Transition and Parsing State and Incrementality in Dynamic Syntax*

Masahiro Kobayashi[a] **and Kei Yoshimoto**[b]

[a]University Education Center, Tottori University, 4-101 Koyama-cho Minami,
Tottori, 680-8550 Japan, kobayashi@uec.tottori-u.ac.jp
and
[b]Center for the Advancement of Higher Education, Tohoku University,
Kawauchi 41, Aobaku, Sendai, 980-8576 Japan, kyoshimoto@mail.tains.tohoku.ac.jp

Abstract. This paper presents an implementation of a gramar of Dynamic Syntax for Japanese. Dynamic Syntax is a grammar formalism which enables a parser to process a sentence in an incremental fashion, establishing the semantic representation. Currently the application of lexical rules and transition rules in Dynamic Syntax is carried out arbitrarily and this leads to inefficient parsing. This paper provides an algorithm of rule application and partitioned parsing state for efficient parsing with special reference to processing Japanese, which is one of head-final languages. At the present stage the parser is still small but can parse scrambled sentences, relative clause constructions, and embedded clauses. The parser is written in Prolog and this paper shows that the parser can process null arguments in a complex sentence in Japanese.

Keywords: Japanese, Dynamic Syntax, implementation, rule application, null argument

1. Introduction

Incremental processing of a sentence as it is inputted from left to right has been taken as most accurately simulating human sentence processing. When we attempt theoretically to account for this incrementality of sentence processing, however, we notice that there is a difference between head-initial languages and head-final languages. What we need to take into account is that in head-initial languages, like English, parsers can *look ahead* to syntactic structures to be established to some extent as a word is consumed. By contrast, this is not the case in head-final languages such as Japanese, because in these languages the syntactic positions of NPs remain unfixed until the matrix verb is inputted finally in the sentence. The Dynamic Syntax approach (Kempson, Meyer-Viol, and Gabbay, 2001; Cann, Kempson, and Marten, 2005), which allows the parser to process a sentence in an incremental fashion, has settled this difficulty by adopting structural underspecification. However, Kempson, Meyer-Viol, and Gabbay (2001) and Cann, Kempson, and Marten (2005) do not provide an explicit algorithm for implementation of the framework as formally as Moot (1999) presents *Grail*, the toolkit for a parser of Categorial Grammar Logics. This paper will delineate the basic idea of a parser for Japanese based on the Dynamic Syntax (hereafter, DS) framework, providing the algorithm of the application of lexical rules and transition

*We would like to thank two anonymous reviewers for valuable comments.
0

Copyright 2007 by Masahiro Kobayashi and Kei Yoshimoto

rules. This paper will also show that the parser is able to cope with null arguments or empty pronouns in embedded clauses. Currently the parser is implemented in Prolog.

The outline of this paper is as follows: the subsequent section will provide a brief introduction to the DS formalism and address issues of parsing inefficiency and the algorithm of rule application, discussing the previous studies. Section 3 will illustrate the implementation of the DS formalism and our algorithm to process head-final languages like Japanese. Section 4 shows how the parser deals with the complex sentences with empty pronouns. Section 5 will be devoted to the discussion. Section 6 will be a conclusion.

2. Dynamic Syntax and Issues of Implementation

2.1. Formalization and Unfixed Nodes

This subsection presents a brief illustration of the DS formalism. DS is a grammar formalism which allows a parser to process a sentence from the onset word to the final word in an incremental way. The (partial) tree structure grows larger and larger, step by step, by the application of transition rules as well as the word consumption; the initial tree structure $\mathcal{T}_0$ shifts to a subsequent structure $\mathcal{T}_1$, and ultimately to the final tree structure $\mathcal{T}_n$ as seen in (1).

(1) $\mathcal{T}_0 \Longrightarrow$ rule application $\Longrightarrow \mathcal{T}_1 \Longrightarrow \cdots\cdots \Longrightarrow \mathcal{T}_{n-1} \Longrightarrow$ rule application $\Longrightarrow \mathcal{T}_n$

The tree structure $\mathcal{T}$ is a set of nodes in the DS formalism, and the initial tree structure consists of a single node $\{Tn(a)$, $?Ty(t)$, $\Diamond\}$ (Kempson, Meyer-Viol, and Gabbay, 2001, pp.57) called the root node, where Tn is a predicate which expresses an $address$ of a node which includes the variable a indicating that a position of this node has not been specified yet. Moreover, the predicate Tn expresses the relationship between nodes; for instance, Tn(01) is the functor node of Tn(0), Tn(00) is the argument node of Tn(0). $Ty(t)$ is a predicate which expresses the type of the node. The question mark '?' prefixed to $Ty(t)$ is called a $requirement$, which expresses that $Ty(t)$ needs to be satisfied by the end of processing: in this initial state, the goal of processing is to meet the condition that this node is of type t. In this sense, the parsing formalism of DS is called goal-directed. The pointer '$\Diamond$' is used to highlight a node and the lexical rules and transition rules are applied to only pointed nodes.

The lexical items themselves take the form of rules such as **IF** A holds **THEN** execute B **ELSE** execute C. These lexical rules as well as transition rules update the current tree structure to the subsequent structure. We assume the following transition rules in this paper for Japanese; LOCAL $*$ADJUNCTION, which is used to introduce an unfixed NP for local scrambling to the tree structure; GENERALISED ADJUNCTION for introducing the embedded clause to the tree; $*$ADJUNCTION for long-distance scrambling; ELIMINATION for the functional application; COMPLETION, used to bring the pointer back to the mother node; THINNING for deleting or satisfying the requirements; LINK ADJUNCTION for introducing the relative clauses to the tree; LINK EVALUATION for building up the semantic representation for relative clause construction; MERGE for the unification of unfixed nodes with the vacuous, fixed nodes.

As we have accounted for, NPs in Japanese need to be kept unfixed until the verb has been consumed when the parser processes a sentence. In (2a) the sentence initial NP is the object of the simple main clause, whereas the first NP of (2b) is the object of the embedded, subordinate clause. However, the parser cannot specify the fixed positions of the initial NP of both (2a) and (2b) when the parser consumes them.

(2) a. Bōru o John ga nageta.
ball ACC John NOM throw-PAST
"John threw the ball."

b. Bōru o John ga nageta to Tarō ga itta.
ball ACC John NOM throw-PAST COMP Taro NOM say-PAST
"Taro said that John threw the ball."

LOCAL ∗ADJUNCTION introduces an unfixed node which is immediately dominated by the root node. By contrast, in (2b), first, the unfixed root node of type t for the embedded clause is introduced by GENERALISED ADJUNCTION, then LOCAL ∗ADJUNCTION introduces an unfixed node which is immediately dominated by the root node of the embedded clause. These unfixed nodes need to find their fixed position by the end of parsing by means of the application of MERGE. Thus, the DS formalism is non-deterministic because the processing of the same sentence initial NP may yield multiple different tree structures. For a more detailed description about the formalism, readers are referred to Kempson, Meyer-Viol, and Gabbay (2001) and Cann, Kempson, and Marten (2005).

2.2. Issues of Implementation

This subsection addresses the issues of implementation of the DS framework. When we try to develop an account of linguistic phenomena with limited data, the paper-and-pencil approach is very inefficient. In implementing the DS framework we have two problems to address. The first issue is how we should handle the unfixed nodes efficiently: in the previous subsection we showed that in some case the sentence initial NP is treated as the locally unfixed node introduced by LOCAL ∗ADJUNCTION, whereas it is treated as the unfixed node dominated by the type t, an unfixed root node for the subordinate clause, introduced by GENERALISED ADJUNCTION. These unfixed nodes all need to find their fixed positions in the course of processing with the transition rule MERGE or other update processes. Originally Kempson, Meyer-Viol, and Gabbay (2001) deals with nodes in the tree structure as a set and thus it is very inefficient to search the pointed node among a set each time a rule is applied, if we implement the grammar this way. Purver and Otsuka (2003) and Otsuka and Purver (2003) present a generation and parsing model for English, in which the tree structure is represented as a set of nodes in Prolog. Since in Japanese we need to keep some nodes unfixed, this is not so efficient an approach to Japanese. Instead if we try to cope with these nodes as a normal binary tree structure, we will get into trouble. For example, in Prolog the normal binary tree structure is often implemented recursively: `node(s, node(np, [], []), node(vp, [], []))` is the well-known Prolog notation for the tree diagram in which S goes to NP and VP. Unfixed nodes are not easy to be handled in this way. We propose a partitioned-parsing state to cope with these unfixed nodes and MERGE operation.

The second issue this paper addresses is the algorithm of lexical rules and transition rules of DS. DS is a kind of an abstract grammar formalism and in the current framework lexical rules and transition rules are applied arbitrarily: DS does not provide any algorithm to specify which rule is applied when. When the parser processes a Japanese sentence such as simple sentences, relative clause constructions, and complex sentences, we assume that 8 rules are needed to establish the semantic representation as we explained in the previous subsection. If the rules are applied in an arbitrary way, the parser has to compute the repeated permutation of the rules, including vacuous application. This results in very inefficient parsing. Although Purver and Otsuka (2003) and Otsuka and Purver (2003) propose a generation and parsing model based on the DS framework, which is written in Prolog, their parser mainly deals with only English. As we explained, in head-final languages such as Japanese various unfixed nodes are introduced and merged in the course of parsing so that we need a different algorithm. In the subsequent section we propose the algorithm to implement a DS grammar for Japanese, a slightly modified version of Kobayashi (2007).

3. Implementation of DS for Japanese

3.1. Parsing State

This subsection illustrates the parsing state. Our parser consists of a triple $\langle W, S, P\rangle$, where W is a list of words to be consumed (words which have not been processed yet), S is a list of (partial) tree structures which the parser obtains until it processes the current word, P indicates the position at which the pointer exists right now. The initial and final position of the pointer is the root node. The parsing successfully ends if and only if W contains no word to be scanned, and the pointer goes back to the root node of the resulting fixed tree structure; P is `pn(fixed, [root])`, and the semantic formula is built up at the root node. The basic idea of the parsing process, from the initial to the final stage, is shown in (3).

(3) $\langle W_0, S_0, \texttt{pn(fixed, [root])}\rangle \Rightarrow \cdots\cdots \Rightarrow \langle \phi, S_n, \texttt{pn(fixed, [root])}\rangle$

In a triple $\langle W, S, P\rangle$, S consists of a double $\langle R, T\rangle$ where R is a list of rules which has been applied to establish the current tree structure. T consists of a triple $\langle F, G, L\rangle$ where F is a fixed tree structure, G is a tree structure which is introduced by GENERALISED ADJUNCTION rule, and L is a tree structure which is introduced by LINK ADJUNCTION. This definition of the parsing state allows the parser to have additional spaces for unfixed nodes and execute MERGE operation efficiently.

3.2. Algorithm of Rule Application

This subsection describes the algorithm for processing Japanese sentence on the basis of the DS framework. We also take a brief look at the approach to English sentences presented by Purver and Otsuka (2003) and Otsuka and Purver (2003). The basic idea of Purver and Otsuka's approach is to divide transition rules into two groups in order to improve efficiency. One group is `always_rules`, which apply forcibly between the transitions, and the other is `possible_rules`, which do not necessarily apply but can be applied. Since Japanese is a typical head-final language, it is not easy to predict what syntactic structure comes next when the parser processes a sentence in an incremental fashion. To put it another way, the syntactic structure of Japanese is determined (or fixed) by the word the parser has processed to some extent. Japanese is lexically driven in that sense. In order to process Japanese sentences efficiently we need to take into account when lexical rules are applied together with transition rules. Therefore, Purver and Otsuka's approach is efficient to parse English sentences, but it does not apply directly to Japanese. This subsection proposes an algorithm for Japanese, which is a slightly revised version of Kobayashi (2007).

The key idea of this algorithm is that the transition rules are classified into two groups: one is `tree_expansion_rules`, which are used to expand the current tree structure, providing, for example, a new unfixed node; the other one is `node_update_rules` used to update the information of the pointed node and to move the pointer up to the mother node (COMPLETION). `tree_expansion_rules` includes LOCAL ∗ADJUNCTION, GENERALISED ADJUNCTION, and LINK ADJUNCTION. By contrast, MERGE, LINK EVALUATION, ELIMINATION, THINNING, and COMPLETION belong to `node_update_rules`. The algorithm is illustrated in Figure 1. In `node_update_rules`, rules are applied in the following order; MERGE, LINK EVALUATION, ELIMINATION, THINNING, and COMPLETION. This is because the parser has to establish the semantic representation with ELIMINATION and delete the requirement with THINNING at the current pointed node before the pointer goes up to the mother node (with COMPLETION). `node_update_rules` can vacuously apply, so the tree structure is passed to the next rule without any modification if some of `node_update_rules` cannot apply. As Figure

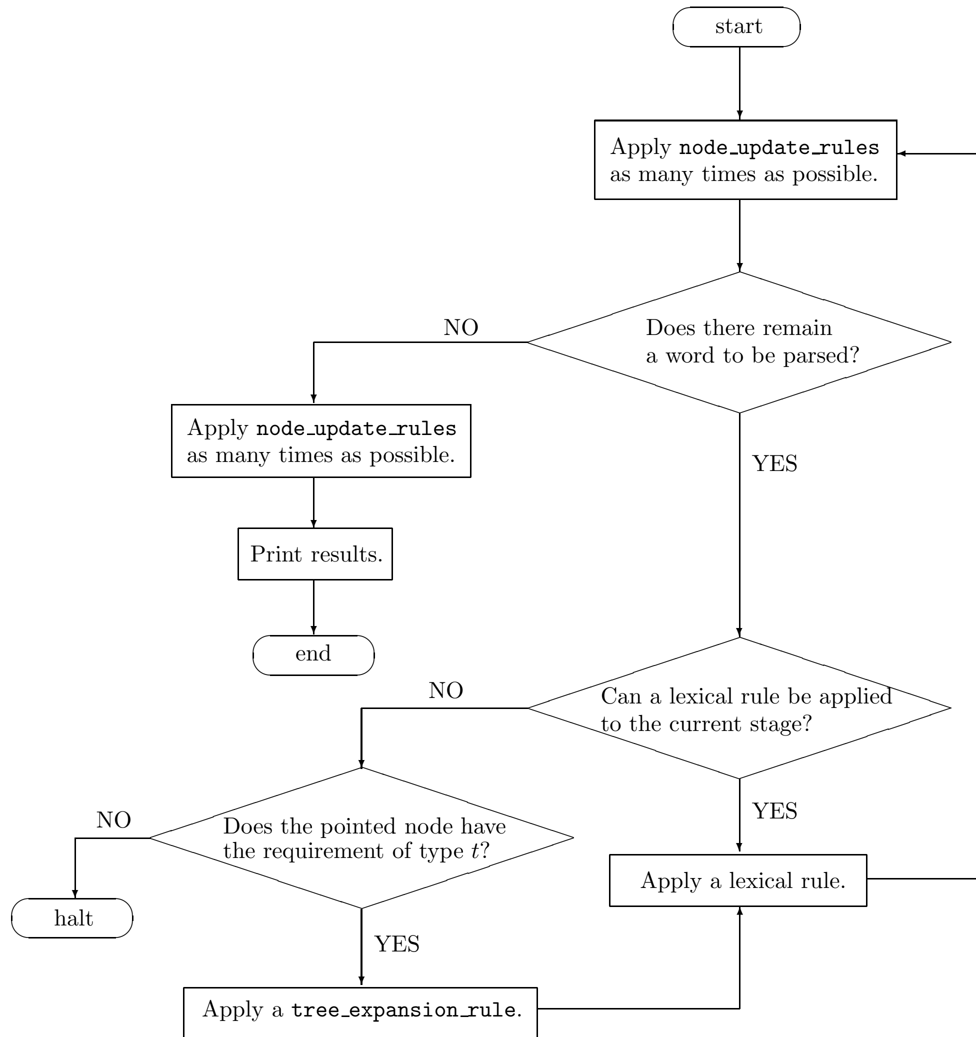

Figure 1: Flow Chart of Algorithm

1 shows, `tree_expansion_rules` apply if a lexical rule cannot apply to the pointed node; `tree_expansion_rules` apply if the parser cannot consume the current word because in DS each lexical item is a lexical rule. This algorithm implements the lexically-driven processing of sentences in Japanese and reflects the prominent difference between English and Japanese.

4. Incremental Processing and Null Arguments

This section shows how the parser works on the basis of the DS framework with special reference to the processing of null arguments. The parser is written in SWI-Prolog on Linux OS. It consists of about 1,200 lines, 41.1 KB except the lexicon. The parser accepts a sentence composed of N words returning N+1 steps, from the initial step to the final one with the semantic representation.

The parser proposed in this paper is still small. Nonetheless, it can parse not only simple sentences but also relative clause constructions, embedded clauses, and scrambled sentences. Let us take a look at how the parser works in processing a simple scrambled sentence (2), repeated here as (4).

(4) Bōru o John ga nageta.
ball ACC John NOM throw-PAST

"John threw the ball."

Figure 2 is a successfully parsed output of example (4). As the figure shows, the sentence consists of five words so that the parser returns the six steps with the semantic representation on the last line. In each step, rules which are applied at each stage are listed in the second line. The tree structure is represented in a tab delimited format.

Incremental processing is obviously crucial to word-order phenomena, but it is also important for the parser to specify (or underspecify) null arguments and pronouns in an incremental way. Let us illustrate how the parser works in processing null arguments in embedded clauses and relative clauses. As seen in Kuroda (1965), Hoji (1998) and other studies, many studies in the literature discuss empty categories and null arguments, especially within the framework of generative grammar. As shown by Takahashi (2006, p.1), unlike English, the subject and object NPs can be empty even in the embedded clauses.

(5) John ga Naomi ni e kyōju ni e shōkaisuru to itta.
John NOM Naomi DAT professor DAT introduce COMP say-PAST

"John told Naomi that he (or someone) would introduce her (or someone) to the professor."

In (5) the subject and object NPs of the embedded clause are omitted. Missing items can refer to someone in the context, but the salient reading is that the omitted subject refers to the subject of the main clause, $John$, and the missing object refers to the object $Naomi$. In this example, the positions in the fixed tree structure of NPs and null arguments cannot be specified until the verb $introduce$ is consumed. Let us show the parsing result of (5) in Figure 3. The figure shows that at step 1 the sentence initial NP $John$ is located at an unfixed node which is introduced by LOCAL ∗ADJUNCTION. Thus this initial NP is tentatively unfixed but will find a fixed position in the main clause (other possibilities cannot survive by the end of parsing). Step 10 is the final stage of the parsing with the semantic representation at the bottom of it. The normal binary tree structure of step 10 is illustrated in Figure 4. As seen in the representation in Figure 3 the parser produced the semantic representation `fo(say(john, introduce(meta_v, meta_v, professor), naomi))` where `meta_v` is a meta variable, which is merged with someone in the context. Although in the framework of generative grammar it has been argued whether ellipsis or empty pronouns are involved in null arguments, DS can process them without setting up a stipulation.

```
?- parse([boru, o, john, ga, nageta], X).

Step 0
Nothing applied.
Pointer: pn(fixed, [root])
Root: [tn([0]), an([?ty(t)]), []]
Gen_adj: []
Linked: link([[], [], []])

Step 1
local_adj, boru applied.
Pointer: pn(fixed, [root, local])
Root: [tn([0]), an([?ty(t)]), [loc([fo(ball), ty(e), ?ty(e)])]]
Gen_adj: []
Linked: link([[], [], []])

Step 2
thinning, o applied.
Pointer: pn(fixed, [root])
Root: [tn([0]), an([\/[1, ty((e->t))], ?ty(t)]), []]
          [tn([0, 1]), an([\/[0, ty(e)], \/[0, fo(ball)], ty((e->t))]), []]
                   [tn([0, 1, 0]), an([fo(ball), ty(e)]), []]
Gen_adj: []
Linked: link([[], [], []])

Step 3
local_adj, john applied.
Pointer: pn(fixed, [root, local])
Root: [tn([0]), an([?ty(t), \/[1, ty((e->t))]]), [loc([fo(john), ty(e), ?ty(e)])]]
          [tn([0, 1]), an([\/[0, ty(e)], \/[0, fo(ball)], ty((e->t))]), []]
                   [tn([0, 1, 0]), an([fo(ball), ty(e)]), []]
Gen_adj: []
Linked: link([[], [], []])

Step 4
thinning, ga applied.
Pointer: pn(fixed, [root])
Root: [tn([0]), an([\/[0, fo(john)], \/[0, ty(e)], ?ty(t), \/[1, ty((e->t))]]), []]
          [tn([0, 0]), an([fo(john), ty(e)]), []]
          [tn([0, 1]), an([\/[0, ty(e)], \/[0, fo(ball)], ty((e->t))]), []]
                   [tn([0, 1, 0]), an([fo(ball), ty(e)]), []]
Gen_adj: []
Linked: link([[], [], []])

Step 5
nageta applied.
Pointer: pn(fixed, [root, 1, 1])
Root: [tn([0]), an([?ty(t), \/[0, fo(john)], \/[0, ty(e)], \/[1, ty((e->t))]]), []]
          [tn([0, 0]), an([fo(john), ty(e)]), []]
          [tn([0, 1]), an([\/[0, ty(e)], \/[0, fo(ball)], ty((e->t))]), []]
                   [tn([0, 1, 0]), an([fo(ball), ty(e)]), []]
                   [tn([0, 1, 1]), an([fo(lambda(ball, lambda(john, throw(john, ball)))), ty((e->e->t))]), []]
Gen_adj: []
Linked: link([[], [], []])

Step 6
completion, elimination, completion, elimination, thinning applied.
Pointer: pn(fixed, [root])
Root: [tn([0]), an([ty(t), fo(throw(john, ball)), \/[1, fo(lambda(john, throw(john, ball)))], \/[0, fo(john)],
      \/[0, ty(e)], \/[1, ty((e->t))]]), []]
          [tn([0, 0]), an([fo(john), ty(e)]), []]
          [tn([0, 1]), an([fo(lambda(john, throw(john, ball))), \/[1, fo(lambda(ball, lambda(john, throw(john,
          ball))))], \/[1, ty((e->e->t))], \/[0, ty(e)], \/[0, fo(ball)], ty((e->t))]), []]
                   [tn([0, 1, 0]), an([fo(ball), ty(e)]), []]
                   [tn([0, 1, 1]), an([fo(lambda(ball, lambda(john, throw(john, ball)))), ty((e->e->t))]), []]
Gen_adj: []
Linked: link([[], [], []])

Semantic Representation: fo(throw(john, ball))
```

Figure 2: Parsing Output of (4)

```
?- parse([john, ga, naomi, ni, kyoujyu, ni, shoukaisuru, to, itta], X).

Step 0
Nothing applied.
Pointer: pn(fixed, [root])
Root: [tn([0]), an([?ty(t)]), []]
Gen_adj: []
Linked: link([[], [], []])

Step 1
local_adj, john applied.
Pointer: pn(fixed, [root, local])
Root: [tn([0]), an([?ty(t)]), [loc([fo(john), ty(e), ?ty(e)])]]
Gen_adj: []
Linked: link([[], [], []])

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


Step 10
completion, elimination, completion, elimination, completion, elimination, thinning applied.
Pointer: pn(fixed, [root])
Root: [tn([0]), an([ty(t), fo(say(john, introduce(meta_v, meta_v, professor), naomi)), \/[1, fo(lambda(john,
      say(john, introduce(meta_v, meta_v, professor), naomi)))], \/[1, ty((e->t))], \/[0, fo(john)],
      \/[0, ty(e)]]), []]
          [tn([0, 0]), an([fo(john), ty(e)]), []]
          [tn([0, 1]), an([fo(lambda(john, say(john, introduce(meta_v, meta_v, professor), naomi))),
          \/[1, fo(lambda(introduce(meta_v, meta_v, professor), lambda(john, say(john, introduce(meta_v,
          meta_v, professor), naomi))))], \/[1, ty((t->e->t))], \/[0, ty(t)], \/[0, fo(introduce(meta_v,
          meta_v, professor))], ty((e->t))]), []]
                    [tn([0, 1, 0]), an([ty(t), fo(introduce(meta_v, meta_v, professor)),
                    \/[1, fo(lambda(meta_v, introduce(meta_v, meta_v, professor)))], \/[0, ty(e)],
                    \/[0, fo(meta_v)], \/[1, ty((e->t))]]), []]
                              [tn([0, 1, 0, 0]), an([fo(meta_v), ?ty(e)]), []]
                              [tn([0, 1, 0, 1]), an([fo(lambda(meta_v, introduce(meta_v, meta_v, professor))),
                              \/[1, fo(lambda(meta_v, lambda(meta_v, introduce(meta_v, meta_v, professor))))],
                              \/[1, ty((e->e->t))], \/[0, ty(e)], \/[0, fo(meta_v)], ty((e->t))]), []]
                                        [tn([0, 1, 0, 1, 0]), an([fo(meta_v), ?ty(e)]), []]
                                        [tn([0, 1, 0, 1, 1]), an([fo(lambda(meta_v, lambda(meta_v,
                                        introduce(meta_v, meta_v, professor)))), \/[1, fo(lambda(professor,
                                        lambda(meta_v, lambda(meta_v, introduce(meta_v, meta_v,
                                        professor)))))], \/[1, ty((e->e->e->t))], ty((e->e->t)), \/[0, ty(e)],
                                        \/[0, fo(professor)]]), []]
                                                  [tn([0, 1, 0, 1, 1, 0]), an([fo(professor), ty(e)]), []]
                                                  [tn([0, 1, 0, 1, 1, 1]), an([fo(lambda(professor,
                                                  lambda(meta_v, lambda(meta_v, introduce(meta_v, meta_v,
                                                  professor))))), ty((e->e->e->t))]), []]
                    [tn([0, 1, 1]), an([fo(lambda(introduce(meta_v, meta_v, professor), lambda(john, say(john,
                    introduce(meta_v, meta_v, professor), naomi)))), \/[1, fo(lambda(naomi,
                    lambda(introduce(meta_v, meta_v, professor), lambda(john, say(john,
                    introduce(meta_v, meta_v, professor), naomi)))))], \/[1, ty((e->t->e->t))], ty((t->e->t)),
                    \/[0, ty(e)], \/[0, fo(naomi)]]), []]
                              [tn([0, 1, 1, 0]), an([fo(naomi), ty(e)]), []]
                              [tn([0, 1, 1, 1]), an([fo(lambda(naomi, lambda(introduce(meta_v, meta_v,
                              professor), lambda(john, say(john, introduce(meta_v, meta_v, professor),
                              naomi))))), ty((e->t->e->t))]), []]
Gen_adj: []
Linked: link([[], [], []])

Semantic Representation: fo(say(john, introduce(meta_v, meta_v, professor), naomi))
```

Figure 3: Abbreviated Parsing Output of (5)

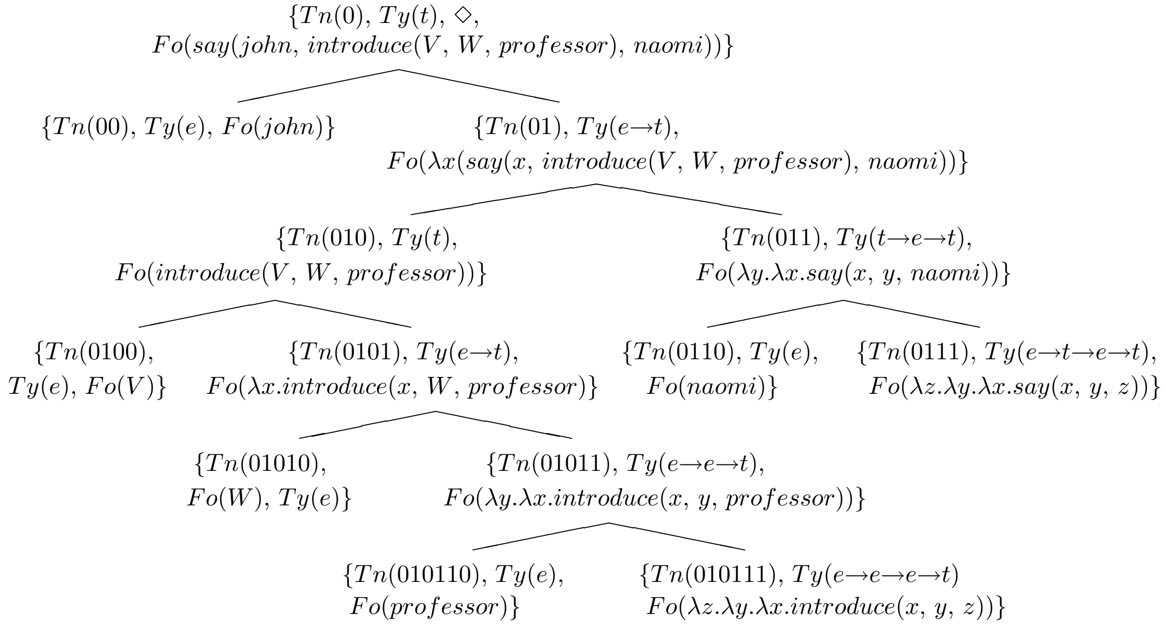

Figure 4: Abbreviated Parsing Output of (5)

5. Discussion

In this paper we have accounted for the algorithm of the application of lexical and transition rules and the implementation of the DS parser for Japanese. This section provides a discussion regarding the current DS grammar, one on the implementation and future research. One of DS's goals is to explore language faculty, and our implementation can contribute to the study of this issue by the simulation of human sentence processing. Although this DS Prolog grammar can parse a simple sentence, relative clause constructions, and embedded structure, currently the coverage of this grammar is very small. As future research we would like to integrate a stochastic approach into the system. Another issue to be tackled is how we should deal with pragmatically-related problems such as pronoun resolution. For instance, the parser provides place-holders such as a `meta_v`, but as far as I know there is no research given in the literature about how the parser merges entities in the context with these variables on the basis of the DS framework. This paper deals with only Japanese and we need to make sure whether this algorithm can be applied to other head-final languages such as Korean.

6. Conclusion

This paper presented the basic implementation of the parser based on the DS framework. The lexical rules and transition rules are applied arbitrarily within the current framework, therefore we provided the algorithm of the application of lexical rules and transition rules for head-final languages like Japanese. This parser adopts the partitioned parsing state, which allows the parser to access the pointed node efficiently. The algorithm is implemented in Prolog. This paper also showed that the parser can process empty pronouns in embedded clauses often observed in Japanese.

References

Cann, R., R. Kempson, and L. Marten. 2005. *The Dynamics of Language*. Elsevier, Oxford.

Hoji, H. 1998. Null object and sloppy identity in japanese. *Linguistic Inquiry*, 29:127–152.

Kempson, R., W. Meyer-Viol, and D. Gabbay. 2001. *Dynamic Syntax: The Flow of Language Understanding*. Blackwell Publishers, Oxford.

Kobayashi, M. 2007. Incremental processing and design of a parser for japanese: A dynamic approach. In *Proceedings of the Fourth International Workshop on Logic and Engineering of Natural Language Semantics (LENLS2007)*, pages 261–273.

Kuroda, S.-Y. 1965. *Generative Grammatical Studies in the Japanese Language*. Ph.D. thesis, MIT.

Moot, R. 1999. Grail: An automated proof assistant for categorial grammars. In R. Delmonte, editor, *Proceedings of Venezia per il Trattamento Automatico delle Lingue '99 (VEXTAL99)*, pages 225–261.

Otsuka, M. and M. Purver. 2003. Incremental generation by incremental parsing. In *Proceedings of the 6th Annual CLUK Research Colloquium*, pages 93–100.

Purver, M. and M. Otsuka. 2003. Incremental generation by incremental parsing: Tactical generation in dynamic syntax. In *Proceedings of the 9th European Workshop on Natural Language Generation*, pages 79–86.

Takahashi, D. 2006. Apparent parasitic gaps and null arguments in japanese. *Journal of East Asian Linguistics*, 15:1–35.