A Common Parsing Scheme For Left- and Right-Branching Languages

Paul T. Sato

Department of Computer Science
North Central College
Naperville, Illinois 60566

This paper presents some results of an attempt to develop a common parsing scheme that works systematically and realistically for typologically varied natural languages. The scheme is bottom-up, and the parser scans the input text from left to right. However, unlike the standard LR(k) parser or Tomita's extended LR(1) parser, the one presented in this paper is not a pushdown automaton based on shift-reduce transition that uses a parsing table. Instead, it uses integrated data bases containing information about phrase patterns and parse tree nodes, retrieval of which is triggered by features contained in individual entries of the lexicon. Using this information, the parser assembles a parse tree by attaching input words (and sometimes also partially assembled parse trees and tree fragments popped from the stack) to empty nodes of the specified tree frame, until the entire parse tree is completed. This scheme, which works effectively and realistically for both left-branching languages and right-branching languages, is deterministic in that it does not use backtracking or parallel processing. In this system, unlike in ATN or in LR(k), the grammatical sentences of a language are not determined by a set of rewriting rules, but by a set of patterns in conjunction with procedures and the meta rules that govern the system's operation.

This paper presents some results of an attempt to develop a common parsing scheme that works systematically and realistically for typologically varied natural languages. When this project was started in 1982, the algorithm based on augmented transition networks (ATNs) codified by Woods (1970, 1973) was not only the most commonly used approach to parsing natural languages in computer systems, but it was also the achievement of computational linguistics which was most influential to other branches of linguistics. For example, researchers of psycholinguistics like Kaplan (1972) and Wanner and Maratsos (1978) used ATN-based parsers as simulation models of human language processing. Bresnan (1978) used an ATN model, among others, to test whether her version of transformational grammar was "realistic". Fodor's theory of "super-strategy" Fodor (1979) was also strongly influenced by the standard ATN algorithm. Indeed, as Berwick and Weinberg (1982) contend, parsing efficiency or computational complexity by itself may not provide reliable criteria for the evaluation of grammatical theories. It is evident, however, that computers can be used as an effective means of simulation in linguistics, as they have proved to be in other branches of science.

Nevertheless, as a simulation model of the human faculty of language processing, the standard ATN mechanism has an intrinsic drawback: unless some *ad hoc*, unrealistic, and efficiency-robbing operations are added, or unless one comes up with a radically different grammatical framework, it cannot be used to parse left-branching languages like Japanese in which the beginning of embedded clauses is not regularly marked.

One may try to cope with this problem by developing a separate parsing algorithm for left-branching languages, leaving the ATN formalism to specialize in right-branching languages like English. However, this solution contradicts our intuition that the core of the human faculty of language processing is universal. Another possible alternative, an ATN-type parser which processes left-branching language's sentences backward from right to left, is also unrealistic. If computational linguistics is to provide a simulation model for theoretical linguistics and psycholinguistics, it must develop an alternative parsing scheme which can

0362-613X/88/010020-30$03.00

effectively and realistically process both left-branching and right-branching languages. Even for purely practical purposes, such a scheme is desirable because it will facilitate the development of machine translation systems which can handle languages with different typological characteristics.

Some limitations of ATN-based parsers for handling left-branching languages are illustrated in section 1. The rest of this paper describes and illustrates my alternative parsing scheme called Pattern Oriented Parser (POP), which can be used for both left-branching and right-branching languages. (POP is a descendant of its early prototype called Pattern-Stack Parser, which was introduced in Sato (1983a.)) A general outline of POP is given in section 2, and its operation is illustrated in section 3, using both English and Japanese examples. Some characteristics of POP are highlighted in section 4, after which brief concluding remarks are made in section 5.

The present version of POP is a syntactic analyzer, and it does not take semantics into consideration. However, the system could be readily augmented with procedures that build up semantic interpretations along with syntactic analysis. One such model was presented in Sato (1983b).

1 Limitations of ATN-Based Parsers

1.1 Case Assignment

One of the greatest obstacles faced when attempting to develop an ATN-based parser for a language like Japanese is the unpredictability caused by the relatively free word order and by the left-branching subordinate clauses which have no beginning-of-clause marker.

Indeed, Japanese word order is not completely free. For example, modifiers always precede the modified, and the verb complex (a verbal root plus one or more ordered suffixes marking tense, aspect, modality, voice, negativity, politeness level, question, etc.) is always placed at the end of the sentence. Moreover, almost all nouns and noun phrases occurring in Japanese sentences have one or more suffixes marking case relationships.[1]

However, Japanese postnominal suffixes, by themselves, do not always provide all the necessary information for case assignment. For example, the direct object of a nonstative verb complex is marked by *-o*, while the direct object of a stative verb complex is usually marked by *-ga*, which also marks the subject.

Compare the two sentences in (1).

(1) a. *Mary-wa John-ga nagusame-ta.* 'As for Mary, John consoled her.' (*-wa* = TOPIC, *nagusame-* 'console' <-STATIVE>), *-ta* = PAST)
b. *Mary-wa John-ga wakar-ta.* 'As for Mary, she understood John.' (*wakar-* 'understand' <+STATIVE>)

An ATN-based parser cannot positively identify the functions of the two noun phrases of these sentences until it processes the verb complex at the end of the sentence.

Examples like (2) also illustrate how little can be deduced from postnominal suffixes before the sentence-final verb complex is processed.

(2) a. *Mary-ga hon-o kaw-ta.* 'Mary bought a book.' (*kaw-* 'buy')
b. *John-ga Mary-ni hon-o kaw-sase-ta.* 'John made Mary buy a book.' (*-sase-* = CAUSE)
c. *Mary-ga John-ni hon-o kaw-sase-rare-ta.* 'Mary was made by John to buy a book.' (*-rare-* = PASSIVE)

The agent of the embedded sentence is marked by *-ni* in (2b), but by *-ga* in (2c).

The relatively free word order of Japanese further complicates the situation, as in the six sentences listed in (3) which are all grammatical and all mean "Mary was made by John to buy a book", but each with different noun phrases given prominence.

(3) a. *Mary-ga John-ni hon-o kaw-sase-rare-ta.* = (2c)
b. *Mary-ga hon-o John-ni kaw-sase-rare-ta.*
c. *John-ni Mary-ga hon-o kaw-sase-rare-ta.*
d. *John-ni hon-o Mary-ga kaw-sase-rare-ta.*
e. *Hon-o Mary-ga John-ni kaw-sase-rare-ta.*
f. *Hon-o John-ni Mary-ga kaw-sase-rare-ta.*

1.2 Embedded Sentences

Embedded sentences in languages like Japanese pose more serious problems because they do not normally carry any sign to mark their beginning. As a result, the beginning of a deeply embedded sentence can look exactly like the beginning of a simple top-level sentence, as illustrated in (4).

(4) a. *Mary-ga sotugyoo-si-ta.* 'Mary was graduated (from school).' (*sotugyoo-si-* 'be graduated')
b. *Mary-ga sotugyoo-si-ta kookoo-ga zensyoo-si-ta.* 'The high school from which Mary was graduated was burnt down.' (*kookoo* 'high school', *zensyoo-si-* 'be burnt down')
c. *Mary-ga sotugyoo-si-ta kookoo-ga zensyoo-si-ta to iw-ru.* 'It is reported that the high school from which Mary was graduated was burnt down.' (*to* = END-OF-QUOTE, *iw-* 'say', *-ru* = NON-PAST)
d. *Mary-ga sotugyoo-si-ta kookoo-ga zensyoo-si-ta to iw-ru sirase-o uke-ta.* '(I/we/you/he/she/they) received news (which says) that the high school from which Mary was graduated was burnt down.' (*sirase* 'news', *uke-* 'receive')
e. *Mary-ga sotugyoo-si-ta kookoo-ga zensyoo-si-ta to iw-ru sirase-o uke-ta Cindy-ga nak-te i-ru.* 'Cindy, who received news that the high school from which Mary was graduated was burnt down, is crying.' (*nak-te i-* 'be crying, be weeping')

In order to process sentences listed in (4), the NP network of an ATN-based parser must be expanded by prefixing to it another state with two arcs leaving from it: a PUSH SENTENCE arc that processes a relative clause, and a JUMP arc that processes noun phrases that do not include a relative clause.

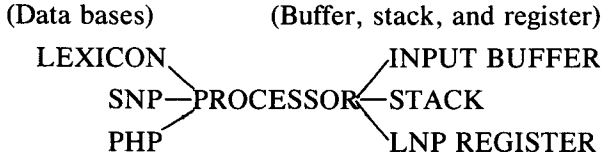
However, as (4) illustrates, there is no systematic way to determine which of the two arcs leaving the first state of this expanded NP network should be taken when the parser encounters the first word of the input. The parser cannot predict the correct path until it has completed processing the entire sentence or the entire relative clause and has seen what followed it. Because there is theoretically no limit to the number of levels of relative clause embedding, the number of combinations of possible arcs to be traversed is theoretically infinite.

2 Overview of Pattern Oriented Parser (POP)

This section presents a quick overview of Pattern Oriented Parser (POP), which I have developed in order to cope with the kind of difficulties mentioned in the previous section.

POP is a left-to-right, bottom-up parser consisting of three data bases, a push-down STACK, a buffer, a register, and a set of LISP programs collectively called here the PROCESSOR that builds the parse tree of the input sentence. The relationship of these components is shown schematically in (5).

(5) Components of POP



The SNP (Sentence Pattern data base) contains a set of parse tree frames, each of which is associated with one class of verbs or verbal derivational suffixes and includes information about the syntactic subcategorization of the members of that class and information about the thematic roles of their arguments. For example, the SNP entry for a class of English verbs which includes *buy* and *sell* looks like (6).

(6) (S (* V)
(AGNT (* NP <+HUMAN>)
(PTNT (* NP <−HUMAN>))

The PHP (Phrase Pattern data base) contains information about the internal structure of noun phrases and adverbial phrases and the procedures for building the parse trees of such phrases. For example, (7) is an English translation of the PHP entry for a Japanese noun phrase which contains a relative clause.[2]

(7) If the CWS is an NP and the TOS is an S, then construct the following noun phrase and push it to the STACK:

(NP (HEAD CWS)
(MOD (rep_emn TOS CWS)))

- CWS is the word or phrase on which the PROCESSOR is currently working.
- TOS is the word or phrase at the top of the STACK.
- (rep_emn TOS *X*) means "pop the TOS and attach *X* to its first matching empty node".
- Each non-empty NP node is given a new index number when it is constructed.

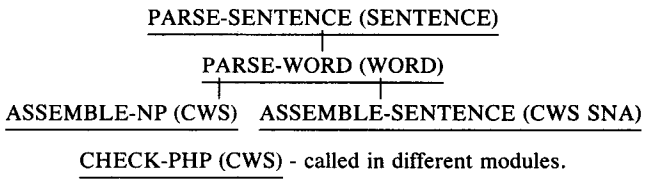
Details of how (7) works will be illustrated in section 3.

The push-down STACK of POP stores partially assembled parse trees and tree fragments, while LNP or the "Last NP" REGISTER temporarily stores a copy of the noun phrase most recently attached to a node in the sentence tree. LNP is necessary to process a noun phrase with a modifier that follows the head noun (e.g., English noun phrases which contain relative clauses). The present version of POP for Japanese does not use an LNP; however, it will prove useful when we try to process parenthetical phrases. The INPUT BUFFER stores the input sentence.

The three data bases of POP are stored on disk and can be updated independently of each other and of the PROCESSOR, while the buffer, the stack and the register are created by the PROCESSOR each time it is invoked.

The major program modules (functions) that constitute the PROCESSOR and their hierarchical calling paths are presented in (8), where the parameters are enclosed in parentheses.

(8) Major Functions of the PROCESSOR



where CWS = the word or the phrase which the PROCESSOR is currently working on
SNA = address of a sentence pattern stored in the SNP

The PROCESSOR is activated when its top-level function, PARSE-SENTENCE, is called with the input sentence as its parameter. PARSE-SENTENCE then creates the STACK, the INPUT BUFFER and the LNP-REGISTER in the memory, puts the input sentence into the INPUT BUFFER, and calls PARSE-WORD. PARSE-WORD searches the LEXICON for an entry which matches the first word in the INPUT BUFFER and, when it is found, calls either ASSEMBLE-NP or ASSEMBLE-SENTENCE, depending on the word type of the entry it finds in the LEXICON, assembles a sub-tree, and pushes the result to the STACK. After that, PARSE-WORD removes the first word from the INPUT BUFFER and repeats the same

process with the next word. In the course of assembling sub-trees, ASSEMBLE-NP uses the PHP, and ASSEMBLE-SENTENCE uses the SNP and the PHP as their data bases. This process continues until the INPUT BUFFER contains only the end-of-sentence mark (EOS), when PARSE-WORD returns control to PARSE-SENTENCE, which pops the assembled sentence from the STACK and sends it to the output device, removes the stack, the buffer and the register from memory, and exits successfully.

As shown in section 3, POP assembles a parse tree primarily by attaching terminal elements (copies of lexical entries) or tree fragments popped from the STACK to the first matching empty node of the matrix tree. All empty nodes of tree frames have an asterisk as their first element, followed by various specifications for matching requirements: (* *ga* (NP <+HUMAN>)) is an empty node for an NP which has a feature specification <+HUMAN> and is flagged with *ga*. To find the first matching empty node, the PROCESSOR conducts a depth-first search for "*" followed by other conditions, and when the first matching empty node is found, it attaches the specified element to that node using the LISP function UNION, thus preventing overlapping elements from being duplicated in the resultant branch. After the attachment is completed, the asterisk is removed from the node.

The use of the LNP REGISTER will be illustrated in subsection 3.3.

3 Operation of POP

This section illustrates the operation of POP more in detail. Subsection 3.1 is a quick walk-through of the overall operation using a simple *yes/no*-question in English as an example, while subsection 3.2 illustrates how POP handles the inherent problems of left-branching languages discussed in section 1, using the Japanese examples presented in that section. Then we turn our attention to English again in subsection 3.3 and illustrate POP's handling of English *wh*-questions and relative clauses.

3.1 Simple English Example

Our first example is (9).

(9) *Did John buy a good book in Boston?*

When PARSE-SENTENCE calls PARSE-WORD and the latter finds *did* in the LEXICON, it makes a copy of the matching lexical entry, (V <+PAST>), and pushes it to the STACK. The next word that PARSE-WORD finds in the INPUT BUFFER is *John*. Therefore, PARSE-WORD searches the LEXICON and gets a copy of the entry that matches this word, ("John"), which is a noun.[3]

Whenever PARSE-WORD encounters a noun, it calls ASSEMBLE-NP with a copy of the lexical entry as its argument. ASSEMBLE-NP assembles a new noun phrase (NP1 "John"), and then it calls CHECK-PHP with the newly assembled NP1 as its argument. CHECK-PHP then examines the PHP data base, and returns NIL to ASSEMBLE-NP because it finds no pattern that matches the string {<V, +PAST> NP} (i.e., the TOS followed by the CWS). Because CHECK-PHP failed to find any matching entry of the PHP, ASSEMBLE-NP pushes NP1 to STACK without conducting any further assembling operation, and returns control to PARSE-WORD. The contents of the STACK at this time are shown in (10).

(10) ((NP1 "John")
(<V, +PAST>))

PARSE-WORD then removes *John* from the INPUT BUFFER, picks up *buy* there, searches the LEXICON, and gets a copy of a matching entry. This is a verb. The lexical entry of every verb or verbal derivational suffix contains an SNA (the SNP address of the sentence pattern associated with it). Therefore, ASSEMBLE-SENTENCE retrieves a copy of the sentence pattern from the address matching the verb's SNA and attaches the verb's remaining lexical entry to its first empty V node (i.e., the first node whose CAR is "*" and the second member is "V"). It then removes the "*" from that node. As mentioned in section 2, the SNP entry for the class of verbs like *buy* and *sell* is (6). Therefore, by attaching (V <"buy">) to the V node of its copy, ASSEMBLE-SENTENCE constructs (11).

(11) (S (V <"buy">)
(AGNT (* NP <+HUMAN>)
(PTNT (* NP <−HUMAN>)))

After (11) is assembled, ASSEMBLE-SENTENCE pops the TOS, attaches it to the first empty node matching its specifications and removes the asterisk at the beginning of that node. The result is (12).

(12) (S (V <"buy">)
(AGNT (NP1 "John")
(PTNT (* NP <−HUMAN>)))

ASSEMBLE-SENTENCE pops TOS again. This time, it is (<V, +PAST>). ASSEMBLE-SENTENCE then examines the PHP and finds two entries (13) and (14) whose conditions match the current state.

(13) If the element popped is a V and if it contains no feature other than tense, number, and/or person, attach it to the V node of the S tree which ASSEMBLE-SENTENCE is currently building.

(14) If there is a tense feature in the element that is popped immediately after the AGNT node (or the OBJ node if the tree has no AGNT node) is filled, attach feature <Q> (i.e., "question") to the main verb of the matrix S.

ASSEMBLE-SENTENCE executes (13) and (14). The result is (15).

(15) (S (V <"buy", +PAST, Q>)
(AGNT (NP1 "John")
(PTNT (* NP <−HUMAN>)))

The STACK is now empty. Therefore, ASSEMBLE-SENTENCE pushes (15) to the STACK and returns control to PARSE-WORD.

PARSE-WORD removes *buy* from the INPUT BUFFER, encounters the indefinite article *a*, gets a copy of the matching lexical entry (DET <-DEF>) from the LEXICON, and pushes it to the STACK. The next word that PARSE-WORD sees is *good*. So a copy of its matching lexical entry (ADJ "good") is pushed to the STACK and *good* is removed from the INPUT BUFFER.

PARSE-WORD then finds *book* in the INPUT BUFFER. Because it is a noun, PARSE-WORD calls ASSEMBLE-NP, which assembles a single-word NP and routinely calls CHECK-PHP. This time, CHECK-PHP finds (16) in the PHP.

(16) If the CWS is an NP and if the TOS is an ADJ, assemble:

(NP (HEAD CWS)
(MOD (pop TOS)))

At this time, the TOS is (ADJ "good"). Therefore, ASSEMBLE-NP pops it and assembles a new noun phrase in accordance with (16) and calls CHECK-PHP again. The new TOS is (DET <−DEF>). CHECK-PHP finds (17) in the PHP which matches this situation.

(17) If the CWS is an NP and if the TOS is a DET, assemble:

(NP (HEAD CWS)
(pop TOS))

ASSEMBLE-NP executes (17). The result is (18).

(18) (NP4 (HEAD (NP3 (HEAD (NP2 "book"))
(MOD(ADJ "good")))
(DET <−DEF>))

Because (18) is an NP, ASSEMBLE-NP calls CHECK-PHP again. This time, the TOS is (15), which is an S tree. CHECK-PHP finds a matching entry in the PHP again, which is (19).

(19) If CWS = NP and TOS = S, pop the TOS and attach the CWS to its first matching empty node.

What is involved here is the assembly of an S, which is outside the domain of ASSEMBLE-NP's responsibility. Therefore, before popping the S from the STACK, ASSEMBLE-NP returns the symbol "AS" to PARSE-WORD. PARSE-WORD then calls ASSEMBLE-SENTENCE substituting (18) for the parameter CWS and "TOS" for the parameter SNA. ASSEMBLE-SENTENCE then builds (20) in the manner explained earlier. The STACK is now empty, and there is no matching PHP entry. Therefore, ASSEMBLE-SENTENCE pushes the newly assembled tree (20) to the STACK.

(20) (S (V <"buy", +PAST, Q>)
(AGNT (NP1 "John")
(PTNT (NP4 (HEAD (NP3 (HEAD (NP2 "book")
(MOD (ADJ "good"))))
(DET <-DEF>))))

The next thing PARSE-WORD sees in the INPUT BUFFER is EOS (end-of-sentence symbol). Therefore, it returns control to PARSE-SENTENCE, which pops (20) from the STACK, and sends it to the output device. Nothing is left in the STACK now. Therefore, PARSE-SENTENCE removes the stack, the buffer and the register from memory and exits successfully.

3.2 JAPANESE EXAMPLES

This section illustrates how POP handles the problems of Japanese sentences discussed in section 1.

3.2.1 CASE MARKING IN SIMPLE SENTENCES

The first example in section 1 was (1a), which is repeated here in (21).

(21) *Mary-wa John-ga nagusame-ta.* 'As for Mary, John consoled her.'
(*-wa* = TOPIC, *nagusame-* 'console' <-STATIVE>), *-ta* = PAST)

POP processes Japanese sentences in basically the same way as it processes English sentences. Therefore, when PARSE-SENTENCE calls PARSE-WORD and PARSE-WORD sees the first word, *Mary-wa*, PARSE-WORD retrieves from the LEXICON a copy of the entry which matches the stem of this word, and calls ASSEMBLE-NP because *Mary* is a noun. ASSEMBLE-NP assembles (NP1 "Mary"), and places its suffix *-wa* in front of the newly assembled NP as its flag. Then CHECK-PHP is called, but it returns NIL because the STACK is still empty. Therefore, ASSEMBLE-NP pushes (*wa* (NP1 "Mary")) to the STACK. The second word, *John-ga*, is processed in the same way, and (*ga* (NP2 "John")) is also pushed to the STACK.

PARSE-WORD then encounters *nagusame-ta* and identifies it as the verb "console" with a past tense suffix. Therefore, PARSE-WORD retrieves a copy of its SNP using the SNA included in the lexical entry, and attaches the lexical entry of *nagusame-ta* to its empty V node. The result is (22).

(22) (S (V <"console", +PAST>)
(PTNT (* *o* (NP <+HUMAN>)))
(AGNT (* *ga* (NP <+HUMAN>))))

ASSEMBLE-SENTENCE then pops the TOS (*ga* (NP2 "John")) and attaches it to the first matching empty node, namely, the AGNT node. The case flag *ga*, which is no longer necessary, is removed.

The next TOS is (*wa* (NP1 "Mary")). As mentioned in section 1, *wa* is a suffix that marks the sentence topic. However, there is no sentence pattern stored in the

SNP which includes a topic (TPIC) node. Instead, it is created by the following instructions (23) retrieved from the PHP.

(23) If the TOS has the flag *wa*:

a. Create a TPIC node which is directly dominated by the topmost S node and attach a "copy" (i.e., the category symbol and its index) of the TOS to this node.
b. Attach the TOS to the first matching empty node.

As is evident from (1a, 1b), the topic marker *wa* absorbs both *ga* and *o*: i.e., the topicalized NP without any other case flag can match both an NP node which is flagged with *o* and an NP node which is flagged with *ga*. Therefore, following (23b), (NP1 "Mary") is attached to the first (and the only) empty node (PTNT) after (23a) is executed. The result is (24), which is the correct parse tree of (21).

(24) (S (V <"console", +PAST>)
(PTNT (NP1 "Mary"))
(AGNT (NP2 "John"))
(TPIC (NP1)))
'As for $Mary_i$, John consoled $Mary_i$.'

Example (1b) is processed in the same way, producing the correct parse tree (25b), although both the PTNT node and the AGNT node of the SNP pattern associated with the stative verb *wakar-* 'understand' are flagged by *ga*, as shown in (25a).

(25) a. SNP pattern associated with *wakar-* "understand"
(S (* V)
(PTNT (* *ga* (NP))
(AGNT (* *ga* (NP <+HUMAN>)))
b. Parse tree of (2-1b) *Mary-wa John-ga wakar-ta.* 'As for Mary, she understood John.'
(S (V <"understand", +PAST>)
(PTNT (NP2 "John"))
(AGNT (NP1 "Mary"))
(TPIC (NP1)))

3.2.2 VERBAL DERIVATIONAL SUFFIX AND CASE MARKING

The next set of examples is (2), repeated here as (26).

(26) a. *Mary-ga hon-o kaw-ta.* 'Mary bought a book.' (*kaw-* 'buy')
b. *John-ga Mary-ni hon-o kaw-sase-ta.* 'John made Mary buy a book.' (*-sase-* = CAUSE)
c. *Mary-ga John-ni hon-o kaw-sase-rare-ta.* 'Mary was made by John to buy a book.' (*-rare-* = PASSIVE)

The SNP pattern associated with *kaw-* 'buy' is (27).

(27) (S (* V)
(PTNT (* *o* (NP)))
(AGNT (* *ga* (NP <+HUMAN>))))

Therefore, the parsing of (26a) to get (28) is straightforward.

(28) (S (V <"buy", +PAST>)
(PTNT (NP2 "book"))
(AGNT (NP1 "Mary")))

The parsing of (26b) is a little more complex because it involves causative suffix *-sase-*, to which is associated another SNP pattern (29) (simplified here for the sake of legibility).

(29) (S (V <CAUSE>)
(PTNT (* or (*ni* (NP = AGNT of S_k))
(*o* (NP = OBJ or PTNT of S_k))))
(AGNT (* *ga* (NP <+HUMAN>)))
(ACTN (* S_k)))
where ACTN = action, S_k = embedded S.

When the PROCESSOR processing (26b) encounters the verb *kaw-sase-ta* 'made to buy', it first retrieves (27) and attaches "buy" to its empty V node to construct the tree frame (30).

(30) (S (V <"buy">)
(PTNT (* *o* (NP)))
(AGNT (* *ga* (NP <+HUMAN>))))

This tree is then incorporated into (29) to obtain the complex tree frame (31). (There is a meta-rule that removes the case flag of a node in the embedded sentence if the node is co-indexed with a node in the matrix sentence.)

(31) (S (V <CAUSE>)
(PTNT (* *ni* (NP_i <+HUMAN>)))
(AGNT (* *ga* (NP <+HUMAN>)))
(ACTN (S (V <"buy">)
(PTNT (* *o* (NP)))
(AGNT (* NP_i <+HUMAN>)))))

By the time the PROCESSOR encounters the verb complex *kaw-sase-ta* 'caused to buy' and constructs the complex tree frame (31), all three noun phrases of the sentence have already been processed and stored in the STACK, as shown in (32).

(32) ((*o* (NP3 "book"))
(*ni* (NP2 "Mary"))
(*ga* (NP1 "John")))

Therefore, when the tree frame (31) is completed, ASSEMBLE-SENTENCE begins to pop elements from the STACK and to attach them to empty nodes of the tree. First, (*o* (NP3 "book")) is popped. The PTNT node of the embedded sentence is the only empty node that matches it, so the popped NP is attached there. Next, (*ni* (NP2 "Mary")) is popped, which is attached to the PTNT node of the matrix sentence and its copy is attached to the co-indexed AGNT node of the embedded sentence. Finally, (*ga* (NP1 "John")) is popped and

attached to the AGNT node of the matrix sentence. The result is (33), which is the correct parse tree of (26b).

(33) (S (V <CAUSE, +PAST>)
(PTNT (NP2 "Mary"))
(AGNT (NP1 "John"))
(ACTN (S (V <"buy">)
(PTNT (NP3 "book"))
(AGNT (NP2)))))
'John made Mary buy a book.'

Example (26c) is a passive of (26b) with passive suffix *-rare-*, with which is associated an SNP pattern (34) (simplified here for the sake of legibility).

(34) (S (V <PASSIVE>)
(PTNT (*ga* (NP = OBJ or PTNT of S_k)))
(AGNT (*ni* (NP = AGNT of S_k)))
(ACTN (S_k)))

Therefore, before beginning to pop elements from the STACK, ASSEMBLE-SENTENCE constructs the complex tree frame (35) by incorporating (31) into (34).

(35) (S (V <PASSIVE, +PAST>)
(PTNT (*ga* (NP_i <+HUMAN>)))
(AGNT (*ni* (NP_j)))
(ACTN (S (V <CAUSE>)
(PTNT (NP_i <+HUMAN>))
(AGNT (NP_j <+HUMAN>))
(ACTN (S (V <"buy">)
(PTNT (*o* (NP)))
(AGNT (NP_i
<+HUMAN>)))))))

At this stage, the contents of the STACK are the same as (32). So when they are popped and attached to the matching nodes according to the principle explained above, we obtain the correct parse tree (36).

(36) (S (V <PASSIVE, +PAST>)
(PTNT (NP1 "Mary"))
(AGNT (NP2 "John"))
(ACTN (S (V <CAUSE>)
(PTNT (NP1))
(AGNT (NP2))
(ACTN (S (V <"buy">)
(PTNT (NP3 "book"))
(AGNT (NP1)))))))
'Mary was made by John to buy a book.'

3.2.3 RELATIVE CLAUSES

As mentioned in section 2, Japanese noun phrases containing a relative clause are processed by the PHP entry presented in (7), repeated here in (37).

(37) If the CWS is an NP and the TOS is an S, then construct the following noun phrase and push it to the STACK:
(NP (HEAD CWS)
(MOD (rep_emn TOS CWS)))

To illustrate how (37) works, we will trace the noun phrase (38), which is included in all sentences cited in (4b) through (4e).

(38) *Mary-ga sotugyoo-si-ta kookoo-ga* 'The high school from which Mary was graduated'
(*sotugyoo-si-* 'be graduated', *-ta* = PAST, *kookoo* 'high school', *-ga* = case suffix)

The SNP pattern associated with *sotugyoo-si-* is (39).

(39)
(S (* V)
(AGNT (* *ga* (NP <+HUMAN>)))
(ABL (* *o* (NP <PLACE, DEF = "school">))))
where ABL = ablative and DEF = default.

Therefore, when the first two words of (38) are processed, (40) is assembled and pushed to the STACK.

(40)
(S (V <"be graduated", +PAST>)
(AGNT (NP1 "Mary"))
(ABL (* *o* (NP <PLACE, DEF = "school">))))

If the next item in the INPUT BUFFER were EOS (as in (4a)), the system pops (40) and, finding that the STACK is now empty, attaches the default value "school" to the empty ABL node, and sends the result to the output device. However, what follows the verb in (38) is a noun. Therefore, ASSEMBLE-NP assembles (*ga* (NP2 "high school")) and calls CHECK-PHP, which finds (37) because the CWS is the noun phrase just assembled and the TOS is (40).

In accordance with (37), (40) is popped from the STACK, and a new noun phrase (41) is assembled and pushed to the STACK.

(41) (*ga* (NP3 (HEAD (NP2 "high school"))
(MOD (S (V <"be graduated", +PAST>)
(AGNT (NP1 "Mary"))
(ABL (NP2))))))

There is no backtracking involved here and, by repeating the same process, POP can process nested relative clauses like those cited in (4) from left to right, without facing any combinatorial explosion.

3.3 *WH*-QUESTION AND RELATIVE CLAUSE IN ENGLISH

The ATN strategy for parsing *wh*-questions and relative clauses in English attracted special attention of many linguists, including Bresnan (1978) and Fodor (1979), because it seemed to support the trace theory and the theory of *wh*-movement transformation. Therefore, we will conclude the illustration of POP by explaining how it handles them.

3.3.1 *WH*-QUESTIONS

No special mechanism is necessary for processing English *wh*-questions like (42) by POP.

(42) a. *Who praised John*?

b. *Who did John praise*?

The SNP pattern associated with the verb *praise* is (43).

(43) (S (* V)
(AGNT (* NP <+HUMAN>))
(PTNT (* NP <+HUMAN>)))

First, we will trace the parse of (42a). The first word, *who*, is processed and the result, (NP1 <+HUMAN, WH, Q>), is pushed to the STACK before the PROCESSOR encounters *praised* and retrieves a copy of (43) from the SNP. Then "praised" is attached to the empty V node of the tree frame, and the TOS is popped and attached to the first matching empty node. Since that NP has the features <WH, Q>, and because the STACK is now empty, the feature <Q> is moved from NP1 node to the V node. The result is (44).

(44) (S (V <"praise", +PAST, Q>)
(AGNT (NP1 <+HUMAN, WH>)
(PTNT (* NP <+HUMAN>)))

Then, *John* is processed in the normal way, and it is attached to the first (and the only) matching node (PTNT), following the ordinary procedure illustrated in section 3.1. The result is the correct parse tree (45).

(45) (S (V <"praise", +PAST, Q>)
(AGNT (NP1 <+HUMAN, WH>)
(PTNT (NP2 "John")))

At first sight, parsing (42b) by POP may seem difficult because the object is placed before the subject in this sentence. However, POP processes the sentence using auxiliary *did* as a clue, just as humans do. In the same way as POP handled the first word of (42a), it processes *who* in (42b) by assembling (NP1 <+HUMAN, WH, Q>) and pushing it to the STACK. And in the same way as it handled *did* in (9), POP assembles (V <+PAST>) and pushes it on top of NP1, after which it processes *John* and pushes (NP2 "John") to the STACK.

The system then encounters *praise* and retrieves (43) from the SNP, pops (NP2 "John") from the STACK, and attaches it to the first matching empty node, which is the AGNT node. Next, (V <+PAST>) is popped, and it is attached to the V node in accordance with (13). Because (V <+PAST>) is an element that is popped immediately after AGNT node is filled and because it contains a tense feature, the feature <Q> is added to this node in accordance with (14). The result is (46).

(46) (S (V <"praise", +PAST, Q>)
(AGNT (NP2 "John"))
(PTNT (* NP <+HUMAN>)))

The TOS is now (NP1 <+HUMAN, WH, Q>), which is popped and attached to the remaining matching node, and its feature <Q> is moved to the V node.[4] The result is the correct parse tree (47).

(47) (S (V <"praise", +PAST, Q>)
(AGNT (NP2 "John"))
(PTNT (NP1 <+HUMAN, WH>)))

3.3.2 RELATIVE CLAUSE

As an example of English sentences which include relative clauses, we will examine (18).

(48) *Joan loves the brilliant linguist who the students respect.*

The first two words are processed and the partial tree (49) is constructed in the usual way, and it is pushed to the STACK.

(49) (S (V <"love", −PAST>)
(AGNT (NP1 "Joan"))
(PTNT (* NP <+HUMAN>)))

The next three words (*the*, *brilliant*, *linguist*) are processed in the ordinary way, and following the PHP instructions cited in (16) and (17), they are assembled into noun phrase (50) and attached to the empty PTNT node of (4-41). The result is (51), and NP4 is the content of the LNP REGISTER.[5]

(50) (NP4 (HEAD (NP3 (HEAD (NP2 "linguist"))
(MOD (ADJ "brilliant"))
(DET <DEF>))))

(51) (S (V <"love", −PAST>)
(AGNT (NP1 "Joan"))
(PTNT (NP4 (HEAD (NP3 (HEAD (NP2 "linguist"))
(MOD (ADJ "brilliant"))
(DET <DEF>)))))

The next word (*who*) is read in. Its lexical entry includes the feature <WH>, and the TOS is (51). Therefore, CHECK-PHP finds (52) which matches these conditions.

(52) If the CWS has a feature <WH> and if the TOS is an S, then
(mark TOS)
and (setq CWS (list (copyi MARKED)
'<REL>))

where - (mark TOS) marks the constituent of the TOS that is equal to the content of the LNP REGISTER
- MARKED represents the constituent of the TOS thus marked
- (copyi X) returns the category index of X.

When (52) is applied, the CWS becomes (53), which is pushed to the STACK.

(53) (NP4 <REL>)

The next two words, *the* and *students*, are processed, and the result (54) is pushed to the STACK in accordance with (17).

(54) (NP6 (HEAD (NP5 "students"))
(DET <+DEF>))

The verb *respect* is encountered, the matching sentence pattern is retrieved, and the verb is attached to its V node. The result is (55).

(55) (S (V <"respect", −PAST>)
(AGNT (* NP <+HUMAN>))
(PTNT (* NP)))

The TOS is popped and attached to the first matching empty node. The result is (56).

(56) (S (V <"respect", −PAST>)
(AGNT (NP6 (HEAD (NP5 "students"))
(DET <+DEF>)))
(PTNT (* NP)))

The next TOS = (53) is popped and attached to the empty node of (56), hence (57).

(57) (S (V <"respect", −PAST>)
(AGNT (NP6 (HEAD (NP5 "students"))
(DET <+DEF>)))
(PTNT (NP4 <REL>)))

CHECK-PHP is called again, which finds matching entry (58).

(58) If the CWS contains <REL> and the TOS contains a marked NP, pop the TOS and replace its marked NP with:

(NP (HEAD MARKED)
(MOD CWS))

Then remove the mark from MARKED and remove feature <REL> from the CWS.

Before (58) is applied, the CWS is (57) and the TOS is (51) of which NP4 is marked in accordance with (52). Following (58), therefore, the daughter of the PTNT node of (51) is replaced by (59).

(59)
(NP7 (HEAD (NP4 (HEAD (NP3 (HEAD (NP2 "linguist"))
(MOD (ADJ "brilliant"))
(DET <DEF>)))
(MOD (S (V <"respect", −PAST>)
(AGNT (NP6 (HEAD (NP5 "students"))
(DET <+DEF>)))
(PTNT (NP4)))))

The result of this replacement is (60), and it is pushed to the STACK.

(60)
(S (V <"love", −PAST>)
(AGNT (NP1 "Joan"))
(PTNT (NP7 (HEAD (NP4 (HEAD (NP3 (HEAD (NP2 "linguist"))
(MOD (ADJ "brilliant"))
(DET <DEF>)))
(MOD (S (V <"respect", −PAST>)
(AGNT (NP6 (HEAD (NP5 "students"))
(DET <+DEF>)))
(PTNT (NP4)))))))

The next element found in the INPUT BUFFER is EOS (end-of-sentence). So the PROCESSOR pops (60) and sends it to the output device.

4 Highlights of Some Characteristics of POP

4.1 Verbal Derivational Suffixes and Case Assignment

As illustrated in (2), the same postnominal suffixes mark different relations in Japanese, depending on the verbal derivational suffixes used in the verb complex. Traditional generative grammarians (like Kuno (1973)) tried to explain this by means of a series of transformational rules such as agentive *ni* attachment, equi-NP deletion, Aux deletion, verb raising, subject marking, object marking, and *ga/ni* conversion, which were applied cyclically. This transformational approach is still widely practiced by researchers of Japanese linguistics. However, as demonstrated by Sato (1983b), this is unsuitable for application to parsing because many of the transformational rules involved here are non-reversible.

A relatively recent approach to this problem is to use a set of rules like (61) which Kuroda (1976) calls Canonical Surface Structure Filters and Miyagawa (1980) calls Case Redundancy Rules.

(61) a. [NP ---] ==> [NP-*ga* ---]
b. [NP NP ---] ==> [NP-*ga* NP-*o* ---]
c. [NP NP NP ---] ==> [NP-*ga* NP-*ni* NP-*o* ---]

These rules are invoked after applying all transformational rules (Kuroda 1976) or all word formation rules (Miyagawa 1980), and they attach suffixes to noun phrases as specified in their output, without regard to the functions of the phrases to which they are attached. The selection of case suffixes and the order of their appearance in the surface structure are determined solely by the number of unmarked noun phrases in the sentence. This approach would work well if Japanese speakers always followed the "canonical word order". However, the so-called canonical word order is not always followed.

Contrary to the theories of Kuroda and Miyagawa which treat Japanese case suffixes as if they were useless appendages which have no syntactic role, POP uses them as integral parts of the input data and, as a result, it does not have to require the input sentences to conform to the "canonical word order". As illustrated in subsection 3.2.2, POP first constructs an expanded sentence tree frame using the SNP patterns that match the SNA's of the derivational suffixes. After this expanded frame is completed, arguments are popped from the STACK and attached to appropriate nodes in the usual manner. Note that the flag specifications on the tree frame are automatically adjusted in course of its expansion, so no further adjustment resorting to the "canonical word order" or scrambling is necessary.

4.2 EMBEDDED SENTENCES

As illustrated in subsection 3.2.3, POP handles Japanese complex sentences with relative clauses without facing combinatorial explosion. Especially noteworthy is the similarity in the PHP instructions to assemble noun phrases with relative clause in Japanese (37) and in English (58), which are paraphrased in (62).

(62) PHP entries for assembling NP with a relative clause
  a. For Japanese = (37):
    1. Pop the TOS (which is a sentence with an empty NP node).
    2. Attach a copy of the CWS (which is an NP) to the first matching empty node of the popped sentence tree.
    3. Assemble a new NP tree with the CWS as its HEAD and the sentence tree assembled in step 2 as its MOD(ifier).
  b. For English = (58):
    1. Pop the TOS (which is a sentence with a marked NP).
    2. Assemble a new NP tree with the marked NP of the sentence popped in step 1 as its HEAD and the CWS (which is a sentence tree containing an NP node co-indexed with the marked NP according to (52)) as its MOD.

The only major difference between the two is that in Japanese (62a) the relative clause is in the STACK when the head NP is encountered, while in English (62b) the head NP is a branch of an S tree in the STACK when the relative pronoun is encountered. This difference is a natural consequence of the difference in word order between the two languages (i.e., left-branching vs. right-branching).

An important fact is that POP for Japanese does not have to know in advance whether the sentence fragment that it is processing is a matrix sentence like (4a) or an embedded sentence like (4b) through (4e).

4.3 COMPARISON WITH MARCUS'S PARSIFAL

The reader may have wondered if there is any direct relationship between POP and Marcus's PARSIFAL Marcus (1980): both are bottom-up parsers, where attachment can be made freely to any matching node in the ACTIVE NODE STACK (Marcus) or the CWS (POP). Therefore, a brief comparison of these two systems may be in order.

When I heard about Marcus's work for the first time, the development of POP was already well under way: its basic algorithm was already completed and coding had already started. Therefore, the similarity between PARSIFAL and POP, if any, is only accidental. Moreover, the basic philosophies of these two systems are different. Marcus's goal was to build a "strictly deterministic" parser for natural language; mine was to build a parser that can handle not only right-branching sentences but also left-branching sentences naturally and without facing a combinatorial explosion. POP does not have any back-tracking or parallel parsing mechanism, but the lack of such mechanism was a consequence of the parser's algorithm and not an intended goal.

In fact, the only significant similarity between PARSIFAL and POP is between the former's pattern/action rules and the latter's PHP entries. The latter can be rewritten using the format of the former. However, the similarity ends here. PARSIFAL's rules are partially ordered by a priority scheme; POP's PHP entries are not ordered nor do they have priority over any other entries in the PHP. In PARSIFAL, a grammar rule activates a packet by attaching it to the constituent at the bottom of the ACTIVE NODE STACK, and the packet of rules remains attached to the node even after the node is pushed up.[6] Such rules remain dormant until the node to which they are attached comes at the bottom of the ACTIVE NODE STACK again. On the other hand, POP's PHP pattern does not remain with any node after a phrase tree (or an S tree) is assembled and pushed to the STACK. A copy of PHP pattern is retrieved from the data base each time it becomes necessary. This strategy saves the memory space in the STACK, although it requires a longer processing time.

POP lacks one of PARSIFAL's most significant characteristics: the distinction between the ACTIVE NODE STACK and the BUFFER. POP also distinguishes the place where trees are actually constructed (which I informally call here the "work space") and the place where the results are stored (i.e., the STACK). However, the similarity again ends here. POP's "work space" is neither a stack nor a buffer, but a machine-dependent temporary memory space where the program (ASSEMBLE-NP, ASSEMBLE-SENTENCE, etc.) retrieves and manipulates partial trees popped from the STACK or lexical entries copied from the LEXICON. Unlike PARSIFAL's ACTIVE NODE STACK, POP's "work space" cannot store any partially completed tree which is not "active". Such inactive partial trees are stored in the STACK.

PARSIFAL's BUFFER is primarily a facility for "look-ahead". Therefore, it contains unprocessed input words as well as phrase trees with no empty node. It contains no phrase tree which has empty nodes, because such trees are stored in the ACTIVE NODE STACK. In contrast, the primary purpose of POP's STACK is to store tree fragments and tree frames. It is not a "look-ahead" facility and therefore does not contain any unprocessed input word. When POP's PROCESSOR looks at an input word, it must process it immediately.

POP can process sentences like (4) without back-tracking or any look-ahead mechanism, while such sentences would remain "garden path sentences" for Marcus's parser even with its limited look-ahead mechanism.

5 Conclusion

POP as presented in this paper is still in its evolving stage, and it needs further refinement. For example, we could include in the common POP core such meta rules as "attach feature <ANIMATE> to AGNT node". As suggested in section 1, we could also augment POP with procedures to build semantic interpretations along with syntactic analysis. Such refinements and improvements will continue.

However, the basic linguistic theory underlying my scheme may not have to undergo a radical change in the process. According to the theory underlying this work, it is not a set of patterns or rewriting rules that singly determines the grammatical sentences of a language. Rather, it is the patterns (SNP) in conjunction with procedures (PHP) and POP's meta rules that do so. In other words, this system points the way to a slightly different view of grammar competence than a basically Chomskian one, in which one provides a competence grammar that incorporates processing while leaving aside details of performance.

References

Berwick, Robert C. and Weinberg, Amy S. 1982 Parsing Efficiency, Computational Complexity, and the Evaluation of Grammatical Theories. *Linguistic Inquiry* 13(2): 165–191.

Bresnan, Joan W. 1978 A Realistic Transformational Grammar. In Halle, Morris; Bresnan, Joan W.; and Miller, G. A., Eds., *Linguistic Theory and Psychological Reality*. MIT Press, Cambridge, Massachusetts: 1–59.

Fillmore, Charles J. 1968 The Case for Case. In Bach, E. and Harms, R.T., Eds., *Universals in Linguistic Theory*. Holt, Rinehart and Winston, New York.

Fodor, Janet D. 1979 Superstrategy. In Cooper, William E. and Walker, Edward C.T., Eds., *Sentence Processing: Psycholinguistic Studies Presented to Merrill Garrett*. Lawrence Erlbaum, Hillsdale, New Jersey: 249–279.

Kaplan, Ronald M. 1972 Augmented Transition Networks as Psychological Models of Sentence Comprehension. *Artificial Intelligence* 3:77–100.

Kuno, Susumu. 1973 *The Structure of the Japanese Language*. MIT Press, Cambridge, Massachusetts.

Kuroda, S-Y. 1976 A lecture given to graduate students and faculty members of the Linguistics Department of the University of Massachusetts at Amherst.

Marcus, Mitchell P. 1980 *A Theory of Syntactic Recognition for Natural Language*. MIT Press, Cambridge, Massachusetts.

Miyagawa, Shigeru. 1980 *Complex Verbs and the Lexicon*. Coyote Papers, Vol. 1. University of Arizona, Tucson, Arizona. (Originally a Ph.D. dissertation, University of Arizona.)

Sato, Paul T. 1982 The Status of "Particles" and Its Typological Implications. *Papers in Japanese Linguistics* 8:191–205.

Sato, Paul T. 1983a On-line Parsing Strategies for English and Japanese. A panel presentation at AAS Symposium on Japanese Language on the Computer, in San Francisco, California.

Sato, Paul T. 1983b Lexicalist vs. Tarnsformationalist Hypothesis on Parsing Japanese Phrases with Complex Verbs. Presented at the Linguistic Conference on East Asian Languages: Verb Phrases, in Los Angeles, California. (Reprinted in Kim, Nam-Kil and Tiee, Henry H., Eds. 1985 *Studies in East Asian Linguistics*. Department of East Asian Languages and Cultures, University of Southern California, Los Angeles, California: 155–165.)

Tomita, Masaru. 1986 *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Boston, Massachusetts.

Wanner, E. and Maratsos, M. 1978 An ATN Approach to Comprehension. In Halle, Morris; Bresnan, Joan W.; and Miller, G.A., Eds., *Linguistic Theory and Psychological Reality*. MIT Press, Cambridge, Massachusetts: 119–161.

Woods, William A. 1970 Transition Network Grammar for Natural Language Analysis. *Communications of the ACM* 13:591–606.

Woods, William A. 1973 An Experimental Parsing System for Transition Networks. In Rustin, R., Ed., *Natural Language Processing*. Algorithmics Press, New York: 111–154.

Notes

1. These postnominal suffixes are usually called "particles", but see Sato (1982).
2. For the sake of readability, I present all PHP entries cited in this paper in their English translation.
3. "John" is an abbreviation of a bundle of features, <N, +PROPER, +HUMAN, +MALE, −PLURAL, . . .>. For convenience' sake, such feature bundles are often rendered in this paper by an English word enclosed in quotation marks.
4. In fact, this <Q> attachment does not add another <Q> to the V node because there is already a <Q> there. Note that POP's attachment function uses UNION.
5. As mentioned in section 2, POP always keeps a copy of the most recently assembled NP in LNP REGISTER, or the "last (assembled) NP register", although I have not indicated this each time it occurred.
6. Marcus (1980) uses the phrase "associate with" instead of "attach to" here. PARSIFAL's ACTIVE NODE STACK grows downward.