자질 기반 구 구조 문법을 위한 문법 개발 환경
(A Grammar Development Environment for Feature-based APSG)

심 광 섭 † 양 재 형 ††
(Kwangseob Shim) (Jaehyung Yang)

요 약 본 논문에서는 자질 기반 구 구조 문법 형식의 자연어 문법 개발에 필요한 환경을 제공하는 GrammE를 소개한다. 문법 개발 단계에서는 텍스트 형식의 문법을 인터프리트하여 구문 분석을 하기 때문에 문법 수정 후 바로 문법을 테스트할 수 있어 문법 개발이 용이해 진다. 일단 문법 개발이 끝나면 GrammE에 포함된 문법 컴파일러를 이용하여 C++로 쓰여진 구문 분석기 프로그램을 생성할 수 있다. 이렇게 해서 만들어진 구문 분석기는 구문 분석을 필요로 하는 여러 가지 자연어 처리 시스템에 활용할 수 있다. GrammE는 언어 독립적인 시스템이며, 현재까지 한국어 문법과 중국어 문법을 개발하는 데 사용되었다.

키워드 : 문법 개발, 구문 분석, 구문 분석기, 도구, 구 구조 문법, GrammE

Abstract This paper presents *GrammE*, a grammar development environment for feature-based APSG. At the stage of a grammar development, analysis are be done by interpreting the grammar under development, given in a text format, it is relatively easy to diagnose the grammar. Once developed, the grammar is compiled, by using the embedded grammar compiler, into a parser program written in C++. The parser program can be used in various types of natural language processing systems requiring syntactic analysis. GrammE is language-independent, and so far has been used for the development of Korean and Chinese grammars.

Key words : Grammar Development, Syntactic Analysis, Syntactic Parser, Tool, Phrase Structure Grammar, GrammE

1. 서 론

자연 언어 처리의 중요 도구 중의 하나인 구문 분석기는 크게 문법 부분과 엔진 부분으로 구성된다. 문법은 자연 언어를 설명하는 문법 규칙들의 집합이며, 엔진은 문법 규칙을 정해진 절차에 따라 적용해 봄으로써 주어진 문장의 적합성을 판별하고 문장 구조를 생성하는 부분이다. 과거에는 구문 분석기를 하나의 독립된 도구로 보지 않고 전체 자연 언어 처리 시스템의 일부분으로 보았기 때문에 새로운 시스템을 구현할 때마다 매번 구문 분석기를 새로 설계하고 구현하는 경우가 많았다. 그런데 다른 시각에서 보면, 문법은 프로그램이며, 엔진은 이 프로그램을 수행하는 인터프리터(interpreter)이다. 따라서 우리가 프로그램을 짤 때마다 인터프리터를 새로 개발하지 않듯이, 문법을 개발할 때마다 새로 엔진을 개발할 필요는 없다.

범용 구문 분석 엔진만 있다면 엔진을 설계, 코딩, 디버깅하는 데 시간을 낭비하지 않고 모든 노력을 문법 개발에 집중할 수는 있는데, 문법은 일반 프로그램과 달리 하향식으로 설계·구현하는 것이 아니라 (1) 주요 문형에 대한 기본적인 문법을 작성한 다음, (2) 여러 가지 다양한 문장에 대하여 분석을 해 보고, (3) 이 과정에서 과분석 혹은 미분석 등의 문제점이 발견되면 기존 문법을 수정하거나 확장하는 과정을 반복하면서 점진적으로 개발된다. 따라서 이러한 과정을 효과적으로 수행할 수 있는 환경이 갖추어 진다면 문법 개발 기간을 단축할 수 있을 것이다. 문법 개발 단계에서는 위에서 말한 것과 같은 과정이 수없이 반복되므로 분석 속도는 다소 느리더라도 수정·확장된 텍스트 형식의 문법을 구문 분석에 바로 적용해 볼 수 있는 인터프리터 방식의 분석 엔진을 사용하는 것이 편리하다. 그런데 문법 개발 목적이 단순히 문법 개발 실습을 하거나 새로운 언어학

· http://cs.sungshin.ac.kr/~shim/demo를 방문하면 본 논문에서 소개한 문법 개발 환경 GrammE를 내려받을 수 있다.

† 종신회원 : 성신여자대학교 컴퓨터정보학부 교수
shim@sungshin.ac.kr
†† 종신회원 : 강남대학교 컴퓨터미디어공학부
jhyang@kns.kangnam.ac.kr

논문접수 : 2003년 4월 3일
심사완료 : 2004년 8월 14일

이론을 검증하기 위한 것이 아니라 실제 응용 시스템에서 사용하기 위한 것이라면 개발 종료된 문법을 C++ 등과 같은 프로그래밍 언어로 변환하여 분석 속도를 높여줄 필요가 있다.

외국의 경우 이와 유사한 목적으로 개발된 여러 가지 도구들이 있는데, 그 중에서 대표적인 몇 가지 사례를 살펴보겠다. 미국의 스탠포드 대학교에서 개발한 PATR-II는 통합 기반 문법 이론을 실제로 구현해 보기 위한 목적으로 개발된 도구로서, 구 구조 규칙(phrase structure rule)과 자질 제약(feature constraint)에 기반한 문법 기술 형식을 지원한다[1]. 영국의 케임브리지 대학교에서 개발한 GDE(Grammar Development Environment)는 lisp로 구현되었으며 GPSG(Generalized Phrase Structure Grammar)와 유사한 문법 기술 형식을 지원한다[2]. GDE는 문법 컴파일러를 가지고 있으나 이는 메타 문법을 확장하여 리스트 형태의 내부 문법 형식으로 바꾸어 주는 것일 뿐 위에서 말한 것과 같이 분석 속도 향상을 위하여 문법을 컴파일하는 것은 아니다. 독일의 Koblenz-Landau 대학교에서 개발한 GTU(Grammatik-Testumgebung)는 DCG(Definite Clause Grammar), ID/LP (Immediate Dominance/Linear Precedence) 문법, LFG (Lexical Functional Grammar) 등의 문법 형식을 지원한다[3]. 하지만 GTU는 텍스트 형식의 문법을 직접 해석하여 구문 분석을 수행하는 것이 아니라 파서 생성기라고 하는 일종의 컴파일러가 문법을 prolog 프로그램으로 변환하면 prolog 인터프리터가 이를 해석하여 구문 분석을 수행하는 방식을 취하였다. 독일의 Tübingen 대학교에서 개발한 ConTroll은 제약 기반 이론(constraint-based theory)을 구현하는데 사용할 수 있는 문법 개발 도구로서 HPSG (Head-driven Phrase Structure Grammar) 문법을 기술하는 데 적합하다[4]. 이 도구도 GTU와 마찬가지로 문법을 직접 해석하는 것이 아니라 문법을 일단 prolog 프로그램으로 컴파일한 다음 prolog 인터프리터가 이것을 수행하는 방식을 취하였다. 문법을 직접 해석하여 구문 분석을 하는 것이 아니므로 문법 내용이 변경될 때마다 문법을 컴파일해야 하므로 매우 번거롭다. 문법 개발 기간을 단축하기 위하여 문법에서 변경된 부분만 선택적으로 컴파일할 수 있는 점진적 컴파일(incremental compilation) 기법을 도입하기는 하였으나 문법을 수정할 때마다 컴파일해야 하는 번거로움은 피할 수 없다. 일본의 동경 대학교에서 개발한 LiLFeS는 TFS(Typed Feature Structure)를 도입한 통합 (unification) 기반 언어로서 HPSG 문법 기술에 적합하다[5]. 앞에서 본 도구와 달리 LiLFeS는 prolog와 같은 특정 언어 위에서 작동하는 것이 아니라 수행에 필요한 모든 것을 가지고 있는 독립된 언어 시스템이다. 미국의 스탠포드 대학에서 LinGO(Linguistic Grammar Online) 프로젝트의 일환으로 개발된 LKB (Linguistic Knowledge Building) 시스템[6]은 TFS에 기반을 둔 문법과 사전 개발 환경이다. LKB도 HPSG 문법 기술에 적합한데, 이 시스템을 사용하여 개발된 대표적인 문법으로 HPSG 기반의 대규모 영어 문법인 ERG (English Resource Grammar)를 들 수 있다. LKB 시스템은 lisp로 구현되어 있으며 차트 기반의 구문 분석기 및 생성기를 포함하고 있다. 또 문법 정보를 계층화된 유형(type)의 다중 상속 체계로 나타내고 이들 간의 통합(unification) 연산을 효과적으로 지원하며 편리한 그래픽 인터페이스를 제공하지만, 컴파일링이나 기타 효율 개선을 위한 고려가 되어 있지 않으며 대규모의 사전 시스템을 위한 대비가 되어 있지 않은, 주로 이론적 탐구를 위한 도구라고 볼 수 있다. TFS를 도입한 통합 기반 도구로는 LiLFeS나 LKB 외에도 ALE(Attribute Logic Engine)나 Profit(Prolog with Features, Inheritance and Templates) 등이 있으나 이들은 prolog 상에서 수행된다[7,8].

지금까지 기존 문법 개발 도구에 대해서 간략하게 살펴보았는데, 이들은 주로 lisp나 prolog 상에서 수행되며 1980년대 이후 언어학계에서 많은 관심을 받았던 LFG, GPSG, HPSG 등과 같은 형식에 따라 문법을 기술할 수 있는 환경을 제공한다는 점에서 유사성을 발견할 수 있다. 이들과 달리 본 논문에서 제시하는 GrammE는 실제 응용 시스템에서 사용할 수 있는 구문 분석기를 단기간 내에 개발하는 데 사용될 수 있는 도구라는 점에서 차이점이 있다. GrammE에서는 자질 기반의 확장된 구 구조 문법 (augmented phrase structure grammar) 형식에 따라 문법을 기술할 수 있어 언어학에 대한 약간의 지식만 있으면 누구나 쉽게 문법을 개발할 수 있다. 또, GrammE는 독립적으로 수행 가능한 문법 개발 환경으로 이 환경 하에서 문법 규칙의 추가, 삭제, 수정이 용이하여 효과적으로 문법을 개발할 수 있다. 문법 개발이 끝나면 GrammE에 포함된 문법 컴파일러를 이용하여 개발된 문법을 C++ 프로그램으로 변환할 수 있다. 이 프로그램을 C++ 컴파일러로 컴파일하고 GrammE와 함께 제공되는 구문 분석 엔진 라이브러리와 링크시키면 독립적으로 수행 가능한 구문 분석기가 만들어진다. 이렇게 만들어진 구문 분석기는 플랫폼에 구애받지 않고 사용할 수 있으며, 실제 응용 시스템에서 사용할 수 있을 정도로 구문 분석 속도 또한 매우 빠르다. GrammE를 사용하여 최소한의 인력으로 한국어 구문 분석기와 중국어 분석기를 단기간에 성공적으로 개발한 바 있는데, 이렇게 개발된 중국어 분석기는

실시간 번역 서비스를 제공하는 상용 웹 응용 시스템에서 사용되고 있다.

2. GrammE의 구조

GrammE의 대략적인 구조는 그림 1과 같다. 이 그림에서 가는 점선으로 표시한 A 부분은 문법 개발 단계에서 사용되는 인터프리터 (interpreter) 방식의 구문 분석기를 표시한 것이다. 여기서 문법은 자질 기반의 확장된 구 구조 문법(Augmented Phrase Structure Grammar) 형식에 따른 다수의 문법 규칙으로 이루어진 텍스트 파일이며, 구문 분석 엔진은 문법을 적용하여 입력 문장에 대한 파스 트리(parse tree)를 생성하는 프로그램이다. 구문 분석 엔진은 상향식 차트 파싱 알고리즘 (bottom-up chart parsing algorithm)을 C++로 구현한 것으로 언어 독립적이다. 문법 개발 단계에서는 문법 규칙을 수정하거나 추가 또는 삭제하는 등의 작업이 빈번하게 일어나므로 구문 분석 엔진을 싸고 있는 문법 인터프리터 층을 두어 텍스트 형식의 문법 파일을 구문 분석에 바로 적용할 수 있도록 하였다. 이 방법은 문법을 컴파일하여 prolog 등의 프로그램으로 변환한 다음 prolog 인터프리터 상에서 구문 분석을 수행하는 기존 방식에 비하여 문법 개발 절차가 단순하므로 문법 개발 기간을 단축할 수 있다.

GrammE에서는 두 가지 종류의 사전을 이용할 수 있는데, 통사 사전은 하위 범주화(subcategorization) 정보와 같이 구문 분석에 필요한 통사 정보를 제공하는 사전이며, 형태소 사전은 형태소 분석기를 이용할 수 없는 경우 형태소 분석기에서 제공되어야 할 정보를 대신 제공하는 사전이다. 형태소 사전을 이용하면 당장 이용할 수 있는 형태소 분석기가 없더라도 문법을 개발하고 테스트할 수 있기 때문에, 형태소 분석기와 문법 개발을 동시에 진행하다가 형태소 분석기 개발이 완료되면 형태소 사전을 형태소 분석기로 대치할 수 있다. 통사 사전과 형태소 사전은 용도만 다를 뿐 사전 정보를 기술하는 형식은 동일하다.

그림 1에서 굵은 점선으로 표시한 B 부분은 문법 개발이 끝난 후 실제 응용 시스템에서 사용될 구문 분석기 부분을 표시한 것이다. 여기서 컴파일된 문법은 문법 컴파일러를 이용하여 텍스트 형태의 문법을 C++ 프로그램으로 변환한 것을 말한다. 컴파일된 문법을 라이브러리 형태로 제공되는 GrammE의 구문 분석 엔진과 링크하면 이후에는 GrammE 없이 독자적으로 수행할 수 있는 구문 분석기가 된다. 이렇게 만들어진 구문 분석기는 인터프리터 방식으로 구문 분석을 수행할 때에 비하여 분석 속도가 약 3 배 가량 빠르며 분석 중에 소비되는 메모리 양도 대폭 줄어들어 구문 분석기를 필요로 하는

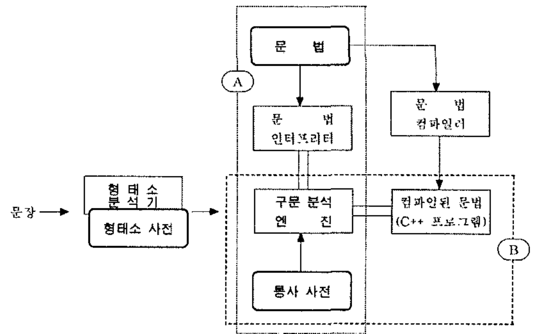


그림 1 GrammE의 구조

응용 시스템에서 사용하기에 적합하다.

그림 1에서 둥근 모서리로 표시한 사전과 문법 부분은 언어 종속적이며, 이것을 제외한 나머지 부분은 언어 독립적이다. 형태소 분석기와 구문 분석 엔진 사이의 인터페이스가 아주 간단하므로 어떤 종류의 형태소 분석기든지 GrammE에 쉽게 연결할 수 있다. 또, 문법을 여러 파일로 나누어 저장할 수도 있으므로 문법을 내용에 따라 구분하여 작성한 다음 각 모듈을 독립적으로 테스트하거나 여러 사람이 공동으로 문법을 개발할 수도 있다.

3. 문법 개발 환경

3.1 자질 구조

GrammE에서 문법을 기술할 때 가정하고 있는 기본 자료 구조는 자질 구조(feature structure)이다. 자질 구조는 이름을 가진 0개 이상의 자질 덩어리로서, 각 자질은 자질 명칭과 자질 값을 가진다. 자질 중에서 + 또는 - 값만을 취하는 것을 특별히 이진 자질(binary feature)이라고 한다. 이진 자질을 제외한 나머지 자질들은 숫자, 스트링, 심볼릭 상수는 물론 이들을 원소로 하는 집합을 값으로 취할 수 있다. 뿐만 아니라 자질 구조 자체를 값으로 취할 수 있으므로 자질 구조는 중첩된 구조가 될 수도 있다. 다음은 자질 구조의 예를 보인 것이다. 여기서 굵은 글씨는 자질 구조의 명칭으로, 이 명칭은 단어나 구의 범주(category)를 나타낸다.

1. (a)
```
PRON
BASE     "he"
AGR      P3SG
CASE     NOM
PROPER   -
```
(b)
```
VERB
BASE     "love"
AGR      P3SG
TENSE    PRES
```
(c)
```
NOUN
BASE     "Mary"
AGR      P3SG
CASE     {NOM, ACC}
PROPER   +
```

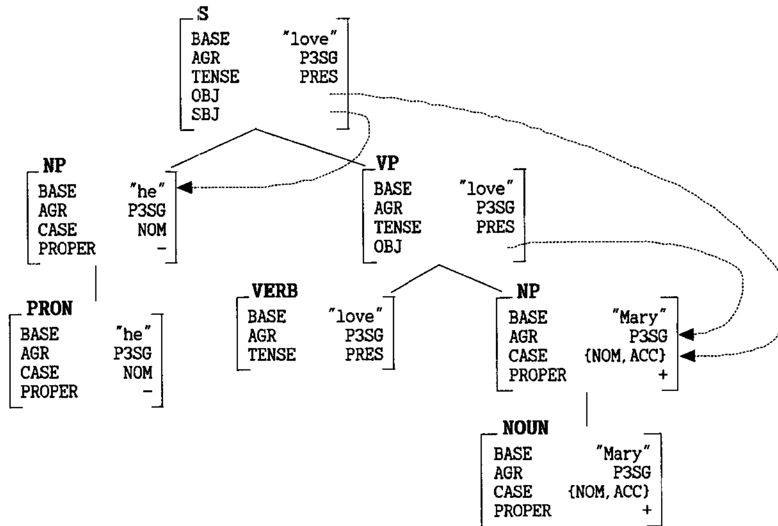


그림 2 구문 분석 결과

위의 예에서 AGR, CASE, TENSE 등의 자질 값으로 주어진 P3SG, NOM, ACC, PRES 등은 심볼릭 상수(symbolic constant)이다. 1의 (c)에서는 CASE란 자질의 값으로 심볼릭 상수의 집합이 주어졌다.

형태소 분석기(또는 형태소 사전)와 통사 사전으로부터 제공된 자질 구조는 구문 분석 과정에서 문법 규칙이 적용되면서 통합된 자질 구조로 발전해 나간다. 문법 규칙이 적용되기 위해서는 문법 규칙의 왼쪽에 명시된 제약 조건이 만족되어야 한다. 모든 제약 조건이 만족되면 문법 규칙의 오른쪽에 명시된 방법에 따라 새로운 통합된 자질 구조가 만들어진다. 그림 2는 "He loves Mary."에 대한 구문 분석 결과를 나타낸 것이다.

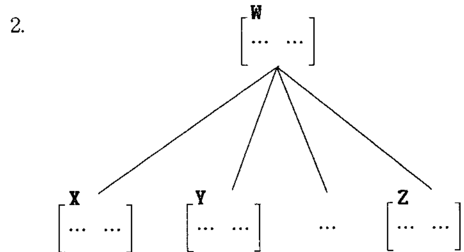
3.2 문법

문법은 자질 기반의 확장된 구 구조 문법 형식에 따른 다수의 문법 규칙으로 이루어진 텍스트 파일이다. 이 문법 파일의 도입부에서는 문법 규칙을 기술하는 데 사용되는 각종 명칭을 선언해 두어야 한다. 먼저 문법 규칙 기술 방법에 대해서 알아본 다음 명칭 선언에 대해서 설명하겠다. 각 문법 규칙은 다음과 같은 형식에 따라 기술한다.

```
<ruleid> X(constraints) Y(constraints) …
Z(constraints) -> W(actions)
```

ruleid는 각 문법 규칙을 유일하게 식별할 수 있는 식별자이다. 이 식별자는 나중에 편집을 위해서 특정 문법 규칙을 불러오거나 디버깅을 위해 각 문법 규칙의 적용 여부를 개별적으로 통제하는 데 필요하다. 또한 구문 분석 결과에서 어느 부분에 어떤 문법 규칙이 적용되었는가를 표시하는 데에도 이 식별자가 사용된다. X, Y, Z, W 등은 자질 구조의 명칭 즉 단어나 구의 범주 명칭이다. 자질 구조 X, Y, Z 등이 문법 규칙에 명시되어 있는 제약 조건 constraints를 만족할 경우 다음 그림과 같이 새로운 자질 구조 W가 생성되는데 이 자질 구조는 actions에 의해 그 모양이 결정된다.

2.



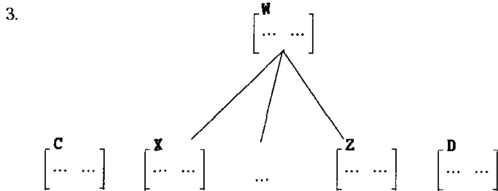
자연 언어를 문맥 자유 문법(context-free grammar)으로 기술할 수 있냐 없냐에 대한 이론적 논란은 아직 끝나지 않았지만, 문맥 의존 문법(context-sensitive grammar)을 효율적으로 처리할 수 있는 알고리즘은 개발되지 않은 반면 문맥 자유 문법을 효율적으로 처리할 수 있는 알고리즘은 개발된 상태이기 때문에 자연 언어를 문맥 자유 문법으로 기술할 수 있다고 보고 문법을 개발하는 것이 보통이다[9,10]. 그런데 실제로 문법을 개발하다 보면 문맥 의존 문법적 요소가 필요한 경우도 있으므로, 비록 제한적이기는 하지만 다음과 같이 문맥 의존 문법 규칙도 표현할 수 있도록 하였다.

```
<ruleid> @C(constraints) X(constraints) …
Z(constraints) @D(constraints) -> W(actions)
```

자질 구조 명칭 앞에 @ 기호를 붙이면 이 자질 구조는 문법 규칙 적용 여부를 판정하기 위한 문맥 역할만

할 뿐 새로 생성될 자질 구조 W의 자식(daughter)에 포함되지는 않는다. 다시 말해서 위 문법 규칙이 적용되려면 자질 구조 X의 왼쪽에는 자질 구조 C가 나와야 하며 자질 구조 Z의 오른쪽에는 자질 구조 D가 나와야 한다. 물론 C와 D에 주어진 constraints가 모두 만족되어야 함은 당연하다. 다음은 이러한 상황을 도시한 것이다.

3.



@ 기호는 문법 규칙 좌측의 양쪽 끝에 최대 한 번씩만 올 수 있으며, 구 수준의 자질 구조 앞에서는 @ 기호를 사용할 수 없다. 이와 같이 제한적으로 문맥을 표현할 수 있도록 한 것은 위에서 말한 것처럼 일반적인 문맥 의존 문법을 위한 효율적인 구문 분석 알고리즘이 아직 개발되지 않았기 때문이다.

4는 아주 간단한 영어 문법을 작성한 예이다. 실제 문법을 개발할 때에는 아래 예보다 훨씬 복잡한 문법 규칙을 정의해야 할 것이다. 논리적 AND를 뜻하는 콤마(,)와 논리적 OR를 뜻하는 수직선(|) 및 괄호를 사용하면 복잡한 형태의 constraints도 표현할 수 있다. 복잡한 형태의 actions를 표현할 때에도 콤마가 사용되는데, 이 때에는 논리적 AND가 아니라 actions의 각 성분을 좌에서 우로 순서대로 처리하라는 의미로 사용된 것이다.

4. (a) <NP1> PRON() -> NP(%PRON)
   (b) <NP2> NOUN() -> NP(%NOUN)
   (c) <VP1> VERB(subcat =~ V1) -> VP(%VERB)
   (d) <VP2> VERB(subcat =~ V2) NP(CASE =~ ACC)
   -> VP(%VERB, OBJ = NP)
   (e) <S1> NP(CASE =~ NOM) VP(AGR =~ AGR(NP))
   PUNC(BASE in {".", "?", "!"})
   -> S(%VP, SBJ = NP)

4에서 subcat은 통사 사전에서 제공되는 하위 범주화 정보를 나타내는 자질이다. (c)와 (d)에서는 subcat 정보를 이용하여 자동사가 목적어를 취한다든지 타동사가 목적어 없이 VP가 되는 것을 막았다. (e)에서는 주어와 동사 사이의 인칭과 수에 대한 검사를 함으로써 "He like Mary."와 같이 인칭과 수가 일치하지 않는 경우에는 S가 생성되지 않도록 하였다.

문법 규칙의 화살표 오른쪽에 나오는 %는 중심어(head)의 자질 구조를 새로 생성되는 부모 자질 구조로 복사하는 데 사용되는 연산자이다. 구문 분석 과정에서 수많은 자질 구조가 생성되는데 새로운 부모 자질 구조가 생성될 때마다 중심어의 자질 정보를 모두 복사해 오는 단순한 방법은 구문 분석 속도를 저하시킬 뿐만 아니라 분석 과정에서 많은 메모리를 낭비하는 요인이 되므로 피해야 한다. 또, 차트 파싱 알고리즘에서 구문 분석 중에 생성된 자질 구조들은 여러 분석 구조에 의해 공유되므로 문법 규칙에 의해 기존 자질 구조의 값이 변경되어야 하는 경우 이 변경은 비파괴적(non-destructive)이어야 한다. 이 두 가지 사실을 고려하고, GrammE에서 이진 자질은 비트(bit)로 표현되며 일반 자질은 연결 리스트(linked list)로 표현된다는 점을 감안하여, 새로운 부모 자질 구조가 생성될 때에는 그림 3과 같이 이진 자질은 복사를 하되 일반 자질은 공유하는 것으로 하였다.

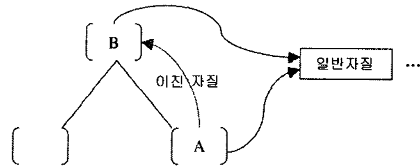


그림 3 자질 구조 복사 및 공유

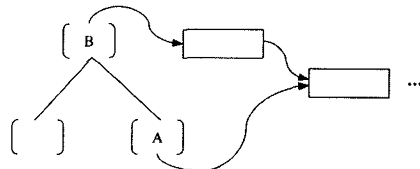


그림 4 일반 자질의 추가 또는 변경

이진 자질은 복사를 하기 때문에 그림 3의 B에서 기존의 이진 자질 값을 변경하거나 새로운 이진 자질을 추가하더라도 A를 공유하는 다른 분석 구조에 아무런 영향을 미치지 않는다. 한편, B에서 일반 자질 값을 변경하거나 새로운 일반 자질을 추가하는 경우 그림 4와 같이 기존 자질 리스트 앞에 삽입되므로 B에서의 어떤 변경을 하더라도 이는 A를 공유하는 다른 분석 구조에 아무런 영향을 미치지 않는다.

4에서 보듯이 문법 규칙에서는 여러 가지 명칭들이 사용되었다. 이러한 명칭은 사용 전에 문법 파일에서 선언되어 있어야 한다. 명칭을 선언해야 하는 이유는, 이렇게 함으로써 각 명칭의 용도를 분명히 할 수 있어 효율적인 구문 분석이 가능하며, 오타로 인해 엉뚱한 명칭이 사용되었을 경우 이를 쉽게 발견할 수 있기 때문이다. 예를 들어 4의 (c)에서 subcat나 V1 등의 명칭을 다음과 같이 잘못 사용한 경우,

5. (a) <VP1> VERB(suvcat =~ V1) -> VP(%VERB)
   (b) <VP1> VERB(subcat =~ B1) -> VP(%VERB)

suvcat나 B1이 다른 용도로 선언되지 않았다면 문법을 적재할 때 이러한 오류를 발견하여 문법 개발자에게 알려 줄 수 있다. 만약 명칭 선언을 하지 않으면 suvcat나 B1이 오타로 인한 오류인지 문법 개발자가 의도적으로 사용하는 명칭인지 구분할 수가 없다. 전자의 경우라면 문법 개발자의 의도와 다른 분석 결과가 생성될 것이다. 각종 명칭은 다음과 같은 형식에 따라 선언한다.

% type id {id}*

id는 명칭을 나타내며 하나 이상이 나와야 한다. type은 선언할 명칭의 타입을 나타낸다. 사용 가능한 타입으로 NODE, PHRASE, BFEATURE, NONHEAD_BFEATURE, VFEATURE, CONST, DLINK 등이 있다. NODE나 PHRASE는 단어나 구의 범주를 나타내는 명칭을 선언할 때 사용된다. BFEATURE는 + 또는 - 값만을 취할 수 있는 이진 자질의 명칭을, VFEATURE는 임의의 값을 취할 수 있는 일반 자질의 명칭을 선언하는 데 사용된다. 머리어(head)의 자질 구조를 부모 노드로 복사할 때 제외되어야 하는 이진 자질이 있다면 이러한 이진 자질은 BFEATURE 대신 NONHEAD_BFEATURE로 선언한다. CONST는 심볼릭 상수를 선언하는 데 사용된다. 심볼릭 상수는 내부적으로 정수로 바뀌어 처리된다. 1에서 P3SG, NOM, ACC와 같은 심볼릭 상수 대신 "P3SG", "NOM", "ACC"와 같은 스트링을 사용해도 똑같은 정보를 표현할 수 있지만 비교 연산을 하거나 자질을 복사할 때 정수에 비하여 시간이 많이 걸리므로 심볼릭 상수를 사용하는 것이 바람직하다. GrammE에서 단어 의존 문법(word dependency grammar)에 따른 의존 관계를 표현할 수도 있는데 DLINK는 의존 관계를 나타내는 명칭을 선언하는데 사용된다. 다음과 같이 (한국어) 문법을 작성했을 때,

6. NN(+sbjmark) VV(-SBJ) -> VV(SBJ = NN)
   NN(+objmark) VV(-OBJ) -> VV(OBJ = NN)

SBJ와 OBJ를 일반 자질로 선언하면 그림 5의 (a)와 같은 구조가 생성되는 데 반하여 이것을 DLINK로 선언하면 그림 5의 (a)와 같은 구 구조는 물론 (b)와 같은 의존 구조도 동시에 생성된다. 후자의 경우 두 가지 구조가 별도 생성되는 것이 아니라 (b)와 같은 구조로도 해석이 가능한 하나의 구조를 생성하는 것이므로 구문분석시 부담해야 하는 별도 비용은 거의 없다고 볼 수 있다.

(a) 구 구조 (b) 의존 구조

그림 5 구문 분석 구조

문법 규칙을 기술할 때 반복적으로 나타나는 표현을 의미 있는 이름으로 바꾸면 간결하고 읽기 쉬운 문법 규칙을 작성할 수 있다. 이것은 프로그래밍 언어에서 매크로(macro)를 사용하는 것과 유사하다. GrammE에서 매크로는 다음과 같은 형식으로 정의한다.

% DEFINE id = definition

DEFINE은 매크로 정의를 알리는 키워드이며 id는 매크로의 명칭이다. definition은 문법 규칙에서 id가 사용되었을 때 id대신 치환될 내용을 정의하는 부분이다.

문법 규칙을 기술하다 보면 문법 규칙의 내용은 동일하지만 이 규칙이 적용되는 범주가 다르기 때문에 따로 문법 규칙을 작성해야 하는 경우가 있다. 예를 들어, 한국어의 경우 체언을 명사, 대명사, 수사, 의존 명사 등과 같은 범주로 구분했을 때 각 범주에 대하여 따로 적용되어야 하는 문법 규칙이 있는가 하면 모든 체언에 대하여 공통적으로 적용되어야 하는 문법 규칙도 있다. 만약, 모든 체언에 대하여 공통적으로 적용되는 문법 규칙을 다음과 같이 범주별로 따로 작성해야 한다면,

7. NN(…) -> NP(…)
   PN(…) -> NP(…)
   NU(…) -> NP(…)
   NX(…) -> NP(…)

처음에 이러한 문법 규칙을 작성하는 것도 번거로운 일이지만, 그 보다는 나중에 체언에 대하여 공통적으로 적용되는 문법 규칙을 수정할 필요가 생길 때마다 위와 같은 네 가지 문법 규칙을 일일이 찾아다니며 수정해야 한다는 것은 대단히 번거로운 일이다. 이러한 번거로움을 피하고자 GrammE에서는 문법적으로 유사한 역할을 하는 범주들을 가상 범주로 통칭하고 이것을 기준으로 문법 규칙을 작성할 수 있도록 하였다. 가상 범주는 다음과 같은 형식으로 정의한다.

% PSEUDO pid = id {| id}+

여기서 pid는 가상 범주 명칭이며 이것은 뒤에 열거된 범주 명칭 id를 통칭하는 역할을 한다. 8에서 (a)는 NN, PN, NU, NX를 통칭하는 ALLN이라는 가상 범주를 정의한 것이며, (b)는 이 가상 범주를 이용하여 7을 다시 작성한 것이다. 이 문법 규칙은 내부적으로 7과 같은 네 개의 문법 규칙으로 확장된다.

8. (a) % PSEUDO ALLN = NN | PN | NU | NX
   (b) ALLN(…) -> NP(…)

지금까지 아주 간단한 문법을 예로 들어 문법을 작성하는 방법에 대하여 알아 보았는데, 문법을 작성하는 구체적인 방법은 [11,12]을 참고하면 된다.[1)]

1) Grammar Writer는 GrammE의 이전 버전이다.

3.3 사전

GrammE에서는 형태소 정보를 제공하는 형태소 사전과 하위 범주화 정보와 같이 구문 분석에 필요한 통사 정보를 제공하는 통사 사전을 둘 수 있도록 하였다. 사전 이용 여부는 어디까지나 선택적인 것으로서, 이용 가능한 형태소 분석기만 있다면 이것을 GrammE에 연결하여 사용할 수 있으므로 형태소 사전을 따로 개발할 필요가 없다. 뿐만 아니라 형태소 분석기가 통사 정보까지 제공한다면 통사 사전도 불필요하다. 형태소 사전이나 통사 사전은 다음과 같은 형식에 따라 만들어진 텍스트 파일이다. 이 사전은 GrammE를 수행할 때 메모리 내에 트라이 (trie) 구조로 적재된다.

```
% LEXICON type
% SEARCHKEY id
word : {cat(feature_descriptions)}+
```

type은 사전의 종류를 나타내는데, 형태소 사전인 경우에는 MORPH라고 지정하며 통사 사전인 경우에는 SYNTAX라고 지정한다. GrammE는 형태소 분석 기능을 가지고 있지 않으며, 문장이 주어지면 이것을 공백 문자를 기준으로 단어 단위로 분리한 다음 형태소 사전으로부터 구문 분석에 필요한 형태소 정보를 가져오기만 하므로 형태소 사전에는 단어의 모든 활용형이 등재되어 있어야 한다. 그런데 통사 사전의 경우에는 기본형에 대해서만 통사 정보가 기술되어 있으므로, 단어의 기본형이 어떤 자질에 저장되어 있는지를 알아야 통사 사전을 탐색할 수 있다. SEARCHKEY는 통사 사전에서만 적용되는 것으로 단어의 기본형이 저장된 자질명을 지정하는 데 사용된다. word는 사전에 등재할 표제어를 나타내며 cat는 이 표제어의 범주를 나타낸다. feature_descriptions는 표제어에 대한 형태 또는 통사 정보를 문법 규칙을 기술할 때와 같은 자질 구조 형식으로 표현한 것이다. 9의 (a)와 (b)는 각각 형태소 사전과 통사 사전을 작성한 예이다.

형태소 사전이든 통사 사전이든 문법에서 필요로 하는 정보를 제공하는 기능을 가지고 있으므로, 사전 정보를 기술할 때에는 문법에서 정의된 명칭만 사용할 수 있다.

앞서 말한 바와 같이 형태소 분석기가 있으면 형태소 사전이 필요 없으며 경우에 따라서는 통사 사전도 필요 없을 수 있다. 사전 대신 형태소 분석기를 이용하려면 어떤 방식으로든 형태소 분석기를 GrammE와 연결하여야 한다. GrammE와의 연결을 위한 자료 구조를 설정해 두고 이를 수용하는 형태소 분석기만 연결하는 방법도 있겠지만, 자료 구조를 설정하는 대신 그림 6과 같이 형태소 분석 결과를 사전과 유사한 형식의 스트링으로 변환하여 GrammE에 전달하도록 하였다.

이 그림에서 ①은 형태소 분석 결과가 내부 자료 구조로 표현된 것을 나타내며, ②는 변환 모듈에 의해 생성된 스트링을 나타낸다.

3.4 사용자 인터페이스

사용자 인터페이스로 현재는 명령어 방식만을 지원하고 있다. 이들 명령어는 문법과 관련된 명령어, 구문 분석과 관련된 명령어, 분석 결과 보기와 관련된 명령어,

9. (a)
```
% LEXICON MORPH
he : PRON(BASE="he", AGR=P3SG, CASE=NOM, -PROPER)
Mary : NOUN(BASE="Mary", AGR=P3SG, CASE={NOM,ACC}, +PROPER)
love : VERB(BASE="love", AGR={P1SG, P2SG, P1PL, P2PL, P3PL}, TENSE=PRES)
loves : VERB(BASE="love", AGR=P3SG, TENSE=PRES)
want : VERB(BASE="want", AGR={P1SG, P2SG, P1PL, P2PL, P3PL}, TENSE=PRES)
       NOUN(BASE="want", AGR=P3SG, CASE={NOM, ACC}, -PROPER)
wants : VERB(BASE="want", AGR=P3SG, TENSE=PRES)
        NOUN(BASE="want", AGR=P3PL, CASE={NOM,ACC}, -PROPER)
```

(b)
```
// V1: VERB, V2: VERB+NP, V3: VERB+NP+NP
// V4: VERB+INF, V5: VERB+NP+INF
% LEXICON SYNTAX
% SEARCHKEY base
love : VERB(subcat=V2)
want : VERB(subcat={V2, V4, V5})
```

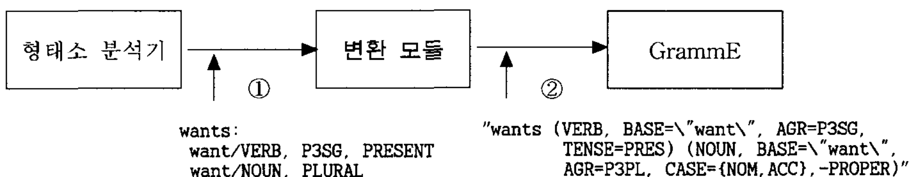


그림 6 형태소 분석기와 GrammE와의 연결

기타 명령어로 구분할 수 있다. 문법과 관련된 명령어는 다음과 같다.

10. (a) load file {file}*
    (b) save file
    (c) ed ruleid
    (d) add [ruleid]
    (e) enable ruleid
    (f) disable ruleid

load는 지정된 문법 파일이나 사전을 GrammE의 작업 메모리로 적재하는 데 사용되는 명령어이며, save는 개발 중인 문법을 지정된 파일로 저장하는 데 사용되는 명령어이다. ed는 지정된 문법을 편집기로 불러와 편집하는 데 사용되는 명령어인데, ruleid는 문법 규칙 식별자를 나타낸다. add는 새로운 문법 규칙을 추가할 때 사용되는 명령어이다. enable과 disable는 지정된 문법 규칙을 활성화시키거나 일시적으로 비활성화시키는 데 사용되는 명령어이다. 이 명령어를 이용하면 문법 파일에서 특정 문법 규칙을 실제로 삭제하지 않더라도 이것을 일시적으로 배제한 채 구문 분석을 시도해 볼 수 있다. 구문 분석과 관련된 명령어는 다음과 같다.

11. (a) run file [nth]
    (b) rerun
    (c) next
    (d) prev
    (e) **
    (f) *
    (g) dribble [file]

run은 지정된 파일로부터 문장을 차례로 읽어 들여 구문 분석을 수행할 때 사용되는 명령어이다. nth를 명시한 경우에는 해당 파일에서 nth번째 문장만 구문 분석을 한다. rerun은 좀 전에 분석했던 문장을 다시 분석할 때 사용되는 명령어이다. run 명령어를 사용하여 nth번째 문장에 대한 구문 분석을 했으면 그 다음부터는 next나 prev 명령어를 사용하여 보다 편리하게 다음 문장 또는 이전 문장을 분석할 수 있다. **는 직전에 분석한 문장을 보여 주는 명령어이며, *는 12에서 보는 바와 같이 직전에 분석한 문장의 단어 사이에 단어 위치를 나타내는 숫자를 넣어 보여 주는 명령어이다.

12. 0 I 1 liKe 2 Mary 3 . 4

단어 위치를 나타내는 숫자는 부분 문장에 대한 분석 구조를 검사할 때 이 부분 문장의 범위를 지정하는 데 필요하다. dribble은 화면으로 출력되는 내용을 지정된 파일에 저장되도록 할 때 사용되는 명령어이다. dribble을 종료하려면 file을 지정하지 않고 이 명령어를 수행하면 된다. 구문 분석 결과 보기와 관련된 명령어는 다음과 같다.

13. (a) print [nth] / lprint [nth]
    (b) print n1 n2 / lprint n1 n2
    (c) print morph
    (d) bfeature [all ¦ no ¦ [+¦-]id]
    (e) vfeature [all ¦ no ¦ [+¦-]id]
    (f) show [phrase ¦ dlink]

print 명령어는 인자를 어떻게 지정하는가에 따라 몇 가지 다른 용도로 사용된다. print 명령어를 (a)와 같은 방법으로 사용하면 다중 분석 구조가 생성되었을 때 그 중에서 nth 번째 분석 구조를 화면으로 출력한다. 만약 nth를 지정하지 않으면 생성된 분석 구조를 모두 다 출력한다. 또 (b)와 같이 사용하면 주어진 문장에서 n1 번째 단어와 n2 번째 단어 사이의 부분 분석 결과를 화면으로 출력한다. lprint 명령어는 print 명령어와 유사하나 비단말 노드에 대한 자질 구조를 볼 필요가 있을 때 사용된다. 14에서 (a)는 구문 분석 결과를 print 명령어로 출력한 것이며, (b)는 lprint 명령어로 출력한 것이다.

lprint로 구문 분석 결과를 출력하면 구(phrase)의 범위와 이 구가 생성되는 데 적용된 문법 규칙을 알 수 있다. 예를 들어, 14의 (b)에서 VP는 주어진 문장에서 단어 1과 단어 3 사이를 포괄하는 구이며 이 구는 문법 규칙 VP2가 적용되어 생성된 것이다. 한편, print 명령을 13의 (c)와 같이 사용하면 형태소 분석 결과를 볼 수 있다. bfeature는 구문 분석 구조를 화면으로 출력할 때 보여 줄 이진 자질을 제어하는 데 사용되는 명령어이다. all은 문법에서 정의된 모든 이진 자질을 보여주도록 설정할 때 사용되며, no는 이진 자질을 출력하지 않도록 설정할 때 사용된다. 이진 자질 명칭 앞에 +나 -를 붙여 특정 이진 자질의 출력 여부를 제어할 수도 있

```
14. (a) (S (NP (PRON* (BASE "he") (AGR P3SG) (CASE NOM) -PROPER))
           (VP* (VERB* (BASE "love") (AGR P3SG) (TENSE PRES) (subcat V2))
                (NP (NOUN* (BASE"Mary") (AGR P3SG) (CASE {NOM, ACC}) +PROPER)))
           (PUNC (base ".")))

    (b) (S [0:4] S1 (BASE "love") (AGR P3SG) (TENSE PRES) (subcat V2) (SBJ NP[0:1]) (OBJ NP[2:3])
           (NP [0:1] NP1 (BASE "he") (AGR P3SG) (CASE NOM) -PROPER
               (PRON [0:1] (BASE "he") (AGR P3SG) (CASE NOM) -PROPER))
           (VP [1:3] VP2 (BASE "love") (AGR P3SG) (TENSE PRES) (subcat V2) (OBJ NP[2:3])
               (VERB [1:2] (BASE "love") (AGR P3SG) (TENSE PRES) (subcat V2))
               (NP [2:3] NP2 (BASE "Mary") (AGR P3SG) (CASE {NOM, ACC}) +PROPER
                   (NOUN [2:3] (BASE "Mary") (AGR P3SG) (CASE {NOM, ACC}) +PROPER)))
           (PUNC [3:4] (BASE ".")))
```

다. vfeature 명령어를 사용하여 일반 자질도 이진 자질과 마찬가지로 출력 여부를 제어할 수 있다. show는 구문 분석 결과를 구 구조 형식(phrase)으로 출력할 것인지 의존 구조 형식(dlink)으로 출력할 것인지를 선택하는 데 사용되는 명령어이다. 14의 (a)를 의존 구조 형식으로 출력하면 15와 같다. 의존 구조 형식으로 출력하려면 문법을 작성할 때 SBJ, OBJ 등을 DLINK 타입으로 선언해 두어야 한다.

```
15. ((SBJ PRON (BASE "he") (AGR P3SG) (CASE NOM)
        -PROPER)
    (OBJ NOUN (BASE "Mary") (AGR P3SG) (CASE
        {NOM, ACC}) +PROPER)
    VERB (BASE "love") (AGR P3SG) (TENSE PRES)
        (subcat V2))
```

4. 구문 분석기 생성

문법 개발 단계에서는 구문 분석 속도가 그다지 중요하지 않지만 문법 개발이 종료된 후 실제 응용 시스템에 사용하는 단계에 이르면 구문 분석 속도가 매우 중요하다. 따라서 문법 개발 단계에서 사용했던 속도가 느린 인터프리터 방식의 구문 분석기는 실제 응용 시스템에 사용하기에 부적절하다. 이 때문에 GrammE는 개발이 종료된 문법을 컴파일하여 C++ 프로그램으로 변환할 수 있는 문법 컴파일러를 지원한다. 컴파일된 문법 즉 C++ 프로그램을 C++ 컴파일러로 컴파일하고 구문 분석 엔진과 링크시키면 완전한 형태의 구문 분석기가 만들어진다.

문법을 컴파일하면 3 개의 헤더 파일과 3 개의 프로그램 파일이 생성된다. 헤더 파일은 문법을 나타내는 C++ 클래스 CCompiled가 정의된 파일과 문법에서 정의한 각종 명칭들이 C++ 심볼릭 상수로 정의된 파일을 포함한다. 프로그램 파일은 문법을 나타내는 CCompiled 클래스의 기본 함수들이 정의된 파일과 개개의 문법 규칙이 C++ 함수로 변환되어 저장된 파일, 그리고 각 문법 규칙 및 명칭을 적재하는 함수가 정의된 파일을 포함한다. 하나의 문법 규칙을 이루고 있는 각 구(phrase)는 별도의 C++ 함수로 컴파일된다.

그림 6은 문법 컴파일러로 4의 (e)를 컴파일하여 얻은 결과이다. 여기서 arc_t는 차트 파싱 알고리즘에서 아크(arc)를 나타내는 데이타 타입이며, constituent_t는 구 구조에서 각 성분을 나타내는 데이타 타입이다. 이들 데이타 타입에 대해서는 [11]에 자세하게 설명되어 있다.

5. 개발 사례 및 성능 평가

GrammE에서는 자질 구조를 기본으로 하고 문맥 자유 문법(context-free grammar) 범주에 속하는 확장된 구 구조 문법 형식으로 문법 규칙을 기술한다. 따라서 GrammE는 언어 독립적이며 문맥 자유 문법으로 표현 가능한 자연 언어의 문법을 개발하는데 사용될 수 있는데, 현재까지 GrammE는 한국어와 중국어 문법을 개발하는데 사용된 바 있다. 한국어 문법은 내용 기반 멀티

```
/* C++ code for Rule <S1> */

bool CCompiled::NP_f1(arc_t *arc, constituent_t *c) /* S1#1 */
{
     if (MATCH(grGetValue(VFID_CASE, c), SCID_NOM))
        return true;
     else return false;
}

bool CCompiled::VP_f2(arc_t *arc, constituent_t *c) /* S1#2 */
{
     if (MATCH(grGetValue(VFID_AGR, c), grGetValue(VFID_AGR, grGetChild(1, arc))))
        return true;
     else return false;
}

bool CCompiled::PUNC_f3(arc_t *arc, constituent_t *c) /* S1#3 */
{
     if (MEMBER(grGetValue(VFID_BASE, c), ".", "?", "!", NULL))
        return true;
     else return false;
}

bool CCompiled::S_f4(arc_t *arc, constituent_t *c) /* S1 */
{
     if (grCopy(grGetChild(2, arc), c) &&
        grSetValue(grGetLValue(VFID_SBJ, c), grGetChild(3, arc))) return true;
     else return false;
}
```

그림 7 컴파일된 문법 규칙

미디어 정보 검색을 위하여 개발된 것으로서, 단어 사이의 의존 관계를 파악할 수 있는 총 164 개의 문법 규칙으로 이루어져 있다. 이 문법에서는 자질 구조 표현을 위해 25 개의 이진 자질과 15 개의 일반 자질이 사용되었다.[2] 한국어 문법이 연구 목적으로 개발된 것이라면, 중국어 문법은 상용 번역 시스템에서 사용할 목적으로 개발된 것이다. 중국어 문법은 총 514 개의 문법 규칙으로 이루어져 있으며, 74 개의 이진 자질과 36 개의 일반 자질이 사용되었다.[3] 이 밖에도 최근에는 GrammE를 사용하여 영어 문법도 개발하고 있는 중이다.

문법을 개발하는 동안에는 문법 개발의 편의성을 도모하기 위하여 GrammE가 인터프리터 방식으로 수행되지만, 개발이 완료되어 응용 시스템의 한 모듈로 사용되는 단계에 이르면 GrammE에 포함된 문법 컴파일러로 문법을 C++ 프로그램으로 변환한다.[4] 이렇게 해서 얻은 C++ 프로그램을 컴파일한 후 라이브러리 형태로 제공되는 GrammE의 구문 분석 엔진과 링크하면 응용 시스템의 한 모듈로 이식하여 사용할 수 있는 구문 분석기를 얻을 수 있다. 이러한 방법으로 위에서 언급한 한국어와 중국어 문법에 대하여 구문 분석기를 만들고, 이 분석기의 구문 분석 속도 및 메모리 사용량에 대한 분석을 실시하였다. 이 분석은 평균 길이가 10.9 어절인 124 개의 한국어 문장과 평균 길이가 13.8 단어인 285 개의 중국어 문장으로 1.13 GHz의 Pentium III CPU와 512 MB의 메인 메모리를 가진 PC 상에서 실시되었다. 실험 결과, 한국어 구문 분석기의 경우 초당 230 문장(2,500 어절)을 분석하며 한 문장을 분석하는데 평균 75 KB의 메모리가 사용되었다. 중국어 구문 분석기의 경우에는 초당 216 문장(3,000 단어)을 분석하며 한 문장을 분석하는데 평균 132 KB의 메모리가 사용되었다. 참고로 문법 개발 단계에서 인터프리터 방식으로 GrammE를 사용할 때에는 위에서와 같은 문장 집합에 대하여 초당 약 70-80 문장씩 분석할 수 있었다.

실제 문서를 보면 길이가 긴 문장들도 꽤 많다. 그래서 이번에는 비교적 긴 문장을 대상으로 구문 분석 속도를 측정하는 실험을 해 보았다. 구문 분석 속도는 문장 길이뿐만 아니라 문법 크기와도 관련이 있으므로 이 실험에서는 한국어 문법에 비하여 완성도도 높고 문법 규모도 큰 중국어 문법을 사용하였다. 이 실험을 위해 285 개의 중국어 문장 가운데 20 단어 이상인 문장을 모두 추출하였다. 추출된 문장은 총 50 개였는데, 평균 길이는 22.1 단어였고, 가장 긴 문장은 26 단어였다. 동일한 중국어 구문 분석기를 사용하여 이들 문장을 분석했을 때 초당 102 문장(2,250 단어)을 분석할 수 있었고, 한 문장을 분석하는데 사용된 메모리는 평균 324 KB였다.

구문 분석 속도는 문법 규칙의 개수와 관련이 있으며, 문법 규칙 개수가 늘어남에 따라 구문 분석 속도도 저하된다. 하지만 문법 규칙의 개수가 많다고 해서 구문 분석기를 실제 응용 시스템에 사용하기에 부적절할 정도로 분석 속도가 느려지는 것은 아니다. 일례로 중국어 문법의 경우 문법 규칙의 개수가 500 개를 넘을 정도로 규모가 꽤 큰 편이지만 웹 상에서 실시간 번역 서비스를 제공하는 응용 시스템에서 사용될 수 있을 정도로 분석 속도가 매우 빠르다. 여기서 문법 규칙의 개수가 더 늘어나더라도 이러한 규칙은 매우 제한적인 조건에서 적용되는 것들이 대부분이기 때문에 이로 인해 분석 속도가 저하되는 정도는 미미할 것으로 예상된다. 한국어와 중국어 문법을 비교해 보면, 규칙의 개수는 한국어 문법 규칙의 개수에 비하여 약 3 배나 많지만 분석 속도는 위에서 본 것처럼 중국어 구문 분석기 쪽이 오히려 더 빠르다. 이는 한국어 문법이 개수는 적지만 일반적인 조건에서 적용되는 정밀도가 떨어지는 규칙이 많아 구문 분석 과정에서 생성되는 중간 분석 구조 및 최종 분석 구조가 더 많기 때문에 발생하는 초래된 결과이다. 실제로 실험을 해 보았더니 한국어의 경우 문장 길이는 더 짧지만 평균 38.8 개의 파스 트리가 생성된 반면 중국어의 경우 문장 길이가 더 긴데도 불구하고 문장 당 생성된 파스 트리의 개수는 27.8 개였다.

기존의 문법 개발 도구들은 주로 lisp나 prolog 상에서 동작을 하기 때문에 분석 속도가 느리다는 단점이 있다. 예를 들어 prolog 인터프리터 상에서 구문 분석을 수행하는 ConTroll의 경우 5 단어로 된 문장을 분석하는 데 약 1-5 초 정도 소요되었으며, 12 단어로 된 문장을 분석하는 데에는 약 10-60 초 정도가 소요되었다[4].[5] 문법을 개발할 때에는 문장을 직접 분석해 봄으로써 문법의 문제점을 파악하는 과정이 수반되는데 이 정도의 분석 속도로는 원활하게 문법을 개발할 수 없어 문법 개발 기간도 길어질 것으로 생각된다. 이들 도구와 달리 GrammE는 lisp나 prolog 상에서 수행되는 것이 아니라 일반적인 플랫폼에서 독자적으로 수행 가능한 도구이며, 인터프리터 방식으로 구문 분석을 하더라도

2) 한국어 문법은 연구 과제의 종결과 함께 개발이 중단된 상태이며, 여기서 밝힌 문법 규칙의 개수는 최종 문법에서 구한 것이다.

3) 중국어 문법은 상용 번역 서비스 중인 응용 시스템의 한 모듈로 사용되고 있으며 현재도 성능 향상을 위한 개발이 계속되고 있는데, 여기서 밝힌 문법 규칙 개수는 논문을 쓰는 시점에서 구한 것이다.

4) 참고로 한국어 문법을 컴파일했을 때에는 4,668 줄의 C++ 프로그램이 생성되었고, 중국어 문법을 컴파일했을 때에는 21,323 줄의 C++ 프로그램이 생성되었다.

5) 이 논문에서는 구문 분석 속도를 측정하는데 사용된 컴퓨터의 사양에 대해서 언급되어 있지 않다.

초당 약 70-80 문장을 분석할 수 있기 때문에 문법 개발 기간을 줄일 수 있을 것으로 본다. 독립적으로 수행할 수 있는 문법 개발 도구로 일본 동경대의 LiLFeS가 있는데 이 도구를 사용하여 개발된 일본어 파서는 DEC Alpha 500/400 MHz 컴퓨터에서 21 단어의 문장 기준으로 초당 3 문장씩 분석할 수 있다[5].[6)]

6. 결론 및 연구 동향

GrammE는 자질 기반의 확장된 구 구조 문법을 개발하는 데 필요한 환경을 제공하는 언어 독립적인 시스템으로 여러 언어의 문법 개발에 사용할 수 있다. 문법 개발 단계에서는 기존의 문법 규칙을 수정하거나 새로운 문법 규칙을 추가하는 등의 작업이 빈번하게 일어나므로 GrammE를 인터프리터 방식으로 수행시키는데 이 수행 방식에서는 문법 수정 후 다른 절차 없이 바로 구문 분석을 해 볼 수 있다. 문법 개발이 종료되면 문법을 C++ 프로그램으로 변환할 수 있는데, 이것을 컴파일하고 구문 분석 엔진과 링크하면 독립적으로 수행할 수 있는 구문 분석기가 된다. 이렇게 만들어진 중국어 구문 분석기로 실험을 했을 때 평균 길이가 13.8 단어인 문장 기준으로 초당 200 문장 이상을 분석할 수 있었다.

기존 문법 개발 도구들은 대부분 lisp나 prolog와 같은 언어 상에서 수행되기 때문에 이것을 사용하기 위해서는 lisp나 prolog 인터프리터를 별도로 설치해야 하는 불편함이 따를 뿐만 아니라, 인터프리터를 통해 구문분석을 수행하기 때문에 분석 속도가 느리다는 문제가 있다. 그러나, GrammE는 Unix나 Windows 등에서 직접 수행되는 독립된 프로그램이기 때문에 사용하기가 편할 뿐만 아니라 구문 분석 속도가 매우 빠르다는 장점이 있다. GrammE는 연구용 또는 상용 구문 분석기를 개발하기 위한 목적으로 개발된 도구이지만, 문법 개발 교육을 위한 실습 도구로도 활용할 수 있다.

문법 개발은 상당히 오랜 기간 동안 지속해야 하는 방대한 작업이다. 그런데 기존에 개발된 문법을 보면 해당 문법을 개발하면서 축적된 자료가 거의 없기 때문에 나중에 다른 사람이 새로운 언어에 대한 문법을 개발하려면 아무 것도 없는 무의 상태에서 새롭게 시작해야 하는 경우가 대부분이다. 그러나 기존 문법으로부터 공통된 지식을 발견하고 이를 문법 개발에 활용할 수 있는 상태로 만들어 두고 여기서부터 문법 개발을 시작할 수만 있다면 무의 상태에서 문법을 개발하는 것보다 훨씬 효과적으로 문법을 개발할 수 있을 것이다. 이러한 노력은 다국어 문법 개발 프로젝트의 일환으로 스탠포드 대학교의 LINGO (Linguistic Grammars Online) 그룹에서 진행되고 있다[13].

6) [5]에 의하면, 수행 속도를 개선할 수 있는 LiLFeS native-code 컴파일러를 개발 중인데, 이를 이용하면 동일 조건에서 초당 10 문장 정도는 분석할 수 있을 것으로 예측하였다.

참 고 문 헌

[1] Stuart M. Shieber, "The Design of a Computer Language for Linguistic Information," Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual Conference of the Association for Computational Linguistics, pp.362-366, 1984.

[2] John Carroll, Ted Briscoe and Claire Grover, A Development Environment for Large Natural Language Grammars, Computer Laboratory Technical Report 233, Cambridge University, 1991.

[3] Martin Volk, Michael Jung and Dirk Richarz, "GTU - A workbench for the development of natural language grammars," Proceedings of the Practical Application of PROLOG, pp.637-660, 1995.

[4] Thilo Götz and Walt Detmar Meurers, "The Controll System as Large Grammar Development Platform," Proceedings of the ACL Workshop on Computational Environments for Grammar Development and Linguistic Engineering, pp.38-45, 1997.

[5] Makino Takaki, Yoshida Minoru, Torisawa Kentaro and Tsujii Jun'ichi, "LiLFeS - Towards a Practical HPSG Parser," Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and the 17th International Conference on Computational Linguistics, pp.807-811, 1998.

[6] Copestake, Ann, Implementing Typed Feature Structure Grammars, CSLI Publications, 2002.

[7] Bob Carpenter and Gerald Penn, *The Attribute Logic Engine, User's Guide, Version 3.2.1*, 2001. (available at http://www.cs.toronto.edu/~gpenn/ale.html)

[8] Gregor Erbach, "ProFIT : Prolog with Features, Inheritance and Templates," Proceedings of the 7th Conference of the European Chapter of the Association for Computational Linguistics, pp.180-187, 1995.

[9] Stanley Peters, "The Use of Context-sensitive Rules in Immediate Constituent Analysis," International Conference on Computational Linguistics COLING 1969, Preprint No. 45, 1969.

[10] Geoffrey K. Pullum, "Context-freeness and the Computer Processing of Human Languages," Proceedings of the 21th Annual Meeting of the Association for Computational Linguistics, pp.1-6, 1983.

[11] 심광섭, 구문 분석기 개발을 위한 통합 환경 구축 및 구문 분석기 자동 생성 시스템 개발, 연구보고서, 과학기술부, 2000.

[12] 심광섭, "구문 분석기 개발 도구 Grammar Writer 소개 및 활용", 제3회 아카데미21 국어정보학세미나 강의자료집, 2002.

[13] Emily M. Bender, Dan Flickinger and Stephan Oepen, "The Grammar Matrix: An Open-Source Starter-Kit for the Rapid Development of Cross-Linguistically Consistent Broad-Coverage Precision Grammars," Proceedings of the Workshop on Grammar Engineering and Evaluation at the 19th International Conference on Computational Linguistics, pp.8-14, 2002.

심 광 섭

정보과학회논문지 : 소프트웨어 및 응용
제 31 권 제 1 호 참조

양 재 형

정보과학회논문지 : 소프트웨어 및 응용
제 31 권 제 1 호 참조