Incremental Processing and Design of a Parser for Japanese: A Dynamic Approach

Masahiro Kobayashi

University Education Center, Tottori University,
4-101 Koyama-chō Minami, Tottori, 680-8550 Japan
kobayashi@uec.tottori-u.ac.jp

Abstract. This paper illustrates a parser which processes Japanese sentences in an incremantal fashion based on the Dynamic Syntax framework. In Dynamic Syntax there has basically been no algorithm which optimizes the application of transition rules: as it is, the rules can apply to a current parsing state in an arbitrary way. This paper proposes both partitioned parsing states allowing easier access to some kind of unfixed nodes and an algorithm to apply transition rules for Japanese. The parser proposed in this paper is implemented in Prolog. The parser is able to process not only simple sentences but also relative clause constructions, scrambled sentences and complex (embedded) sentences.

1 Introduction

Incremental and dynamic approaches to sentence comprehension and grammar formalisms have attracted a great deal of attention both in psycholinguistics and natural language processing. For example, Lombardo, Mazzei and Sturt (2004) discuss the relationship between competence and performance in incremental parsing based on Tree Adjoing Grammar. This paper is meant to describe a parser and its algorithm for a fragment of Japanese based on the incremental, left-to-right parsing formalism in the Dynamic Syntax framework (Kempson, Meyer-Viol and Gabbay 2001, Cann, Kempson and Marten 2005) and show how the parser processes some of the constructions in Japanese. In advocating Dynamic Syntax, Kempson et al. (2001) and Cann et al. (2005) have shown that the formalism is able to cope with many constructions regarding word order phenomena, such as cross-over constructions and topicalizations in English and also that it has a broad typological perspective. However there is no algorithm specified for computational implementation. In other words, Dynamic Syntax is an abstract grammar formalism with lexical rules and transition rules, and no algorithm has been proposed which optimizes the application of transition rules. In addition, the formalism makes the parser keep NPs unfixed within the tree structures at a certain parsing stage and narrow down the possibilities of outputs by specifying their positions. This means that the parser needs to deal with not only usual binary tree structures but also unfixed tree structures whose positions must be fixed in the final tree structure.

K. Satoh et al. (Eds.): JSAI 2007, LNAI 4914, pp. 174–186, 2008.

This paper has two aims. The first is to propose partitioned parsing states with easier access to fixed and unfixed nodes. The second aim is to propose an algorithm to implement the application of lexical rules and transition rules for Japanese. Purver and Otsuka (2003) and Otsuka and Purver (2003) proposed a parsing and generation model for English based on the Dynamic Syntax formalism, but as far as I know, there is no research so far which attempts to implement Dynamic Syntax for Japanese. The parser can currently process not only simple sentences but also relative clause constructions and embedded clauses. A Prolog implementation of the parser and efficient parsing algorithm will also be presented including a comparison to Purver and Otsuka (2003) and Otsuka and Purver (2003).

The outline of this paper is as follows: the subsequent section will be devoted to a brief look at the Dynamic Syntax formalism and issues this paper will deal with. This section will also describe previous studies. Section 3 will illustrate the parser for Japanese on the basis of Dynamic Syntax and propose an algorithm as well as show how the parser processes sentences. Section 4 discusses the effectiveness of this approach and further issues. Section 5 concludes the discussion.

2 Dynamic Syntax and Problems of Parsing

2.1 Dynamic Syntax and Formalism

Before getting into the discussion of the issues I address, a brief illustration of the formalism of Dynamic Syntax (hereafter DS) is needed. The DS formalism allows the parser to process a sentence in a left-to-right fashion: it consumes the words from the onset, building up semantic representation as the scan of the string proceeds. The grammar formalism is called goal-directed in the sense that in each node there exists *requirement*(*s*) prefixed by "?" and every requirement needs to be satisfied or canceled until the parsing has been finished. The initial state of parsing consists of the single node $\{Tn(a), ?Ty(t), \Diamond\}$ (quoted from Kempson et al. 2001: p.57), where Tn is the tree node identifier and $?Ty(t)$ means that this node will be associated with a type t formula; the parser is going to parse the sentence. The node includes the pointer $\Diamond$ indicating that the pointed node is highlighted or active so that lexical rules and transition rules are applied to this node. Summing up, processing is defined as (1). A parsing state, which is a (partial) tree structure, shifts to the next one and grows larger and larger from the initial state $\mathcal{T}_0$ to the final state $\mathcal{T}_n$ through the application of lexical and transition rules.

(1) $\mathcal{T}_0 \rightarrow$ rule application $\rightarrow \mathcal{T}_1 \rightarrow \ldots \ldots \rightarrow$ rule application $\rightarrow \mathcal{T}_n$

In the tree structure, nodes are represented as sets, and the relation between nodes are represented by the tree node predicate Tn and the node modalities. The lexical items themselves are defined as rules and take the form of **IF** *Action*$_1$ **THEN** *Action*$_2$ **ELSE** *Action*$_3$ which updates the current partial

tree structure. In Cann et al. (2005) the main transition rules are as follows: Local ∗Adjunction, Generalised Adjunction, ∗Adjunction, Introduction, Prediction, Elimination, Anticipation, Completion, Thinning, Link Adjunction, Link Evaluation and Merge.

The other characteristic worth noting is *syntactic* underspecification. When the parser processes the string in a left-to-right fashion, especially in head-final languages like Japanese, the parser cannot specify whether the NP at the beginning of a sentence will be fixed to a main or subordinate or relative clause. For example, the initial NP *hon-o* "book-ACC" of (2a) is the object NP of a main clause but in (2b) it is the object of a relative clause in a subordinate clause.

(2) a. Hon o gakusei ga katta.
book ACC student NOM buy-PAST
"The student bought the book."

b. Hon o katta gakusei ga koronda to Taro ga itta.
book ACC buy-PAST student NOM fall down-PAST COMP Taro NOM say-PAST
"Taro said that the student who bought the book fell down."

DS has a mechanism which enables the parser to keep NPs unfixed in the partial tree structure. This paper adopts three transition rules; Local ∗Adjunction, Generalised Adjunction and ∗Adjunction (Cann et al. 2005). Local ∗Adjunction provides an unfixed position for local scrambling, and Generalised Adjunction is used to introduce a subordinate clause because it provides an unfixed node associated with a type t formula and the unfixed node dominates all the words which belong to the subordinate clause. ∗Adjunction rule provides a tree structure with an unfixed node for long distance scrambling. The snapshot of the partial tree consuming *hon* of (2a) is shown in Figure 1 and that of (2b) in Figure 2. The unfixed node introduced by Local ∗Adjunction is connected to the mother node by the dashed line, while the dotted line means that the unfixed node is provided by the Generalised Adjunction rule in each figure. Such unfixed nodes are required to find their fixed positions by the end of the processing with tree-update actions. Readers who wishes to see a more detailed description of the DS formalism, are referred to Kempson et al. (2001) and Cann et al. (2005).

2.2 Problems of Parsing in DS

This subsection will outline the issues addressed in this paper. This paper mainly deals with two problems. The first is the issue of how we should express (partial) tree structures in implementing the DS grammar. To put it another way, the problem is how we should efficiently implement the grammar of head-final languages such as Japanese within the DS framework in Prolog. In Prolog notation, a complete binary tree structure is often represented as `node(s, node(np, [], []), node(vp, [], []))`, which is the equivalent of the usual notation of the tree structure in which S goes to NP and VP. When unfixed nodes,

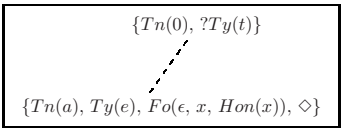

Fig. 1. The parsing tree of *hon* "book" in (2a)

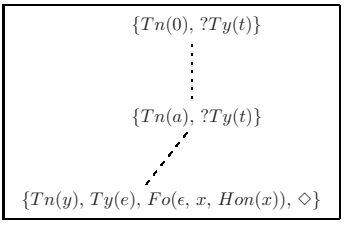

Fig. 2. The parsing tree of *hon* "book" in (2b)

which are introduced by Local ∗adjunction, Generalised adjunction and ∗Adjunction, come in this notation, then it would not be easy to search a pointed node and merge the unfixed nodes with other fixed positions. Particularly in a head-final language like Japanese, the number of nodes dominated by the unfixed node can be far larger than in English because, for example, the type t node of a sentence sometimes needs to be unfixed until the complementizer *to* has been consumed. If the parser represents the tree structures as sets of nodes, as in DS originally, it would not be easy to parse the string efficiently. This problem has a close relation to how we should represent the processing state.

The second issue this paper addresses is an algorithm for the application of the transition rules and lexical rules. DS is a grammar formalism that allows a parsing state to be updated to a subsequent state, and currently in the DS framework transition rules can apply to any state in an arbitrary way; there is no algorithm which specifies how and when lexical and transition rules are applied. As illustrated in Figures 1 and 2, application of different transition rules leads to different partial tree structures. For example in Figure 2, Local ∗adjunction is applied after applying Genralised adjunction, while only Local ∗adjunction is applied in Figure 1. In implementing the grammar, we need an algorithm to derive such partial structures as these two. This paper proposes an algorithm of rule applications for the implementation of the DS parser and provides some improvements for efficiency.

2.3 Previous Studies

Although the generation and parsing model of Purver and Otsuka (2003) and Otsuka and Purver (2003) based on DS formalism can be found in the literature, they mainly deal with English. Purver and Kempson (2004) and Purver, Cann and Kempson (2006) go a step further and propose a context-dependent parsing and generation model to account for the transition from a hearer to a speaker and vice versa with reference to shared utterances and some context-dependent phenomena such as VP ellipsis.

A brief description of their model and implementation is as follows. In their implementation in Prolog, the tree structure is represented as a set of nodes. Figure 3 is their Prolog notation of a partial tree structure, while its DS conterpart is illustrated as Figure 4 (both are cited from Otsuka and Purver 2003: p.98).

```
tree( [node('0', [?ty(t), ?([\/0],ty(e)), ?([\/1],ty(e>t))] ),
       node('00', [ty(e), fo(john), +male] ),
       node('01', [ty(e>t), fo(X^snore(X)), +pres] )] ).
```

Fig. 3. Prolog notation of Otsuka and Purver (2003: p.98)

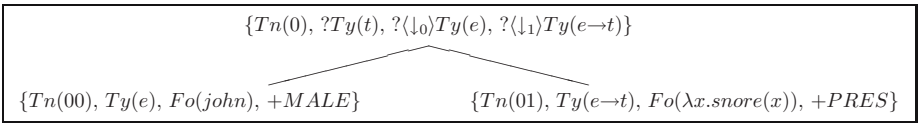

Fig. 4. DS notation of Figure 3 (Otsuka and Purver 2003: p.98)

Although the tree structure is represented as a set of nodes, as Figure 3 shows, the relation between the mother node and its daughter node is represented as the tree node identifier (Tn): the root node has the identifier $Tn(0)$ and the argument daughter has the identifier $Tn(00)$, and the functor $Tn(01)$.

There are two problems I can point out about their model, ones related to the two issues illustrated in subsection 2.2. The first is about Prolog tree notation and a formal definition of the parsing state. According to Purver and Kempson (2004: p.77) a parsing state of their parser is defined as a set of triples $\langle T, W, A\rangle$, where T is a tree, W, words which have already been consumed and A, rules which have already been applied. As described above, any node (and any partial tree) is defined as an element of set T. Therefore, once we attempt to search a certain node, e.g., the pointed node, among the partial tree, it turns out to be inefficient especially as the tree gets bigger and bigger. Moreover, as described in the previous subsection, in Japanese the parser needs to cope with many unfixed nodes other than fixed nodes; this makes the treatment of a tree as a set of nodes inefficient for processing head-final languages. In the subsequent section of this paper a slightly different, structured parsing state approach to DS tree structure will be proposed which enables easier access to the nodes.

The second problem is related to the algorithm and the application of transition rules. Otsuka and Purver (2003) and Purver and Otsuka (2003) try to improve the efficiency of parsing by assuming `always_rules` which apply forcibly and `possible_rules` which don't necessarily apply. In the subsequent section of this paper I propose a different transition rule application system for Japanese and an algorithm to process fragments of Japanese sentences.

3 Incremental Parsing for Japanese

3.1 Definition of the Parser

This subsection describes development of the parser proposed in this paper, an issue closely related to the representation of the tree structures we have already

discussed. Unlike the model of Purver and Kempson (2004), the model proposed here has the following parsing state structure: it is a set of triples $\langle W, S, P\rangle$ where W is a string to be consumed and P a node address of the pointed node at a current stage including the feature `fixed, local, gen` and `link`; `fixed` means that the pointer is in the fixed tree structure, `local` that the pointer is in a node introduced by Local ∗Adjunction, `gen` that the pointer is in a node introduced by Generalised Adjunction, and `link` that the pointer is in the linked structure. S is a multitier parsing state which consists of a set of doubles $\langle R, T\rangle$ where R is a set of transition and lexical rules which have been used to establish the current tree. T consists of a triple $\langle F, G, L\rangle$ where F is a fixed tree structure, G a tree structure whose root node is introduced by Generalised Adjunction, and L a linked tree structure. F has the binary tree structure as shown in section 2, such as `node(Mother, node(Argument, [], []), node(Functor, [], []))`, but each node, e.g., `Mother`, has a single place for a node introduced by Local ∗Adjunction, while Otsuka and Purver (2003) treat trees as a set of nodes. The pictorial image of the parsing state is illustrated in Figure 5. The initial parsing state is $\langle W_0, S_0, \texttt{pn(fixed, [root])}\rangle$; S_0 consists of a double $\langle \phi, A\rangle$ where A is the initial parsing state, which contains only the requirement $?Ty(t)$ and the pointer $\Diamond$. The final parsing state is $\langle \phi, S_n, \texttt{pn(fixed, [root])}\rangle$. In S_n the generalised tree structure and the link structure are empty, and the semantic representation is established at the root node of the fixed tree structure. The pointer needs to go back to the root node of the fixed tree structure for the parsing to be successful as indicated by `pn(fixed, [root])`.

The separate treatment of fixed, generalised, and link structure and the pointed node address indicator P enable us to search the pointed node and manipulate tree-update actions efficiently: this approach helps the parser process SVO languages like Japanese, particularly when a considerable amount of nodes is dominated by the unfixed mother node.

3.2 Transition Rules and Algorithm

This subsection deals with the second issue I addressed: the application of the transition rules and its algorithm. The parser currently has the following transition rules: Local ∗Adjunction, Generalised Adjunction, ∗Adjunction, Introduction, Prediction, Elimination, Anticipation, Completion,

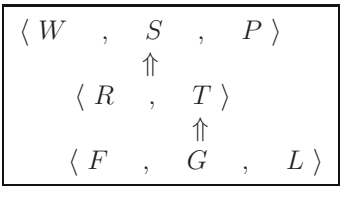

Fig. 5. Structure of parsing state

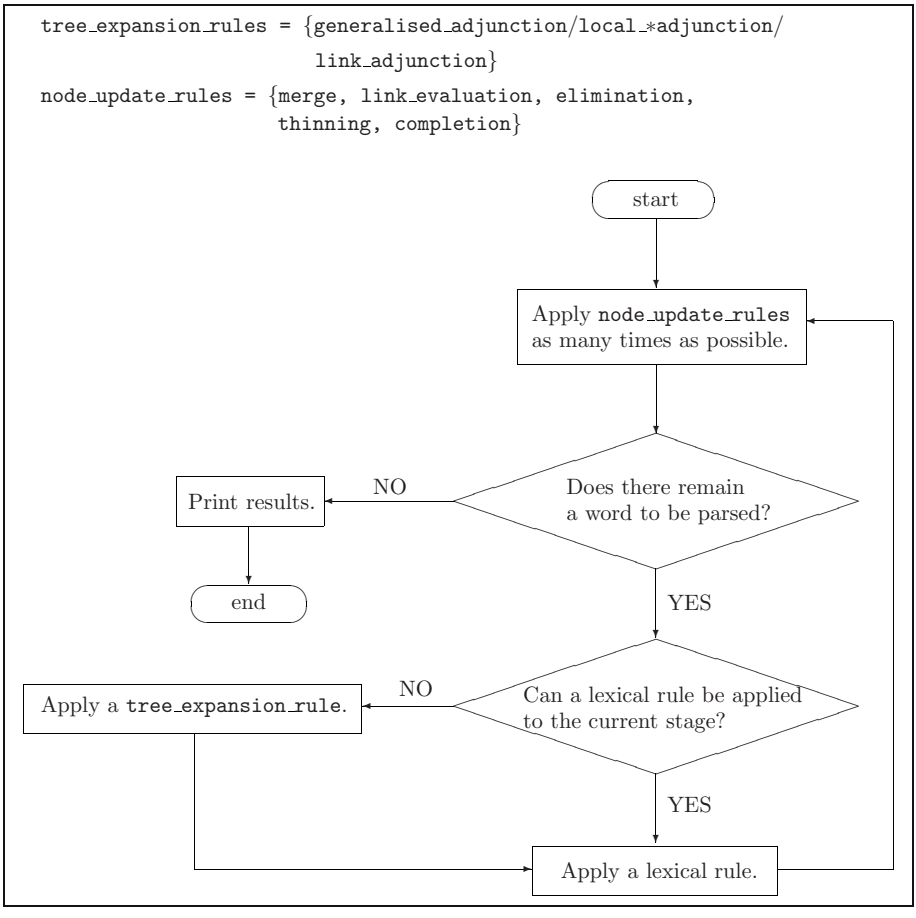

Fig. 6. Algorithm of processing

Thinning, Link Adjunction, Link Evaluation, and Merge. This paper assumes that among those rules Introduction, Prediction, and Anticipation are not used for processing Japanese sentences.

As mentioned earlier, there is no algorithm which specifies when and how the transition rules apply: they can be applied to any state in an arbitrary way in the current DS formalism. This subsection presents an algorithm, and the subsequent subsection shows that the algorithm is sufficient to parse simple sentences, relative clause constructions, and embedded clauses. Unlike Otsuka and Purver (2003) and Purver and Otsuka (2003), the approach proposed here divides the transition rules into two groups: one is those rules which expand partial tree structure, `tree_expansion_rules`, and the other is those which do not expand the tree but update the node information, `node_update_rules`. The former group, `tree_expansion_rules`, includes Local ∗Adjunction, Generalised

```
ds_parse([], [[R, L1]|Result], [[R2, L2], [R, L1]|Result]) :-
     apply_tree_expansion_rules([[[]], L1], [], [R2, L2]),
     satisfy(L2), %Checks the tree has no requirement
     pretty_print([[R2, L2], [R, L1]|Result]), nl.

ds_parse([H|T], [[X, T1]|T2], Result) :-
     apply_node_update_rules([[[]], T1], [], [R, Mid]),
     (lexical_rule(H, [R, Mid], [R1, Mid3]);
      (tree_expansion_rule([R, Mid], [R2, Mid2]),
       lexical_rule(H, [R2, Mid2], [R1, Mid3])
      )
     ),
     ds_parse(T, [[R1, Mid3], [X, T1]|T2], Result).
```

Fig. 7. Prolog code of the algorithm

Adjunction, ∗Adjunction, and Link Adjunction. Other transition rules, Merge, Link Evaluation, Elimination, Thinning, and Completion belong to `node_update_rules`. The shared characteristic among the former `tree_expansion_rules` group is that they require that the pointed node have the requirement of type t.

The algorithm proposed here is diagramed as Figure 6, and the algorithm implemented in the Prolog code is illustrated in Figure 7. The upper four lines of Figure 7 are the base rule, and the bottom eight lines are the recursive rule. The first argument of the `ds_parse` predicate is the string to be consumed: when there remains no word to be consumed, the processing ends. The second argument is the list consisting of the pair of rules applied in that stage and the tree structure. The algorithm defined in Figure 6 considerably improves the efficiency of parsing because the `tree_expansion_rules` are not applied if the pointed node does not have the requirement of type t. The important characteristic worth noting is the application order of the `node_update_rules`. The parser tries to apply rules in the following order; Merge, Link_Evaluation, Elimination, Thinning, and Completion, though there might be the possibility that some of them cannot apply. When one of the rules, e.g., Merge, cannot apply, the tree structure is passed to the next rule, in this case Link_Evaluation, without any modification to the tree structure. This ensures that there is always an output tree structure at the end of each rule application in `node_update_rules`. This application order especially works in the situation where the parser brings the type and semantic information up to the mother node with the β-reduction after consuming the verb: Elimination generates the type and formula of the node and Thinning deletes the requirement, and Completion brings the information up to the mother node.

3.3 How the Parser Works

This subsection illustrates how the parser works. The parser is written in SWI-Prolog on Linux. The current grammar is small but the source code is composed

```
?- parse([john, ga, hon, o, katta], 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 5
katta applied.
Pointer: pn(fixed, [root, 1, 1])
Root: [tn([0]), an([?ty(t), \/[1, ty((e->t))], \/[0, fo(john)], \/[0, ty(e)]]), []]
          [tn([0, 0]), an([fo(john), ty(e)]), []]
          [tn([0, 1]), an([\/[0, ty(e)], \/[0, fo(book)], ty((e->t))]), []]
                    [tn([0, 1, 0]), an([fo(book), ty(e)]), []]
                    [tn([0, 1, 1]), an([fo(lambda(book, lambda(john, buy(john, book)))),
                     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(buy(john, book)), \/[1, fo(lambda(john, buy(john, book)))],
       \/[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, buy(john, book))),
           \/[1, fo(lambda(book, lambda(john, buy(john, book))))],
           \/[1, ty((e->e->t))], \/[0, ty(e)], \/[0, fo(book)], ty((e->t))]), []]
                    [tn([0, 1, 0]), an([fo(book), ty(e)]), []]
                    [tn([0, 1, 1]), an([fo(lambda(book, lambda(john, buy(john, book)))),
                     ty((e->e->t))]), []]
Gen_adj: []
Linked: link([[], [], []])

Semantic Representation: fo(buy(john, book))
```

Fig. 8. Snapshot of the output

of about 1,200 lines except the lexicon, and the total size is 41.1 KB. The parser can process simple sentences such as (3a, b) as well as also the relative clause constructions (3c) and complex sentences (3d).

(3) a. John ga hon o katta.
John NOM book ACC buy-PAST
"John bought the book."

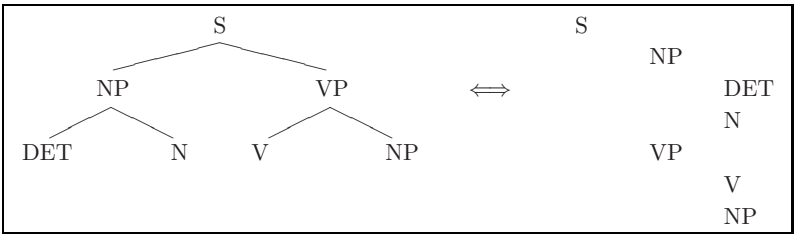

Fig. 9. Tree notation and command line notation

b. Hon o John ga yonda.
book sc acc John NOM read-PAST
"John read the book."
c. Hon o katta gakusei ga hashitta.
book ACC buy-PAST student NOM run-PAST
"The student who bought the book ran."
d. John ga hon o katta to Tom ga itta.
John NOM book ACC buy-PAST COMP Tom NOM say-PAST
"Tom said that John bought the book."

The parser returns all steps of the processing, N+1 steps in total, from step 0 (the initial state) to step N+1, for the N words input. Figure 8 is a simplified snapshot of the processing of the example sentence (3a). The parsing result is printed from step 0 (the initial state) to step 6 (final state), although steps 2, 3 and 4 are abbreviated. The semantic formula is printed at the bottom line as `fo(buy(john, book))`, which means *John bought the book* (ignoring tense). Some other notational conventions are given in Figure 10. As you can see in the figure the tree structure grows from the onset to the final stage monotonically. In each step, the tree structure is represented as lines whose dominance relationships are shown with the tab spacing. Figure 9 shows us the corresponding, standard DS representation.

The parser can return more than one semantic representation for ambiguous sentences. As Kempson et al. (2001: p.70) discusses, when an appropriate context is given, the initial NP in (4) can be the subject of the main clause or that of the subordinate clause or both of them.

(4) John ga hon o katta to itta.
John NOM book ACC buy-PAST COMP say- PAST
1: "John said that (Tom or Mary or someone else) bought the book."
2: "(Tom or Mary or someone else) said that John bought the book."
3: "John$_i$ said that he$_i$ bought the book."

The parser returns two semantic representations by Prolog's backtracking after processing (4); one is `fo(say(john, buy(meta_v, book)))`, and the other `fo(say(meta_v, buy(john, book)))`, where `meta_v` is a meta-variable which will be substituted for *John* or other entities in the context by pragmatic actions.

DS notation	Parser notation
$\langle\downarrow_0\rangle Ty(e)$	\/[0, ty(e)]
$\langle\uparrow_1\rangle Ty(t)$	/\[1, ty(t)]
$\langle\downarrow_0\rangle Fo(John)$	\/[0, fo(john)]
$Tn(0110)$	tn([0, 1, 1, 0])

Fig. 10. Notational convention of features in DS and parser

Let me explain briefly how the algorithm in Figure 6 processes the sentence using one of the possible readings in (4), "John said that (Tom or Mary or someone else) bought the book." As Figure 6 shows, the first step the parser attempts to take is to apply `node_update_rules` as many times as possible. This step results in a vacuos application; in this case, it returns tree structures without any modification compared to the previous stage, since none of the `node_update_rules` are applied. Then in the next step the lexical specification of the NP *John* fails to apply to the current tree structure where the pointer is situated in the type t node, since *John* requires that the type e node be the pointed node. As the next step `tree_expansion_rules`, in this case Generalised Adjunction or Local $*$Adjunction, apply because the pointed node is of type t (when Generalised Adjunction is set to work, the semantic representation would be `fo(say(meta_v, buy(john, book)))`, while Local $*$Adjunction leads to the other reading `fo(say(john, buy(meta_v, book)))`). Let us assume that the Local $*$Adjunction applies to the current stage. After consuming the NP *John*, `tree_expansion_rules` takes on a role again, because the pointer has gone back to the root, type t node, and Generalised Adjunction gives rise to another type t node which would be the root node of the embedded clause in a later stage. After scanning the verb within the embedded clause *katta* "buy-past", the repetitive application of `node_update_rules` generates the meaning of the subordinate clause `fo(buy(meta_v, book))` in the type t node introduced by Generalised Adjunction, bringing the pointer back to the type t node for the subordinate clause. Then the complementizer *to* integrates the unfixed subordinate tree structure to the fixed, topmost type t node. After returning the pointer to the type t node, as shown in Figure 6, not `tree_expansion_rules` but the lexical rule *itta* "say-past" applies to the current node because `tree_expansion_rules` are triggered only if a lexical rule cannot be applied.

As this algorithm shows, the application of these transition rules is restricted by the types of the pointed node. In Japanese, rules which expand tree structures are able to apply to the tree only if the pointed node is of type t.

4 Discussion

This section is devoted to discussing my approach and possibilities for future research. This paper presented a partitioned parsing state for Japanese based

on the DS framework. This approach to locally unfixed nodes and non-local unfixed nodes, such as a type t node which establishes a new embedded clause in the course of incremental processing, realizes more efficient actions for merging unfixed nodes with fixed nodes as well as easier access to the pointed node.

This paper also presented the algorithm of application of the transition rules and lexical rules. I speculate that the algorithm is basically applicable to other head-final languages like Korean. Purver and Otsuka (2003) proposes the generation and parsing model for English on the basis of DS, but as far as I know this paper is the first implementation for Japanese within the DS framework.

The parser (and the DS framework itself) has some problems to be overcome in future research. The most important one to be addressed in the future is the quantitive evaluation of the parser. Although the grammar and lexicon are still small, the parser will be evaluated using corpus, and some stochastic treatment should be added to improve its efficiency.

In the DS formalism the core engine of the parser does not provide any pragmatics-related action, and there have been few in-depth studies presented on this issue. Furthermore, the idea and treatment of *discourse* or *context* is not very clear in DS. Therefore, the other important issue would be the formalization and implementation of *pragmatic actions* giving rise to the merging of meta-variables with appropriate entities in the context.

5 Conclusion

This paper delineated a parser for fragments of Japanese sentences on the basis of Dynamic Syntax. The parser proposed separate treatments of nodes in fixed (binary) trees and unfixed nodes for Japanese, a proposal essential for head-final languages which often may have multiple unfixed nodes at some parsing stages. This allows easier access to the pointed node. This paper also proposed the algorithm of an application of transition rules which makes the parsing more efficient. The transition rules have been classified into `node_update_rules` and `tree_expansion_rules`. The application of these rules is restricted by the position of the pointer in the tree structure. The parser, implemented in Prolog, is able to process simple sentences as well as relative clause constructions and complex sentences.

Acknowledgements

I would like to thank Kei Yoshimoto for valuable suggestions and comments on an earlier draft. I would also like to thank anonymous reviewers for their comments.

References

Cann, R., Kempson, R., Marten, L.: The Dynamics of Language. Elsevier, Amsterdam (2005)

Kempson, R., Meyer-Viol, W., Gabbay, D.: Dynamic Sytnax: The Flow of Language Understanding. Blackwell Publishers, Oxford (2001)

Lombardo, V., Mazzei, A., Sturt, P.: Competence and Performance Grammar in Incremental Processing. In: Keller, F., et al. (eds.) Proceedings of the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together, pp. 1–8 (2004)
Otsuka, M., Purver., M.: Incremental Generation by Incremental Parsing. In: Proceedings of the 6th Annual CLUK Research Colloquium, pp. 93–100 (2003)
Purver, M., Cann, R., Kempson, R.: Grammars as Parsers: Meeting the Dialogue Challenge. Research on Language and Computation 4, 2–3, 259–288 (2006)
Purver, M., Kempson., R.: Incremental Parsing, or Incremental Grammar? In: Proceedings of the ACL Workshop on Incremental Parsing., pp. 74–81 (2004)
Purver, M., Otsuka., M.: Incremental Generation by Incremental Parsing: Tactical Generation in Dynamic Syntax. In: Proceedings of the 9th European Workshop on Natural Language Generation, pp. 79–86 (2003)