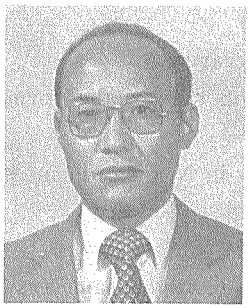


吳 吉 祿
韓國電子技術研究所
컴퓨터 연구부장/工博

한글 자연어 처리 시스템

"인간과 컴퓨터와의 대화에 있어서 컴퓨터 위주의 Assembly 언어나 Programming 언어가 중심이 되어 사용자에게 불편함이 있었으나 이제는 자연언어 처리 시스템의 개발로 일상생활에서 컴퓨터와의 대화가 가능하게 되었으며, 학계와 연구소간의 부단한 연구와 교류가 그 근간을 이루고 있다."

1. Introduction

가. Overview

컴퓨터와 인간의 초기의 대화방식은 컴퓨터가 알아 듣기 쉬운 Binary Code에 의해 이루어져 왔다. 그 이후로 어셈블리 언어, High Level의 컴퓨터 프로그래밍 언어가 개발되어 보다 인간이 편하게 컴퓨터와 대화할 수 있게 되었다. 그러나 이제까지의 방법은 모두 컴퓨터가 이해하기 쉬운 방식으로 대화하게 되어 있어서 인간에게는 보통 불편한 문제가 아니었다. 그러나 컴퓨터가 발달하고 인공지능 분야에 관심이 쏠리게 됨에 따라 인간이 일상 생활에서 사용하는 자연언어로써 컴퓨터와의 대화를 시도하게 되었다.

컴퓨터가 자연언어를 처리하는 데 있어서는 Formalize된 언어의 형태 즉, Syntax가 존재하게 되고 그에 따른 Parser가 필요하게 되었다. 그 뿐만 아니라 언어의 의미에 대한 분석과 표현을 필요로 하게 되어 이제까지의 보통의 프로그래밍 언어가 컴파일되는 단계와 비슷한 과정을 거치게 되었다.

그러나 자연언어의 해석에 관한 연구는 이미 오래전부터 컴퓨터 프로그래밍 언어와는 별도로 시작되어 왔다.

초기의 자연언어에 관한 연구는 1940년에 컴퓨터 Linguistics의 연구가 시작되어 주로 Machine Translation에 관해서 연구가 많이 이루어져왔다. 그리고 1960년대에 와서는 Human Language를 Knowledge-based 시스템으로 모델화하기에 이르렀다.

자연언어처리에 관한 여러 시도는 다음과 같이 크게 묶어 볼 수 있다.

(1) First Program : 제한된 Domain을 나타내는 Adhoc한 데이터 구조를 사용하였음. 입력

되는 문장은 간단한 서술형이거나 질문형이며 미리 정해진 Key Word나Pattern에 의해 Scan됨.

ex) Green's Baseball, Lindsay's SAD-SAM, Bobrow's Student, Weizenbaum's ELIZA

(2) Second Program : Text 자체의 Representation을 데이터베이스에 저장하고, 여러 가지의 Index Scheme을 사용하여 특별한 단어나 문장을 찾아낸다.

ex) Simmons, Burger and Long's PROTOSYNTHEX-I, Quillian's Semantic Memory

(3) Third Program : 데이터 베이스에는 어떤 Formal한 형태의 정보가 저장되어 있고 입력되는 문장은 이러한 내부 형태로 바뀐다. Limited-logic 시스템으로서 논리 추론의 일부만을 할 수 있다.

ex) Raphael's SIR, Quillian's TLC, Thompson's DEACON, Kellogg's CONVERSE

(4) Fourth Program : Logic, Procedural Semantics, Sematic Networks, or frame같은 Knowledge Represent 방법을 사용한 Knowledge-based 시스템.

자연언어처리에 관한 연구는 1950년에 Chomsky가 처음으로 Generative 문법의 이론을 소개함으로써 더욱 활성화되었다. 비교적 최근의 자연언어 처리에 관한 연구에 대하여 살펴보면, 1970년에 Wiliiam Woods가 달에서 가져온 암석에 대한 질문의 답을 해주는 프로그램인LUNAR라는 시스템을개발하였으며, Terry Winograd's 로보트를 Manipulation 하는 시스템인 SHRDLU를 발표하였다. 또한 1977년에는 Bobrow가 비행기 여행자들의 여행계획에 대한 자문 역할을 하는 GUS라는 시스템을 개발하였으며, 1990년에 와서는 Roger Schank가 Conceptual dependency 이론이 기본된 MARGIE, SAM 이라는 시스템을 개발하였다. 근래에 와서는 Script와 frame-based 시스템이 자연언어처리 분야에서는 가장 활발한 분야이다.

나. Mathematical Theory

a. Formal Grammar의 4 가지 형태

(1) Type 0 : Unrestricted grammar.
Turing machine

(2) Type 1 : Context Sensitive grammars.
if each Production in P is of the form $\alpha \rightarrow \beta$, where $|\alpha| \leftarrow |\beta|$ α, β is in $(NU\Sigma)*$

(3) Type 2 : Context free grammars
if each Production in P is of the form $A \rightarrow \alpha$, where A is in P and is in $(NU\Sigma)*$

(4) Type 3 : Regular grammar
if each Production in P is of the form $A \rightarrow xB$ or $A \rightarrow x$,
where A and B are in N and x is in *

b. Phrase-structure 문법의 부적격성

define)

Context-free와 context sensitive grammars는 Phrase-Structure Grammars 라고도 부른다.

영어는 regular language도 아니고 Context-free Language 도 아니다. 그 이유는 제한된 형태의 문법은 어떤 Common Construction을 generate할 수 없다. Context-sensitive 같은 더 강력한 문법이 영어의 문장을 정교하게 Generate할 수 있게 쓰여지는지는 확실하지 않지만 다음과 같은 이유로 그러한 문법은 적당하지 않다.

(1) 영어의 구조를 나타내기에는 너무 귀찮고 복잡하다.

(2) 다른 뜻의 문장도 같은 구조를 가질 수 있다.

The Picture was Painted by a new technique.

The Picture was Painted by a new artist.

(3) Phrase-structure 문법은 같은 의미를 가지는 문장들이 어떤 기본 구조로 나타나지 않아 다음과 같은 문장들의 동질성과 이질성을 나타낼 수 없다.

John ate an apple.
Did John eat an apple ?
What did John eat ?
Who ate an apple ?

c. Propositions

Logic의 가장 기본이 되는 것은 진실과 거짓이다. 그런데 어떤 문장이 있어, 이 문장은 진실일 수도 있고 거짓일 수도 있다. 그 예를 들어

보면 "달은 초록색 치즈로 만들어졌다." 혹은 "3 더하기 4 는 7 이다."라는 문장이 있을 수 있다. 이러한 문장들을 Proposition 이라 한다. 그러나 "x+y>5", "그는 내일 떠날 것인가?" 라는 문장처럼 진실과 거짓을 밝힐 수 없는 문장은 Proposition이라고 할 수 없다.

d. Predicate Calculus

Propositional logic은 문장의 진실과 거짓만을 다루는 데 제한되어 있다. 그러나 Predicate calculus는 Propositional logic의 확장으로서 변수의 개념을 포함하고 있다. 즉 Predicate는 다음과 같이 쓸 수 있고,

Predicate (individual, individual, ···)

여기서 말하는 individual이 변수가 될 수 있다. 그 예를 들어보면, 다음과 같은 Propositional을 Predicate로 나타내 볼 수 있다.

"철수가 영희를 좋아한다" → 좋아한다(철수, 영희) → 좋아한다(X, 영희).

여기서 X가 철수로 주어지면 이 문장은 진실이 되지만 값이 주어지면 거짓이 된다.

e. First-order logic

First-order logic은 Predicate logic에 두 가지의 개념을 더한 것이다. 그 하나는 function의 개념이다. Function이라 함은 일정하게 주어진 갯수의 argument를 넘겨 받았을 때 그에 맞는 값을 넘겨 주는 것을 말하며 Proposition과 달리 진실과 거짓의 값을 가지지 않는다. 그 예를 들어 보면 다음과 같다. 만약 다음과 같은 Function이 있을 때

"좋아한다(X, 영희)."

X에는 철수가 Return되어 온다. 또 하나의 요소는 Predicate Equals의 개념을 갖는다.

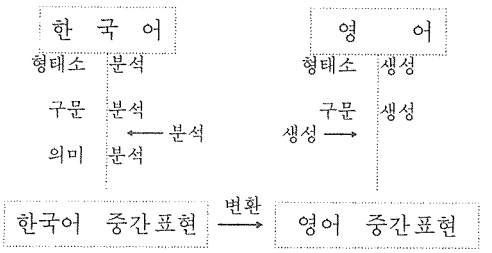
즉 Individual X, Y가 Equal이라 함은 모든 Predicate P에 대하여 P(X)=P(Y)이고, 모든 Function F에 대하여 F(X)=F(Y)의 관계를 갖는 것을 뜻한다.

다. Machine Translation

어떤 언어에서 어떤 언어로 변환시키는 기계번역에 대한 연구는 컴퓨터의 역사보다 더 오래전에 시작되었다. 1930년에 이미 이 분야에 대하여 P. P. Smirnov-Troyansky, G. B. Artsouni라는 사람들에 의하여 시도되었으며 Digital 컴퓨터의 소개로 이들의 연구는 더욱 활기를 띠게 되었다. 1949년에는 Weaver에 의해서 "Translation"이라는 보고서가 발표되었으며, 1952년에는 가상의 Intermediate 언어인 Machinese를 제안하였다. 그 이후 1957년 Chomsky에 의해 Transformational Grammar가 발표되고, 1960년에는 ALGOL과 LISP가 개발됨으로써 자연언어 처리 분야에 큰 발전이 있었다.

그리고 1970년대에는 Knowledge Representation 방법을 사용하여 개발한 시스템들이 나오기 시작하였으며 현재는 여러 선진국에서 다소 완전한 번역을 시도하고 있다.

기계번역의 처리 과정은 일반적으로 다음 그림에서 보는 것과 같이 해석, 변환, 생성의 세 부분으로 나누어진다. 이것은 한영 번역 시스템의 예를 보인 것이다.



a. 해 석

해석은 번역 대상문을 해석하여, 문장의 구조나 의미구조를 추출한다. 문장의 구문구조를 추출함으로써, 문장의 주술관계나 문장의 형 (평서문 / 명령문 / 의문문) 등이 인식될 수 있다.

이것을 구문해석 또는 통어해석이라 한다. 문장의 의미 구조를 추출함으로써, 문장을 구성하는 각 단어가 문장중에서 어떤 의미적 역할을 하는가가 결정될 뿐 아니라 의미적으로 이상한 구문 구조를 제거할 수 있다. 이것을 의미해석이라 한다.

문장의 의미 구조를 추출하기 위해서는 생략되어 있는 단어를 추정한다든지, 지시대명사가 지칭하는 것을 결정할 필요가 있다. 이것은 문맥해석 또는 담화해석이라고 부르며 언어학과 자연언어 처리기술의 관점에서 보아도 미숙한 연구 분야이다.

b. 변 환

변환은 해석에서 얻어진 번역 대상문장의 해석 결과를 변환하여, 번역 목표문장이 합성되기

쉬운 구조로 바꾸어 나간다. 이 부분에서는 번역 대상언어의 단어를 번역 목표언어의 어떤 단어로 변환할 것인지의 역어선발의 과정을 포함한다. 또 번역 대상문장의 해석 결과로써 얻어진 구조의 일부를 변환하여 번역 목표문장이 합성되기 쉬운 구조로 바꾸는 과정도 포함할 수 있다. 역어의 선택에서는 해석에서 이루어 놓은 의미적인 해석이 다시 한 번 필요로 하는 경우가 많다.

c. 생 성

합성(생성)은 변환 과정을 받아서 그것을 일렬로 늘어 놓아 번역 목표문장을 만들어내는 과정이다. 번역 대상문장과 번역 목표문장과의 어순의 차이는 합성의 단계에서 조정한다. 합성의 과정에서 변환의 구조 변환을 포함하는 경우가 많다.

또 용언의 활용 변화나 음변변화를 시행하며 번역 목표문장이 일본어일 경우에는 명사구에 적당한 조사를 부가하는 일 등을 행한다.

2. 자연언어 처리를 위한 문법 규칙

가. Definite Clause Grammar

이 절에서는 Definite Clause Grammar의 Formalism에 관해 서술하고자 하는데 그 기본적인 생각은 Kowalski와 Colmerauer에 의해 출발되었다. 이러한 DCG의 Formalism을 얻기 위해 Definite Clause와 일치성을 유지하는 방법으로 Context-Free Grammars를 일반화한다.

1) 표 기

DCG의 표기는 다음과 같은 방법으로 Context-free Grammar의 표기를 확장한다.

(1) Non-terminals는 Context-free 한 경우에 단순 Atoms가 허용되는 외에 복합 Term으로도 허용되어진다.

np(X, S) sentence(S)

(2) 규칙의 오른편에, Non-terminals와 Terminals의 List 외에 Brackets '{', '}' 내에 쓰여진 일련의 Procedure calls가 사용될 수 있다. 이 Procedure들은 규칙에서 만족되어야 하는 특별한 조건을 표현하는 데 사용된다.

noun(N)→[W], {rootform(W, N), is-noun(N)}.

두번째 예는 noun(N)의 구는 단어(W) 로서 구성되고 N은 단어(W)의 원형이고 N이 명사임을 보여준다. 규칙의 오른편에 Non-terminals, Terminals, Procedure calls는 총괄적으로 Goal로서 간주된다.

2) Definite clause로서 DCG 표기의 의미

DCG의 규칙에서 Terminal symbols는 앞에서 처럼 번역되어 지고 Narity의 non-terminal은 같은 이름을 가지는 N+2 Place Predicate로 번역된다. 여기서 N Arguments는 non-terminal에서 외부적으로 나타나는 의미이고 나머지 두 개의 Arguments는 Context-free non-terminal의 번역에서와 같은 의미이다. 또한 규칙의 오른편에 있는 Procedure calls는 단순히 그들 자신의 의미를 갖는다. 예를 들어 다음과 같은 규칙은

noun(N)→[W], {rootform(W, N), is-noun(N)}.

다음과 같은 Clause를 의미한다.

noun(N, SO, S):-connects(SO, W, S), rootform(W, N), is-noun(N).

3. Definite Clause Grammar의 사용

DCG는 언어 분석에서 Parser Tree, 구문구성의 특별한 조건, 문맥의 독립성을 제공한다.

가. Building Structure

Non-terminal의 Argument는 문법 규칙에 Tree 구조의 Structure를 제공한다. Non-terminal은 문법 규칙에 Matching 함에 의해 확장되고, Unification 과정을 통해 Tree 구조가 만들어진다. 여기서 단순한 예를 들어보면 다음의 Context-free Grammar는 각 절이 단순한 Parser tree를 만들기 위해 변형된다.

sentence → noun-phrase, verb-phrase.
noun-phrase → determiner, noun, rel-clause.
noun-phrase → name.
verb-phrase → trans-verb, noun-phrase.
verb-phrase → intrans-verb.
rel-clause → [that], verb-phrase.
rel-clause → [ ].
determiner → [every].
denterminer → [a].
noun → [man].

noun → [woman].

name → [mary].

trans-verb → [loves].

intrans-verb → [lives].

또한 단어에 일치하는 Non-terminal을 정의하는 규칙인 Dictionary를 표현함에 의해 더 치밀하고 효율적인 방법을 소개한다. 일반적으로 다음과 같은 형태의 규칙 대신에 :

category (arguments) → [word].

Category에 있는 각 단어에 대해 일반적인 규칙을 쓴다.

category (arguments) → [W],
{cat (W, arguments)}.

그리고 다음과 같은 형태의 절로 구성된 cat (dictionary procedure)를 정의한다.

cat (word, arguments).

변형된 예에 대한 규칙은 :

- sentence (s (NP, VP)) → noun-phrase (NP), verb-phrase (VP).
- noun-phrase (np (Det, Noun, Rel ((→ determiner (Det), noun (Noun), rel-clause (Rel).
- noun-phrase (np (Name)) → name (Name).
- verb-phrase (vp (TV, NP)) → trans-verb (TV), noun-phrase (NP).
- verb-phrase (vp (IV) → intrans-verb (IV).
- rel-clause (rel (that, VP)) → [that], verb-phrase (VP).
- rel-clause (rel ([ ])) → [ ].
- determiner (det (W)) → [W], {is-determiner (W)}.
- noun- (n (W)) → [W], {is-noun (W)}.
- name (name (W)) → [W], {is-name (W)}.
- trans-verb (tv (W)) → [W], {is-trans (W)}.
- intrans-verb (iv (W) → [W], {is-intrans (W)}.

NP의 의미를 가지는 명사절로서 Noun-phrase (NP) 같은 Non-terminals를 확장하고 위의 첫번째 규칙의 Argument s (NP, VP)는 명사절 (NP)과 동사절 (VP)로서 구성됨을 보여주며 다음은 부수적인 Dictionary 절이다.

is-determiner (every).

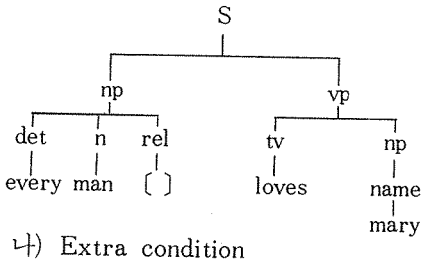
is-noun (man).

is-name (mary).

is-trans (loves).

is-intrans (lives).

위와 같은 규칙을 가질 때 입력 문장 "Every man loves Mary"는 다음과 같은 Parser tree를 구성한다.



나) Extra condition

규칙의 Body에서 외부적인 Procedure call은 다음과 같은 규칙에 의해 표현된다.

date (D, M) → month (M), [D],
{integer (D), 0 〈D, D〉32}.

위의 규칙은 M 달의 D일이라는 문장이 M달과 다음 Symbol D에 의해 표현되고 여기서 D는 0 보다 크고 32보다 작은 정수를 나타낸다.

다) 문맥의 종속성

DCG에서 Non-terminal arguments는 Tree 구조 외에 문맥의 정보를 테스트하는 데 사용된다. 예를 들면 한정사 (Determiners), 명사 그리고 동사 사이에서 요구되는 수의 일치를 처리하기 위해 가) 절의 예를 변형할 수 있다.

변형된 문법은 "Every man loves some girl" "All men like girls" 같은 문장을 처리할 수 있지만 "All men that lives love a woman" 같은 문법에 틀린 문장은 처리할 수 없다. 수의 일치를 처리하기 위해 어떤 Non-terminal은 단수나 복수의 값을 취하는 Argument를 사용할 수 있고 사전에서 수의 일치와 품사의 원형을 넘겨줄 수 있는 Arguments를 가진다.

변형된 규칙은 다음과 같다.

- sentence (s (NP, VP)) → noun-phrase (N, NP), verb-phrase (N, VP).
- noun-phrase (N, np (Det, Noun, Rel)) → determiner (N, Det), noun (N, Noun), rel-clause (N, Rel).
- noun-phrase (singular, np (Name)) → name (Name).
- verb-phrase (N, vp (TV, NP)) → trans-verb (N, TV), noun-phrase (NI, NP).
- verb-phrase (N, vp (IV)) → intrans-verb (N, IV).
- rel-clause (N, rel (that, VP)) → [that], verb-phrase (N, VP).

- rel-clause(N, rel([ ])) → [ ].
- determiner(N, det(W)) → [W], {is-determiner(W, N)}.
- determiner(plural, det([ ])) → [ ].
- noun(N, n(Root)) → [W], {is-noun(W, N, Root)}.
- name(name(W)) → [W], {is-name(W)}.
- trans-verb(N, tv(Root)) → [W], {is-trans(W, N, Root)}.
- intrans-verb(N, iv(Root)) → [W], {is-intrans(W, N, Root)}.
- is-determiner(every, singular).
- is-determiner(all, plural).
- is-noun(man, singular, man).
- is-noun(men, plural, man).
- is-name(mary).
- is-trans(likes, singular, like).
- is-trans(like, plural, like).
- is-intans(live, plural, live).

나. Augmented Transition Network

1) 소 개

지난 몇년 동안 Augmented Transition Network(ATN) Grammar는 종이에 쓰여진 글과 사람의 입으로 통해 나오는 말을 해석하는 자연어 이해 시스템에 많이 사용되어져 왔다. 이 문법은 쓰기 쉽고 고치기 쉬우며 넓은 범위의 문장 구조를 다룰 수 있을 뿐 아니라 다른 시스템과의 Interface도 쉽게 할 수 있다. ATN 문법은, William Woods에 의해 발표되었으며 지금까지 영어의 자연어 처리 분야에서 가장 많이 사용된 문법 중에 하나이다.

2) Recursive Transition Network

Recursive Transition Network는 arc와 State로 된 Directed Graph로써 특별한 State인 Start State와 Final State를 가진다.

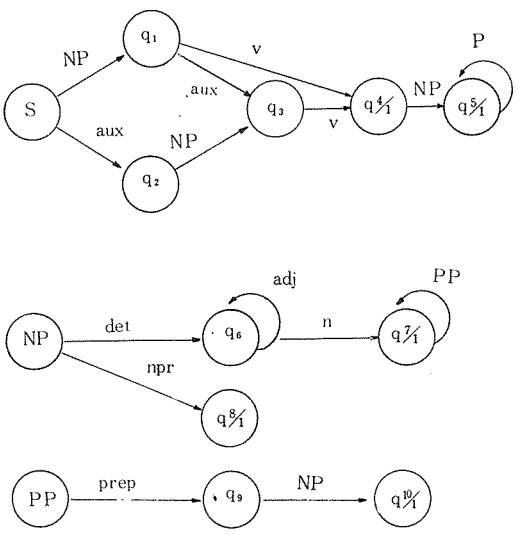
다음 그림은 영어의 Subset에 대한 RTN이 주어졌으며 이는 "John washed the car"와 "Did the red barn collapse?"이라는 문장을 받아들일 수 있다. 이 Network를 보면 우리는 쉽게 어떤 종류의 문장을 받아들일 수 있는가를 알 수 있게 된다.

"Did the red barn collapse?"라는 문장을 받아 들이기 위해서 이 Network는 State S에서 출발한다. 첫번째는 조동사 "did"를 보고 q2로 가고 q2에서는 다음 입력 String이 NP가 되어야만 q3로 갈 수 있다. State NP에서는 Determiner "the"를 보고 State q6 로 간다.

여기에서 형용사 "red"는 q6에서 Loop를 돌고 명사 "barn"을 보고 q7으로 간다. q7은Final State이기 때문에 State NP에서 pop된다. RTN 문법은 이러한 과정을 거쳐서 한 문장을 분석할 수 있다.

3) Augumented Transition Network

위에서 설명한 바와 같이 RTN은 Context Free Grammar와 거의 같아서 자연언어 처리에 필요한 문법을 나타내기에는 적당치 않아, Network의 각 arc에 필요한 Condition을 더하여 더욱 복잡한 문법을 나타낼 수 있게 한 것이 Augumented Transition Network이다. 이 ATN에서 사용되는 Action은 문장 분석의 결과에 의해 문장의 구문구조를 만드는 역할을 하고 Condition과 다른 Arc에서 사용할 수 있는 임시 정보를 제공해 준다.



Condition은 문장을 분석하는 데 필요없는 분석을 하지 않도록 두 개의 규칙이 공통부분을 가질 때 첫번째 규칙이 수행되어 결과를 얻어

놓으면 두번째 규칙은 이미 얻어 놓은 정보를 이용할 수 있어 공통 부분에 대해서는 반복 수행할 필요가 없게 된다. Action과 Condition은 다같이 Register를 사용하는데 Action이 만드는 구문구조의 일부를 저장하기도 하고 Condition이 사용하는 Flag나 Indicator를 저장하기도 한다. 다음은 ATN을 표현하는 language의 Specification이다.

⟨Transition network⟩ → (⟨arc set⟩
⟨arc sety⟩*)
⟨are set⟩ → (⟨state⟩ ⟨arc⟩*)
⟨arc⟩ → (CAT⟨category name⟩ ⟨test⟩
⟨action⟩*⟨terminal action⟩) (PUSH⟨state⟩
⟨test⟩ ⟨action⟩*⟨terminal action⟩)|
(POP⟨form⟩ ⟨test⟩)
⟨action⟩ → (SETR⟨register⟩ ⟨form⟩)
⟨terminal action⟩ → (TO⟨state⟩)
(JUMP⟨state⟩)
⟨Form⟩ → (GETR⟨register⟩) |*|
(BUILDQ⟨fragment⟩ ⟨register⟩*) |
(LIST⟨form⟩*) |
(APPEND⟨form⟩ ⟨form⟩) |
(QUOTE⟨arbitrary structure⟩)

4) An Illustrative example

다음은 위에서 소개한 Language로 나타낸 ATN이다. 이것은 그림의 S/, Q1, Q2, Q3, Q4 그리고 Q5의 state로 구성된 Transition Network의 확장된 부분이다. 이러한 Augumented Network는 문장이 단언문인지, 의문문인지를 나타내는 Type (DCL 혹은 Q)과 주체 명사구, 보조구(NIL일 수 있다), 동사구로 구성되는데 이러한 표현은 문장 내에서 주체 명사구와 보조구의 순서에 관계없이 생성된다.

```
(S/ (PUSH NP/T
      (SETR SUBJ*)
      (SETR TYPE (QUOTE DCL))
      (TO Q1))
    (CAT AUX T
      (SETR AUX*)
      (SETR TYPE (QUOTE Q))
      (TO Q2)))
(Q1 (CAT V T
      (SETR AUX NIL)
      (SETR V *)
      (TO Q4))
    (CAT AUX T
      (SETR AUX *)
      (TO Q3)))
(Q2 (PUSH NP/T
      (SETR SUBJ *)
      (TO Q3)))
(Q3 (CAT V T
      (SETR V *)
      (TO Q4)))
(Q4 (POP (BUILDQ (S+++(VP+))
      TYPE SUBJ AUX V) T)
      (PUSH NP/T
      (SETR VP (BUILDQ (VP (V+) *) V))
      (TO Q5)))
(Q5 (POP (BUILDQ (S++++)
      TYPE SUBJ AUX VP) T)
      (PUSH PP/T
      (SETR VP (APPEND (GETR VP)
      (LIST *)))
      (TO Q5)))
```

3. Parsing 전략

가. 프로그램 언어로서의 논리
(Definite Clause Subset)

1) Syntax, Semantics

가) Terms

Terms는 언어의 데이터 Object를 말하고 상수이거나 변수, 복합 Term으로 나누어진다.

상수는 정수(0 1 999)이거나 Atoms (a void = := 'Algol-68' [ ]) 이다. Atoms의 Symbol은 일련의 Character로서 구성이 되는데 변수나 정수와 구분하기 위해 따옴표를 사용한다.

변수는 두문자가 대문자에 의해 구분되어 진다. (X Value A A1) 변수는 알려지지 않은 Object를 나타내기 위해 사용되어 지고, 대부분의 프로그램언어에서 같이 할당될 수 있는 기억 장소가 아니라 데이터 Object의 Local name 이다. 복합 Term은 언어의 구조화된 데이터 Object로서 한개의 Functor와 여러 개의 Argument로 구성이 된다. Functor는 name (atom)

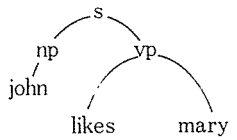
과 arity (argument의 수)로 표현된다.

Point (X, Y, Z) : arity가 3이고 name이 Point 인 Functor

복합 Term은 나무 구조로서 표현될 수 있다. 예를 들면 다음과 같은 term은

s (np (john), vp (vp (v (likes) np (mary)))

아래와 같은 나무 구조로 표현되어 진다.



가끔 infix 표현을 사용하여 복합 term을 쓰는 것이 편리하다.

X+Y (P;Q) X〈 Y

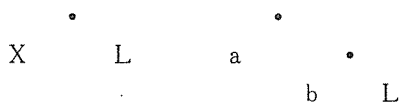
대신에 :

+(X, Y); (P, Q) 〈 (X, Y)

List는 데이터 구조에서 중요한 역할을 하는데 Lisp 언어에서 List와 같은 의미이다. List는 Empty List를 나타내는 Atom ([ ]) 이거나 List의 Functor (.)와 Head와 Body인 두 개의 Argument를 가지는 복합 Term이다.

[X|L] [a, b|L]

는 다음과 같은 나무 구조를 나타낸다.



나) Clauses

논리 프로그램의 기본적인 요소는 Goal 혹은 Procedure Call이다. 예를 들면

Gives (Tom, Apple, Teacher) Reverse ([1, 2, 3], X)

Goal은 단순히 Term의 특별한 형으로 프로그램 내에서 나타나는 위치에 의해 구분되어 진다. Goal의 Functor를 Predicate라 하고 일반 프로그램 언어에서 Procedure의 이름과 일치한다.

논리 프로그램은 Clause라 불리어지는 일련의 문장으로 구성되고 Caluse는 Head와 Body로 나누어 진다. Head는 Single Goal이거나 Empty이고 Body는 0 혹은 여러개의 Goal로 구성된다.

Clause의 Head나 Body가 Empty가 아니면 Non-unit Clause라 하고 다음과 같은 형태로 쓴다.

P : -Q, R, S

여기서 P는 Head Goal이고 Q, R 그리고 S는 Body의 Goal로서 Declarative하게 그리고 Procedure하게 해석된다.

2) Declarative and Procedural Semantics

Pereira와 Warren의 정의에 의하면 Declarative한 Definite Clause의 의미는 :

"어떤 Goal이 참이 되기 위해 그 Goal이 어떤 Clause의 Head이고, 그 Clause Instance의 Body에 있는 각 Goal이 True이어야 한다. 여기서 Clause Instance란 0개 이상의 변수가 있으면 그 변수의 모든 Occurrence에 새로운 Term을 대치함으로써 생기는 Clause를 말한다."

또한 Procedure한 의미는 Prolog가 Goal을 수행하는 방법을 정의한다.

"Goal을 수행하기 위해 Prolog는 먼저 어떤 Clause의 Head가 Goal과 Matching하는 첫번째 Goal을 찾는다. 그러한 Match가 발견되면 Body에 있는 각 Goal을 왼편에서 오른편으로 수행함으로써 Matching Clause Instance가 활성화된다. 여기서 Goal에 대한 Match를 발견하지 못하면 Backtracking한다. 이때 가장 최근에 활성화되었던 Clause에 대하여 그 Clause의 Head와 Match함으로써 생긴 모든 Substitution을 원래 상태로 만든다. 다음에 그 Clause를 활성화시켰던 Goal에 대해 Alternative한 Clause를 찾는다."

3) 논리언어의 특성

Warren (1977)에 의하면 논리 언어에서는 일반 프로그램 언어에서 볼 수 없는 구문과 의미의 단순성을 가지는데, 여기서는 Grammar Writing에 관한 특별한 특성을 서술한다.

(1) 일반 언어에서 사용되는 데이터에 대한 Selector와 Constructor Function을 제공하는 Pattern Matching (Unification)을 가진다.

(2) 여러 개의 입, 출력 변수를 처리하는 Procedure를 가진다.

(3) Procedure의 입, 출력 Arguments는 구분될 필요가 없고, 한 Procedure에서 다른 P-

rocedure로 Call할 때 변화하는 Argument를 가져 다목적의 Procedure를 가진다.

(4) Procedure는 Backtracking을 통하여 선택적인 결과를 얻는데 이러한 Procedure를 Non-determinate 라고 한다.

(5) Procedure는 그 결과가 변수인 것처럼 불완전한 결과를 Return할 수 있다.

(6) 같은 형태의 'Program'과 'Data'를 가진다.

나. 문법 규칙적으로부터 구문 해석 프로그램의 자동 생성

1) 소개

Prolog는 인공 지능적인 언어로서 주목을 모으고 있고, 자연언어 처리에 사용할 수가 있다.

필자등이 사용하고 있는 DEC System-10 Prolog에서는 DCG라고 불리워지는 Top-Down Parser가 내장되어 있고, 이것을 자연언어의 구문 해석 시스템으로 이용하는 방법이 제안되고 있다. 이 방법에 의하면 하나의 문맥자유문법 규칙을 Prolog의 하나의 문(Horn절)에 대응하고, Prolog의 Backtracking기능을 이용해서 해석이 행해지고 있기 때문에 특별한 구문해석 프로그램을 필요로 하지 않는다. 그러나 1) Left-Recursive한 문법규칙은 취급하지 않는다. 2) 사전과 문법 규칙의 취급을 구별하지 않는다. 등의 결점이 존재한다.

DCG의 잇점을 가급적 살려서, 이러한 결점을 보충해서 보완된 것이 Bottom up Parser, BUP이다. BUP에서는 DCG와 같이 문맥자유문법 규칙과 Prolog 프로그램이 대응하고 있기 때문에, DCG의 문법 규칙으로부터 BUP의 문법 규칙에의 변환이 가능하다. 이 변환 프로그램을(BUP Translator) DEC System -10 Prolog로 작성했으며 본고에서 여기 대해서 보고한다. 또, ICOT에서도 같은 Translator를 작성하고 있으나 변수명이 보존되지 않기 때문에 변환 후의 프로그램을 읽기 힘들다. 본고에서 소개하는 Translator는 처리시간보다 변환 후의 프로그램이 읽기 쉬운 방법이 고려되어 있다. 2.가.3)나) 에서 표시하는 바와 같이 구조의 OR Type의 문법 규칙도 변환할 복잡한 수가 있다.

2) BUP

BUP는, Prolog의 Backtracking 기능을 이용하면서, 문맥 자유문법을 Bottom-Up적으로 적용해가는 Parser이다. 각 문법 Category가 Prolog의 술어에 변환되어 있기 때문에, 문법 Category에 인수를 가지게 하든지 Prolog프로그램을 문법규칙 가운데 집어넣을 수가 있고 문맥 자유문법의 기술능력을 보충하고 있다. Left-Recursive한 문법 규칙을 처리할 수 있고, 사전과 문법 규칙의 취급도 분리하고 있다. 본장에서는 BUP의 기본적인 동작 원리에 대하여 설명하겠다.

가) BUP의 기법

BUP에서는 문맥 자유 문법의 규칙을 Prolog Program으로 변환한 형태로 쓰인다. 문맥자유 문법이란 문법 G=(N, , P, S)에 있어서, 문법규칙 P의 요소가 A → a형(단, AN 및 a (NU)*)와 같은 문법이다. 이 문법 규칙은 기본적으로는 다음의 두 가지 형태로 표현할 수 있다.

(1) C → a1, a2, ……am (m>=1)

(2) C → C1, C2……Cn (n>=1)

여기서 ai는 Terminal Symbol, C및 Ci 는 Non-Terminal Symbol로 표시한다. DCG 에서는 이러한 규칙을 대부분 그대로의 형태로 기술할 수 있고 그리고, 또한 Non-Terminal Symbol에 임의 인수를 가지게 할 수 있다. 또한 Terminal Symbol은 괄호([ ])및 List 형식을 취한다.

(1') c(Args) → [a1, a2……am]

(2') c(Args) → C1(Args1), c2(Args2), …… cn(Argsn).

이 기법은 DEC System-10 Prolog에 기초해서, 정수는 소문자, 변수는 대문자로 시작이 된다. BUP에서는 위의 (1)의 형태를 사전항목, (2)의 형태를 문법 규칙으로서 각각 다음과 같은 형식에 변환되어 사용한다.

(1") dict(c, [Args], [a1, a2, …… am[X], X)

(2") c1(G, [Args1], I, X0, X1)

……

goal(cn, [Argsn], Xn-2. Xn-1),

c(G, [Args], I, Xn-1, X)

(1"), (2")는 DCG에 의해 각각 다음과 같이

할 수도 있다.

(1") Dict(c, [Argsl]) → [a1, a2, ……am]
(2") Cl(G, [Argsl], I) → Goal(c2, [Args 2]),
.....
goal(cn, [Argsn]),
c(G, [Args], I).

실제 본고에 소개하는 Translator는 (1'), (2')와 같은 DCG의 기호를 (1"), (2")와 같은 BUP의 기호로서 변환한다.

나) BUP의 알고리즘

BUP의 문법 규칙 중에 나타나는 술어 Goal은 Bottom-UP Parsing의 과정을 Control하는 것으로 다음과 같이 정의되고 있다.

goal(G, A, X, Z) :- dict(C, A1, X, Y), Link(C, G),
P=..[C, G, A1, A, Y, Z], Call(P).

술어 Goal의 제1인수는 Goal이 Call되는 시점에 있어서의 Goal Category, 제2인수는 Goal Catgeory에 주어진 인수의 List, 제3, 제4인수는 Goal Category에 대응하는 종단 기호열을 그대로의 차로서 표현하는 List이다. 술어 Goal이 불리워지는 Dict(C, A1, X, Y)에 의해 List X의 최초의 요소에 Match되는 사전항목을 찾아서 그 문법 Category Call을 변수 C에 속박한다.

Goal(G, A, [Word1, Word2, ... Wordn],
  ↓ [ ])
Dict(G, A1, [Word1, Word2, ... Wordn],
  ↕ Y)
Dict(Cat1, [Arg1, ... Argm], [Word1|X],
  X)

다음에 Link[C, G]에 의해서 Cat1로부터 Goal Category의 도달가능성을 조사해서, 만약 가능성이 없으면 Back Tracking을 일으킨다.

'=..'은 우변의 List의 제1요소를 술어명, 기타 요소를 그 인수로 한 술어표현을 구성하는 Operator, Call(P)에 의해서 Cat1(G, A1, A, Y, Z)가 Call된다. 이미 기술한 바와 같이 BUP에서는 각 문법 Category가 Prolog의 술어명에 되어 있기 때문에 Cat1(G, A1, A, Y, Z)과 좌변이 Match하는 BUP형식의 문법 규칙을 찾는다. 이와 같은 문법 규칙이 복수개 존재하는 경우에는, 이 중에서 하나를 끄집어 내어서 우변의 Goal(Cat2, [Args], I, Xn-1, X)가 Call된다. 이것들이 모두 성공하면 1단의 문법 Category Cat의 부분목이 구성되고, 다음의 Cat(G, [Args], Xn-1, X)가 불려진다. 만약 하나라도 실패하면 Backtracking을 일으켜 다른 적용가능한 문법 규칙을 시험한다. 이와 같이 BUP의 알고리즘은 Backtracking을 행한 Bottom-up+Depth-first법이다. DCG의 Top-down+Depth-first와 비교하면 Left Recursive한 문법 규칙을 무한 Loop에 빠질 염려는 없다. 또한 DCG에서는 사전 찾기의 Timing이 확정되지 않으나, BUP에서는 술어 Goal이 불려지는 때 한했기 때문에 사전과 문법규칙의 취급을 분리할 수 있다.

다) 정지 조건절

전절에서는 BUP의 알고리즘의 개략에 대해서 설명했으나 Bottom-up적인 구문목의 성장의 정지성에 대해서는 설명하지 않았다. BUP에서는 항상 Goal이 되는 문법 Category를 설정해서 이것을 향해서 구문목을 성장해가기 때문에 Goal Category에 대한 부분목이 구성된 시점에서 처리가 성공하지 않으면 아니된다. 이때문에 도입한 것이 정지 조건절이다. 정지 조건절은 문법 규칙 중에 나타나는 모든 문법 Category에 대하여 주어지며 다음과 같이 기술된다.

C(C, I, I, X, X).

이것은 각 문법 Category에 대응하는 술어의 제1인수인 Goal Category가 자기 자신이면 이 술어의 호출을 성공시킨다. 정지조건절은 이성격상 다른 문법 규칙보다 먼저 적용하지 않으면 아니된다.

라) Link절

나)절에서 표시한 술어 Goal의 정의 가운데 Link라는 술어를 사용해서 Goal Category에의 도달가능성을 조사했으나 이 관계를 표시한 것이 Link절이다. Link절에서는 각 문법 규칙에서 직접 얻는 것과 이것들의 추이율을 적용해서 얻는 것이 있다. Link절을 사용한 문법 Category간의 도달 가능성의 Check는 각 문법 규칙 중에도 삽입되어 불필요한 문법 규칙의 적용을

방해하고 있다.

마) Program항

BUP는 문맥 자유 문법에 기초하고 있으나 자연언어의 구문해석 시스템으로서 사용하기 때문에 문맥 의존적인 처리가 필요하다. 거기서 DCG와 같이 BUP에서도 Non-Terminal항 및 Terminal항 이외의 Prolog프로그램을 문법 규칙 중에 자유롭게 기술할 수 있다. 이것을 프로그램 항이라 부르고, ( )형식으로 기술한다.

프로그램항은 문법규칙 중의 문법 Category에 주어진 인수간의 제약조건(의미처리 프로그램 포함)을 기술함에도 사용된다.

4. 한글 자연어 처리 시스템의 구현

가. 한글 문법

컴퓨터를 이용하여 한국어의 구문구조를 분석하는 데는 많은 문제점이 있는데 여기서는 문장성분간의 결합 관계를 이용하여 문법규칙을 형식화함에 의해 한국어 구문구조의 컴퓨터 처리를 위한 구문분석(Syntactic analysis)에 관하여 서술한다.

구문분석은 기본부분과 변형부분으로 구성되는데, 기본부분은 문장의 내면구조(Deep Structure)를 생성하고 변형부분은 표면구조(Surface Structure)를 생성한다 또 기본부분은 범주규칙(Categorial Rule)과 어휘부(Lexicon)로 구성되는데, 범주규칙은 내면구조의 문법적 관계를 결정하고, 어휘부는 문법적 관계, 구문의 의미 또는 환경적인 의미에 적절한 어휘형태를 결정하는 Subcategorial Rule로서 내부에 고유한 구문적인 의미를 가진다. 한국어 문장에 대한 구문규칙은 일반화된 것은 없으나 다음과 같다.

(1) S → NP+VP
(2) NP → (Det) (Adj*)N(Kasus) | NP+VP|
(3) VP → (NP*)V | V+NP | NP+Cop
(4) V → Vst+T+SE
(5) T → Present(현재)
Past(과거)
Future(미래)
(6) SE → Declarative(단언문)
Interrogative(의문문)
Imperative(명령문)
Exclamate(감탄문)
(7) Kasus →
SM(subjective marker : 주격)
OM(objective Marker : 목적격)
LM(Locative Marker : 처격)
DM(Dative marker : 여격)
CM(commitative Marker : 공동격)
IM(instrumentative marker : 도구격)
AM(allative marker : 향격)
(8) Cop → 이다 | 아니다

여기서 SE(sentence end)는 종결어미로서 단언형(declarative), 의문형 (Interrogative) 명령형 (Imperative), 감탄형 (Exclamative)을 의미하고, T는 시제(Tense)로서 현재(Present), 과거(Past), 미래(Future)를 각각 의미한다. 또 격표지(Case Marker)는 다음과 같다.

(1) 주격(SM)……이 / 가
(2) 관형격(GM)……의
(3) 목적격(OM)……를 / 을
(4) 보격(CM)……이 / 가
(5) 부사격
처격(LM)……에 / 에서
향격(AM)……로
여격(DM)……(에)게
조격(IM)……로(써)
자격격(QM)……로(서)
공동격(CM)……와 / 과
인용격( )……라고
비교격( )……처럼 / 같이 / 보다
시발격( )……부터
도달격( )……까지
(6) 특수조사
는 / 은, 도, 만, 조차, 마저, 뿐, 야, 나마, 밖에, ……

품사의 분류에 대해 알아보면, 단어는 구문을

형성하는 의미의 최소적 단위로서 단어의 규정 문제는 구문연구의 기본으로서 지금까지 한국어의 언어학자에 대한 분류 방식이 통일되어 있지 않다. 그러나 단어에 대한 공통적인 의견을 통일하면 의미적인 Minimum Free Morpheme Form으로 분리성을 가지고 있다. 여기서는 주로 사용되는 9 품사로서 품사분류를 따른다.

형태 분류	의미 분류	기 호
체 언(N)	명 사	No
	대 명 사	Ns
	수 사	Nn
용 언(V)	동 사	Vb
	형 용 사	Va
수식언(A)	관 형 사	D
	부 사	Ad
관계언	조 사	K
독립언	감 탄 사	

위의 단어들이 어떤 규칙하에서 조합되어 어절을 생성하는데 주로 체언의 조사 첨용형태와 용언의 활용형태로 분류된다. 이러한 어절의 형태를 형식화하면 다음과 같다.

Type 1 N+Aux(체언+조사)
〈단어〉 → 〈체언〉〈조사〉
〈체언〉 → 〈명사〉 | 〈대명사〉 | 〈수사〉

Type 2 V+Suf(용언+활용어미)
〈단어〉 → 〈용언〉〈활용어미〉
〈용언〉 → 〈동사〉〈형용사〉
〈활용어미〉 → 용언의 활용어미

Type 3 V+N. Suf+Aux(용언+명사형 어미+조사)
〈단어〉 → 〈용언〉〈명사형 어미〉〈조사〉
〈용언〉 → 〈동사〉〈형용사〉
〈명사형 어미〉 → ㅁ | 음 | 기

Type 4 N+〔하, 되, 시키, 이〕+명사형 어미+조사
〈단어〉 → 〈명사〉〈Nov〉〈명사형 어미〉〈조사〉
〈Nov〉 → 하 | 되 | 시키 | 이

Type 5 NO(명사)+〔하, 되, 시키, 이〕+활용형 어미
〈단어〉 → 〈명사〉〈Nov〉〈활용 어미〉
〈Nov〉 → 하 | 되 | 시키 | 이

Type 6 N(체언), A(수식언)의 단독 형태
〈단어〉 → 〈체언〉 | 〈수식언〉
〈수식언〉 → 〈관형사〉 | 〈부사〉

나. 구 현

1) Lexical Analysis

위에서 제시한 품사 분류 및 형태소 분석에 따라 다음과 같이 Token을 만들어 준다.

품사 분류

체언과 조사를 분리하는 Algorithme

n=1

(1) 입력 어절을 받아 List 형태로 바꾼 다음, 그 List를 Reverse한다.

(2) 조사 테이블에서 1 개의 조사(1=음소의 갯수)를 가져와 Null이면 Goto (5) 그 조사의 Reverse List를 구한다.

(3) 입력 List의 nth Atom과 조사 List의 n-th 의 nth Atom을 비교하여
같지 않으면 Goto 2)
만약 N=1 이면 Goto 4)
N=N+1, Goto 3)

(4) Successfully Ended

(5) 체언+조사 외의 형태

(예)

Type 1 명사+조사

?-In-put(철수가, A, B).
A=철수
B=가;

Type 2 그외 형태

?-In-put(주었다, A, B).
A=주었다
B=〔 〕;

형태소 분석

Vst(용언의 어간), Tense(시제), SE(sentence end)

?-Morpheme(주었다, A, B, C)
A=주
B=과거

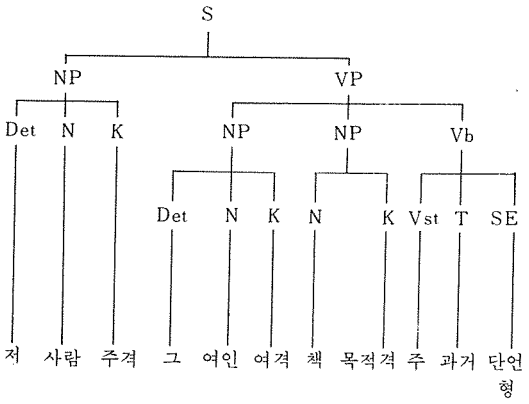
C=단언형

2) Syntax Analysis

위에서 제시한 한국어에 대한 문법을 Prolog Interpreter가 제공하는 DCG (Definitite Clause Grammar)의 문법형태로 바꾸어 Parsing 한다. 다음은 한국어의 입력 문장에 대한 출력 Parsing Tree를 나타낸다.

입력문장 : 저 사람이 그 여인에게 책을 주었었다.

출력결과 : 문장 (명사구(관형사(저), 명사(사람), 주격), 동사구(명사구, (관형사(그), 명사(여인)', 여격), 명사구(명사(책), 목적격), 타동사(주다, 과거, 단언형)



5. 결 론

이제까지 한 일에 대하여 살펴보면, 한글의 단순문에 대한 거의 모든 문장은 분석할 수 있는 시스템을 개발하였다. 그 구체적인 내용을살펴보면 한글의 문장에서 조사의분리를 하고 Parser에 필요한 한글 Token을 만들어주는 Lexical Analizer, 한글의 동사에 대한 활용형을 분리해 내는 Morphosis분석기, 그리고 한글의 단문을 Parsing할 수 있는 Bottom-up Parser의 개발이다. 이는 한글의 데이터를 처리할 수 있는 한글 Prolog로 Implement하였다. 앞으로의 할 일은 이제까지 개발해 놓은 Parser에 한글의 복문 및 중문에 대한 것도 Parsing할 수 있는 기능을 추가하는 작업이 있어야 하겠으며, 현재의 간단한 예에 적용되는 사전을 확장하여 모든 한글언어를 포함한 수 있는 사전을 작성하는 일이 필요하다. 문제점으로는 한글의 문법에 대한 연구원들의 지식이 부족함으로 언어학자들과의 교류가 있어, 서로 공동 연구를 할수 있어야 하겠다.

현재 한글 자연어 처리에 관심을 가지고 연구하고 있는 곳은 한국 전자기술연구소 컴퓨터 연구부를 비롯하여 Kaist전산학과, 서울대 전산기공학과, 한양대전자공학과, Kaist 시스템 공학센터 등이다.