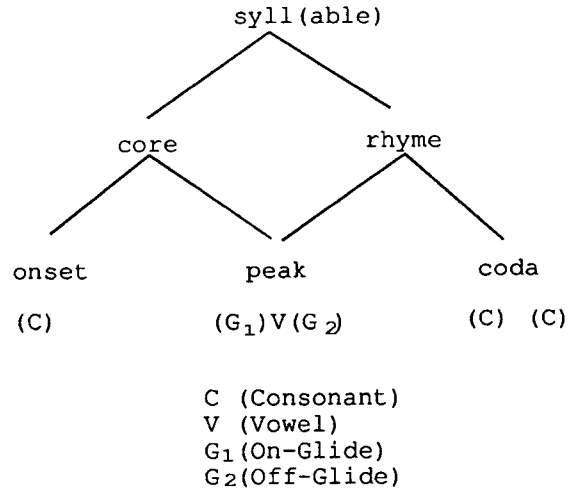



그림 2: 한국어를 위한 음절 구조

**RULE 1** wa(core,i) := p + a ⇔ [a | o](rhyme,i-1) & i = 2
/ [left [cat==verb,adjective][irregular==p]]

RULE 1를 이해하기 위해서는 먼저 그림 2와 같은 음절 구조를 아는 것이 필요하다. 그림 2의 음절 구조는 오른쪽 분기 음절 구조[1] 와 왼쪽 분기 음절 구조[2] 가 혼합된 형태이다. 이 음절 구조는 한국어의 철자 변화를 좀 더 자연스럽고 간단하게 기술할 수 있도록 해 준다. 이는 한국어의 철자 규칙이 RULE 1과 같이 코어(core)나 운(rhyme)을 직접적으로 참조하는 경우가 많기 때문이다. 결과적으로, RULE 1은 다음을 나타낸다: 두 번째(i = 2) 음절의 코어 wa('와')가 받침 p('ㅂ')로 대치되어 하나의 형태소를 이루고 다음 형태소가 a('아')로 되기 위한 필요·충분 조건은(wa(core,i) := p + a ⇔) 첫 번째 음절의 운이 일차적으로 a('아')나 o('오')여야 하고([a | o](rhyme, i-1)), 분리된 왼쪽의 형태소가 'ㅂ'-불규칙인 동사나 형용사여야([left [cat==verb, adjective] [irregular==p]]) 한다.

규칙 컴파일러에 의해 RULE 1은 ACTION을 갖는 결정적인 유한 오토마타로 변환된다. 그림 3은 그러한 유한 오토마타를 도식적으로 표현한 것이다. 이 오토마타의 S5는 RULE 1의 종결상태(final state)를 나타내며, 다음과 같은 ACTION을 포함한다:

- 규칙 RULE 1의 ACTION: TRIE의 현재 상태에서 자소 p('ㅂ')를 가지고 그 TRIE를 예비 탐색(look-ahead)한다. 만약, 그 자소가 하나의 형태소를 이루게 하면, 그 형태소의 어휘 정보가 품사 정보로 동사나 형용사를 갖고, 불규칙 코드로 'p'를 갖는지를 확인한다. 확인 결과,

[1]오른쪽 분기 음절 구조는 음절핵(Peak)과 음절후부(Coda)가 운(Rhyme)이라고 불리는 단위를 형성하고, 이것이 음절전부(Onset)와 함께 하나의 음절을 이루는 구조이다.

[2]왼쪽 분기 음절 구조는 음절전부(Onset)와 음절핵(Peak)이 코어(Core)라고 불리는 단위를 형성하고, 이것이 음절후부(Coda)와 함께 하나의 음절을 이루는 구조이다.

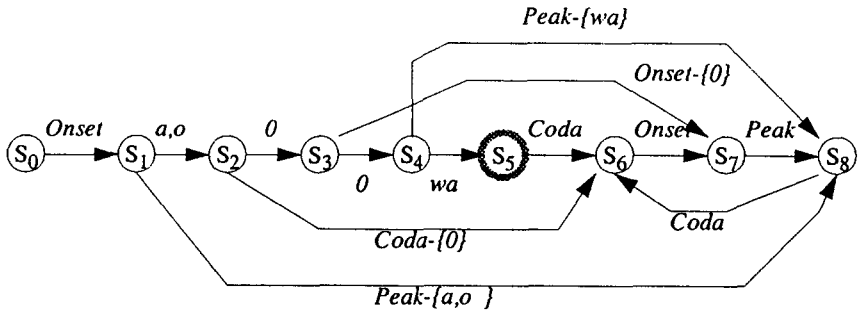



그림 3: 규칙 컴파일러에 의한 SBFL 규칙 R2의 번역

그것이 사실이면, 입력 단어의 현재 위치에서 규칙이 적용되는 범위까지를 a('아')로 대체한 다음 *TRUE*를 리턴하고, 그렇지 않다면, *FALSE*를 리턴한다.

SBFL 규칙 컴파일러는 개개의 철자 규칙을 ACTION을 갖는 오토마타로 바꾸고, 오토마타 이론에 의해 이들을 모두 합쳐 하나의 커다란 오토마타를 구성한다. 이것은 새로운 상태 S에서 각 오토마타의 초기 상태로 가는 $\epsilon$이 있는 NFA를 가정한 다음, 이 NFA를 DFA로 바꾸는 알고리즘[8]에 의해 이루어진다. 물론, 이 과정에서 오토마타의 최소화도 행해진다.

# 4. 형태소 분절 알고리즘

형태소 분절 과정은 주어진 어절로부터 그 어절을 이루고 있는 모든 가능한 형태소들을 분리해 내는 과정이다. 이 과정은 보통 사전 검색과 함께 다루어진다. 여기에서 제안하는 형태소 분절 알고리즘은 동적 프로그래밍 기법을 이용하여 사전의 중복 탐색을 방지하고, 결정적인 유한 오토마타에 바탕을 두어 철자 변화 현상을 효율적으로 처리하는 방법이다. 먼저, 이 알고리즘에서 사용하는 광역 변수들을 선언하면 다음과 같다:

1. **input**: word의 집합. 여기에서 하나의 word는 (onset × peak × coda)*이다.
2. **chart**: chartNode의 집합. 하나의 chartNode는 형태소 (morph), 다음 형태소의 위치 (nextPosition), 형태소의 어휘정보 (lexicalInformation)를 이룬다.
3. **agenda**: agendaItem의 집합으로 된 큐. 하나의 agendaItem은 규칙 번호 (ruleNumber), 규칙의 적용 위치 (ruleTriggeredPosition), 규칙의 적용 범위 (ruleAppliedRange)를 이룬다.

4. **mTrie**: 메모리에 상주하는 trieNode의 집합.

5. **lexicalTable**: chartNode의 리스트로 된 테이블. 이 테이블의 크기는 최대 word의 길이이다.

6. **GetAgenda(position)**: 주어진 입력 word의 position에서 그 word의 끝 위치까지 철자 규칙의 오토마타를 수행하여 agendaItem를 구한 다음, 이를 리턴한다.

알고리즘의 주 모듈 *MainBody*는 주어진 입력으로부터 어절을 얻고, 철자 규칙 오토마타를 이용하여 그 어절에 나타날 수 있는 규칙들의 집합을 구한 다음, 이들을 다음 모듈에 넘겨줘 입력 어절에 대한 형태소 분절 결과를 얻는다. 이 결과는 리스트 차트 구조(chart)에 남는다.

```
1 Procedure MainBody()
2     while ((word ← GetWordPhrase(input)) is not EOF)
3         agenda ← GetAgenda(0) {'0'은 word의 시작 위치임.}
4         chart ← SegmentationOfFirstString(0)
5     end while
6 end Procedure
```

```
1 Procedure SegmentationOfFirstString(position)
2     초기화: truncationFlag ← FALSE { Trie 탐색 절단 정보 }
3             offset ← 0 { 탐색된 trieNode의 mTrie 내에서의 절대위치 }
4             attachFlag ← ToChild { 생성될 chartNode의 접속 정보 }
5     agendaItem ← dequeue from agenda
6     for each i ← position to length of word do
7         offset를 일시적으로 보관한다.
8         { 순수 사전 탐색 }
9         case i번째 word의 자소로 mTrie 탐색 { offset 변경 }
10            실패 : truncationFlag ← TRUE
11            성공 & trieNode가 종결노드 : 현재 상태를 모두 스택에 보관하고, 종결
12                노드가 가르키는 형태소를 가지고 하나의 chartNode를 생성한다. 또한,
13                i+1를 키로 하여 생성된 chartNode를 lexicalTable에 등록한다.
14                SegmentationOfRestString(i+1)
15                스택으로부터 이전 상태를 모두 복원한다.
16                attachFlag ← ToSibling
17        end case
18        { 계층 1의 규칙 적용 }
19        while (agendaItem.ruleTriggerdPosition = i)
20            if (agendaItem.ruleNumber의 규칙 적용)
21                현재 상태를 모두 스택에 보관하고, 규칙 적용에 따라 입력 word을 변경한다.
22                규칙 적용으로 구한 형태소를 가지고 하나의 chartNode를 생성하고, i를
23                키로 하여 생성된 chartNode를 lexicalTable에 등록한다.
24                SegmentationOfRestString(i)
25                스택으로부터 이전상태를 모두 복원한다.
```

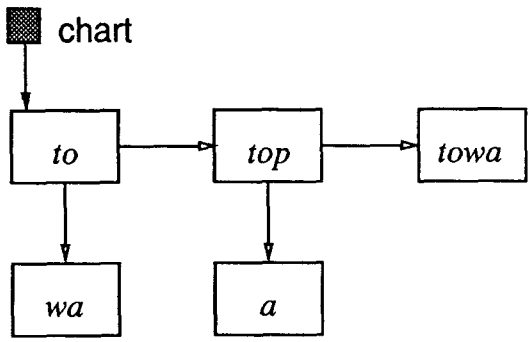
```
26                  attachFlage ← ToSibling
27              end if
28              agendaItem ← dequeue from agenda
29          end while
30          if (truncationFlag = TRUE) return end if
31      end for
32      { word의 맨 마지막 위치에서 규칙이 발생될때 }
33      while (agendaItem.ruleTriggeredPosition = i) { i is length of word }
34          if (agendaItem.ruleNumber의 Rule 적용)
35              현재 상태를 모두 스택에 보관하고, 규칙 적용에 따라 입력 word을 변경한다.
36              규칙 적용으로 구한 형태소를 가지고 하나의 chartNode를 생성하고, i를
37              키로 하여 생성된 chartNode를 lexicalTable에 등록한다.

38              SegmentationOfRestString(i)

39              이전상태 복원
40              attachFlag ← ToSibling
41          end if
42          agendaItem ← dequeue from agenda
43      end while
44 end Procedure

1 Procedure SegmentationOfRestString(position)
2      if (position = length of word) return end if
3      if (RestString이 lexicalTable에 존재) { 그 전 과정에서 이미 처리된 상태 }
4          현재 chartNode.child가 이전 node를 가리킨다. { 새로운 chartNode를 생성 안함 }
5          return
6      end if
7      계층 2의 규칙 적용. { 분절된 형태소와 잔여 스트링 사이의 음운 현상 처리 }
8      agenda ← GetAgenda(position)
9      SegmentationOfFirstString(postion)
10 end Procedure
```

**예제 3** 어절 "도와"의 형태소 분절 결과



이 분절 알고리즘의 효율은 리스트 차트 chart와 어휘 참조 테이블 lexicalTable에 의해 유지된다. 이 두 데이타 구조는 형태소 분절 과정에서 일어나는 같은 어휘에 대한 재탐색과 같은 중복 계산을 최대한 방지하게 한다. 이것에 의해 알고리즘의 계산적인 복잡도는 최악의 경우 $O(n^2)$이다. 여기에서 n은 입력 word의 길이이다. 이것에 대한 증명은 다음과 같다.

- (증명) 먼저, 1개의 글자가 1개의 형태소를 이룰 수 있고, n개의 글자로 된 단어가 $2^{n-1}$개의 형태소 배열이 가능한 최악의 경우에 대하여 고려해 보자.

  1. n이 0일 때, 사전 탐색 비용 $T_0 = 0$.
  2. n이 1일 때, 사전 탐색 비용 $T_1 = 1$.
  3. n이 2일 때, 사전 탐색 비용 $T_2 = 2 + 1 = 3$.
     예를 들어, n이 2인 단어 AB에 대하여 가능한 형태소 배열 A+B와 AB를 보자. TRIE 사전을 이용하면 A와 AB는 한번에 탐색될 수 있으므로 A와 AB를 탐색하는데 드는 비용은 2이다. B는 별도로 탐색해야 하므로 1의 비용이 든다. 따라서, n이 2일 때의 사전 탐색 비용은 3이다.
  4. n이 3일 때, 사전 탐색 비용 $T_3 = 3 + 2 + 1 = 6$.
     예를 들어, n이 3인 단어 ABC에 대하여 가능한 형태소 배열 A+B+C, A+BC, AB+C, ABC에 대하여 보자. TRIE 사전을 이용하면 A, AB, ABC는 한번에 탐색될 수 있으므로, 그 비용은 3이다. 마찬가지로, B와 BC는 한번에 탐색될 수 있으므로, 그 비용은 2이다. 문제는 2개의 C($A+B+$ C, $AB+$ C)를 탐색하는데 드는 비용이 얼마인가에 있다. 보통의 경우에 2개의 C를 탐색하는데 드는 비용은 2이다. 그러나, 이 알고리즘에서는 중복 계산 방지를 위한 lexicalTable 때문에 1의 비용만이 든다. 따라서, n이 3일 때의 사전 탐색 비용은 6이다.
  5. 위 과정을 일반화하면, n일 때의 사전 탐색 비용 $T_n = T_{n-1} + n = \frac{n(n+1)}{2}$.

# 5. 어절 구조와 형태소 배열 검사

형태소 배열 검사는 분절된 형태소들의 가능한 배열들에 대하여 그것이 옳은 배열인지 아닌지를 검사하는 과정이다. 이 형태소 배열 검사에는 어절의 구조가 직접적으로 이용된다. 일반적으로 어절의 구조는 정형의 형태소 배열을 그대로 반영한다. 그러므로 가능한 어절의 모든 구조를 아는 것이 무엇보다도 중요하다.

한국어의 일반적인 어절의 구조는 다음과 같이 BNF 형태로 표현될 수 있다.

```
wordPhrase    ::  singleWord
              |   inflWord
              |   complexWord
singleWord    ::  체언
              |   부사
```

| | | |
|---|---|---|
| | \| | 관형사 |
| | \| | 독립언 |
| inflWord | :: | inflNoun |
| | \| | inflVerb |
| | \| | 부사 조사 |
| inflNoun | :: | 체언 조사 |
| | \| | 체언 체언접미사 조사 |
| | \| | 체언 체언접미사 어말어미 |
| | \| | 체언 어말어미 |
| | \| | 체언 서술격조사 suffStem |
| | \| | 체언 서술격조사 suffStem 조사 |
| | \| | 체언 체언접미사 서술격조사 suffStem |
| | \| | 체언 체언접미사 서술격조사 suffStem 조사 |
| | \| | 체언 조용사 suffStem |
| | \| | 체언 조용사 suffStem 조사 |
| inflVerb | :: | 용언 suffStem |
| | \| | 용언 suffStem 조사 |
| | \| | 용언 용언접미사 suffStem |
| | \| | 용언 용언접미사 suffStem 조사 |
| suffStem | :: | 어말어미 |
| | \| | pends 어말어미 |
| pends | :: | 선어말어미 |
| | \| | pends 선어말어미 |
| complexWord | :: | 접두사 inflNoun |
| | \| | compoundNoun inflNoun |
| | \| | 접두사 compoundNoun inflNoun |
| compoundNoun | :: | 체언 |
| | \| | compoundNoun 체언 |

위와 같은 BNF 형태의 어절의 구조는 형태소 해석 엔진에서의 형태소 배열 검사를 위하여 다음과 같은 두 가지 형태로 변환된다.

- **탐색 절단을 위한 digramTable**
  인접해 나올 수 있는 어휘 범주를 쌍으로 하는 테이블.

- **형태소 배열 검사를 위한 wordStructureAutomaton**
  위의 어절 구조를 표현한 결정적인 유한 오토마타.

형태소 배열 검사는 형태소 분절에서 구한 chart를 입력으로 받아 digramTable과 wordStructureAutomaton을 이용하여 행한다. 개략적인 알고리즘은 다음과 같다.

```
1 Procedure MorphotacticChecking(chart) { child 우선으로 chart 탐색 }
2         if (chart = NULL) return end if
3         { chart 탐색 절단 검사 }
4         digramTable을 이용하여 현재 chartNode와 parent간의 인접 가능성을 검사하여
5         그 둘간의 인접 가능성이 없으면, return한다.
```

```
6           MorphotacticChecking(chart.child)
7               현재 chartNode가 중단 노드이면, wordStructureAutomaton을 이용하여
8               형태소 배열 검사을 수행한다.
9           MorphotacticChecking(chart.sibling)
10 end Procedure
```

**예제 4** *"도와"의 형태소 분절 결과(예제 3)에 대한 형태소 배열 검사.*
먼저 각 형태소의 어휘 범주를 국어대사전[4]에 바탕을 두고 보면 다음과 같다.

- 도*:* 명사, 조사, 접미사, 접두사.
- 와*:* 조사, 부사.
- 돕*:* 동사.
- 아*:* 감탄사, 조사, 접미사, 어미.
- 도와*:* 명사.

이에 대한 형태소 배열 검사 과정은 다음과 같다.

1. 도 *(명사)* + 와 *(조사): wordStructureAutomaton*에 의해 성공.
2. 도 *(명사)* + 와 *(부사): wordStructureAutomaton*에 의해 실패.
3. 도 *(조사): digramTable*에 의해 *chart* 탐색 절단 *(실패).*
4. 도 *(접미사): digramTable*에 의해 *chart* 탐색 절단 *(실패).*
5. 도 *(접두사)* + 와 *(조사): wordStructureAutomaton*에 의해 실패.
6. 도 *(접두사)* + 와 *(부사): wordStructureAutomaton*에 의해 실패.
7. 돕 *(동사)* + 아 *(감탄사): wordStructureAutomaton*에 의해 실패.
8. 돕 *(동사)* + 아 *(조사): wordStructureAutomaton*에 의해 실패.
9. 돕 *(동사)* + 아 *(접미사): wordStructureAutomaton*에 의해 실패.
10. 돕 *(동사)* + 아 *(어미): wordStructureAutomaton*에 의해 성공.
11. 도와 *(명사): wordStructureAutomaton*에 의해 성공.

이 형태소 배열 검사 알고리즘의 계산적인 복잡도는 최악의 경우에 $O(2^m)$으로 분석된다[1]. 여기에서 m은 어절 내의 형태소의 최대수이다. 이러한 결과는 주어진 어절에 대하여 모든 가능한 해석을 출력하기 때문에 나오는 것이다.

# 6. 결론

이 논문에서 우리는 한국어 형태소 해석 시스템 DDAG를 소개했다. DDAG의 주요 특징은 다음으로 요약될 수 있다.

- 동적 프로그래밍 기법에 의한 계산의 중복성 방지.
- 철자 규칙에 의한 철자 변화형 어절의 처리.
- 음절에 기반한 철자 규칙 기술 언어 SBFL 제공.
- 규칙 컴파일러에 의한 SBFL 규칙의 오토마타 자동 생성.
- 알고리즘과 언어학적인 지식의 완전 분리에 의한 시스템의 모듈화.
- 어절 구조에 의한 형태소 배열 검사.
- 어절 구조의 점진적 정의를 위한 문법 기술 언어 지원.
- 사전의 확장성을 고려한 사전 컴파일러의 지원.

DDAG는 Sparc-10의 UNIX 환경에서 C언어로 구현되어 실험되었다. 현재의 시스템 구성 및 성능은 다음과 같다:

- 철자 규칙은 한국어 굴절 형태론에서 발생하는 것만을 대상으로 하여, 계층 1의 규칙이 불규칙 현상 및 축약 현상을 다루는 규칙들로 49개가 작성되고, 계층 2의 규칙이 모음 조화 현상과 '으'-삽입 현상을 다루는 규칙들로 4개가 작성되었다. 이 때, 컴파일된 규칙의 크기는 약 80Kbyte이다.
- 사전은 체언이 49600 단어, 용언이 8100 단어 등 59300 단어로 구성되며, 그것의 컴파일된 사전의 크기는 TRIE 사전이 약 2.6 M이고, INFO 사전이 약 700K이다. INFO 사전이 TRIE 사전보다 작은 이유는 현재 사전의 정보가 어휘 범주 정도의 정보밖에 없기 때문이다. 그러나, 잠정적으로는 INFO 사전이 TRIE 사전보다 훨씬 커질 것이다.
- 한국어 어절의 구조는 5장에서 제시한 정도가 이용되었다.
- 27388개의 유일한 서술 어절[3] (형태소 해석의 입력으로써 가장 복잡한 경우)에 대하여 초당 60 단어 정도를 처리한다. (체언으로만 이루어진 약 50000 단어에 대해서는 초당 114 단어 정도를 처리한다.) 결과에 대한 자세한 분석이 아직 끝나지 않아 정확하게 평가를 내릴 수는 없지만, 대체로 거의 모든 철자 변화를 처리하고, 과잉 해석도 거의 없는 등 만족할 만한 성능을 보이고 있다.

앞으로 해야 할 일들로는 다음과 같은 것들이 있다:

[3] 국민학교 교과서에서 추출.

1. 어휘 범주의 세분류와 그러한 범주에 의한 한국어 코퍼스로부터의 실질적인 어절의 구조를 획득하는 작업.

2. 사전 정보의 보완 및 확장 작업.

3. 분절된 형태소들의 개별적인 의미로부터 전체 의미를 생성하는 모델에 관한 연구.

# 참고 문헌


[1] 김덕봉. A Survey on Morphological Chart Parsing Methods, CS-Seminar-6-15, 1993.

[2] 이성진. Two-Level 한국어 형태소 해석, 한국과학기술원 석사학위논문, 1992.

[3] 이은철. CYK법에 기반한 한국어 형태소 분석에서의 개선 기법, 포항공대 석사학위논문, 1992.

[4] 이희승. 국어대사전, 서울: 민중서림, 1991.

[5] 이희승, 안병희. 한글 맞춤법 강의, 서울: 신구문화사, 1991.

[6] Antworth, E. *PC-KIMMO: A Two-Level Processor for Morphological Analysis,* Occasional Publications in Academic Computing 16, Summer Institute of Linguistics, 1990.

[7] Barton, G. E. *Computational Complexity in Two-Level Morphology,* Proc. of ACL-86, pp. 53-59, 1986.

[8] Holub, A. L. *Compiler Design in C,* Prentice-Hall, 1990.

[9] Koskenniemi, K. *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production,* Ph.D. Thesis, Univ. of Helsinki, 1983.

[10] Ritchie, G. D., G. J. Russell, A. W. Black, and S. G. Pulman. *Computational Morphology,* MIT Press, 1991.