A Study of a Simple Generalized Transition Network Parser for the Japanese Language (I)

Akira Mikami

Abstract

This paper presents a parsing system for the Japanese language. The JPARS system applys a Generalized Transition Network parser to a portion of the grammar of the Japanese language. Its grammatical description is focused on syntactic concepts. It is hoped that the JPARS can be elaborated by further research and used by learners of the Japanese language to test the accuracy of their syntactic knowledge.

1.0. Introduction

The foundation of the field of Computational Linguistics was laid in the 1950's right after the emergence of computers. In the field of natural language understanding, intelligent computer systems such as ELISA, HAL, Eo-HAL, PARRY and K. M. Colby's interviewing program have been constructed [Allen, 1987; Borden, 1987]. The majority of these have been built on the concept of Augmented Transition Network [ATN] parsing [Wood, 1969]. The JPARS system uses a generalization of the basic ATN formalism.

The remainder of this paper will be organized in the following sections: 2.0 an overview of natural language processing, 3.0 the features of Generalized Transition Networks [GTNs], 4.0 the Japanese Language Parsing System, 5.0 a discussion of Japanese grammar structure which is relevant to the program, 6.0 a general summary and conclusion, and 7.0 further research. (5.2, 6.0, 7.0 and references will be dealt with in the continuing paper.)

2.0. Natural Language Processing

Natural languages are the languages used by people in the course of their daily affairs, such as English and Japanese. In other words, natural languages are human languages. The word "natural" is used in contrast with "artificial." Musical notations and computer programming languages are considered to be artificial languages.

According to Tennant [1981] there are two reasons to research natural language processing. Firstly, computers that can use natural languages could be useful tools.

Usually in order to manipulate computers, people have to know a computer language. If computers could be used without learning the nuances of a programming language, computers would be more accessible to people. People would be freed from the annoyance of learning a programming language. Secondly, natural language research will increase our understanding of how human languages work. Without a precise natural language processing system, flaws, inconsistencies and areas of incompleteness in a theory might be left unnoticed.

Already in the 1950's, a strong interest in using computers for translating text from one language to another existed. The United States federal government funded the research heavily. As a result of the enthusiasm of the federal government and researchers in machine translation, research began on translating Russian, German, and French into English. A similar effort was under way in the Soviet Union and in Europe.

In the 1960's researchers received great help from advances in computer technology. Natural language processing systems are strongly dependent on memory. Since the sixties, researchers enjoy a virtual address space of more than 1 billion words, a six-order of magnitude increase over earlier model computers. Also a programming language which is well adapted to building a natural language processing system emerged in the late 1950's; that language is LISP [the LISt Processing language].

Since then natural language processing has been thought as an interdisplinary field drawing upon linguistics, Artificial Intelligence [AI], philosophy, psychology, and the neural sciences.

3.0. Parser grammar

There are reasons why natural languages can be processed by computer programs. In its written form it is composed of linearly arranged discrete entities that occur only in particular combinations. [Sager, 1981] For this reason a grammar can be used in a procedure to recognize the syntactic structure of sentences.

Two things must be considered to examine how the structure of a sentence can be computed. They are the grammar [or a formal specification of the structures allowable in the language] and the parsing technique [or the method of analyzing a sentence to determine its structure according to the grammar].

There are three types of grammar which have been used in natural language processing; Context-Free Grammars, Recursive Transition Networks, and Augmented Transition Networks.

3.1. Context-Free Grammars

Context-Free Grammars [CFGs] are grammars consisting entirely of rules of the form "<symbol> $\longleftarrow$ <symbol>$_1$ .... <symbol>$_n$" for n>=1 or of the form "<symbol> $\longrightarrow$ <symbol>$_1$ .... <symbol>$_n$" for n>=1. The former form is used

for bottom-up parsing and the latter for top-down parsing. [Allen, 1987]

This grammar seems plausible but it is clearly limited; for example, there is no mechanism for marking "dogs" as the plural of "dog." [Loritz, 1987]

3.2. Recursive Transition Networks

A simple parser can be modeled after the transition network formalism. In figure 3.2., the parsing begins at node S. The first word is tested on the first arc. If it satisfies the conditions on the arc, the arc is taken, the word is consumed, and the parsing goes to the next node. Otherwise, if there are more than one arcs leaving the node, the next arc is attempted. If all the words in a sentence are consumed and the parsing reaches the last node, the last arc performs a (SEND)[1] and the sentence is accepted by the grammar that the network represents.

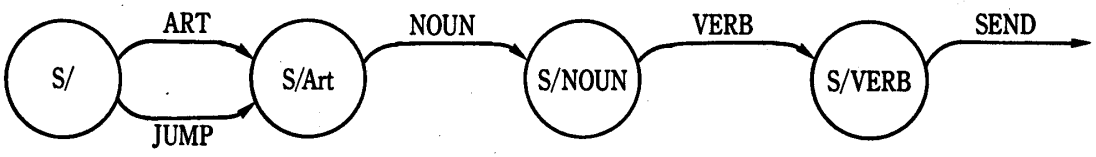

Figure 3.2. RTN diagram

In the course of the parsing, a representation of the structure of the sentence is obtained. When the words in the sentence are consumed, the appropriate arc assigns the word being consumed a grammatical function. For example, the word consumed on the verb arc of figure 3.2. is the verb of the sentence. This representation of the structure of a sentence is displayed in the form of a parse tree.

As the grammar gets bigger, repeated operations in the grammar are made into subroutine calls [i.e. routines or networks called by another routine or subnet]. The noun phrase subroutine is called by (SEEK NP), and when the parsing in the noun phrase network is done, it (SEND)s back to the routine which performed the (SEEK NP). Thus more than one noun phrase network does not have to be built in the program. This contributes to the economy of the program.

Because subroutine networks are called repeatedly, these transition networks are called Recursive Transition Networks [RTNs].

RTN grammar uses several networks to represent the grammar. As a result, it is easy to write and understand. To change the rules concerning the noun phrase, instead of looking at all networks for noun phrases, only one NP network needs to be examined.

3.3. Augmented Transition Networks

Augmented Transition Networks [ATNs] are RTNs with two mechanisms added to facilitate in handling word order changes, insertions, and deletions. The two

mechanisms are the use of registers in recognition, and the addition of arbitrary conditions on the arcs. [Tennant, 1981]

The use of registers is extended to enable storing condition flags and to permit holding temporary structures in memory to facilitate their relocation in underlying structure. The registers, set through conditions and actions on the arcs of the networks during parsing, will be tested to see if a certain condition is already achieved. These registers, in another words, hold features and fragments of the sentence being parsed.

The addition of arbitrary conditions on the arcs extends the condition checking beyond being dependent only on the current state and the next word or phrase in the sentence.

An ATN grammar was introduced by Thorne *et al.* [1968] and Bobrow and Fraser [1969]. Their model was elaborated by William Woods [1969]. [Bates, 1978].

In ATN parsing, if an arc is taken and later proves not to lead to a successful parse of the sentence, the parser backs up to the last choice point and tries an alternative choice. The backing up, however, is not economical if all the arcs taken before have to be reparsed. So if a subnet is successfully parsed, a well-formed flag is set to true and this well-formed subnet will not be parsed again for efficiency's sake. Also the order of the arcs between two nodes should be arranged so that the most economical arc is taken first.

ATN grammars have proved to be flexible, easy to write and debug, able to handle a wide variety of syntactic constructions, and easy to interface to other components of a total system.

3.3.1. Function in ATNs

Just like RTNs, ATNs have arcs and nodes. Each arc has tests and actions for the arc. There are several types of arcs which have different functions. A (CAT) arc is the arc which tests if the current word belongs to the syntactic category. If the arc is taken, the word is consumed and parser goes to the next node.

A (JMP) [jump] arc takes the parser to the next node without consuming a word.

A (SEEK) arc is the arc which tests if several words starting from the current word belong to the syntactic subnet. If the subnet is satisfied, the several words are consumed in the subnet and the subnet (SEND)s the parser back to the original network. In other words the parser stores the current information in the stack and after the subnet (SEND)s the parser back, the tsack is popped and the parsing is continued. After coming back to the original arc, the parser (JMP)s to the next node. [The words are already consumed in the subnet. The original arc does not consume any words.]

An arc can utilize the formerly parsed structure of the sentence and check the current condition. Even if the grammatical category is satisfied, if the other tests are

not satisfied, the arc may not be taken.

An arc also stores information about sentence structure into the registers in order to use it on later arcs. If a sentence is passive, this information is stored in the registers when the parser parses the verb. The information will be used later. For example, even if the parser meets the phrase "by someone" in course of the parsing, unless the parser previously met "be + past participle" structure, the phrase may not be taken as a passive subject.

3.3.2. The GTN functions in Parlisp

Parlisp is a Lisp environment for building Generalized Transition Network [GTN] parsers. Unlike most ATN parers, GTN parsers are capable of both top-down and bottom-up, left-to-right and right-to-left, and depth-first and breadth-first analyses. In particular, JPARS deviates from most ATN parsers by performing right-to-left analysis. The GTN system in Georgetown University is constructed in TLC Lisp. Parlisp provides utilities for building GTN parsers; researchers can build their grammar without worrying about the minute functions of GTNs. By taking advantage of this GTN environment the JPARS system was efficiently constructed and tested.

4.0. The lexicon

The lexicon contains several grammatical features. The first feature is parts of speech, which is mandatory. This lexicon contains nouns, demonstrative adjectives, particles, two types of verbs, three types of adjectives, auxiliary verbs, and conjunctions. The other features are optional. The second feature is the form of verbs and adjectives. This feature will be discussed later. The third feature is information on how words are connected. The fourth feature is case grammar information. Although this system is not fully equipped to use case grammar, this system introduces case grammar analysis to overcome some parsing problems as Hellwig [1987] suggests. The other features help analyze negative, interrogative, passive sentences.

The lexicon of this system is included in the program. As the lexicon is expanded, the lexicon will need to be stored in a separate file.

4.1. The form of verbs

The Japanese language has inflected forms of verbs, adjectives, and auxiliary verbs. McClain [1981] categorizes these forms into six groups; negative base, continuative base, conclusive & attributive base, conditional base, imperative base, and tentative base. Conventional Japanese grammar books describe these six bases in a different way; negative & tentative base, continuative base, conclusive base, attributive base, conditional base, and imperative base. This system uses seven. The negative base is called FORM M [after the Japanese term "Mizenkei"], the continuative base FORM Y [from "renYookei"], the conclusive base FORM E [from "Ending"], the

attributive base FORM T [from "renTaikei"], the conditional base FORM K [from "Kateikei"], the imperative base FORM R [from "meiReikei"], and the tentative base FORM S [from "Suiryookei"].

4.2. Verbs

McClain [1981] divides verbs into three categories according to inflected forms; vowel-stem verbs, consonant-stem verbs, and irregular verbs. Conventional Japanese grammar divides verbs into four categories; "5 dan katuyoo" verbs, "kami 1 dan katuyoo" verbs, "simo 1 dan katuyoo" verbs, and irregular verbs. Because Japanese syllabaries are unable to represent consonants without their accompanying vowels, consonant-stem verbs are further divided into two; "kami 1 dan katuyoo," and "simo 1 dan katuyoo." "Kami 1 dan katuyoo" has the vowel [i] affixed to the stem, and "simo 1 dan katuyoo" has [e] affixed. In this lexicon, vowel-stem verbs are coded as V1 and consonant-stem verbs are coded as V2. There are only two irregular verbs; "kuru," or "come" and "suru," or "do." Although these verbs are not coded in this lexicon, *kuru* should be coded as V2, and *suru* as V1 from the type of following auxiliary verbs.

Japanese verbs have inflection, but lack conjugation. Verbs do not change according to person, number, or gender.

4.3. Adjectives

Conventional Japanese grammar has two types of adjectives; "adjectives," and adjectival verbs. Both types behave differently from adjectives of the English language. They function without being accompanied with verbs. In other words, when they are positioned as predicates, the copulae are excluded.

McClain [1981] categorizes adjectival verbs as copular nouns which are 1) modified by adverbs, not by adjectival words, 2) not used as independent words in sentences but always followed by copulas, and 3) never used as the subjects or the direct objects. It seems reasonable, however, to classify the adjectival verbs as adjectives. In this lexicon, conventional adjectives are coded as A1, adjectival verbs as A2, and a few adjectives which are categorized as both "adjectives" and adjectival verbs are coded as A3.

4.4. Case grammar

Case grammar is coded for nouns, particles, verbs, and adjectives. Fillmore [1968] suggested that the deep case system consists of semantic roles which nouns play in the meaning of sentences. The JPARS system is strictly grammar based. Deep coding of case grammar was avoided. Case grammar, however, is a powerful tool for a parser. For this system, a partial implementation of case grammar could not be avoided. This system uses agents, patients [objects], recipients and locatives

following Cook [1989].

4.5. Others

Interrogative sentences have both interrogative particles and interrogatives in the Japanese language. Interrogative particles are crucial, but interrogatives are optional. Interrogativity is coded in both.

Interrogative sentences are not different in word order from declarative sentences. Even if interrogatives are used, the word order remains the same.

```
(pkg "simpl" :)
;                    * * *  Lexicon  * * *
(progn
(putprop 'simpl:ano          'features '((pos dem)))
(putprop 'simpl:inu          'features '((pos n)))
(putprop 'simpl:dare         'features '((pos n)(stype qu)))
(putprop 'simpl:anata        'features '((pos n)))
(putprop 'simpl:georgetown   'features '((pos n)(sens place)))
(putprop 'simpl:toki         'features '((pos n)(sens time)))
(putprop 'simpl:ha           'features '((pos part)(tp t)))
(putprop 'simpl:ga           'features '((pos part)(ag t)))
(putprop 'simpl:wo           'features '((pos part)(pt t)))
(putprop 'simpl:ka           'features '((pos part)(form e)(stype qu)))
(putprop 'simpl:ni           'features '((pos part)(rcp t)(pvs t)(lc t)(adv time)))
(putprop 'simpl:de           'features '((pos part)(adv place)))
(putprop 'simpl:no           'features '((pos part)(ps t)))
(putprop 'simpl:hoe          'features '((pos v2)(form m)(form y)(case ((ag t)(pt t)))))
(putprop 'simpl:hoeru        'features '((pos v2)(form e)(form t)(case ((ag t)(pt t)))))
(putprop 'simpl:hoere        'features '((pos v2)(form k)(case((ag t)(pt t)))))
(putprop 'simpl:hoero        'features '((pos v2)(form r)(case((ag t)(pt t)))))
(putprop 'simpl:da           'features '((pos v1)(form e)(case((ag t)(pt t)))))
(putprop 'simpl:noni         'features '((pos conjn)(add t)))
(putprop 'simpl:ba           'features '((pos conjn)(add k)))
(putprop 'simpl:to           'features '((pos conjn)(add e)))
(putprop 'simpl:temo         'features '((pos conjn)(add y)))
(putprop 'simpl:ta           'features '((pos aux)(form e)(form t)(add y)))
(putprop 'simpl:tara         'features '((pos aux)(form k)(add y)))
(putprop 'simpl:taro         'features '((pos aux)(form m)(form s)(add y)))
(putprop 'simpl:u            'features '((pos aux)(form e)(add s)))
(putprop 'simpl:rareru       'features '((pos aux)(form e)(form t)(add m)(stype pvs)(ad v2)))
(putprop 'simpl:rare         'features '((pos aux)(form m)(form s)(form y)(add m)(stype pvs)(ad v2)))
(putprop 'simpl:rarere       'features '((pos aux)(form k)(add m)(stype pvs)(ad v2)))
(putprop 'simpl:kawaikaro    'features '((pos a1)(form m)(case((ag t)))))
(putprop 'simpl:kawaikat     'features '((pos a1)(form y)(case((ag t)))))
(putprop 'simpl:kawaiku      'features '((pos a1)(form y)(case((ag t)))))
(putprop 'simpl:kawaii       'features '((pos a1)(form e)(form t)(form k)(case((ag t)))))
(putprop 'simpl:kirei        'features '((pos a2)(form s)(case((ag t)))))
(putprop 'simpl:kireidaro    'features '((pos a2)(form m)(case((ag t)))))
(putprop 'simpl:kireidat     'features '((pos a2)(form y)(case((ag t)))))
(putprop 'simpl:kireide      'features '((pos a2)(form y)(case((ag t)))))
(putprop 'simpl:kireini      'features '((pos a2)(form y)(case((ag t)))))
(putprop 'simpl:kireida      'features '((pos a2)(form e)(case((ag t)))))
(putprop 'simpl:kireina      'features '((pos a2)(form t)(case((ag t)))))
(putprop 'simpl:kireinara    'features '((pos a2)(form k)(case((ag t)))))
```

Listing 4.1. The sample lexicon of the JPARS

Negative sentences have only negative particles or negative adjectives. For further elaboration, the system includes the logic which accepts negative adverbs, which need agreement with the particles and the adjectives.

Passive sentences do not require changes of word order in Japanese. Particles, however, function differently in passive sentences. Passive auxiliaries are coded in this lexicon.

In order to facilitate the parsing of adverb phrases, semantic information, whether a noun has time or place sense, is also coded.

5.0. The JPARS system

The characteristics of the Japanese language are 1) there are no articles, 2) nouns usually do not have special plural forms, 3) pronouns are not treated differently from nouns and are omitted if they can be understood from the context, 4) verbs do not have special forms to indicate person or number, 5) adjectives are closely related to verbs, and take endings according to their tense and mood, 6) there are no cases for nouns and pronouns; particles are used to indicate the relationships between words instead, 7) basic word order is SOV, although the order is quite flexible, 8) conjunctions come at the end of the clause they govern, 9) subordinate clauses must come first in the sentence, and 10) special forms are used to indicate shades of courtesy, respect, and formality. [Bleiler, 1963]

The Japanese language is a left branching language. All modifiers are placed before modified words. A lot of information on the structure of a sentence comes at the end of each segment or phrase ["bunsetsu"]. The most important information on the structure of a sentence is in the verb phrase. For this reason, JPARS parses "backwards" from the end of the input sentence.

Although Sato [1988] says "an ATN-style parser which processes left-branching language's sentences backward from right to left, is also unrealistic," the backward parser is feasible. Firstly as is mentioned above the information needed to parse comes at the end of each phrase, while the most important information comes at the end of the sentence. Native Japanese speakers are told at school to speak the end of a sentence clearly because it is the most important part.

Secondly human brains are not serial processors but parallel distributed processors [Grossberg, 1983; Borchardt, 1988]. A sentence is not simply processed linearly from the beginning of the sentence to the end, but also processed simultaneously. Processing a sentence from the beginning is also not different from processing it from the end as far as a machine is concerned. A parser starts processing an input sentence as a whole when the return key is pressed.

The Japanese language has a variety of forms to indicate shades of courtesy, respect, and formality, but these are dependent on the relationship between a speaker and a listener, a speaker and a referent, and so on. The JPARS only parses one

sentence at a time. As it is not given the luxury of using the context of the sentence, it is not possible to take these human relationships into consideration. The parser only parses sentences of neutral or normal politeness.

The JPARS uses the Japanese conventional transcription which is further romanized to facilitate Parlisp environment. The Japanese conventional transcription uses "wo" and "ha" for articles which are pronounced /o/ and /wa/ respectively. In a phonetically precise spelling system, "shi," "chi," "tsu," and so on should be used, but the conventional transcription uses "si," "ti," "tu," and so on.

The parser does not have the ability to cut a sentence into words. Although a Japanese sentence is usually written without spaces between words, to parse in the JPARS words have to be already divided by spaces.

5.1. The GTNs of this system

There are three networks in the parser; the sentence network, the noun phrase network, and the relative clause ntwork.

5.1.1. Sentence Network

The sentence network [figures 5.1.1.] starts with the node S/ and ends with the node S/S.

Node S/:ARC 1. Proceeding from the end of the input sentence, the node S/ checks if the sentence ends with a particle. If it does, the particle is registered. If the particle indicates that the sentence is a negative, interrogative, and/or passive sentence, the information is also registered as STYPEXNG, STYPEXQU, and STYPEXPV [Sentence TYPE in auXiliary NeGative, QUestion, and PassiVe). After taking this arc, the parser goes back to the node S/.

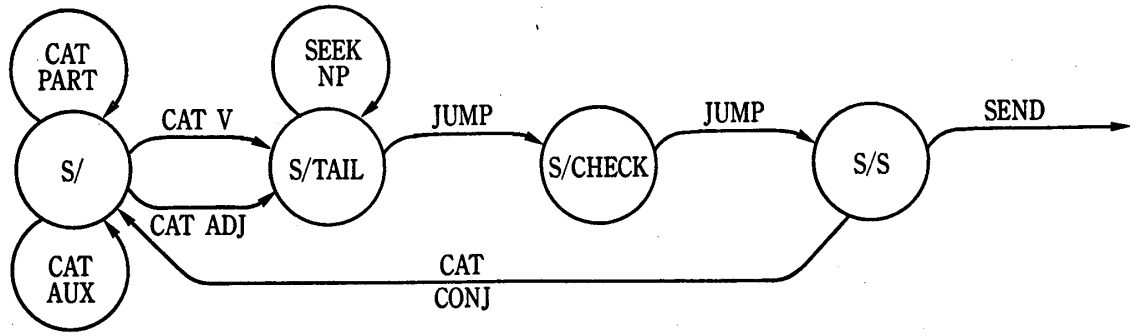

Figure 5.1.1. The diagram of the Sentence Network

Node S/:ARC 2. The node S/ then checks if an auxiliary verb is found. Because Japanese auxiliary verbs are inflected, the inflected forms have to agree with the grammatical context. If an auxiliary verb is detected, the form of the auxiliary verb is checked to see if it is an ending form. If a sentence ends with an auxiliary verb or an auxiliary verb folowed by a particle, the auxiliary verb has to be the ending form. If a second auxiliary verb is detected, the parser checks the agreement

between the two auxiliary verbs. The second auxiliary verb [which is closer to the beginning of the sentence in this parser] must be the correct form to connect to the first auxiliary verb [which is closer to the end of the sentence]. For example, a negative auxiliary verb "nai" has to be preceded by (FORM M), so the combination "rare nai" is grammatically correct, but "rareru nai" is incorrect. If the form is correct, this information is stored as (AGREE) information. Just like particles, the information about the word itself and negativity, interrogativity, and passivity is stored in the registers. This arc also goes back to the node S/.

Node S/:ARC 3. If a verb is found, the parser checks the form of the verb. If the sentence ends with the verb or a particle, the verb has to be in the ending form. If the sentence ends with an auxiliary verb, the verb form has to meet the requirement of the auxiliary verb. For example, the form M of the verb "hoeru" [bark] is "hoe". So "hoe nai" is correct, but not "hoeru nai." Again the information of the verb is stored. The parser goes to the next node S/tail.

Node S/:ARC 4. Second, an adjective may precede an auxiliary verb or a particle. The Japanese adjectives behave differently from those of English. The Japanese adjectives are more like verbs, and do not have to be preceded by verbs. In node S/, an adjective is sought only when no verb is found. All procedures are almost the same as the ones for a verb.

If no verb or adjective is found, the parsing fails.

Node S/tail:ARC 1, 2, 3, and 4. The node S/tail invokes the NP network. If an NP is found, the parser assigns the NP to grammatical functions of NPs; topics, agents, patients, recipients, adverbs, and locations. To facilitate backing up the parser, NP seeking is done in four arcs. This allows a NP to be assigned to every possible function.

Node S/tail:ARC 5. The parser proceeds to the next node S/CHECK.

Node S/CHECK: The node S/CHECK checks if the NPs fit into the NP slots of the verb or the adjective. The Japanese language does not have to have all the case slots for the verb or the adjective filled, but if an NP exists, the verb or the adjective must have a case slot to accept that NP. ["Case slot" is derived from "frames" and their "slots" in AI]. In other words, even if a verb has a case of an agent and a patient, NPs for an agent and a patient do not have to exist, but if an agent NP and a patient NP exist, the verb or the adjective has to be coded for the cases of agent and patient. Every verb or adjective has its own slots for NPs, and NPs have to fit into the slots. Empty slots, however, are acceptable. The parser proceeds to the next node S/S.

Node S/S:ARC 1. The parser looks for a conjunction. If it finds one, it clears the register, and returns to the node S/.

Node S/S:ARC 2. If no word to parse is left, the parsing ends.

If the parsing is successfully completed, the parser displays "OK" on the screen.

All final register information in this sentence network is stored in TREE. After parsing, PP TREE will display all this information.

5.1.2. Noun Phrase Network

Node NP/:Arc 1. The node NP/ checks if a particle is detected. The particle attaching to a noun is different from the one attaching to a verb or an auxiliary verb. The particle plays a very important role in deciding if a noun is a topic, an agent, a patient, a recipient, an adverb or a location. When the particle is found, the particle is registered.

Node NP/:Arc 2. After registering a particle, the node NP/ also checks if a noun is detected. If no noun is found, the NP network fails. Otherwise the parser proceeds to the next node NP/N.

Node NP/N:Arc 1. For program efficiency, if no word to parse is left, the parser immediately goes to the node NP/NP without seeking another modifier.

Node NP/N:Arc 2, 3, 4, and 5. The node NP/N checks if a demonstrative adjective, an adjective or adjectives, a possessive noun phrase, and/or a relative clause optionally modify the noun. All modifiers found are also registered.

Node NP/NP: In the node NP/NP, the register information obtained in NP network is sent up to the original network to be utilized for further parsing.

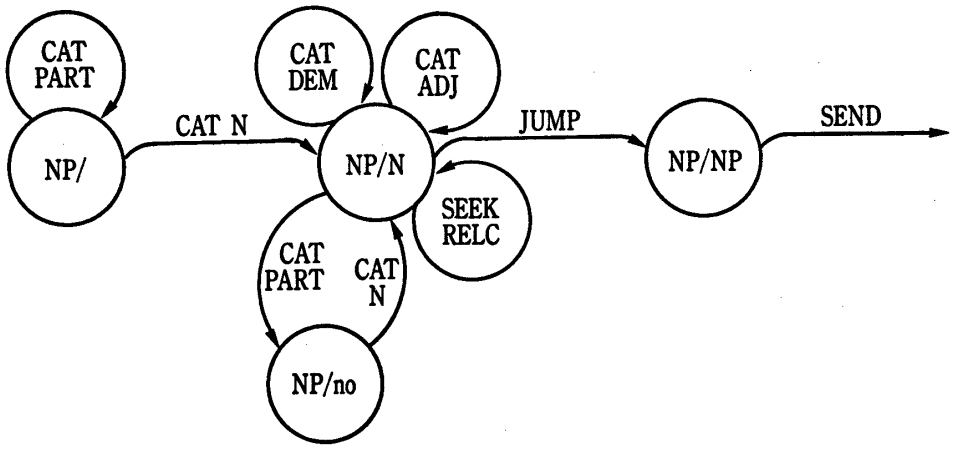

Figure 5.1.2. The diagram of the Noun Phrase Network

5.1.3. Relative Clause Network

The relative clause network [figure 5.1.3.] is similar to the sentence network. The differences are 1) a particle is not added at the end of the clause, 2) the particle "ni" is used for a patient, 3) the possessive particle "no" is used for an agent, and 4) the node RELC/RELC does not take a conjunction into consideration. The last difference is only for the ease of programming this parser. The Japanese language does not have relative pronouns. As the consequence, it is semantics which decides if a clause before a relative clause is a relative clause connected by a conjunction, or another

sentence connected by a conjunction. This parser is a grammar parser which does not check any semantic aspect. Because of this restriction more than one relative clause connected by a conjunction is not allowed in this parser.

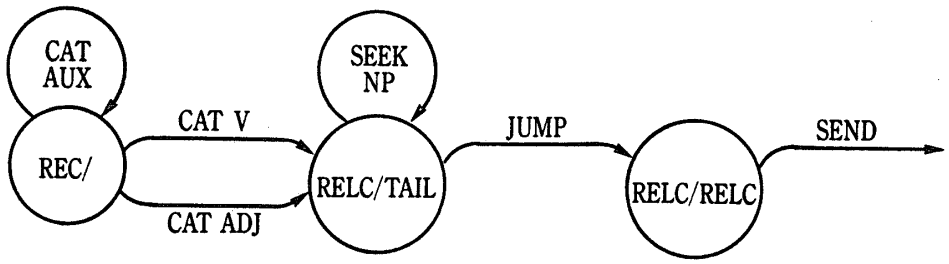

Figure 5.1.3. The diagram of the Relative Clause Network

The final information stored in this relative clause network is stored in TREER. To get this TREER after parsing, PP TREER will give the content.

```
;================  *  *  THE GRAMMAR  *  *  *  ===============
(defstate 'START/ '(
   (1 (t (seek 's/)    )
      (a  t)           ) )
))))))))))

;   * * * *  Sentence Level  * * * *
(defstate 's/ '(
  (1  (t  (and (newcat 'part)
               (null (getr 'part))
               (formcheck * 'form 'e) )                          )
      (a  (if (equal (getfeature * 'stype) 'ng)
              (setr   'stypexng  'ng) )
          (if (equal (getfeature * 'stype) 'qu)
              (setr   'stypexqu  'qu) )
          (setr 'part *)
          (to   's/)                                             )   )
  (2  (t  (newcat  'aux)
          (or (and (null  (getr 'aux) )
                   (formcheck * 'form 'e))
              (and (getr  'aux)
              (checkagree)))                                     )
      (a  (if (equal (getfeature * 'stype) 'ng)
              (setr   'stypexng  'ng))
          (if (equal (getfeature * 'stype) 'qu)
              (setr   'stypexqu  'qu))
          (if (equal (getfeature * 'stype) 'pvs)
              (setr   'stypexpv  'pvs))
          (setr 'agree (getfeature * 'add))
          (setr 'aux *)
          (to   's/)                                             )   )
  (3  (t  (or (newcat 'v1)
              (newcat 'v2))
          (or (and (null (getr 'aux))
                   (formcheck * 'form 'e))
              (and (null (getr 'stypexpv))
                   (getr  'aux)
                   (checkagree))
              (and (getr  'aux)
                   (checkagree)
                   (getr 'stypexpv)
                   (equal (getfeature (getr 'aux) 'ad)
                          (getfeature * 'pos))))                 )
      (a  (setr 'tail *)
          (to   's/tail)                                         )   )
  (4  (t  (or (newcat 'a1)
              (newcat 'a2)
              (newcat 'a3))
          (or (and (null (getr 'aux))
              (formcheck * 'form 'e))
          (and (getr  'aux)
               (checkagree)))                                    )
      (a  (setr 'tail *)
```

Listing 5.1.4. The grammar of the JPARS

```
          (to   's/tail )                                          )
) )  )))))))))

(defstate 's/tail '(
  (1  (t   (or (newcat 'n)
               (and (and (newcat 'part)
                         (null (getfeature * 'form)))
                    (or  (and (null (getr 'topic))
                              (getfeature  * 'tp))
                         (and (null (getr 'agent))
                              (getfeature  * 'ag))
                         (and (null (getr 'agent))
                              (getr 'stypexpv)
                              (getfeature * 'pvs))
                         (and (null (getr 'patient))
                              (getfeature * 'pvs))
                         (and (null (getr 'patient))
                              (getfeature * 'pt )))))
           (sendr 'tail)
           (seek  'np/)                                            )
      (a   (or (and (null (getr 'topic))
                    (getfeature  (getr 'part) 'tp)
                    (setr 'topic (getr 'head)))
               (and (getr 'stypexpv)
                    (null (getr 'patient))
                    (getfeature  (getr 'part) 'ag)
                    (setr 'patient (getr 'head)))
               (and (null (getr 'agent))
                    (getfeature  (getr 'part) 'ag)
                    (setr 'agent (getr 'head)))
               (and (getr 'stypexpv)
                    (null (getr 'agent))
                    (getfeature  (getr 'part) 'pvs)
                    (setr 'agent (getr 'head)))
               (and (null (getr 'patient))
                    (getfeature  (getr 'part) 'pvs)
                    (setr 'patient (getr 'head)))
               (and (null (getr 'patient))
                    (getfeature  (getr 'part) 'pt)
                    (setr 'patient (getr 'head))))
           (setr 'tailnt *nt)
           (jmp  's/tail)                                          )  )
  (2  (t   (and (newcat 'part)
                (null (getfeature * 'form))
                (null (getr 'loc))
                (getfeature * 'lc))
           (seek 'np/)
           (equal (getfeature (getr 'head) 'sens) 'place)          )
      (a   (setr 'loc (getr 'head))
           (setr 'tailnt *nt)
           (jmp  's/tail)                                          )  )
  (3  (t   (and (newcat 'part)
           (null (getfeature * 'form))
```

Listing 5.1.4. The grammar of the JPARS [continued]

```
          (null (getr 'adv))
          (getfeature  * 'adv)  )
          (seek 'np/)
          (equal (getfeature (getr 'head)  'sens) 'time)  )
      (a  (setr 'adv (getr 'head))
          (setr 'tailnt *nt)
          (jmp  's/tail)                                    )   )
 (1  (t   (and (newcat 'part)
               (null (getfeature * 'form))
               (null (getr 'rcp))
               (getfeature  * 'rcp))
          (seek 'np/)                                       )
      (a  (setr 'rcp (getr 'head))
          (setr 'tailnt *nt)
          (jmp  's/tail)                                    )   )
  (5  (a  (jmp  's/check)                                   )   )
))))))))))))))))))))

(defstate 's/check '(
  (1  (t   (or (if (and (getr 'topic) (null (getr 'agent)))
                   (setr 'agent (getr 'topic))                  nil)
               (if (and (getr 'topic) (null (getr 'patient)))
                   (setr 'patient (getr 'topic))                nil)
               (if (null (getr 'topic))
                   t                                            nil)
           (and (if (getr 'agent)
                    (get  (getr 'tail) 'case 'ag)   t)
                (if (getr 'patient)
                    (get  (getr 'tail) 'case 'pt)   t)
                (if (getr 'rcp)
                    (get  (getr 'tail) 'case 'rcp)  t)
                (if (getr 'loc)
                    (get  (getr 'tail) 'case 'lc)   t)  )
           (or  (and (getr 'stypenng)
                     (equal (getr 'stypexng)
                            (getr 'stypenng) ) )
                (and (getr 'stypenqu)
                     (equal (getr 'stypexqu)
                            (getr 'stypenqu) ) )
                (and (null  (getr 'stypenng) )
                     (null  (getr 'stypenqu) ) )  )         )
 (a  (jmp 's/s)                                             )   )
))   )))))))))

(defstate 's/s '(
  (1 (t  (newcat 'conjn)                                    )
     (a  (if (null tree1)
         (setq tree1 (car regs))
         (setq tree1 (append (car regs) tree1)))
         (setq regs nil)
         (setr 'aux *)
         (setr 'conjn *)
         (setr 'agree (getfeature * 'add))
```

Listing 5.1.4. The grammar of the JPARS [continued]

```
            (to 's/)                                                    ) )
    (2 (t   (null *)                                                    )
       (a   (if (null tree1)
                (setq  tree  (car regs))
                (setq  tree  (append (car regs) tree1)))
            (send)                                                      ) )
))    )))))))))

;          * * * *  Noun  Phrase  Level  * * * *
(defstate 'np/ '(
  (1  (t (newcat 'part)
         (or  (null (getr 'part))
              (and  (getr 'part)
                    (equal (getr 'agree)
                    (getfeature * 'form))))                             )
       (a (setr  'part *)
          (setr  'agree (getfeature * 'add) )
          (to 'np/)                                                     ) )
  (2  (t (newcat 'n)                                                    )
       (a (if (equal (getfeature * 'stype) 'ng)
              (setr 'stypenng 'ng))
          (if (equal (getfeature * 'stype) 'qu)
              (setr 'stypenqu 'qu))
          (if (and (null (getr 'part))
                   (equal (string (getr 'tail)) "da"))
              (setr 'part 'simpl:wo))
          (setr 'head *)
          (to 'np/n)                                                    ) )
))    ))))))))))
(defstate 'np/n '(
 (1   (t   (null *)                                                     )
      (a   (jmp   'np/np)                                               ) )
 (2   (t   (seek 'relc/)                                                )
      (a   (setr 'relc *nt)
           (jmp   'np/n)                                                ) )
 (3   (t   (null (getr 'dem))
           (newcat 'dem)                                                )
      (a   (if (equal (getfeature * 'stype) 'ng)
               (setr 'stypenng 'ng))
           (if (equal (getfeature * 'stype) 'qu)
               (setr 'stypenqu 'qu))
           (setr 'dem *)
           (to 'np/n)                                                   ) )
 (4   (t   (or (newcat 'a1)
               (newcat 'a2)
               (newcat 'a3)  )
           (formcheck * 'form 't)                                       )
      (a   (setr 'adj *)
           (to 'np/n)                                                   ) )
 (5   (t   (newcat 'part)
           (getfeature * 'ps)                                           )
      (a   (to 'np/no)                                                  ) )
 (6   (a   (jmp 'np/np)                                                 ) )
```

Listing 5.1.4. The grammar of the JPARS [continued]

```
)) )))))))))

(defstate 'np/no '(
  (1  (t (newcat 'n))
      (a (setr 'poss *)
         (to 'np/n)                                          )   )
)) ))))))

(defstate 'np/np '(
  (1  (a (upr  'part)
         (upr  'head)
         (if (getr 'adj)  (upr  'adj))
         (if (getr 'poss) (upr  'poss))
         (if (getr 'dem)  (upr  'dem))
         (if (getr 'stypenqu) (upr  'stypenqu))
         (if (getr 'stypenng) (upr  'stypenng))
         (send)                                              )   )
 ))))))

;     * * * * * * * Relative Clause * * * * * * *
(defstate 'relc/ '(
  (1  (t  (newcat  'aux)
          (or (and (null (getr 'aux))
                   (formcheck * 'form 't))
              (and (getr 'aux)
                   (checkagree)))                            )
      (a  (if (equal (getfeature * 'stype) 'ng)
              (setr  'stypexng 'ng))
          (if (equal (getfeature * 'stype) 'qu)
              (setr  'stypexqu 'qu))
          (if (equal (getfeature * 'stype) 'pvs)
              (setr  'stypexpv 'pvs))
          (setr 'agree (getfeature * 'add))
          (setr 'aux *)
          (to 'relc/)                                      )   )
  (2  (t  (or (newcat  'v1)
              (newcat  'v2))
          (or (and (null (getr 'aux))
                   (formcheck * 'form 't))
              (and (null (getr 'stypexpv))
                   (getr  'aux)
                   (checkagree))
              (and (getr  'aux)
                   (checkagree)
                   (getr 'stypexpv)
                   (equal (getfeature (getr 'aux) 'ad)
                   (getfeature * 'pos))))                   )
      (a  (setr 'tail  *)
          (to   'relc/tail)                                )   )
  (3  (t  (or (newcat  'a1)
              (newcat  'a2)
              (newcat  'a3))
          (or (and (null (getr 'aux))
```

Listing 5.1.4. The grammar of the JPARS [continued]

```
                    (formcheck * 'form 't))
                (and (getr  'aux)
                     (checkagree)))                                    )
        (a  (setr 'tail *)
            (to   'relc/tail)                                          )   )
) )  )))))))))

(defstate 'relc/tail '(
  (1  (t   (or (newcat 'n)
               (and (and (newcat 'part)
                         (null (getfeature * 'form)))
                    (or  (and (null (getr 'topic))
                              (getfeature  * 'tp))
                         (and (null (getr 'agent))
                              (getfeature  * 'ag))
                         (and (null (getr 'agent))
                              (getr 'stypexpv)
                              (getfeature  * 'pvs))
                         (and (null (getr 'patient))
                              (getfeature  * 'pvs))
                         (and (null (getr 'patient))
                              (getfeature  * 'pt ))
                              (getfeature  * 'ps )) ) )
           (seek 'np/)                                                 )
       (a   (or (and (null (getr 'topic))
                     (getfeature  (getr 'part) 'tp)
                     (setr 'topic (getr 'head)))
                (and (getr 'stypexpv)
                     (null (getr 'patient))
                     (getfeature  (getr 'part) 'ps)
                     (setr 'patient (getr 'head)))
                (and (null (getr 'agent))
                     (getfeature  (getr 'part) 'ps)
                     (setr 'agent (getr 'head)))
                (and (getr 'stypexpv)
                     (null (getr 'patient))
                     (getfeature  (getr 'part) 'ag)
                     (setr 'patient (getr 'head)))
                (and (null (getr 'agent))
                     (getfeature  (getr 'part) 'ag)
                     (setr 'agent (getr 'head)))
                (and (getr 'stypexpv)
                     (null (getr 'agent))
                     (getfeature  (getr 'part) 'pvs)
                     (setr 'agent (getr 'head)))
                (and (null (getr 'patient))
                     (getfeature  (getr 'part) 'pvs)
                     (setr 'patient (getr 'head)))
                (and (null (getr 'patient))
                     (getfeature  (getr 'part) 'pt)
                     (setr 'patient (getr 'head))))
          (setr 'tailnt *nt)
          (jmp  'relc/tail)                                            )   )
```

Listing 5.1.4. The grammar of the JPARS [continued]

```
  (2  (t (and (newcat 'part)
              (null (getfeature * 'form))
              (null (getr 'loc))
              (getfeature  * 'lc))
         (seek 'np/)
         (equal (getfeature (getr 'head) 'sens) 'place)    )
      (a (setr 'loc (getr 'head) )
         (setr 'tailnt *nt)
         (jmp  'relc/tail)                                  )   )
  (3  (t (and (newcat 'part)
              (null (getfeature * 'form))
              (null (getr 'adv))
              (getfeature  * 'adv))
         (seek 'np/)
         (equal (getfeature (getr 'head) 'sens) 'time)     )
      (a (setr 'adv (getr 'head) )
         (setr 'tailnt *nt)
         (jmp  'relc/tail)                                  )   )
  (4  (t (and (newcat 'part)
              (null (getfeature * 'form))
              (null (getr 'rcp))
              (getfeature * 'rcp))
         (seek 'np/)                                        )
      (a (setr 'rcp (getr 'head) )
         (setr 'tailnt *nt)
         (jmp  'relc/tail)                                  )   )
  (5  (a (jmp  'relc/relc)                                  )   )
) )  )))))))))

(defstate 'relc/relc '(
  (1  (t  (or (if (and (getr 'topic) (null (getr 'agent)))
                  (setr 'agent (getr 'topic))
                  nil)
              (if (and (getr 'topic) (null (getr 'patient)))
                  (setr 'patient (getr 'topic))
                  nil)
              (if (null (getr 'topic))
                  t
                  nil) )
          (and (if  (getr 'agent)
                    (get (getr 'tail) 'case 'ag)  t)
               (if  (getr 'patient)
                    (get (getr 'tail) 'case 'pt)  t)
               (if  (getr 'rcp)
                    (get (getr 'tail) 'case 'rcp) t)
               (if  (getr 'loc)
                    (get (getr 'tail) 'case 'lc)  t) )
          (or  (and (getr 'stypennng)
                    (equal (getr 'stypexng)
                    (getr 'stypennng)))
               (and (getr 'stypenqu)
                    (equal (getr 'stypexqu)
                           (getr 'stypenqu)))
```

Listing 5.1.4. The grammar of the JPARS [continued]

```
                (and (null  (getr 'stypenng))
                     (null  (getr 'stypenqu))))                    )
        (a  (if (null treer1)
                (and (setq  treer1 (car regs))
                (setq  treer  (car regs)) )
            (setq treer (append (car regs) treer1)))
            (send)                                                 )   )
))))))))))
```

Listing 5.1.4. The grammar of the JPARS [continued]

[The remainder of this paper will be continued.]

1. Parentheses are used in Lisp to delimit lists and function calls. Parentheses will be used here and elsewhere to distinguish Lisp functions, function calls and lists. Parenthesized English text will be enclosed in brackets [ ].