1984

# An efficient all-paths parsing algorithm for natural languages

Masaru Tomita
*Carnegie Mellon University*

CMU-CS-84-163

# An Efficient All-paths Parsing Algorithm for Natural Languages

**Masaru Tomita**
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

25 October 1984

## Abstract

An extended LR parsing algorithm is introduced and its application to natural language processing is discussed. Unlike the standard LR, our algorithm is capable of handling arbitrary context-free phrase structure grammars including ambiguous grammars, while most of the LR parsing efficiency is preserved. When an input sentence is ambiguous, it produces all possible parses in an efficient manner with the idea of a "graph-structured stack." Comparisons with other parsing methods are made.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

# Table of Contents

# 1. Introduction

When a parser encounters an ambiguous input sentence, it can deal with that sentence in one of two ways. One way is to produce a single parse which is the most preferable. Such parsers are called *one-path parsers*. On the other hand, parsers that produce all possible parses of the ambiguous sentence are called *all-paths parsers*. One-path parsers are, naturally, much faster than all-paths parsers because they look for only one parse. There are, however, situations where all-paths parsers should be used. For example, consider the following short story.

I saw the man with a telescope.
He bought it at the department store.

When the first sentence is read, there is absolutely no way of resolving the ambiguity[1] at that time. The only action the system can take is to produce two parses and store them somewhere for later disambiguation.

Another situation where all-paths parsers should be used is what we call *interactive parser*, which originally motivated this work. An interactive parser disambiguates structurally ambiguous input sentences by asking its user questions interactively. For example, an interactive parser asks a question such as the following to disambiguate the sentence "I saw a man with a telescope."

1) The action "I saw a man" takes place "with a telescope"
2) "a man" is "with a telescope"
NUMBER?

The technique to implement this is described in [20] (also in Appendix). In order to ask such a question, all possible structures of the ambiguous sentence must be available, and therefore all-paths parsing is required.

In this paper, we introduce an efficient all-paths parsing algorithm named *MLR*, which is an extension of LR. The LR parsing algorithm is a very efficient one-path parsing algorithm which is much faster than the Cocke-Younger-Kasami algorithm [1] and Earley's algorithm [9] especially when the grammar is significantly large. The LR parsing algorithm, however, has seldom been used for natural language processing, because the LR parsing algorithm is applicable only to a small subset of context-free grammars, and usually it cannot apply to natural languages. Our MLR parsing algorithm, while most of the LR parsing efficiency is preserved, can apply to arbitrary context-free grammars, and is therefore applicable to natural languages.

One might wonder, by the way, whether natural languages can be specified in context-free phrase structure. It had been thought that natural languages are not context-free, before the recent

[1] "I" have the telescope, or "the man" has the telescope.

literature [11] showed that the belief is not necessarily true and there is no reason for us to give up the context-freedom of natural languages. We do not discuss this matter further, because even if natural languages are not context-free, a fairly comprehensive grammar for a subset of natural language sufficient for practical systems can be written in context-free phrase structure.

In section 2, we briefly review LR parsing, and discuss the problem that arises when applied to natural languages. In section 3, we introduce the MLR parsing algorithm, and in section 4 we describe how to represent parse trees efficiently and how to produce them using MLR parsing. Section 5 compares MLR parsing with other existing parsing methods. Finally, we enumerate future tasks to be completed in section 6 and potential contributions in section 7.

# 2. LR parsing

LR parsers [1, 2] have been developed originally for programming languages. An LR parser is a shift-reduce parser which is deterministically guided by a parsing table indicating what action should be taken next. The parsing table can be obtained automatically from a context-free phrase structure grammar, using an algorithm first developed by DeRemer [7, 8]. We do not describe the algorithm here, referring the reader to Chapter 6 in Aho and Ullman [3].

## 2.1. An example

An example grammar and its LR parsing table obtained by the algorithm are shown in Figure 2-1 and 2-2, respectively.

```
-------------------------
 (1)   S --> NP VP
 (2)   S --> S PP
 (3)  NP --> *det *n
 (4)  PP --> *prep NP
 (5)  VP --> *v NP
-------------------------
```

Figure 2-1: Example Grammar

| State | *det | *n | *v | *prep | $ | NP | PP | VP | S |
|---|---|---|---|---|---|---|---|---|---|
| 0 | sh3 | | | | | 2 | | | 1 |
| 1 | | | | sh5 | acc | | 4 | | |
| 2 | | | sh6 | | | | | 7 | |
| 3 | | sh8 | | | | | | | |
| 4 | | | | re2 | re2 | | | | |
| 5 | sh3 | | | | | 9 | | | |
| 6 | sh3 | | | | | 10 | | | |
| 7 | | | | re1 | re1 | | | | |
| 8 | | | re3 | re3 | re3 | | | | |
| 9 | | | | re4 | re4 | | | | |
| 10 | | | | re5 | re5 | | | | |

action table goto table

Figure 2-2: LR Parsing Table

Grammar symbols starting with "*" represent pre-terminals. Entries "sh *n*" in the action table (the left part of the table) indicate the action "shift one word from input buffer onto the stack, and go to state *n*". Entries "re *n*" indicate the action "reduce constituents on the stack using rule *n*". The entry "acc" stands for the action "accept", and blank spaces represent "error". Goto table (the right part of the table) decides to what state the parser should go after a reduce action. The exact definition and operation of the LR parser can be found in Aho and Ullman [3].

Let us parse a simple sentence "My car has a radio" using the LR parsing table. The trace of the LR parsing is shown in Figure 2-3.

Inputbuffer = MY CAR HAS A RADIO $

```
STACK                          NA    NW
----------------------------------------
0                          sh3  MY

0 *det 3                       sh8   CAR

0 *det 3 *n 8                  re3   HAS

0 NP 2                         sh6   HAS

0 NP 2 *v 6                    sh3   A

0 NP 2 *v 6 *det 3             sh8   RADIO

0 NP 2 *v 6 *det 3 *n 8        re3   $

0 NP 2 *v 6 NP 10              re6   $

0 NP 2 VP 7                    re1   $

0 S 1                          acc   $
----------------------------------------
```

Figure 2-3: Trace of LR Parsing

The number on the top (rightmost) of the stack indicates the current state. Initially, the current state is 0. The inputbuffer initially contains the input sentence followed by the end marker "$".

Since the parser is looking at the word "MY", whose category is "*det", the next action "shift and goto state 3" is determined from the action table. The parser takes the word "MY" away from the inputbuffer, pushes the preterminal "*det" onto the stack, and goes to state 3 pushing the number "3" onto the stack. The inputbuffer is now "CAR HAS A RADIO $".

The next word the parser is looking at is "CAR", whose category is "*n", and "shift and goto state 8" is determined from the action table as the next action. Thus, the parser takes the word "CAR" from the inputbuffer, pushes the preterminal "*n", and goes to state 8 pushing the number "8" onto the stack. The inputbuffer is now "HAS A RADIO $".

The next word is "HAS", and from the action table, "reduce using rule 3" is determined as the next action. So, the parser reduces the stack using the rule "NP --> *det *n". The current state "2" is determined by the goto table from the state before the removed constituents. The inputbuffer is still "HAS A RADIO $", because we have not shifted the word "HAS" yet.

Since the parser is still looking at the word "HAS", and it is in state 2, the next action is "shift and goto state 6". So, the parser shifts the word "HAS" from the inputbuffer onto the stack, and goes to state 6.

Several steps later, the parser eventually finds the action "accept", which is the signal for the parser to halt the process.

### 2.2. Problem in Applying to Natural Languages

Several advantages of LR parsing exist which makes it attractive to use in natural language processing. As we have seen in the example, the LR paring is one of the most efficient parsing algorithms. It is totally deterministic and no backtracking or search is involved. The algorithm to build an LR parsing table is well-established, and there is a practical program called YACC (Yet Another Compiler Compiler) [12] running on Unix.

Unfortunately, we cannot directly adopt the LR parsing technique for natural languages. This is because not all context-free phrase structure grammars (CFPSG's) can have an LR parsing table. Only a small subset of CFPSG's called *LR grammars* (see figure 2-4) can have such an LR parsing table. Every ambiguous grammar is not LR, for example. And since natural language grammars are almost always ambiguous, they are not LR; therefore we cannot have an LR parsing table for natural language grammars.

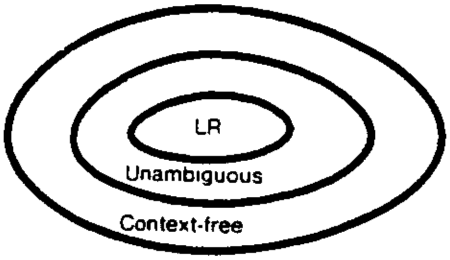


**Figure 2-4:** Context-free Grammars and LR grammars

If a grammar is non-LR, its parsing table will have multiple entries[2] ; one or more of the action table entries will be multiply defined. Figures 2-5 and 2-6 show an example non-LR grammar and its parsing table.

[2] They are often called *conflict*.

```
(1)   S --> NP VP
(2)   S --> S PP
(3)  NP --> *n
(4)  NP --> *det *n
(5)  NP --> NP PP
(6)  PP --> *prep NP
(7)  VP --> *v NP
```

Figure 2-5: An Example Ambiguous Grammar

| State | *det | *n | *v | *prep | $ | NP | PP | VP | S |
|---|---|---|---|---|---|---|---|---|---|
| 0 | sh3 | sh4 | | | | 2 | | | 1 |
| 1 | | | | sh6 | acc | | 5 | | |
| 2 | | | sh7 | sh6 | | | 9 | 8 | |
| 3 | | sh10 | | | | | | | |
| 4 | | | re3 | re3 | re3 | | | | |
| 5 | | | | re2 | re2 | | | | |
| 6 | sh3 | sh4 | | | | 11 | | | |
| 7 | sh3 | sh4 | | | | 12 | | | |
| 8 | | | | re1 | re1 | | | | |
| 9 | | | re5 | re5 | re5 | | | | |
| 10 | | | re4 | re4 | re4 | | | | |
| 11 | | | re6 | re6,sh6 | re6 | | 9 | | |
| 12 | | | | re7,sh6 | re7 | | 9 | | |

Figure 2-6: LR Parsing Table with Multiple Entries

We can see that there are two multiple entries in the action table; on the rows of state 11 and 12 at the column labeled "*prep". It has been thought that, for LR parsing, multiple entries are fatal because once a parsing table has multiple entries, deterministic parsing is no longer possible and some kind of non-determinism is necessary. However, in the following section, we shall introduce an extended LR parsing algorithm, named MLR, that can handle parsing tables with multiple entries using a graph-structured stack. The MLR parsing algorithm, while it can apply to arbitrary CFPSG's, preserves most of the efficiency of the standard LR parsing algorithm.

# 3. MLR parsing

As mentioned above, once a parsing table has multiple entries, deterministic parsing is no longer possible and some kind of non-determinism is necessary. The first subsection describes a simple non-determinism, i.e. pseudo-parallelism (breath-first search), in which the system maintains a number of stacks simultaneously. We call the list of stacks *Stack List.* The next subsection describes the idea of stack combination, which was introduced by Tomita's paper [21] (also in Appendix), to make the algorithm efficient and feasible. With this idea, stacks are represented as trees (or a forest). Finally, a further refinement, the graph-structured stack, is described to make the algorithm even more efficient.

## 3.1. With Stack List

The basic idea is to handle multiple entries non-deterministically. We adopt pseudo-parallelism (breath-first search), maintaining a list of stacks called *Stack List.* The pseudo-parallelism works as follows.

A number of *processes* are operated in parallel. Each process has a stack and behaves basically the same as in standard LR parsing. When a process encounters a multiple entry, the process is split into several processes (one for each entry), by duplicating its stack. When a process encounters an error entry, the process is killed, by removing its stack from the stack list. All processes are synchronized; they shift a word at the same time so that they always look at the same word. Thus, if a process encounters a shift action, it waits until all other processes also encounter the shift action.

Figure 3-1 shows a snapshot of the stack list right after shifting the word "with" in the sentence "I saw a man on the bed in the apartment with a telescope" using the grammar in Figure 2-5 and the parsing table in Figure 2-6.

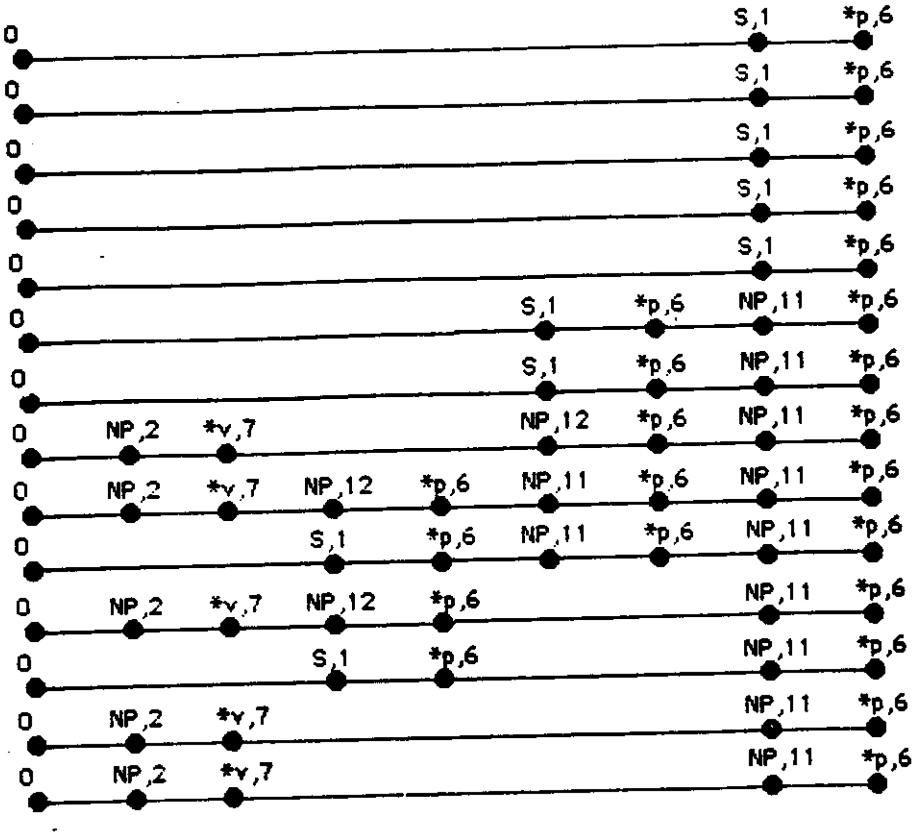


**Figure 3-1:** Stack List

For the sake of convenience, we denote a stack with nodes and edges. The leftmost node is the bottom of the stack, and the rightmost node is the top of the stack. Each node, except the leftmost node, has two labels, a grammar symbol and a state number. The leftmost node has only a state number, and it is always 0. The distance between nodes (length of an edge) does not have any significance, except it may help the reader understand the stacks' status.

We notice that some stacks in the stack list appear to be identical. They are, however, internally different because they have reached the current state in different ways. Although we shall describe a method to compress them into one stack in the next section, we consider them to be different in this section.

A disadvantage of the stack list method is that there are no interconnections between stacks (processes) and there is no way for a process to utilize what other processes have done already. Therefore, the number of stacks in the stack list grows exponentially as ambiguities are encountered. For example, these 14 processes in Figure 3-1 will parse the rest of the sentence "the telescope" 14 times in exactly the same way. This can be avoided by using a tree-structured stack, which is described in the following subsection.

### 3.2. With a Tree-structured Stack

If two processes are in a common state, that is, if two stacks have a common state number at the rightmost node, they will behave in exactly the same manner until the node is popped from the stacks by a reduce action. To avoid this redundant operation, these processes are unified into one process by combining their stacks. Whenever two or more processes have a common state number on the top of their stacks, the top nodes are unified, and these stacks are represented as a tree, where the top node corresponds to the root of the tree. We call this a tree-structured stack. When the top node is popped, the tree-structured stack is split into the original number of stacks. In general, the system maintains a number of tree-structured stacks in parallel, so stacks are represented as a forest. Figure 3-2 shows a snapshot of the tree-structured stack right after shifting the word "with".

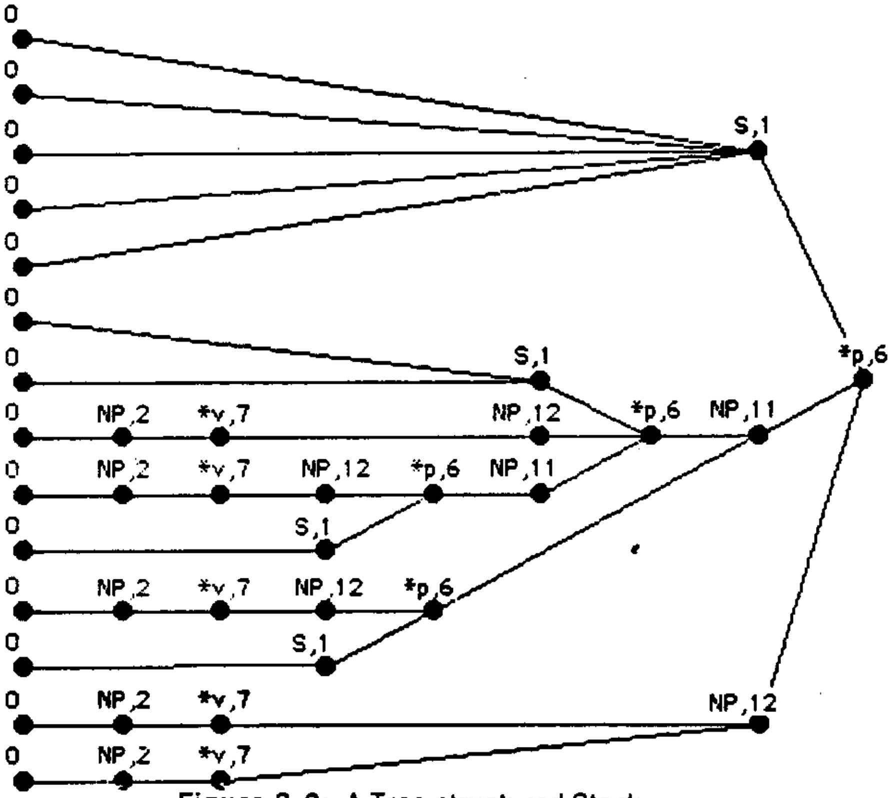


**Figure 3-2:** A Tree-structured Stack

Although the number of stacks is reduced significantly by the stack combination technique, the number of branches of the tree-structured stack (the number of bottoms of the stack) that we must maintain still grows exponentially as ambiguities are encountered. In the next subsection, we describe a further modification in which stacks are represented as a directed acyclic graph.

### 3.3. With a Graph-structured Stack

So far, when we split a stack, we make a copy of the whole stack. However, we do not necessarily have to copy the whole stack: Even after different parallel operations on the tree-structured stack, the bottom portion of the stack may remain the same. Only the necessary portion of the stack should therefore be split. When a stack is split, the stack is thus represented as a tree, where the bottom of the stack corresponds to the root of the tree. With the stack combination technique described above, stacks are represented as a directed acyclic graph. Figure 3-3 shows a snapshot of the graph stack.

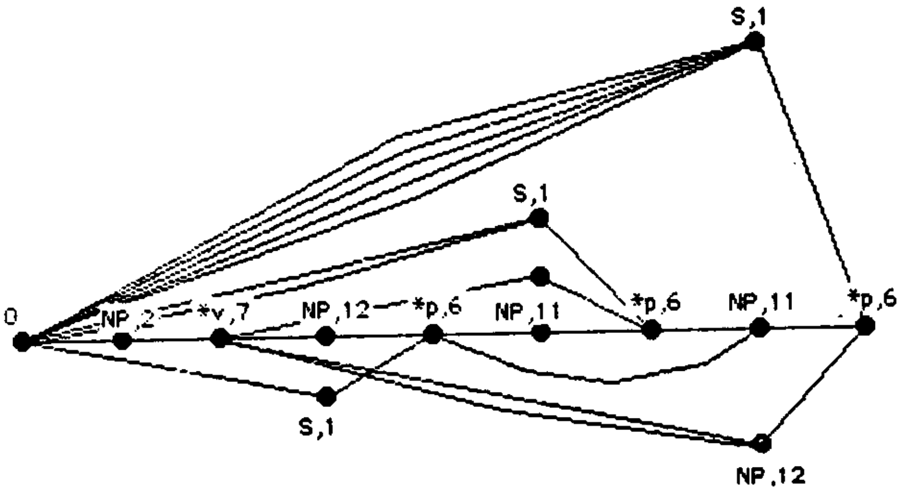


Figure 3-3: A Graph-Structured Stack

It is easy to show that the MLR parser with the graph-structured stack does not parse any part of an input sentence more than once in the same way. This is because if two processes had parsed a part of a sentence in the same way, they would have been in the same state, and they would have been combined as one process.

So far, we have focussed on how to accept or reject a sentence. In practice, however, the parser must not only simply accept or reject sentences, but also must build the syntactic structure(s) of the sentence (parse trees). In the next section, we describe how to represent the syntactic structure and how to build it with the MLR parser.

## 4. An Efficient Representation of Parse Trees

The ambiguity (the number of parses) of a sentence grows exponentially as the length of a sentence grows. Thus, one might notice that, even with an efficient parsing algorithm such as the one we described, the parser would take exponential time because exponential time would be required merely to print out all parse trees. We must therefore provide an efficient representation so that the size of the representation does not grow exponentially.

In this section, we describe two techniques for providing an efficient representation: sub-tree sharing and local ambiguity packing. It should be mentioned that these two techniques are not completely new ideas, and some existing systems already adopted these techniques. We shall therefore focus on how to implement these techniques with the MLR parsing algorithm.

### 4.1. Sub-tree Sharing

If two or more parse trees have a common sub-tree, the sub-tree should be represented only once. For example, parse trees for the sentence "I saw a man with a telescope" should be represented as follows:

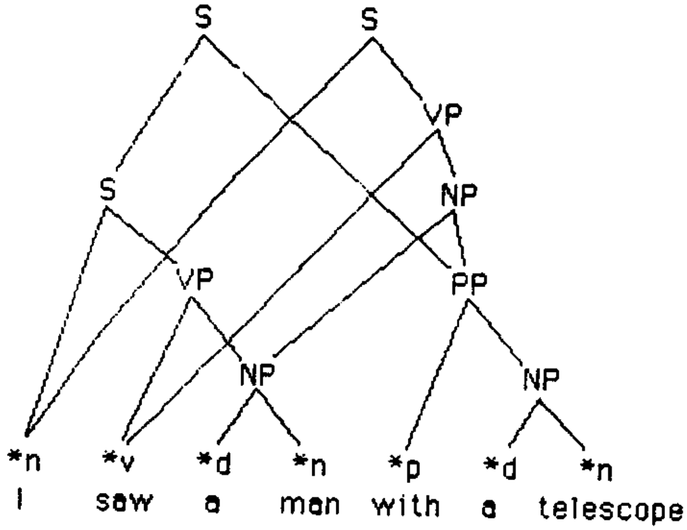


**Figure 4-1:** Shared Trees

Our MLR parsing is very well suited for building this kind of shared tree as its output, as we shall see in the following.

To implement this, we no longer push grammatical symbols on the stack; instead, we push pointers to a node[3] of shared trees. When the parser "shifts" a word, it creates a leaf node labeled with the

[3] One must not be confused "nodes" of shared trees and "nodes" of a graph-structured stack. Hereinafter, the term "node" will be used as of shared trees unless otherwise noted.

word and the pre-terminal, and instead of pushing the pre-terminal symbol, a pointer to the newly created leaf node is pushed onto the stack. If the exact same leaf node (i.e. the node labeled with the same word and the same pre-terminal) already exists, a pointer to this existing node is pushed onto the stack, without creating another node. When the parser "reduces" the stack, it pops pointers to shared tree nodes from the stack, creates a new node whose successive nodes are pointed to by those popped pointers, and pushes a pointer to the newly created node onto the stack.

Using this relatively simple procedure, the MLR parser can produce the shared trees as its output without any other special book-keeping mechanism, because the MLR parser never does the same reduce action twice in the same manner.

### 4.2. Local Ambiguity Packing

We define that two subtrees represent *local ambiguity* if they have common leaf nodes and their top nodes are labeled with the same non-terminal symbol. That is to say, a fragment of a sentence is locally ambiguous if the fragment can be reduced to a certain non-terminal symbol in two or more ways. If a sentence has many local ambiguities, the total ambiguity would grow exponentially. The *local ambiguity packing* is a technique to avoid this, and works in the following way. Two subtrees that represent local ambiguity are merged and treated by higher-level structures as if there were only one subtree. Examples of shared trees without and with the local ambiguity packing are shown in Figure 4-2.

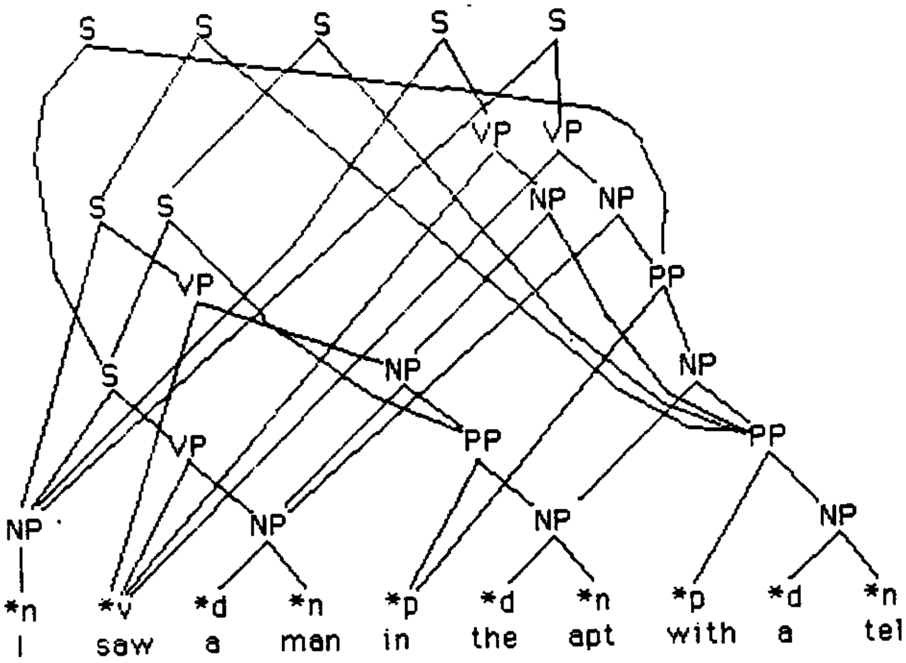


**Figure 4-2(a):** Unpacked Shared Trees

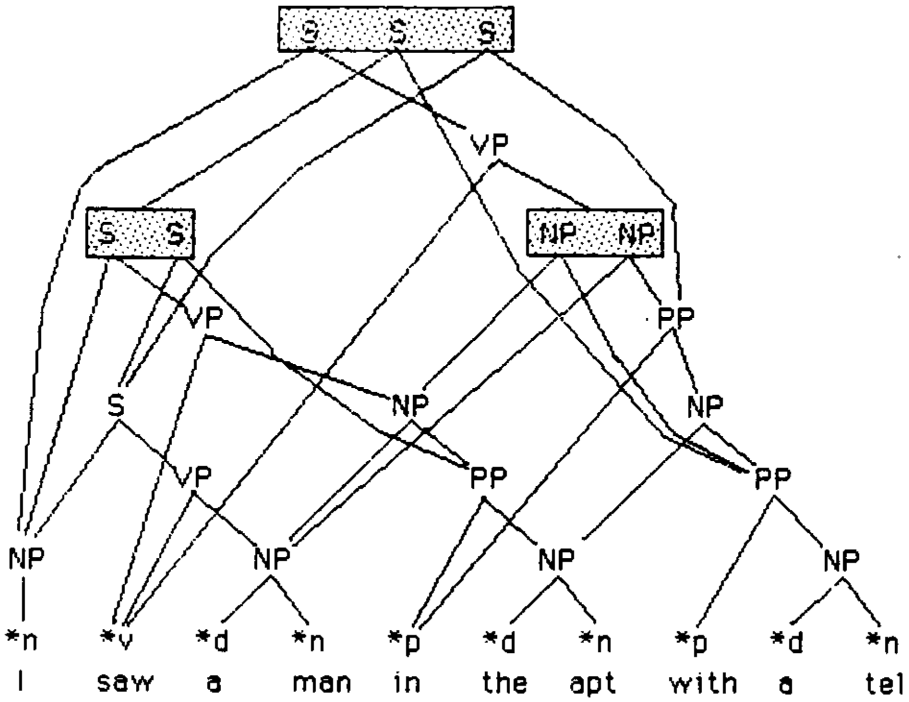


**Figure 4-2(b):** Packed Shared Trees

The local ambiguity packing can be easily implemented with the MLR parsing as follows. In the graph-structured stack, if two or more edges have a common starting point and a common ending point, they represent local ambiguity, and the parser considers them as if there is only one edge. In Figure 3-3 for example, we see one 5-way local ambiguity and two 2-way local ambiguities. Figure 4-3 shows the snapshot of the graph-structured stack and the shared trees right after shifting the word "with" in the sentence "I saw a man on the bed in the apartment with a telescope."

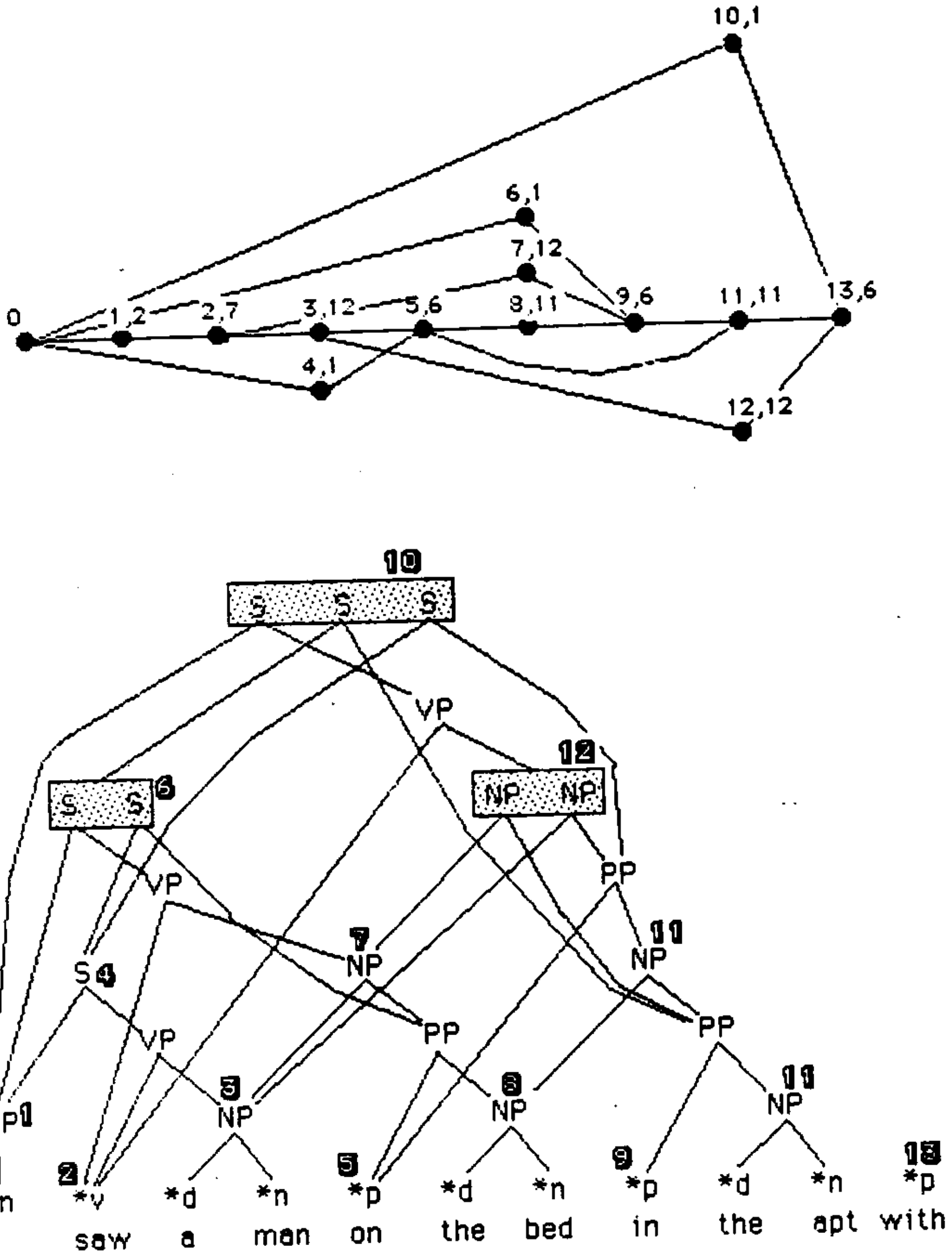


**Figure 4-2:** Graph-Structured Stack and Packed Shared Trees

# 5. Comparison with Other Methods

In this section, we compare the MLR parsing with other existing parsing methods such as LR, ATN, Cocke-Younger-Kasami, Earley and Chart parsing.

## 5.1. LR parsers

There are several kinds of LR parsers: LR(0), SLR(1), LALR(1), LR(1), LR(k), etc. LR(0) is the simplest and the least efficient LR parser, because it does not manage "look ahead" at all. On the other hand, LR(k) is the most efficient, but its parsing construction is the most complex. YACC is LALR(1). These LR parsers differ from each other only by the LR table construction algorithm; the parsing algorithm itself is common to all these LR parsers. Since the parsing table construction of the MLR parser does not differ from that of LR parsers, we can also think of several kinds of MLR parsers: MLR(0), MSLR(1), MLALR(1), MLR(1), MLR(k), etc., according to the parsing table construction algorithm. The MLR parser whose parsing table is obtained by YACC is therefore MLALR(1).

LR parsing can be considered as a special case of MLR parsing. If the grammar is LR, MLR parsers behave in exactly the same manner as LR parsers.

## 5.2. Other One-path Parsers

Although the LR parsers and PARSIFAL [15] (theoretically LR(k) [4]) cannot handle ambiguous sentences, other one-path parsers, including ATN [27], DCG [17], BUP [16] and LINGOL [18], can simulate the all-paths parsing, producing all possible parses by backtracking. However, the number of times they must backtrack grows exponentially as the sentence ambiguity grows, and they become unrealistic when a sentence has hundreds of possible parses.

## 5.3. Cocke-Younger-Kasami Algorithm

Cocke-Younger-Kasami (CYK) algorithm operates basically bottom-up: all constructable subtrees are built exhaustively regardless of whether they are actually utilized by some higher structure trees. Thus, this algorithm produces many meaningless subtrees which are never used, wasting time and space. Moreover, while the MLR parser can detect ungrammaticality of a sentence as soon as an inconsistent word is read, the CYK algorithm cannot detect the ungrammaticality until the whole sentence is read.

### 5.4. Earley's Algorithm

Earley's algorithm is somewhat more efficient than CYK algorithm in the sense that it does not build all possible subtrees exhaustively. Also, Earley's algorithm is capable of detecting ungrammaticality as soon as an inconsistent word is read.

The disadvantage of this algorithm, compared with MLR, is that each time a word is read, the system must compute "a set of items" (cf. pp.207 in [3]). Computing sets of items requires the system to look all over the grammar rules, and it is particularly inefficient when the grammar is large. In MLR parsing, on the other hand, such sets of items are pre-computed at the parsing table construction time, and the results of the pre-computing are encoded implicitly in the parsing table. Therefore, the MLR parser does not have to compute sets of items during parsing.

### 5.5. Chart Parsing

To avoid confusion, it should be noted that there is no such thing as a "chart parsing algorithm". A *chart* [13] is the name of a data structure that represents the syntactic structure of a sentence. The most popular algorithm for chart parsing is the CYK algorithm because it is simple. However, Earley's algorithm can also be used for the chart parsing, and such a parser is called the *Active Chart Parser* [26]. Since we have already mentioned these two algorithms, we do not compare the MLR parser with chart parsers incorporating these two algorithms.

The graph-structured stack in the MLR parsing looks very similar to the chart. Although we have viewed MLR parsing as a generalized and extended version of LR parsing, we can also view MLR parsing as an extended version of chart parsing, which is guided by an LR parsing table. The major extension is that nodes in the chart contain information about the LR state as well as information about position of the node in the sentence. Thus, unlike a conventional chart, there may be more than one node at any position of a sentence. New edges and nodes are then created according to the LR parsing table. It might be interesting to redesign the MLR parsing algorithm as the extended chart parsing, although we do not discuss this matter further.

# 6. Future Work

In this section we enumerate future work to be completed in a year or so. In brief, we will implement the algorithm, experiment with a fairly large English grammar, compare our algorithm with other parsing algorithms and discuss its applications. The following subsections describe the future work in more detail.

## 6.1. Implementation of the algorithm

We will implement our algorithm fully. The program will be written in Mac Lisp on CMU-CS-C (Tops-20). Moreover, we will implement the following extension in order to parse natural language in a more flexible manner.

- Handling Multi-part-of-speech Words: Some words in English such as "saw" have more than one grammatical categories. We will solve this problem elegantly as follows. When such a word is encountered, the parsing table can immediately tell us which of its categories are legal. And if more than one of its categories are legal, the parser behaves as if a multiple entry were encountered. Little effort should be required to extend our parser to handle this.

- Handling Unknown Words: Moreover, our parser will be able to parse a sentence with unknown words. Unknown words can be considered as a special multi-part-of-speech word whose categories can be any legal categories.

## 6.2. Experiment on the Parser Efficiency

We will construct parsing tables for a tiny (10 rule), a small (60 rule) and a fairly large (400 rule) English grammar. We will either run YACC or write our own program for table construction. If we utilize YACC, we need to write a program that converts YACC output into a parsing table in our representation.

We will then have the parser actually parse an appropriate set of English sentences, in order to find out:

1. The parsing time with respect to the length of a sentence.

2. The parsing time with respect to the ambiguity of a sentence.

3. The parsing time with respect to a grammar size.

### 6.3. Comparison with Other Algorithms

We will compare our algorithm in detail with at least two other algorithms: CYK algorithm and Earley's algorithm. The comparison will be made in one of the following ways.

1. To prove mathematically that our algorithm dominates the other algorithm(s) under some reasonable assumptions.

2. To define some reasonable "primitive operations" and compare the number of operations. Note that we need not to implement the other algorithm(s).

3. To implement the other algorithm(s) on the same machine and in the same programming language and compare their execution times.

### 6.4. Discussion of Applications

The following discussions will be made.

- Incorporation of Semantics: We will discuss incorporation of Knowledge Representation Language (KRL) [5] with our parsing algorithm so that only semantically correct parses are produced.

- Application to Interactive Parser: We will discuss the technique of interactive sentence disambiguation [20] and discuss incorporation of this technique with our parsing algorithm. This technique requires having all possible parsers in advance, out of which the system asks its user questions to disambiguate a sentence.

- Application to Machine Translation: We will discuss an application of the interactive parser to personal/interactive machine translation systems [25, 22, 23, 24, 19].

# 7. Potential Contributions

Finally, we point out potential contributions of our work.

## 7.1. To Parsing Theory

We extend the LR parsing algorithm so as to handle an arbitrary context-free grammar with little loss of the LR efficiency. Although its upper bound on the time needed in the worst case is O($n^3$) as others, its coefficient is significantly reduced due to utilization of an LR parsing table.

## 7.2. To Practical Systems

We give an efficient all-paths parsing algorithm, which would be required in practical systems such as personal/interactive machine translation systems with the interactive sentence disambiguation.

## 7.3. To Computational Linguistics

An obvious application is to the possible implementation of GPSG [10], in which grammar rules are eventually represented purely as a context-free phrase structure grammar. Also, our algorithm could apply to implementation of functional grammars such as Lexical Functional Grammar (LFG) [6] and Unification Grammar (UG) [14], in which a base structure is specified by a context-free phrase structure grammar. Our algorithm would be particularly suitable in case sentences are highly ambiguous.

# Acknowledgements

I would like to thank Jaime Carbonell, Herb Simon, Phil Hayes, Ralph Grishman, Takehiro Tokuda and Osamu Watanabe for thoughtful comments on an earlier version of this paper and Cynthia Hibbard for helping to produce this document.

# 8. Appendix

## 8.1. LR parsers for Natural Languages

## 8.2. Disambiguating Grammatically Ambiguous Sentence by Asking

# References

[1] Aho, A. V. and Ullman, J. D.
*The Theory of Parsing, Translation and Compiling.*
Prentice-Hall, Englewood Cliffs, N. J., 1972.

[2] Aho, A. V. and Johnson, S. C.
LR parsing.
*Computing Surveys* 6:2:99-124, 1974.

[3] Aho, A. V. and Ullman, J. D.
*Principles of Compiler Design.*
Addison Wesley, 1977.

[4] Berwick, R. C.
A Deterministic Parser with Broad Coverage.
*Proceedings of IJCAI83* ;pp.710, August, 1983.

[5] Bobrow, D. G. and Winograd, T.
An Overview of KRL, a Knowledge Representation Language.
*Studies in Cognitive Science* Vol.1(No.1), 1977.

[6] Bresnan, J. and Kaplan, R.
*Lexical-Functional Grammar: A Formal System for Grammatical Representation.*
MIT Press, Cambridge, Massachusetts, 1982, pages pp. 173-281.

[7] Deremer, F. L.
*Practical Translators for LR(k) Languages.*
PhD thesis, MIT, 1969.

[8] DeRemer, F. L.
Simple LR(k) grammars.
*Comm. ACM* 14:7:453-460, 1971.

[9] Earley, J.
An Efficient Context-free Parsing Algorithm.
*Communication of ACM* (6:8):94-102, February, 1970.

[10] Gazdar, G.
*Phrase Structure Grammar.*
D. Reidel, 1982, pages 131-186.

[11] Gazdar, G.
Phrase Structure Grammars and Natural Language.
*Proceedings of IJCAI83* v.1, August, 1983.

[12] Johnson, S. C.
*YACC -- Yet Another Compiler Compiler.*
Technical Report CSTR 32, Bell Laboratories, 1975.

[13] Kay, M.
*The MIND System.*
Algorithmics Press, New York, 1973, pages pp.155-188.

[14] Kay, M.
Functional Grammar.
In *Fifth Annual Meeting of the Berkeley Linguistic Society*, pages pp. 142-158. Berkeley Linguistic Society, MIT Press, Berkeley, California, February, 1979.

[15] Marcus, M. P.
*A Theory of Syntactic Recognition for Natural Language.*
The MIT Press, Cambridge, Massachusetts, 1980.

[16] Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H. and Yasukawa, H.
BUP: A Bottom-Up Parser Embedded in Prolog.
*New Generation Computing* 1:pp.145-158, 1983.

[17] Pereira, F. and Warren, D.
Definite Clause Grammar for Language Analysis.
*Artificial Intelligence* 13:pp.231-278, May, 1980.

[18] Pratt, V. R.
LINGOL -- A Progress Report.
In *Proc. of 4th IJCAI*, pages pp.327-381. August, 1975.

[19] Saito, H. and Tomita, M.
On Automatic Composition of Foreign Letters.
In *IPSJ Symposium on Natural Language Processing*. Information Processing Society of Japan, 1984.

[20] Tomita, M.
Disambiguating Grammatically Ambiguous Sentences by Asking.
In *10th International Conference on Computational LInguistics (COLING84)*. 1984.

[21] Tomita, M.
LR Parsers For Natural Language.
In *10th International Conference on Computational LInguistics (COLING84)*. 1984.

[22] Tomita, M., Nishida, T. and Doshita, S.
User Front-End for disambiguation in Interactive Machine Translation System.
In *IPSJ Symposium on Natural Language Processing*. Information Processing Society of Japan, (in Japanese), 1984.

[23] Tomita, M., Nishida, T. and Doshita, S.
An Interactive English-Japanese Machine Translation System.
*Transactions of Information Processing Society of Japan* (in Japanese; submitted), 1984.

[24] Tomita, M., Nishida, T. and Doshita, S.
Interactive Approach to Machine Translation.
*(Forthcoming)* , 1984.

[25] Tomita, M.
*The Design Philosophy of Personal Machine Translation System.*
Technical Report, Computer Science Department, Carnegie-Mellon University, 1984.

[26] Winograd, T.
*Language as a Cognitive Process.*
Addison-Wesley, 1983.

[27] Woods, W. A.
Transition Network Grammars for Natural Language Analysis.
*CACM* 13:pp.591-606, 1970.

10-th International Conference on
Computational Linguistics, 1984

# LR Parsers For Natural Languages[1]

**Masaru Tomita**
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

## Abstract

MLR, an extended LR parser, is introduced, and its application to natural language parsing is discussed. An LR parser is a shift-reduce parser which is deterministically guided by a parsing table. A parsing table can be obtained automatically from a context-free phrase structure grammar. LR parsers cannot manage ambiguous grammars such as natural language grammars, because their parsing tables would have multiply-defined entries, which precludes deterministic parsing. MLR, however, can handle multiply-defined entries, using a dynamic programming method. When an input sentence is ambiguous, the MLR parser produces all possible parse trees without parsing any part of the input sentence more than once in the same way, despite the fact that the parser does not maintain a chart as in chart parsing. Our method also provides an elegant solution to the problem of multi-part-of-speech words such as "that". The MLR parser and its parsing table generator have been implemented at Carnegie-Mellon University.

## 1 Introduction

LR parsers [1, 2] have been developed originally for programming language of compilers. An LR parser is a shift-reduce parser which is deterministically guided by a parsing table indicating what action should be taken next. The parsing table can be obtained automatically from a context-free phrase structure grammar, using an algorithm first developed by DeRemer [5, 6]. We do not describe the algorithm here, reffering the reader to Chapter 6 in Aho and Ullman [4]. The LR parsers have seldom been used for Natural Language Processing probably because:

1. It has been thought that natural languages are not context-free, whereas LR parsers can deal only with context-free languages.

2. Natural languages are ambiguous, while standard LR parsers can not handle ambiguous languages.

The recent literature [8] shows that the belief "natural languages are not context-free" is not necessarily true, and there is no reason for us to give up the context-freedom of natural languages. We do not discuss on this matter further, considering the fact that even if natural languages are not context-free, a fairly comprehensive grammar for a subset of natural language sufficient for practical systems can be written in context-free phrase structure. Thus, our main concern is how to cope with the ambiguity of natural languages, and this concern is addressed in the following section.

## 2 LR parsers and Ambiguous Grammars

If a given grammar is ambiguous,[2] we cannot have a parsing table in which every entry is uniquely defined; at least one entry of its parsing table is multiply defined. It has been thought that, for LR parsers, multiple entries are fatal because they make deterministic parsing no longer possible.

Aho *et. al.* [3] and Shieber [12] coped with this ambiguity problem by statically[3] selecting one desired action out of multiple actions, and thus converting multiply-defined entries into uniquely-defined ones. With this approach, every input sentence has no more than one parse tree. This fact is desirable for programming languages.

For natural languages, however, it is sometimes necessary for a parser to produce more than one parse tree. For example, consider the following short story.

I saw the man with a telescope.
He should have bought it at the department store.

When the first sentence is read, there is absolutely no way to resolve the ambiguity[4] at that time. The only action the system can take is to produce two parse trees and store them somewhere for later disambiguation.

In contrast with Aho *et. al.* and Shieber, our approach is to extend LR parsers so that they can handle multiple entries and produce more than one parse tree if needed. We call the extended LR parsers MLR parsers.

[1] This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

[2] A grammar is ambiguous, if some input sentence can be parsed in more than one way.

[3] By "statically", we mean the selection is done at parsing table construction time.

[4] "I" have the telescope, or "the man" has the telescope.

## 3 MLR Parsers

An example grammar and its MLR parsing table produced by the construction algorithm are shown in fig. 1 and 2, respectively. The MLR parsing table construction algorithm is exactly the same as the algorithm for LR parsers. Only the difference is that an MLR parsing table may have multiple entries. Grammar symbols starting with "*" represent pre-terminals. "sh *n*" in the action table (the left part of the table) indicates the action "shift one word from input buffer onto the stack, and go to state *n*". "re *n*" indicates the action "reduce constituents on the stack using rule *n*". "acc" stands for the action "accept", and blank spaces represent "error". Goto table (the right part of the table) decides to what state the parser should go after a reduce action. The exact definition and operation of LR parsers can be found in Aho and Ullman [4].

We can see that there are two multiple entries in the table; on the rows of state 11 and 12 at the column of "*prep". As mentioned above, once a parsing table has multiple entries, deterministic parsing is no longer possible; some kind of non-determinism is necessary. We shall see that our dynamic programming approach, which is described below, is much more efficient than conventional breath-first or depth-first search, and makes MLR parsing feasible.

Our approach is basically pseudo-parallelism (breath-first search). When a process encounters a multiple entry with n different actions, the process is split into n processes, and they are executed individually and parallelly. Each process is continued until either an "error" or an "accept" action is found. The processes are, however, synchronized in the following way: When a process "shifts" a word, it waits until all other processes "shift" the word. Intuitively, all processes always look at the same word. After all processes shift a word, the system may find that two or more processes are in the same state; that is, some processes have a common state number on the top of their stacks. These processes would do the exactly same thing until that common state number is popped from their stacks by some "reduce" action. In our parser, this common part is processed only once. As soon as two or more processes in a common state are found, they are combined into one process. This combining mechanism guarantees that any part of an input sentence is parsed no more than once in the same manner. This makes the parsing much more efficient than simple breath-first or depth-first search. Our method has the same effect in terms of parsing efficiency that posting and recognizing common subconstituents of different parses have in the chart parsing method [10, 11]. The idea should be made clear by the following example.

## 4 An Example

In this section, we demonstrate, step by step, how our MLR parser processes the sentence:

I SAW A MAN WITH A TELESCOPE

using the grammar and the parsing table shown in fig 1 and 2. This sentence is ambiguous, and the parser should accept the sentence in two ways.

Until the system finds a multiple entry, it behaves in the exact same manner as a conventional LR parser, as shown in fig 3-a below. The number on the top (rightmost) of the stack indicates the current state. Initially, the current state is 0. Since the parser is looking at the word "I", whose category is "*n", the next action "shift and goto state 4" is determined from the parsing table. The parser takes the word "I" away from the input buffer, and pushes the preterminal "*n" onto the stack. The next word the parser is looking at is "SAW", whose category is "*v", and "reduce using rule 3" is determined as the next action. After reducing, the parser determines the current state, 2, by looking at the intersection of the row of state 0 and the column of "NP", and so on.

| STACK | NEXT-ACTION | NEXT-WORD |
|---|---|---|
| 0 | sh 4 | I |
| 0 *n 4 | re 3 | SAW |
| 0 NP 2 | sh 7 | SAW |
| 0 NP 2 *v 7 | sh 3 | A |
| 0 NP 2 *v 7 *det 3 | sh 10 | MAN |
| 0 NP 2 *v 7 *det 3 *n 10 | re 4 | WITH |
| 0 NP 2 *v 7 NP 12 | re 7, sh 6 | WITH |

Fig 3-a

At this point, the system finds a multiple entry with two different actions, "reduce 7" and "shift 6". Both actions are processed in parallel, as shown in fig 3-b.

```
(1)   S --> NP VP
(2)   S --> S PP
(3)  NP --> *n
(4)  NP --> *det *n
(5)  NP --> NP PP
(6)  PP --> *prep NP
(7)  VP --> *v NP
```

Fig 1

| State | *det | *n | *v | *prep | $ | NP | PP | VP | S |
|---|---|---|---|---|---|---|---|---|---|
| 0 | sh3 | sh4 | | | | 2 | | | 1 |
| 1 | | | | sh6 | acc | | 5 | | |
| 2 | | | sh7 | sh6 | | | 9 | 8 | |
| 3 | | sh10 | | | | | | | |
| 4 | | | re3 | re3 | re3 | | | | |
| 5 | | | | re2 | re2 | | | | |
| 6 | sh3 | sh4 | | | | 11 | | | |
| 7 | sh3 | sh4 | | | | 12 | | | |
| 8 | | | | re1 | re1 | | | | |
| 9 | | | re5 | re5 | re5 | | | | |
| 10 | | | re4 | re4 | re4 | | | | |
| 11 | | | re6 | re6,sh6 | re6 | | 9 | | |
| 12 | | | | re7,sh6 | re7 | | 9 | | |

Fig 2

```
--------------------------------------------------------------
0 NP 2 VP 8                                 re 1         WITH
0 NP 2 *v 7 NP 12 *prep 6                   wait            A

0 S 1                                       sh 6         WITH
0 NP 2 *v 7 NP 12 *prep 6                   wait            A

0 S 1 *prep 6                               sh 3            A
0 NP 2 *v 7 NP 12 *prep 6                   sh 3            A
--------------------------------------------------------------
```

Fig 3-b

Here, the system finds that both processes have the common state number, 6, on the top of their stacks. It combines two processes into one, and operates as if there is only one process, as shown in fig 3-c.

```
--------------------------------------------------------------
0 S 1             *prep 6                  sh 3            A
0 NP 2 *v 7 NP 12

0 S 1             *prep 6 *det 3           sh 10   TELESCOPE
0 NP 2 *v 7 NP 12

0 S 1             *prep 6 *det 3 *n 10     re 4            $
0 NP 2 *v 7 NP 12

0 S 1             *prep 6 NP 11            re 6            $
0 NP 2 *v 7 NP 12
--------------------------------------------------------------
```

Fig 3-c

The action "reduce 6" pops the common state number 6, and the system can no longer operate the two processes as one. The two processes are, again, operated in parallel, as shown in fig 3-d.

```
--------------------------------------------------------------
0 S 1 PP 5                                  re 2           $
0 NP 2 *v 7 NP 12 PP 9                      re 5           $

0 S 1                                       accept
0 NP 2 *v 7 NP 12                           re 7           $
--------------------------------------------------------------
```

Fig 3-d

Now, one of the two processes is finished by the action "accept". The other process is still continued, as shown in fig 3-e.

```
--------------------------------------------------------------
0 NP 2 VP 8                                 re 1           $
0 S 1                                       accept
--------------------------------------------------------------
```

Fig 3-e

This process is also finished by the action "accept". The system has accepted the input sentence in both ways. It is important to note that any part of the input sentence, including the prepositional phrase "WITH A TELESCOPE", is parsed only once in the same way, without maintaining a chart.

## 5 Another Example

Some English words belong to more than one grammatical category. When such a word is encountered, the MLR parsing table can immediately tell which of its categories are legal and which are not. When more than one of its categories are legal, the parser behaves as if a multiple entry were encountered. The idea should be made clear by the following example.

Consider the word "that" in the sentence:

That information is important is doubtful.

A sample grammar and its parsing table are shown in Fig. 4 and 5, respectively. Initially, the parser is at state 0. The first word "that" can be either "*det" or "*that", and the parsing table tells us that both categories are legal. Thus, the parser processes "sh 5" and "sh 3" in parallel, as shown below.

```
  STACK                         NEXT ACTION  NEXT WORD
-------------------------------------------------------
0                               sh 5, sh 3     That

0                                  sh 5        That
0                                  sh 3        That

0 *det 5                           sh 9      information
0 *that 3                          sh 4      information

0 *det 5 *n 9                      re 2          is
0 *that 3 *n 4                     re 3          is

0 NP 2                             sh 6          is
0 *that 3 NP 2                     sh 6          is
-------------------------------------------------------
```

Fig. 6-a

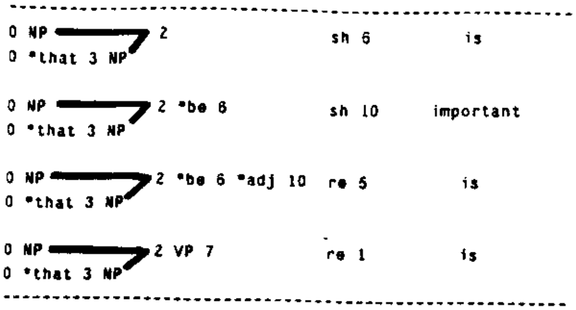
At this point, the parser founds that both processes are in the same state, namely state 2, and they are combined as one process.

```
------------------------
 (1)   S --> NP VP
 (2)  NP --> *det *n
 (3)  NP --> *n
 (4)  NP --> *that S
 (5)  VP --> *be *adj
------------------------
```

Fig. 4

| State | *adj | *be | *det | *n | *that | $ | NP | S | VP |
|---|---|---|---|---|---|---|---|---|---|
| 0 | | | sh6 | sh4 | sh3 | | 2 | 1 | |
| 1 | | | | | | acc | | | |
| 2 | | sh6 | | | | | | | 7 |
| 3 | | | sh5 | sh4 | sh3 | | 2 | 8 | |
| 4 | | re3 | | | | | | | |
| 5 | | | | sh9 | | | | | |
| 6 | sh10 | | | | | | | | |
| 7 | | re1 | | | | re1 | | | |
| 8 | | re4 | | | | | | | |
| 9 | | re2 | | | | | | | |
| 10 | | re6 | | | | re6 | | | |

Fig. 5



```
-------------------------------------------------------
0 NP          2                      sh 6          is
0 *that 3 NP

0 NP          2 *be 6                sh 10      important
0 *that 3 NP

0 NP          2 *be 6 *adj 10        re 5          is
0 *that 3 NP

0 NP          2 VP 7                 re 1          is
0 *that 3 NP
-------------------------------------------------------
```

Fig. 6-b

The process is split into two processes again.

```
-------------------------------------------------------
0 NP 2 VP 7                          re 1        is
0 *that 3 NP 2 VP 7                  re 1        is

0 S 1                              #ERROR#       is
0 *that 3 S 8                        re 4        is
-------------------------------------------------------
```

Fig. 6-c .

One of two processes detects "error" and halts; only the other process goes on.

```
-------------------------------------------------------
0 NP 2                               sh 6          is
0 NP 2 *be 6                         sh 10      doubtful
0 NP 2 *be 6 *adj 10                 re 5          $
0 NP 2 VP 7                          re 1          $
0 S 1                                acc           $
-------------------------------------------------------
```

Fig. 6-d

Finally, the sentence has been parsed in only one way. We emphasize again that, in spite of pseudo-parallelism, each part of the sentence was parsed only once in the same way.

## 6 Concluding Remarks

The MLR parser and its parsing table generator have been implemented at Computer Science Department, Carnegie-Mellon University. The system is written in MACLISP and running on Tops-20.

One good feature of an MLR parser (and of an LR parser) is that, even if the parser is to run on a small computer, the construction of the parsing table can be done on more powerful, larger computers. Once a parsing table is constructed, the execution time for parsing depends weakly on the number of productions or symbols in a grammar. Also, in spite of pseudo-parallelism, our MLR parsing is theoretically still deterministic. This is because the number of processes in our pseudo-parallelism never exceeds the number of states in the parsing table.

One concern of our parser is whether the size of a parsing table remains tractable as the size of a grammar grows. Fig. 6 shows the relationship between the complexity of a grammar and its LR parsing table (excerpt from Inoue [9]).

| | XPL | EULER | FORTRAN | ALGOL60 |
|---|---|---|---|---|
| Terminals | 47 | 74 | 63 | 66 |
| Non-terminals | 51 | 45 | 77 | 99 |
| Productions | 108 | 121 | 172 | 205 |
| States | 180 | 193 | 322 | 337 |
| TableSize(byte) | 2041 | 2587 | 3662 | 4264 |

Fig. 6

Although the example grammars above are for programming langauges, it seems that the size of a parsing table grows only in proportion to the size of its grammar and does not grow rapidly. Therefore, there is a hope that our MLR parsers can manage grammars with thousands of phrase structure rules, which would be generated by rule-schema and meta-rules for natural language in systems such as GPSG [7].

## Acknowledgements

I would like to thank Takehiro Tokuda, Osamu Watanabe, Jaime Carbonell and Herb Simon for thoughtful comments on an earlier version of this paper.

## References

[1] Aho, A. V. and Ullman, J. D.
*The Theory of Parsing, Translation and Compiling.*
Prentice-Hall, Englewood Cliffs, N. J., 1972.

[2] Aho, A. V. and Johnson, S. C.
LR parsing.
*Computing Surveys* 6:2:99-124, 1974.

[3] Aho, A. V., Johnson, S. C. and Ullman, J. D.
Deterministic parsing of ambiguous grammars.
*Comm. ACM* 18:8:441-452, 1975.

[4] Aho, A. V. and Ullman, J. D.
*Principles of Compiler Design.*
Addison Wesley, 1977.

[5] Deremer, F. L.
*Practical Translators for LR(k) Languages.*
PhD thesis, MIT, 1969.

[6] DeRemer, F. L.
Simple LR(k) grammars.
*Comm. ACM* 14:7:453-460, 1971.

[7] Gazdar, G.
*Phrase Structure Grammar.*
D. Reidel, 1982, pages 131-186.

[8] Gazdar, G.
Phrase Structure Grammars and Natural Language.
*Proceedings of the Eighth International Joint Conference on Artificial Intelligence* v.1, August, 1983.

[9] Inoue, K. and Fujiwara, F.
On LLC(k) Parsing Method of LR(k) Grammars.
*Journal of Information Processing* vol.6(no.4):pp.206-217, 1983.

[10] Kaplan, R. M.
*A general syntactic processor.*
Algorithmics Press, New York, 1973, pages 193-241.

[11] Kay, M.
*The MIND system.*
Algorithmics Press, New York, 1973, pages 155-188.

[12] Shieber, S. M.
Sentence Disambiguation by a Shift-Reduce Parsing Technique.
*Proceedings of the Eighth International Joint Conference on Artificial Intelligence* v.2, August, 1983.

10-th International Conference on
Computational Linguistics, 1984

# Disambiguating Grammatically Ambiguous Sentences By Asking

**Masaru Tomita**
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

## Abstract

The problem addressed in this paper is to disambiguate grammatically ambiguous input sentences by asking the user, who need not be a computer specialist or a linguist, without showing any parse trees or phrase structure rules. Explanation List Comparison (ELC) is the technique that implements this process. It is applicable to all parsers which are based on phrase structure grammar, regardless of the parser implementation. An experimental system has been implemented at Carnegie-Mellon University, and it has been applied to English-Japanese machine translation at Kyoto University.

## 1. Introduction

A large number of techniques using semantic information have been developed to resolve natural language ambiguity. However, not all ambiguity problems can be solved by those techniques at the current state of art. Moreover, some sentences are *absolutely* ambiguous, that is, even a human cannot disambiguate them. Therefore, it is important for the system to be capable of asking a user questions interactively to disambiguate a sentence.

Here, we make an important condition that an user is neither a computer scientist nor a linguist. Thus, an user may not recognize any special terms or notations like a tree structure, phrase structure grammar, etc.

The first system to disambiguate sentences by asking interactively is perhaps a program called "disambiguator" in Kay's MIND system [2]. Although the disambiguation algorithm is not presented in [2], some basic ideas have been already implemented in the Kay's system[2]. In this paper, we shall only deal with grammatical ambiguity, or in other words, syntactic ambiguity. Other ambiguity problems, such as word-sense ambiguity and referential ambiguity, are excluded.

Suppose a system is given the sentence:

"Mary saw a man with a telescope"

and the system has a phrase structure grammar including the following rules <a> - <g>:

```
<a>     S --> NP + VP
<b>     S --> NP + VP + PP
<c>     NP --> *noun
<d>     NP --> *det + *noun
<e>     NP --> NP + PP
<f>     PP --> *prep + NP
<g>     VP --> *verb + NP
```

The system would produce two parse trees from the input sentence (I. using rules <b>,<c>,<g>,<d>,<f>,<d>; II. using rules <a>,<c>,<g>,<e>,<d>,<f>,<d>). The difference is whether the preposition phrase "with a telescope" qualifies the noun phrase "a man" or the sentence "Mary saw a man". This paper shall discuss on how to ask the user to select his intended interpretation without showing any kind of tree structures or phrase structure grammar rules. Our desired question for that sentence is thus something like:

1) The action "Mary saw a man" takes place "with a telescope"
2) "a man" is "with a telescope"
NUMBER ?

The technique to implement this, which is described in the following sections, is called *Explanation List Comparison*.

## 2. Explanation List Comparison

The basic idea is to attach an *Explanation Template* to each rule. For example, each of the rules <a> - <g> would have an explanation template as follows:

```
          Explanation Template

<a>       (1) is a subject of the action (2)
<b>       The action (1 2) takes place (3)
<c>       (1) is a noun
<d>       (1) is a determiner of (2)
<e>       (1) is (2)
<f>       (1) is a preposition of (2)
<g>       (2) is an object of the verb (1)
```

Whenever a rule is employed to parse a sentence, an explanation is generated from its explanation template. Numbers in an explanation template indicate n-th constituent of the right hand side of the rule. For instance, when the rule <f>

```
    PP --> *prep + NP
```

matches "with a telescope" (*prep = "WITH"; NP = "a

[1] This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

[2] Personal communication.

**telescope**"), the explanation

"(with) is a preposition of (a telescope)"

is generated. Whenever the system builds a parse tree, it also builds a list of explanations which are generated from explanation templates of all rules employed. We refer to such a list as an *explanation list*. The explanation lists of the parse trees in the example above are:

**Alternative I.**

<b> The action (Mary saw a man) takes place (with a telescope)
<c> (Mary) is a noun
<g> (a man) is an object of the verb (saw)
<d> (A) is a determiner of (man)
<f> (with) is a preposition of (a telescope)
<d> (A) is a determiner of (telescope)

**Alternative II.**

<a> (Mary) is a subject of the action (saw a man with a telescope)
<c> (Mary) is a noun
<g> (a man with a telescope) is an object of the verb (saw)
<e> (a man) is (with a telescope)
<d> (A) is a determiner of (man)
<f> (with is a preposition of (a telescope)
<d> (A) is a determiner of (telescope)

In order to disambiguate a sentence, the system only examines these Explanation Lists, but not parse trees themselves. This makes our method independent from internal representation of a parse tree. Loosely speaking, when a system produces more than one parse tree, explanation lists of the trees are "compared" and the "difference" is shown to the user. The user is, then, asked to select the correct alternative.

## 3. The revised version of ELC

Unfortunately, the basic idea described in the preceding section does not work quite well. For instance, the difference of the two explanation lists in our example is

1)
The action (Mary saw a man) takes place (with a telescope),
(a man) is an object of the verb (saw);
2)
(Mary) is a subject of the action (saw a man with a telescope),
(a man with a telescope) is an object of the verb (saw),
(a man) is (with a telescope);

despite the fact that the essential difference is only

1) The action (Mary saw a man) takes place (with a telescope)
2) (a man) is (with a telescope)

Two refinement ideas, *head* and *multiple explanations*, are introduced to solve this problem.

### 3.1. Head

We define *head* as a word or a minimal cluster of words which are syntactically dominant in a group and could have the same syntactic function as the whole group if they stood alone. For example, the head of "VERY SMART PLAYERS IN NEW YORK" is "PLAYERS", and the head of "INCREDIBLY BEAUTIFUL" is "BEAUTIFUL", but the head of "I LOVE CATS" is "I LOVE CATS" itself. The idea is that, whenever the system shows a part of an input sentence to the user, only the head of it is shown. To implement this idea, each rule must have a *head definition* besides an explanation template, as follows.

| Rule | Head |
|---|---|
| <a> | [1 2] |
| <b> | [1 2] |
| <c> | [1] |
| <d> | [1 2] |
| <e> | [1] |
| <f> | [1 2] |
| <g> | [1 2] |

For instance, the head definition of the rule <b> says that the head of the construction "**NP + VP + PP**" is a concatenation of the head of 1-st constituent (NP) and the head of 2-nd constituent (VP). The head of "A GIRL with A RED BAG saw A GREEN TREE WITH a telescope" is, therefore, "A GIRL saw A TREE", because the head of "A GIRL with A RED BAG" (NP) is "A GIRL" and the head of "saw A GREEN TREE" (VP) is "saw A TREE".

In our example, the explanation

(Mary) is a subject of the action (saw a man with a telescope)

becomes

(Mary) is a subject of the action (saw a man),

and the explanation

(a man with a telescope) is an object of the verb (saw)

becomes

(a man) is an object of the verb (saw),

because the head of "saw a man with a telescope" is "saw a man", and the head of "a man with a telescope" is "a man".

The difference of the two alternatives are now:

1)
The action (Mary saw a man) take place (with a telescope);
2)
(Mary) is a subject of the action (saw a man),
(a man) is (with a telescope);

### 3.2. Multiple explanations

In the example system we have discussed above, each rule generates exactly one explanation. In general, multiple explanations (including zero) can be generated by each rule. For example, rule <b>

```
S --> NP + VP + PP
```

should have two explanation templates:

```
(1) is a subject of the action (2)
The action (1 2) takes place (3),
```

whereas rule <a>

```
S --> NP + VP
```

should have only one explanation template:

```
(1) is a subject of the action (2).
```

With the idea of head and multiple explanations, the system now produces the ideal question, as we shall see below.

### 3.3. Revised ELC

To summarize, the system has a phrase structure grammar, and each rule is followed by a head definition followed by an arbitrary number of explanation templates.

| Rule | Head | Explanation Template |
|---|---|---|
| <a> | [1 2] | (1) is a subject of the action (2) |
| <b> | [1 2] | (1) is a subject of the action (2)<br>The action (1 2) takes place (3) |
| <c> | [1] | <<none>> |
| <d> | [1 2] | (1) is a determiner of (2) |
| <e> | [1] | (1) is (2) |
| <f> | [1 2] | (1) is a preposition of (2) |
| <g> | [1 2] | (2) is an object of the verb (1) |

With the ideas of head and multiple explanation, the system builds the following two explanation lists from the sentence "Mary saw a man with a telescope".

**Alternative I.**

<b> (Mary) is a subject of the action (saw a man)
<b> The action (Mary saw a man) takes place (with a telescope)
<g> (a man) is an object of the verb (saw)
<d> (A) is a determiner of (man)
<f> (with) is a preposition of (a telescope)
<d> (A) is a determiner of (telescope)

**Alternative II.**

<a> (Mary) is a subject of the action (saw a man)
<g> (a man) is an object of the verb (saw)
<e> (a man) is (with a telescope)
<d> (A) is a determiner of (man)
<f> (with is a preposition of (a telescope)
<d> (A) is a determiner of (telescope)

The difference between these two is

The action (Mary saw a man) takes place (with a telescope)

and

(a man) is (with a telescope).

Thus, the system can ask the ideal question:

1) The action (Mary saw a man) takes place (with a telescope)
2) (a man) is (with a telescope)
Number?

## 4. More Complex Example

The example in the preceding sections is somewhat oversimplified, in the sense that there are only two alternatives and only two explanation lists are compared. If there were three or more alternatives, comparing explanation lists would be not as easy as comparing just two.

Consider the following example sentence:

Mary saw a man in the park with a telescope.

This sentence is ambiguous in 5 ways, and its 5 explanation lists are shown below.

**Alternative I.**

(a man) is (in the park)
(the park) is (with a telescope)
: :
: :

**Alternative II.**

(a man) is (with a telescope)
(a man) is (in the park)
: :
: :

**Alternative III.**

The action (Mary saw a man) takes place (with a telescope)
(a man) is (in the park)
: :
: :

**Alternative IV.**

The action (Mary saw a man) takes place (in the park)
(the park) is (with a telescope)
: :
: :

**Alternative V.**

The action (Mary saw a man) takes place (with a telescope)
The action (Mary saw a man) takes place (in the park)
: :
: :

With these 5 explanation lists, the system asks the user a question twice, as follows:

1) (a man) is (in the park)
2) The action (Mary saw a man) takes place (in the park)
NUMBER? 1

1) (the park) is (with a telescope)
2) (a man) is (with a telescope)
3) The action (Mary saw a man) takes place (with a telescope)
NUMBER? 3

The implementation of this is described in the following.

We refer to the set of explanation lists to be compared, $\{L_1, L_2, \ldots\}$, as $A$. If the number of explanation lists in $A$ is one ; just return the parsed tree which is associated with that explanation list. If there are more than one explanation list in $A$, the system makes a *Qlist* (Question list). The Qlist is a list of explanations

$$\text{Qlist} = \{ e_1, e_2, \ldots, e_n \}$$

which is shown to the user to ask a question as follows:

1) $e_1$
2) $e_2$
. .
. .
n) $e_n$
**Number?**

Qlist must satisfy the following two conditions to make sure that always exactly one explanation is true.

- Each explanation list $L$ in $A$ must contain at least one explanation $e$ which is also in Qlist. Mathematically, the following predicate must be satisfied.

  $$\forall L \exists e (e \in L \wedge e \in \text{Qlist})$$

  This condition makes sure that at least one of explanations in a Qlist is true.

- No explanation list $L$ in $A$ contains more than one explanation in a Qlist. That is,

$$\neg(\exists L \exists e \exists e'(L \in A \wedge e \in L \wedge e' \in L$$
$$\wedge\, e \in \text{Qlist} \wedge e' \in \text{Qlist} \wedge e \neq e')$$

This condition makes sure that at most one of explanations in Qlist is true.

The detailed algorithm of how to construct a Qlist is presented in Appendix.

Once a Qlist is created, it is presented to the user. The user is asked to select one correct explanation in the Qlist, called the *key explanation*. All explanation lists which do not contain the key explanation are removed from $A$. If $A$ still contains more than one explanation list, another Qlist for this new $A$ is created, and shown to the user. This process is repeated until $A$ contains only one explanation list.

## 5. Concluding Remarks

An experimental system has been written in Maclisp, and running on Tops-20 at Computer Science Department, Carnegie-Mellon University. The system parses input sentences provided by a user according to grammar rules and a dictionary provided by a super user. The system, then, asks the user questions, if necessary, to disambiguate the sentence using the technique of Explanation List Comparison. The system finally produces only one parse tree of the sentence, which is the intended interpretation of the user. The parser is implemented in a bottom-up, breath-first manner, but the idea described in the paper is independent from the parser implementation and from any specific grammar or dictionary.

The kind of ambiguity we have discussed is *structural* ambiguity. An ambiguity is structural when two different structures can be built up out of smaller constituents of the same given structure and type. On the other hand, an ambiguity is *lexical* when one word can serve as various parts of speech. Resolving lexical ambiguity is somewhat easier, and indeed, it is implemented in the system. As we can see in the Sample Runs below, the system first resolves lexical ambiguity in the obvious manner, if necessary.

Recently, we have integrated our system into an English-Japanese Machine Translation system [3], as a first step toward user-friendly interactive machine translation [6]. The interactive English Japanese machine translation system has been implemented at Kyoto University in Japan [4, 5].

## Acknowledgements

I would like to thank Jaime Carbonell, Harb Simon, Martin Kay, Jun-ich Tsujii, Toyoaki Nishida, Shuji Doshita and Makoto Nagao for thoughtful comments on an earlier version of this paper.

## Appendix A: Qlist-Construction Algorithm

```
input  $A$ : set of explanation lists
output Qlist : set of explanations
local  $e$ : explanation
       $L$ : explanation list (set of explanations)
       $U, C$ : set of explanation lists

1: $C \Leftarrow \phi$
2: $U \Leftarrow A$
3: Qlist $\Leftarrow \phi$
4: if $U = \phi$ then return Qlist
5: select one explanation $e$ such that
      $e$ is in some explanation list $\in U$,
      but not in any explanation list $\in C$;
      if no such $e$ exists, return ERROR
6: Qlist $\Leftarrow$ Qlist + $\{e\}$
7: $C \Leftarrow C + \{L \mid e \in L \wedge L \in U\}$
8: $U \Leftarrow \{L \mid e \notin L \wedge L \in (U)\}$
9: goto 4
```

- The input to this procedure is a set of explanation lists, $\{L_1, L_2, \ldots\}$.

- The output of this procedure is a list of explanations, $\{e_1, e_2, \ldots, e_n\}$, such that each explanation list, $L_i$, contains exactly one explanation which is in the Qlist.

- An explanation list L is called *covered*, if some explanation $e$ in L is also in Qlist. L is called *uncovered*, if any of the explanations in L is not in Qlist. $C$ is a set of covered explanation lists in $A$, and $U$ is a set of uncovered explanation lists in $A$.

- 1-3: initialization. Let Qlist be empty. All explanation lists in $A$ are uncovered.

- 4: if all explanation lists are covered, quit.

- 5-6: select an explanation $e$ and put it into Qlist to cover some of uncovered not explanation lists. $e$ must be such that it does exist in any of covered explanation lists (if it does exist, the explanation list has two explanation in $A$, violating the Qlist condition).

- 7-8: make uncovered explanation lists which are now covered by $e$ to be covered.

- 9: repeat the process until everything is covered.

## References

[1] Kay, M.
*The MIND System.*
Algorithmic Press, New York, 1973, .

[2] Nishida, T. and Doshita, S.
An Application of Montague Grammar to English-Japanese Machine Translation.
*Proceedings of conference on Applied Natural Language Processing* :156-165, 1983.

[3] Tomita, M., Nishida, T. and Doshita, S.
An Interactive English-Japanese Machine Translation System.
*Forthcoming* (in Japanese), 1984.

[4] Tomita, M., Nishida, T. and Doshita, S.
User Front-End for disambiguation in Interactive Machine Translation System.
In *Tech. Reports of WGNLP.* Information Processing Society of Japan, (in Japanese, forthcoming), 1984.

[5] Tomita, M.
*The Design Philosophy of Personal Machine Translation System.*
Technical Report, Computer Science Department, Carnegie-Mellon University, 1983.

## Appendix B: Sample Runs

```
(transline '(time flies like an arrow in Japan))

(---END OF PARSE-- 10 ALTERNATIVES)

(The word  TIME (1) is:)
(1 : VERB)
(2 : NOUN)
NUMBER> 2

(The word  FLIES (2) is:)
(1 : VERB)
(2 : NOUN)
NUMBER> 1

(1 : (AN ARROW) IS (IN JAPAN))
(2 : THE ACTION (TIME FLIES) TAKES PLACE (IN JAPAN))
NUMBER> 2

(S (NP (TIME *NOUN))
           (FLIES *VERB)
           (PP (LIKE *PREPOSITION) (NP (AN *DETERMINER) (ARROW *NOUN)))
           (PP (IN *PREPOSITION) (JAPAN *NOUN)))

(transline '(Mary saw a man in the apartment with a telescope))

(---END OF PARSE-- 5 ALTERNATIVES)

(1 : (A MAN) IS (IN THE APARTMENT))
(2 : THE ACTION (MARY SAW A MAN) TAKES PLACE (IN THE APARTMENT))
NUMBER> 1

(1 : (A MAN) IS (WITH A TELESCOPE))
(2 : (THE APARTMENT) IS (WITH A TELESCOPE))
(3 : THE ACTION (MARY SAW A MAN) TAKES PLACE (WITH A TELESCOPE))
NUMBER> 3

(S (NP (MARY *NOUN))
           (VP (SAW *VERB)
               (NP (NP (A *DETERMINER) (MAN *NOUN))
                   (PP (IN *PREPOSITION)
                       (NP (THE *DETERMINER) (APARTMENT *NOUN)))))
           (PP (WITH *PREPOSITION)
               (NP (A *DETERMINER) (TELESCOPE *NOUN))))
```